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

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

## Abstract

Web 2.0 applications place new and different demands on servers compared to their Web 1.0 counterparts. Simultaneously, the definitive arrival of pay-as-you-go "cloud computing" and the proliferation of application development stacks present new and different degrees of freedom in deploying and tuning software-as-a-service. We first identify non-obvious challenges and caveats to performing "apples-to-apples" comparisons of Web 2.0 application deployments. We then offer Cloudstone, a toolkit consisting of an open-source Web 2.0 social application (*Olio*), a set of automation tools for generating load and measuring its performance in different deployment environments, and a recommended set of constraints for computing a metric we believe makes more sense, *dollars per user per month*. By way of example we present preliminary measurements of both Rails and PHP versions of Cloudstone on a variety of Amazon Elastic Compute Cloud (EC2) configurations, discussing the challenges of comparing platforms or software stacks and how Cloudstone can help quantify the differences.

## 1. Why We Need New Workloads

Existing Web benchmarking tools (*ab, httperf,* SPECWeb) and applications (RuBiS, PetStore) are becoming less relevant to current practice in three ways. First, Web 2.0 application functionality induces new characteristics in 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 more broadly applicable. Lastly, debate continues over the performance differences among different development stacks, but we 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 through AJAX (Asynchronous JavaScript And XML), in which JavaScript code communicates with the server in the background, generates extra work on the server during what would otherwise be the Web user's "think time". On the other hand, well-coded AJAX may result in less rendering work on the server. A benchmark should help quantify these effects.

### 1.2. Cloud Computing is Different

Pay-as-you-go storage and computing, such as Amazon's EC2 and S3, change the economics of service deployment in two important ways. One is the much lower cost of "instant" incremental scalability. Another is that capacity can be quickly un-deployed to save money; developers need neither provision for 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. A Web 2.0 Application, Workload and Metrics

The *Cloudstone* toolkit addresses these requirements with three components:

**(1)** *Olio*[1], an open-source incubator project recently launched by Sun and UC Berkeley, consisting of two complete implementations of a social-event calendar web application (Ruby on Rails and PHP), and a sophisticated open-source workload generator, Faban[2], that can scale to thousands of simulated users and supports fine-grained time-varying workloads. Both 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, exposing architectural as well as implementation-specific deployment and tuning choices such as caching and database tuning parameters.

**(2)** A set of automation tools, including database population, metric gathering, etc., for running large *Olio*

[1] http://incubator.apache.org/projects/olio.html
[2] http://faban.sunsource.net

experiments on cloud computing environments such as Amazon Elastic Compute Cloud (EC2).

**(3)** A recommended methodology and set of parameters for computing the key metric of *dollars per user per month* in a deployment of the application.

Cloudstone consists 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. We include some preliminary results of Cloudstone measurement on Amazon EC2. It is our expectation that CloudStone can be used to systematically investigate questions such as: How do different development stacks trade 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 manycore 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

Our goal is to capture "typical" Web 2.0 functionality in a datacenter or cloud computing environment, and provide for consistent evaluation of how many ***dollars per user per month*** an installation can support under typical production conditions. To this end we provide a realistic workload generator, allow flexibility in deployment to support a range of best practices for caching and tuning, and allow for testing and data collection of a variety of scenarios, including stress testing, linear scaling, "hockey stick" surges, etc.

*Non-goal:* arguing for any one development stack over another. Many factors influence the choice of a development stack, and at best CloudStone will help developers systematically quantify some of the effects of those choices.

*Non-goal*: emulating legacy applications. Our choice of application features and development stacks reflects popular design points for Web 2.0 application design today.

*Non-goal:* providing the "best" (fastest, most memory-efficient, etc.) implementation of this particular application. Our goal is code that exploits each platform's strengths and idioms 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

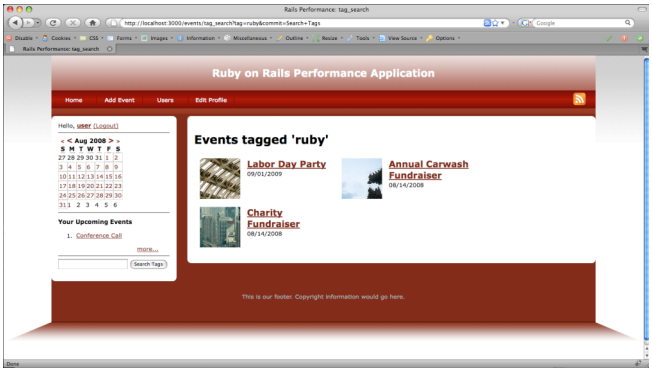


**Figure 1.** The *Olio* "social events" application's functionality and implementation are representative of Web 2.0 in both implementations: Ruby on Rails and PHP.

