# Framework for Monitoring and Testing Web Application Scalability on the Cloud

Martti Vasar
Institute of Computer Science
University of Tartu
J. Liivi 2, Tartu, Estonia
martti.vasar@ut.ee

Satish Narayana Srirama
Institute of Computer Science
University of Tartu
J. Liivi 2, Tartu, Estonia
satish.srirama@ut.ee

Marlon Dumas
Institute of Computer Science
University of Tartu
J. Liivi 2, Tartu, Estonia
marlon.dumas@ut.ee

## ABSTRACT

By allowing resources to be acquired on-demand and in variable amounts, cloud computing provides an appealing environment for deploying pilot projects and for performance testing of Web applications and services. However, setting up cloud environments for performance testing still requires a significant amount of manual effort. To aid performance engineers in this task, we developed a framework that integrates several common benchmarking and monitoring tools. The framework helps performance engineers to test applications under various configurations and loads. Furthermore, the framework supports dynamic server allocation based on incoming load using a response-time-aware heuristics. We validated the framework by deploying and stress-testing the MediaWiki application. An experimental evaluation was conducted aimed at comparing the response-time-aware heuristics against Amazon Auto-Scale.

## Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; H.3.5 [**Online Information Services**]: Web-based services; H.4 [**Information Systems Applications**]: Miscellaneous; H.5.2 [**User Interfaces**]: Benchmarking

## General Terms

Performance

## Keywords

Cloud computing, web application, performance testing, auto-scaling

## 1. INTRODUCTION

Testing the quality of service (QoS) of web applications can be a grievous task. Using on-premise servers does not allow one to quickly re-configure the infrastructure. Also, the cost of short performance experiments for evaluating QoS is much higher using an on-premise infrastructure compared to the cloud, due to the large number of servers needed. Besides, the maintenance of the on-premise servers entails higher staffing levels, power consumption, and other cost-related factors. Cloud computing gives an appealing platform for developing and deploying pilot applications and for configuring them on the fly, for example to support performance testing under varying configurations. Cloud computing promises to reduce the cost of performance experiments by providing elasticity combined with a pay-per-use model.

While the cloud environment gives an appealing platform for performance testing, we still to set-up appropriate tools to monitor the performance and resource usage of the servers in the cloud in order to extract relevant parameters, such as how jobs enter into system and how they are divided across resources. Also, we need to test the system under varying configurations and loads. This requires having a streamlined process for re-configuring the servers on the cloud and re-running tests under different configurations and loads.

In this context, we studied different monitoring and configuration tools to understand their pros and cons vis-a-vis of different systems and proposed workflows. Most often the solutions found are targeted at particular cloud providers and access to the core resources is limited. To address these issues, we built a new framework supporting tasks required for performance and scalability testing such as managing (turning on/off) and monitoring the servers and extracting parameters related to the scalability of the system under test. The framework is designed to runn in Eucalyptus[1] private clouds as well as the Amazon EC2 public cloud[2]. Amazon EC2 is a popular public cloud where customers are charged per instance-hours. An instance is a virtual machine that is executed by a hypervisor (XEN [1] in the case of Amazon EC2) using virtual images provided by the user.

The proposed framework is designed to ease the burden of setting-up and executing performance and scalability tests of web applications on the cloud. This paper discusses our experiments with verifying the features of the framework with the MediaWiki[3], a popular web application simulating the Wikipedia setup. In the process, we also discuss the auto-scaling mechanism integrated in the proposed framework and we compare it with the Amazon Auto-Scale mechanism.

The paper is organized as follows. Section 2 briefly dis-


*WICSA/ECSA 2012* August 20-24, Helsinki, Finland.