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 *Olio* 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; both implementations share similar CSS stylesheets and identical XHTML markup, 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 uses MySQL and includes a loader application to populate it with deterministic dummy data.

### 2.4. Workload Generation

Faban is an open source[3] Markov-chain, closed-loop [6] , session-based [3] synthetic workload generator. (See [4] for an overview of approaches to Web load testing.) Unlike simpler workload generators such as *ab* or *httperf*, Faban 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.). An $N \times N$ matrix $M$ gives the probability $M_{ij}$ that workflow $j$ will follow workflow $i$; this matrix can be derived from site-specific estimates [1] or by clustering information in web server logs [4] . Many parallel Faban agents on different machines can act under the control of a central coordinator, allowing distributed workload generation. The number of simulated users can be changed at any time during the course of the run according to a text file specifying a workload profile. Faban uses a standard inter-arrival time model for spacing of operations that may incorporate multiple requests.

[3] http://faban.sunsource.net

### 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 90$^{th}$ and 99$^{th}$ percentiles 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 and graphs the results using *fenxi*. All test data can be exported for further analysis.

Cloudstone includes automation scripts that implement *deploy, undeploy, restart* and *configure* actions for databases, web servers, application servers, and load balancers in a deployment environment. The scripts currently work with EC2, but are easily modified for other environments.

## 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. For example, Rails provides built-in abstractions for caching, whereas PHP leaves it to the developer to design and implement a caching strategy. Providing multiple tuning options allows developers to compare different strategies and determine how best to implement these strategies in their applications.

A *platform comparison* compares the behavior of the same software in different hardware environments, e.g. on different server types; different hardware topologies and virtualization complicate this comparison. To help potential users of Cloudstone, we outline our tuning methodology and point out caveats where the choice of platform or other tuning can trump other effects or otherwise distort results.

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 to hand-code all SQL queries, which allows more optimizations for experienced developers but increases the burden on less-experienced developers (who may write inefficient queries). In contrast, 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 some query optimizations. Even within a framework, different versions may require changes to database query strategy; for example, the synthesis of join queries 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

The preferred deployment topologies are different for Rails and PHP, as the figures below illustrate. 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 (www.nginx.net), to name a few.

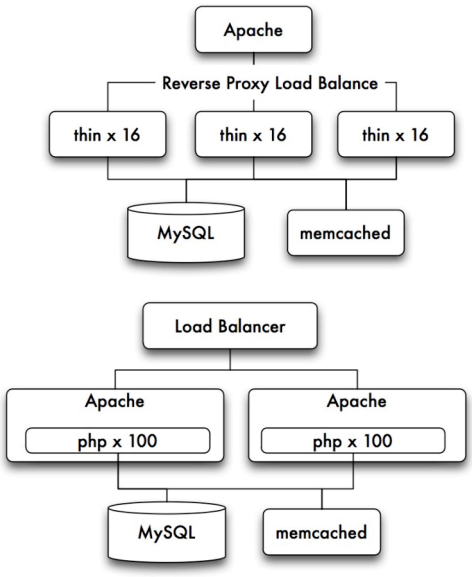


**Figure 2.** Whereas the preferred RoR deployment uses Apache process as a load balancer and multiple Rails application servers, the preferred PHP deployment uses many Apache+ *mod_php* workers.

Although we did not make use of a hardware SSL accelerator, such components are external to the application stack and would affect all measurements equivalently.

### 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. At any rate, in this paper we report results using RAM-based caching only.

## 4. Computing Dollars per User per Month

Given all these degrees of freedom, even defining the general measurement of *dollars per user per month* is not trivial. We therefore propose the following methodology for computing and reporting this metric.

### 4.1. Benchmark Run to Compute Number of Users

We first characterize the system *size* as U=$\log_{10}$(# of user accounts in database). As of this writing, for Facebook, U=8 (~110 million users). Results are only valid for U≥3.

The Cloudstone-supplied Faban configuration file defines response latency targets for each operation type in the workload. We define two SLA levels. SLA-1 (resp. SLA-2) requires 90% (resp. 99%) of the operations in any 5-minute window to complete within their allowed response times. Separate runs should be performed to report the maximum number of concurrent users that can be supported at SLA-1 and SLA-2; each run must be no shorter than 5 minutes. The CloudStone mix matrix is constructed such that at any time, the number of concurrent logged-in users is roughly 10% of the total number of user accounts in the database.

### 4.2. Dollars Per Month

Computing the dollar cost per month is installation-dependent. Cloud computing services such as EC2 make it relatively straightforward: all usage is charged by the hour or by number of bytes stored or transferred, with no startup or capital charges to depreciate. If hardware is racked and purchased, capital costs must be considered. Therefore, the cost per user per month must be given as three measurements: based on 1 month of operation, based on 1 year, and based on 3 years (the standard hardware-depreciation period for tax purposes), with a detailed description of how the cost per month was calculated. The table below summarizes the benchmark parameters and required and optional reporting.

**Table 1: CloudStone parameters & reporting**

| Metric | CloudStone value |
|---|---|
| U ($\log_{10}$(#users)) | Depends on system under test; minimum U=3 (1000 users) |
| SLA response-time threshold | Set per-operation in CloudStone mix matrix; varies from 1-4 seconds |
| SLA percentile | 90 and 99 (report both) |
| window for measuring SLA compliance | 5 minutes |

**Required reporting:**

1. CloudStone version and app variant (Rails or PHP)
2. Configuration file settings for database, front end, application server, etc. if different from defaults
3. Value of *U* ($\log_{10}$(# user accounts))
4. $/user/month for 1, 12, 36 months, at SLA-1 (90%ile)
5. $/user/month for 1, 12, 36 months, at SLA-2 (99%ile)

## 5. Experimental Setup

As the previous section illustrates, the many deployment options and components make exact comparisons between frameworks difficult. We focus on two questions: (1) *How many (concurrent) users can be served for a fixed dollar cost per month on Amazon EC2?* (2) *How many can be supported without replicating the MySQL database?*

### 5.1. Experimental Setup

Figure 4 shows the deployment template we use, with slight differences for running the Rails vs. PHP version of Cloudstone. Following accepted practice, since Rails processes run best in a dedicated single-threaded application server rather than as a part of a full-featured web server, we run the Rails application using a single load balancer/front end process and several *thin*[4] application server processes. We initially used Apache with *mod_proxy* as the load balancer but quickly switched to *nginx*. In contrast, the most common PHP deployment uses the *haproxy* load balancer and several Apache/*mod_php* processes as application servers, with Apache dynamically deciding how many *mod_php* workers to run, up to a specified maximum.

[4] http://code.macournoyer.com/thin

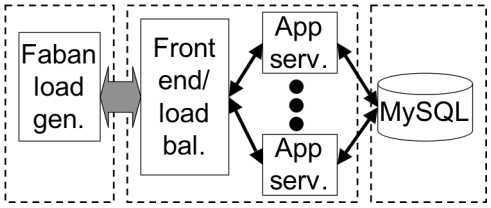


**Figure 3.** Experimental deployment setup on EC2; dotted lines are EC2 instance boundaries.

We measure two configurations on EC2. In the first configuration we limit ourselves to three EC2 instances: one for the load generator, one for the MySQL database, and one for the HTTP front end and application server processes. The goal of this configuration is to answer the question: *How many (concurrent) users can be supported for a fixed dollar amount?* In the second configuration we add up to two more EC2 instances hosting additional application server processes. The goal of this configuration is to answer the question: *How many users can be supported with a single nonreplicated MySQL instance, even if this means adding more application servers?,* that is, "balancing the pipeline" to keep a single MySQL busy.

We make use of two instance types available on EC2. Both are virtualized 64-bit x86 machines with the same (according to Amazon) I/O performance and 1.5TB of virtualized storage, but the M1.XL type has 15GB RAM and 8 "compute units" of CPU[5] (4 virtual cores x 2 units each) whereas the C1.XL type has 7GB RAM and 20 compute units (8 cores x 2.5 units each). This distinction will prove interesting, especially since the two instance types *cost the same to use,* namely $0.80 per instance-hour.