[1] http://www.eucalyptus.com
[2] http://aws.amazon.com/ec2/
[3] http://www.mediawiki.org

cusses related work in this field. Section 3 presents the framework's configuration and its performance measurement model as well as the MediaWiki application, which is used as a sample application to validate the framework. Section 4 describes experiments aimed at comparing the performance of different Amazon instance types. Section 5 describes performance and scalability experiments on the MediaWiki application, including experiments involving the framework's auto scaling mechanism. Section 6 concludes the paper and outlines future research directions.

## 2. RELATED WORK

Over the past years organizations have been moving infrastructure to the cloud with the aim of reducing infrastructure ownership and maintenance costs and to take advantage of the elasticity offered by the cloud. In this respect, the Amazon cloud provides services such as Auto Scale and Amazon Elastic Load Balancer that allow customers to setup scalable and fault-tolerant applications that take advantage of this elasticity.

Several commercial tools that use Amazon Cloud or other cloud infrastructure support the dynamic provision of services. For example, Scalr[4] and RightScale[5] are cloud management services where customers can create configurable web applications, assign different roles and instance types depending on the workload and setup. Both services are paying, but RightScale is provided as open source, meaning that it can be run separately on in-house infrastructure.

However, tools provided by the cloud are limited so that only small portion of parameters can be used to determine how the application should be scaled. A study by Dejun et al. [5] states that dynamic resource provisioning is made difficult as each virtual instance has its own individual performance characteristics. Standard resource provisioning techniques provided by the cloud providers do not take into account instance heterogeneity and therefore the load is not properly distributed, resulting in wasted resources. Multitier application scaling is much more difficult as there is no configuration option provided by Amazon to deploy and scale databases under the master-slave paradigm.

A novel approach for stress testing web applications using cloud computing is proposed by BenchLab [4]. Benchlab is able to simulate realistic traffic through web browsers. Many benchmark tools do not fully use AJAX capabilities and, therefore, do not reflect real life traffic. BenchLab emulates full browser support while stress testing the underlying system, it will also simulate AJAX and javascript requests. Still further input is required from the user as BenchLab does not provide an out-of-the-box solution for load balancing and server provisioning.

There are also a number of studies and mathematical models for dynamic server provisioning. These models aim to reduce the cost of running a given service or a flow of incoming jobs with the smallest number of resources (e.g. servers), while meeting a given SLA (Service Level Agreement). For example, there are various models for dynamically turning on/off physical servers in view of serving a given flow of incoming jobs within a given SLA, while minimizing electricity consumption [8, 11, 3, 2]. There are also models for dynamic allocation of virtual servers (instances) in order to run a service within a given SLA while minimizing the number of server-hours used [6, 9]. The framework proposed in this paper is complementary to the above ones. The framework collects and monitors arrival rates (for front-end and back-end servers), CPU, memory, I/O and network utilization, thus providing the necessary input data to use one of the above models for dynamic server provisioning.

## 3. THE FRAMEWORK

The framework is intended to simplify the deployment, monitoring, stress-testing and scaling of web applications on the cloud. The primary aim is to reduce the time spent on configuring the application and finding the optimal setup. To this end, the framework provides several scripts for managing activities like turning on/off cloud instances, extracting the performance parameters and varying the deployment setup of web applications, with the help of simple JAVA API. For measuring the performance of the running cloud instances, several tools like Collectd[6] and Cacti[7] were studied and several experiments were conducted to actually come up with the final mechanism adopted in the framework [13].

The framework provides two possibilities for deploying software and infrastructure into the cloud. First is the more manual approach, where all the necessary tools, software and infrastructure are deployed into a single instance, to be more precise into the cloud image, and only configuration files are modified by the framework to produce the intended deployment. The second approach is more dynamic, but will use more time while setting up the experiment and installing necessary software into the instances. Using the second approach is discouraged as the time to get instances running and get fully configured can take more time. The procedure is also error-prone, depending on several issues like the network speed and whether the applications repository is available or not (mirror or site is down).