Prior to deployment, we used *sysbench* on MySQL and found that it achieves better performance on an C1.XL instance than an M1.XL instance (suggesting it is CPU-bound), topping out at 800 queries/second for a read-only workload and about 600 queries/second for a read-write workload. Therefore in all experiments we ran MySQL on its own C1.XL instance, the Faban load generator on a separate C1.XL instance, and several application server processes distributed over one to three additional C1.XL instances. Table 2 summarizes the measured configurations, including the two experiments where more than 1 instance was used to host application servers.

| # | App | Type | Comments (see text) |
|---|---|---|---|
| 1 | RoR | M1.XL | *mod_proxy* (load balancing) bottleneck |
| 2 | RoR+caching | M1.XL | |
| 3 | RoR | C1.XL | |
| 4 | RoR+caching | C1.XL | |
| 5 | RoR | C1.XL | 44 *thins*, 3 servers |
| 6 | RoR+caching | C1.XL | 28 *thins*, 2 servers |
| 7 | PHP+caching | C1.XL | |

**Table 2.** Summary of configurations measured.

[5] According to Amazon, 1 "compute unit" is equivalent to a 1.0–1.2 GHz Opteron or Xeon core in 2007.

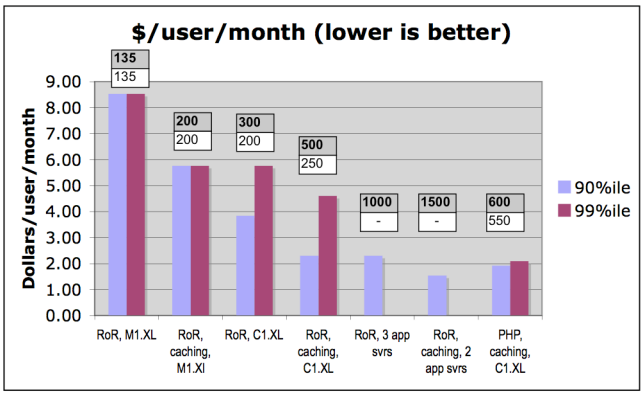


**Figure 5.** Cost per user per month for configurations of Table 2, at 90th and 99th percentile SLA. Numbers in shaded & unshaded boxes give number of simulated concurrent users at 90th and 99th percentile respectively.

### 5.2. Results and Observations

In this paper we report a subset of our results; we have collected and are publishing data for many more scenarios [7] . Figure 5 summarizes our answer to the two questions *How many users can be supported on a fixed dollar budget?* and *How many users can be supported using a single non-replicated MySQL instance?* To produce each data point, we manually perform various runs to "zero in" on the largest number of concurrent users that will still meet the 90th and 99th percentile SLA's for each configuration in Table 2. (We are working on simple scripts to automate this binary search process.) We then compute the monthly cost of the configuration by adding up the number of instances used (in all cases, 1 instance for the database and 1 for the application servers, except in scenarios 5 and 6 where we used respectively 2 and 3 instances for application servers) and assuming $0.80 per hour for 24x7 usage over a 30-day month. To simplify reporting we are not including a bandwidth cost estimate, but since outbound bandwidth varies directly with number of concurrent users, it will affect all measurements by a constant-factor multiple. Cloudstone requires reporting the monthly cost per user for 1, 12, and 36 months, but for EC2 these would all be the same due to its strictly pay-as-you-go pricing. Thus, for example, configuration #4 allows us to support 500 users for $2.30 per user per month if we wish to meet the 90th percentile SLA, or 250 users at $4.61 per user per month at the stricter 99th percentile SLA.[6]

In experiments 1–4 (all using the Rails version of Cloudstone), we limit ourselves to a single EC2 instance for the application servers to ask how many users can be supported for $1.60 per hour (one instance hosting appservers and one hosting MySQL, excluding bandwidth charges). In the first experiment, we quickly ran into an unexpected bottleneck due to the *mod_proxy* load balancer, and switched to *nginx* thereafter. But comparing experiments 2 and 4 is instructive, since the only difference is the choice of an M1.XL vs. a C1.XL instance (the two cost the same). In scenario 4 we