Both approaches have thoroughly been tested. However we mostly used the first approach for conducting the experiments as the configuration phase is much faster and all the servers have the same properties. It means that applications installed are with same versions and with the same configuration, having consistency through all the servers started for a setup. This will ensure equal load distribution, system should behave in a similar manner and should remove potential bottlenecks that might happen, while using different configuration and software builds.

We used our private cloud, SciCloud [12], to develop necessary infrastructure, configuration scripts and to validate different tools and configurations, before migrating the framework to the public cloud Amazon EC2. SciCloud runs on Eucalyptus, uses XEN hypervisor and is compatible with Amazon EC2 REST and SOAP queries. This gave us a suitable platform for pre-validating the framework.

MediaWiki, a popular web application supporting the Wikipedia service, was used to thoroughly analyze the framework. Several load curves were simulated on Amazon Cloud to test the usage of the framework. Since MediaWiki is extensively used in analyzing the proposed framework, the following subsection briefly discusses the application. Apart from MediaWiki, several experiments [10, 13] were conducted with the help of the framework to quickly test different pro-

[4]http://www.scalr.net
[5]http://www.rightscale.com
[6]http://www.collectd.org
[7]http://www.cacti.net

posed mathematical models and theories for running web services in the cloud.

## 3.1 MediaWiki

MediaWiki is a web application that supports the Wikipedia service. It is written in PHP and uses MySQL for storing the content. The code base of the application is complex and rendering the web page for every request is a CPU-intensive task. In order to decrease the amount of work needed for handling incoming requests, MediaWiki uses caching. There are several caching options to use: (i) file caching, where pages are stored on the hard drive, (ii) database caching, where content is stored in the database and (iii) memcached, where information is stored in the physical memory. Wikipedia uses memcached for caching rendered pages in the memory to improve service speed.

The experiments were set up in a way that replicates the Wikipedia[8] configuration. The configuration does not include reverse-proxy cache (*e.g.* Squid or Varnish) because of the nature of the experiments. Under a relatively small amount of requests, the content is quickly cached and the reverse-proxy cache is doing most of the job meaning next requests are not passed to the back-end servers. It was not our goal to simulate the real world configuration with hundreds of servers and with thousands of requests per second, as it would need an unpractical amount of resources and time, not to mention the time that might be taken to upload the entire Wikipedia dump into the database and other bottlenecks that might happen while setting up such a system.

The experiments are done in a much smaller scale to test the performance of the application and its scalability. We were interested to have a stable system, which means that the cache hit ratio for each page visit should be relatively large. A stable system will present common web page, where most of the pages have been visited by the users and editing the pages (invalidating the cache) is with low frequency. Caching is beneficial for MediaWiki, as it can reduce response times at least 4× and the variation of response time for different pages is not large. It was observed that without caching and using `c1.medium` instance on Amazon EC2, the response time varied between 55 ms to 1 second, where the fastest pages were simple re-directs and the slowest pages had a complex structure and took large amount of CPU cycles to render the content.

### 3.1.1 Layout of the MediaWiki configuration

Figure 1 shows the layout of the framework running MediaWiki in the Amazon Cloud. The setup consists of a load-balancer for routing the requests generated by the benchmarking clients to the Apache HTTP Server. The database and the memcached nodes are also shown in the figure. Table 1 shows the configuration deployed in the Amazon EC2 Cloud instances with Ubuntu 11.04 (Natty) OS. Two separate instances were bundled for 32-bit and 64-bit architectures, to support a wider variety of instance types provided by the Amazon as the lower-cost instances use a 32-bit architecture. Software displayed in the table 1 was installed on both instances, for each role, only necessary software and services are started by the framework. SUN Java is used to run the framework and it runs along with nginx[9] to allow

[8]http://www.wikipedia.org
[9]http://nginx.org

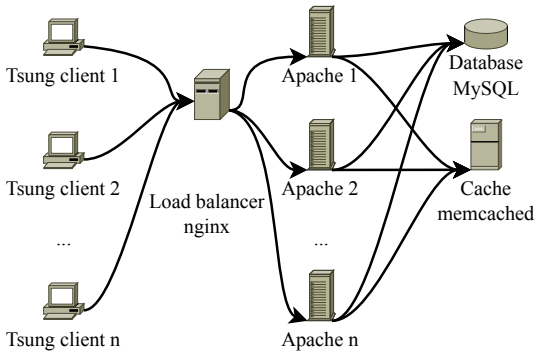


**Figure 1: Servers located in the cloud and how the requests are forwarded.**

**Table 1: Software used for deploying infrastructure and their roles**

| Role | Software |
|---|---|
| HTTP server | Apache 2.2.14<br>PHP 5.3.2<br>XCache 1.3.2 |
| Load-balancer | nginx 0.8.32<br>Fair module<br>SUN Java 1.6.0 |
| Database | MySQL 5.1.41 |
| Cache | memcached 1.4.2 |
| Benchmark tool | Tsung 1.4.2 |

easier configuration of the load-balancer, while provisioning servers depending on the incoming load.

To reduce latency, all the servers were running in the same availability zone, but this is not necessary and different availability zones, regions and even different clouds can be used. User has to be aware that using different regions and availability zones, the traffic between different servers is charged. It is strongly suggested to use compression (`gzcompress`) of pages for PHP to reduce traffic amount. Our experiment set consisted of uncompressed pages that have average size of 55 KB. Using compression this can be brought down to 13 KB (4× smaller) and will reduce cost of conducting tests with traffic from outside of the cloud.

## 3.2 Interactions of the Framework

As already mentioned, the proposed framework is capable of configuring servers on-fly, depending on the roles. Here we describe how interactions between different types of servers and the framework are configured.

**MySQL.** Framework has a MySQL configuration file that the user can change. When the framework starts the configuration phase, it first copies the modified or new configuration file to the MySQL server and starts/restarts the service. If

correct MySQL credentials are entered, the framework can grab MySQL statistics from the database and log the statistics. It is also possible to add or remove commands for each role, depending if reconfiguration or starting additional services is required.

**Memcached.** The memcached service can be configured and started from the command line. The framework will automatically start the service and log cache hit/miss statistics.

**Nginx.** The Nginx configuration file contains the back-end Apache IP addresses.The framework is fully aware of the running servers and regularly updates the back-end pool list to match with the provisioning decisions. The framework also connects with the nginx `HttpStubStatusModule` to fetch the arrival rates and restarts/starts service whenever it is needed (e.g. new Apache servers becoming available).

**Apache.** The MediaWiki application is configured with correct MySQL and memcached IP addresses. The framework can also change PHP and Apache configuration files (e.g. maximum users, memory limit). The framework constantly acquires statistics from Apache `mod_status` and provides information such as how many connections have been done, how many active connections there are and the current bandwidth.

**Load Generator.** The framework copies configuration file of the load generator to the correct servers. Starting the benchmark tool is not yet fully automatic and the user has to provide input and check if the experiment is started correctly. The framework constantly monitors the performance metrics for each server and logs CPU, network, I/O and memory usage, as explained in the next subsection.

### 3.3 Measuring the Performance of the Web Applications Using the Framework

Package `sysstat` [7] is installed on the instances to support measuring and monitoring the performance and health of the servers running in the cloud. This package provides a variety of tools to easily grab information about CPU, memory, network and hard disk usage. One option for the framework is to use SSH (Secure Shell) with private key to connect to the running servers in the cloud, for getting the monitoring information. It is a secure method for connecting with other servers and getting output of the command line, but it is inefficient, meaning there is at least 5-10 second delay to get the data, and under high load the overall time to collect data from one instance could take up to 4 minutes or became completely unresponsive.