[6] We ran out of time to calculate the 99th percentile values for experiments 5 and 6, as we were using an older version of Faban that reported only the 90th percentile.

support twice as many users for the same price, confirming the anecdotal evidence that Rails is CPU-limited rather than memory-limited. A tentative conclusion might be: *other things being equal, Rails deployment will benefit from spending money for more CPU rather than more RAM.*

Enabling caching using Rails' abstractions for page, action and fragment-level caching (scenarios 2, 4, 6 compared to 1, 3, 5) improves throughput as expected, but more interesting is that we can support twice as many users at the 90th percentile SLA compared to the stringent (but more representative for larger sites) 99th percentile SLA, clearly indicating the "cost of an extra 9" of performance.

Lastly, in order to measure how many users can be supported until the database saturates, in scenarios 5 and 6 we add additional servers hosting more Rails application servers (*thin*s). As expected, until the database saturates, adding more appservers simply amortizes the cost of the database more effectively, so the cost per user decreases further in these scenarios. In general we experience diminishing returns on throughput with more than two *thin* servers per (virtual) core. (With PHP, we did not need to add additional application server machines to saturate the database.)

While "apples to apples" comparisons are impossible due to the number of variables involved and the comparison of two independent implementations of the same application, a tentative summary of Scenario 7 might be: "Rails is less hardware-efficient than PHP by a factor of 2 to 3." While imprecise, this suggests a quantitative response to the anecdotal skepticism that "Rails doesn't scale."

### 5.3. Discussion

Somewhat to our surprise, the process of running these benchmarks uncovered various interesting behaviors. We note that while some of these are consistent with anecdotes from colleagues, we now have a way to reproducibly quantify the effects and capture (in an Amazon virtual machine image) the specific configuration that illustrates each effect.

- Apache+*mod_proxy* was causing a premature bottleneck at only 135 concurrent users. Switching to the high-performance *nginx* load balancer fixed this.
- Saturating a single MySQL on an C1.XL instance required between two and three application instances.
- Rails is CPU-bound: f we can serve more users by hosting the *thin* servers on an C1.XL instance rather than an identically-priced M1.XL instance, whereas the Apache/PHP version is memory bound and does not exhibit this behavior.
- Turning off logging resulted in an approximately 20% throughput increase. We speculate that the I/O overheads associated with virtualization are responsible, but we need to investigate further to understand this effect.
- At 1,000 concurrent users, we approach the capacity limit of gigabit Ethernet. Note that this corresponds to the non-trivial volume of approximately 17 million hits per day.

## 6. Conclusions and Future Work

We have identified many degrees of freedom in deploying Web 2.0 applications—caching, hardware configuration, tuning, and options we haven't even considered such as hardware SSL acceleration and alternative interpreters (JRuby, IronRuby, etc.) Given all these variables and the controversy about which development stack to use, conventional "apples to apples" benchmarks may not be the right goal for cloud computing benchmarks if Web applications are the workload of interest.

We have contributed a "canonical" application, load generator and instrumentation, and a proposal for measuring *dollars per user per month* under representative constraints, in order to capture salient aspects of the performance of a particular deployment of Web 2.0 applications in cloud computing environments. We have already learned a great deal about many open-source software components and we will explore dynamic workloads ("hockey stick" growth, flash crowds, etc.) in future work. We hope others will find these tools as useful and illuminating as we have, and will consider adopting our methodology and metrics of merit for evaluating cloud computing deployment options.

## 7. Download & Acknowledgments

We will be making the Amazon machine images used for these experiments available at http://radlab.cs.berkeley.edu/wiki/Projects/Cloudstone. We hope others join the Olio incubator project and create implementations using other technology stacks.

Thanks to Peter Bodík for help with Faban modifications and Rean Griffith for discussion of benchmark metrics.

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 matching funds from the State of California 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) and the UC Discovery Grant (COM07-10240).

## 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

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

[7] W. Sobel et al. Scaling Ruby on Rails in a Cloud Computing Environment. UC Berkeley Technical Report EECS-2008-130 (in preparation).

[8] A. Sucharitakul and S. Subramanyam. *Cadillac or Nascar? A Non-Religious Investigation of Modern Web Technologies.* Proc. O'Reilly Velocity'08, http://en.oreilly.com/velocity2008