In order to work around this problem, a separate simple web service in Java was developed as part of the framework, that gathers all the necessary monitoring information and pushes the information out from a predefined TCP port. This allowed quick connection with the instance and collecting the monitoring information under 10 ms, giving the possibility to fetch information in smaller intervals and for more servers. Our web service collecting the data has only soft-state, meaning there is no I/O overhead incurred and should not affect overall performance of the system. Retrieved statistics from the web service averaged around 330 bytes, indicating very low network overhead and should not really affect network performance, while the main node collects data from a large number of instances. Using 30 second interval for retrieving metrics, it will use on average 88 bits/s of the traffic for one instance.

Tools like collectd and Cacti were also tried for measuring the performance. However, it was not possible to fetch the numerical values in a fast way with these tools as they used RRD files for storing the monitored values. To get to those binary files giving numerical values, different RRD tools had to be used, but processing the information from binary affected the performance of the service. These tools certainly give good overview of the service performance and can be used for drawing graphs. One option would be to write additional library to *collectd* to push out lastly gathered values that can be easily collected by the framework.

## 4. PERFORMANCE TESTING OF AMAZON EC2 INSTANCES

To come up with ideal deployment configuration and scaling up for web applications, we need to have base numbers showing the performance of the cloud instances. Different sets of experiments were conducted to profile a variety of instance types provided by the Amazon EC2 Cloud platform to see the performance gain via vertical scaling (*e.g.* changing slower machine with faster machine). Two typical experiments were conducted to measure (i) service time and (ii) maximum throughput of the MediaWiki application. Both experiments were conducted with fixed 1000 random pages fetched from the database and memcached was filled to ensure low response time of the requests.

To measure service time, one job per second was generated by the benchmark tool. We assumed that one page should not take more than a second to process the request. This helps to measure minimum time it takes to process a request by MediaWiki with minimal load as only one job enters the system. The service time is fetched from the average response time gathered during the experiment.

Second test was to measure maximum throughput of the servers. This was achieved by generating a ramp up experiment with increasing load, each time the epoch inter-arrival rate between requests was decreased. The test was conducted until a certain maximum limit was discovered, where the service is saturated (*i.e.* more jobs in the system than it is capable of handling). At that point, further increase in arrival rate will degrade the service and throughput starts to drop (maximum throughput).

These experiments used large numbers of maximum allowed clients in the server configuration to see how it affects the service while it is under heavy load. Without limitations and reaching the maximum throughput, while decreasing further the inter-arrival rate the service maximum throughput starts to decrease, as there are more jobs in the system and fewer CPU cycles are divided for each request.

With default Apache and PHP configuration provided in the Ubuntu repository we were able to crash `m1.small` instance in Amazon Cloud. The reason behind crashing was that PHP uses large amount of memory for its worker processes and with larger number of requests entering into the system, instance memory quickly ran out and the operating system started to swap memory on to the hard drive, this introduces backlog, making serving new requests very slow and eventually crashing the system.

### 4.1 Comparison of Amazon Instances

Table 2 summarizes the results of the tests conducted in the Amazon EC2 Cloud. To have a good comparison

**Table 2: Results of the experiments with different instance types**

| measurement | m1.small | c1.medium | m1.large |
|---|---|---|---|
| min. response time | 62 ms | 57 ms | 57 ms |
| avg. response time | 132 ms | 71 ms | 71 ms |
| max. response time | 459 ms | 278 ms | 367 ms |
| CPU model | E5430 | E5410 | E5506 |
| CPU clock | 2.66GHz | 2.33GHz | 2.13GHz |
| compute units | 1 | 5 | 4 |
| no. of cores | 1 | 2 | 2 |
| max. CPU steal | 56.04 % | 6.23 % | 25.14 % |
| cost of server | 0.08 $ | 0.165 $ | 0.32 $ |
| max. throughput | 6 rps | 28 rps | 18 rps |
| price per req. ($10^{-6}$) | 3.704 $ | 1.637 $ | 4.938 $ |

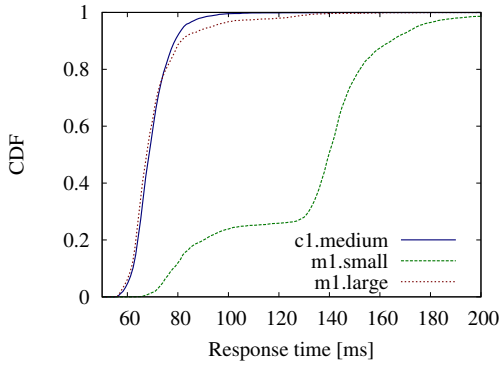


**Figure 2: Cumulative distribution of response times for different instances**

between instances, different characteristics and results are shown in the table. Price per request was calculated using theoretical value, how many requests in one hour the selected instance should be capable of serving, while taking into account maximum throughput and cost of one server for one hour. Results show that using `c1.medium` instances for running MediaWiki application is the cheapest, when looking how much a single request costs, indicating that it is not reasonable to run more `m1.small` servers than using single `c1.medium`.

Figure 2 shows response time cumulative distribution for different instance types in Amazon EC2 Cloud. Instances `c1.medium` and `m1.large` are acting similarly, behaviour for `m1.small` is a bit different, as one third of the requests are twice faster than the other part. It seems that smaller pages with less complexity are using less CPU and can serve content without virtualization (CPU steal) taking away available CPU cycles. While processing complex pages, more CPU usage is affected by the CPU steal, therefore taking much longer to process the request.

# 5. TESTING SCALABILITY

One of the critical QoS characteristics of any web application is how well the application scales to varying loads. To verify this feature, the framework supports different mechanisms. Our intention was to see, how well the framework is capable of handling random inter-arrival rates to the system. Load-balancer nginx was used to direct traffic to back-end servers. Using `HTTPStubModule` for nginx, it was possible to fetch arrival rate to the system. This was used to provision servers in the cloud.

The framework supports easy re-configuration of all the servers running in the cloud with a little effort allowing to conduct different experiments in a short period. Web applications are using resources differently, depending how it was built and handling the requests. Some of the applications can consume more memory; others are more CPU-intensive. Our framework helps to measure different parameters and run several experiments to see the best provisioning and service configuration for different applications.

Experiments done with MediaWiki have shown that it is mostly a CPU-bound application and will definitely need machines with fast processor. It will help us to clarify proper provisioning policy using CPU, memory or arrival rate as an indicator. Knowing the upper limits for each machine, the limits are relatively easy to set. Tests have shown for `c1.medium` that the average response time for one request is 70 ms, meaning with two core machines the theoretical maximum throughput of the system in one second is 28 requests per second. To confirm the value, a ramp up experiment was conducted to see if it holds. Using the retrieved value, we can calculate proper amount of servers needed to cope with incoming arrivals.

In the rest of the section, we describe the various elements of the auto-scaling mechanism incorporated in the proposed framework. This mechanism basically has to decide based on the collected data, if adding or removing servers from the cloud is necessary. The mechanism relies on prediction algorithms (e.g. double exponential smoothing) to get information about the trend of the requests and allow to efficiently provision servers as discussed below.

## 5.1 Predicting Arrival Rates

Websites traffic is largely fluctuating in small time periods, but has an underlying trend on a larger scale. The trend can be either increasing or decreasing, depending if the current hour arrival rate increased compared to last hour or not. One way to predict or smooth arrival rate for next hour is to use weighted average, where at the end values (closer to the ending hour) have more importance. If the average arrival rate increases at the last part of the hour, the arrival rate retrieved by this calculation is slightly larger and should follow the trend.

We made some experiments to see, if weighted average can improve predicting arrival rate for next hour compared to taking arrival rate from the last hour. We found out that in smaller scale (running 20 servers and having at maximum 600 request per second) it did not really affect the end results as the provisioned amount of servers stayed the same, slightly varying between higher increases and decreases, but not significantly changing performance of the service.

Another option is to use double exponential smoothing that will use previous arrival rate values and error between calculated prediction and real arrived traffic to smooth the next hour curve depending on the error of the prediction and in the trend. These kinds of algorithms need at least previous day data to improve prediction. Using our cur-

rent configuration and predicting arrival rates with double exponential smoothing did not really improved the service performance and cost of running the servers compared to two previously mentioned methods. It can reflect that fluctuation within an hour is too large to correctly calculate the "optimal" amount of servers that will be capable of serving all the incoming jobs and in the same time hold running cost low as possible. All the algorithms and methods still need some spare servers in order to cope with fluctuating traffic.

## 5.2 Adding/Terminating Instances

In order to provision servers in the cloud, we employ an approach similar to the one described in [10]. The algorithm is as follows:

1. Five minutes before the full hour, fetch arrival rate from nginx and calculate amount of servers needed to cope with the traffic.
2. Using previous value, check if amount of servers calculated is lower than currently running, if yes, remove proper amount of servers from the cloud (terminate instances) and reconfigure load balancer.
3. At the full hour, check if the amount of servers calculated from step 1 is more than running, if yes, request new servers from the cloud and store retrieved instance ID codes for tracking purpose.
4. Track requested servers until all the pending servers have changed state to running and return to step 1. If server is changing state from pending to running, it gets private and public IP from the pool. Use retrieved IP to connect with the instance, change database and cache IP addresses and start necessary services. If this is done, add new server into load balancer server list and restart the service.

The above steps are built into the framework and are automatically done. The framework uses arrival rate and user-defined service time to calculate proper amount of servers. If the servers are too loaded, larger service time can be used to allow running more servers. Decision for removing servers before full hour (look figure 3) was done to simplify removing the servers as we do not need to look, which server is the closest to the full hour. This ensures us, that while removing servers, we do not need to pay for an extra hour, because the termination process was called out later or took more time.

An important step for adding/removing servers is to ensure that the process is transparent and no incoming connection is lost. While the instance is going to be terminated, the job might be forwarded to the server, and it never returns. It is also possible that when reloading load balancer, already existing jobs in the balancer might be dropped. Last possibility is, when new servers are added too quickly to load balancing pool and are not yet properly configured (Apache is not started or MediaWiki configuration database and memcached IP addresses are not changed), the request will fail.

Using nginx as load balancer ensures that all the jobs entering the system are processed even if nginx is restarted or terminated. The worker processes will first get a termination signal and will wait until all the requests are sent back, while nginx master will create new worker processes with new configuration and all the new requests are forwarded

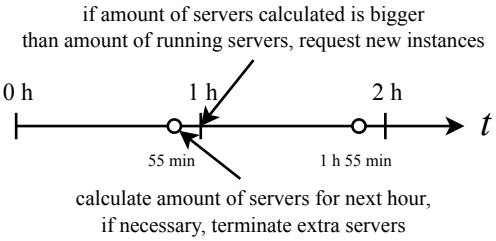


**Figure 3: Server allocation by the framework [10]**

there. This will ensure that while reloading the load balancer, none of the requests are lost. Our framework also takes care of adding and removing servers, making sure that instance going to be removed is first removed from the nginx configuration server pool and service itself is restarted and instances are added only if the configuration step was fully processed, meaning that the back-end Apache server is ready to process requests.

## 5.3 Dynamic resource allocation policy

Our policy for server allocation uses service time and arrival rate to determine sufficient amount of servers, but at the same time maintains reasonable response time meeting the SLA. We use queuing theory to calculate average service time of the current configuration. The system has $M/M/c/c$ queue and all the servers $c$ are considered homogeneous.

$$s = \frac{r}{1 + \frac{\frac{\lambda}{c} \times r}{n}} \tag{1}$$

Using Equation 1 [5], we can calculate theoretical response time of the current configuration, where $r$ is service time in ms (70 ms), $n$ is amount of cores (2), $\lambda$ is arrival rate and $c$ is amount of servers. Our interest is to hold the average response time below 250 ms (3.5× slower response time than `c1.medium` experiment showed). To find a solution, variable $c$ in Equation 1 is increased until $s \leq 250ms$.

Using too small epoch for allocating servers may oscillate the system as the arrival rate is not deterministic and is rather noisy. Larger time spans might have too stable system meaning that the servers are allocated too late and we can lose traffic with rising trend and have more idle servers running with decreasing trend. One hour is suitable point for allocating the servers, as Amazon EC2 charges customers at that rate and closer look for different traces have shown that within one hour the fluctuation in traffic is relatively small.

Additional policies could be fit into the framework, such as for example policies aimed at maximizing revenue on behalf of a software-as-a-service provider [10].

## 5.4 Configuration and Results of the Experiment

Clarknet traces[10] were used for generating loads for the experiments. They contain information about arrival rates for 2 weeks. The 10th day was used for generating the traffic,

[10]http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html

where at the beginning and end, the arrivals are low indicating at night times, there are few visitors and in the middle of the day, peak traffic rate is reached. Figure 4 shows the rate at which the requests were injected into the system. Inter-arrivals were changed for each minute resulting in largely fluctuating traffic.

Tsung is used as load generator to make `HTTP GET` requests against nginx load-balancer. Complex Tsung XML configuration file with 1 minute resolution for `arrivalphase` is set up to follow Clarknet trace curve. Randomized set of `URI` addresses, excluding the redirects, are fetched from the MediaWiki database and are used to generate the requests. Tsung was deployed in a distributed cluster, composed of three nodes (one master and two secondary), running on the cloud. This ensures that load generator's overall load on the server is small and does not affect the measurement of average response time of requests. Master node takes care of executing the test plan (Tsung XML configuration file), forwards request logic to secondary nodes and collects various statistics from secondary nodes, including average response times into one place. In this way information is combined and can be analyzed into a single report using Tsung-plotter utility or custom scripts.

A one day experiment was conducted to see benefits and load curve changes in the system, while dynamically allocating servers in the cloud. For the first experiment we used Amazon Auto Scale [6] and for second experiment we used our simple optimal heuristic method. Amazon Auto Scale was configured to scale with breach time 15 minutes. This means that if CPU usage threshold is exceeded at least for 15 minutes, a new server allocation or server termination is done. We set Amazon Auto Scale threshold to 60% and 70% of the average CPU for down-scaling and up-scaling respectively. Second experiment was using average service time as 70 ms to allocate correct number of servers.

Table 3 summarizes the metrics gathered for 24 hour experiments. These experiments show that it is possible to reduce cost of running the service 28% to 35%, but with the cost reduction, more jobs are rejected as the servers are working with larger CPU utilization. Average CPU utilization looks normal even with Amazon Auto Scale, but the traffic strongly fluctuates, indicating that in certain time periods the servers are overloaded and dropping excessive connections. Over 22 million requests were generated, thus losing 343,763 jobs for Auto Scale represents only 1.5% of the jobs lost, which can be acceptable depending on the SLA. For the optimal policy and running all 20 servers for 24 hours, job loss is relatively small (under 0.15%).

Average arrival rate in this experiment was 255 requests per sec. We were also interested in observing the yield of each policy. Because the cost of one request is relatively small, we compare the values for 1 million requests. Using Auto Scale with 70% up-scaling and 60% down-scaling, serving million requests pays 2.305$ for the service provider. The cost for optimal is 2.527$ and for running all the server, it is much higher, around 3.491$. These results are experimental and might not give the same results, when using with real people, and it is hard to predict user behaviour in such case. Users might refresh page immediately, generating much larger load for that time period or may leave to other pages, resulting in much lower load.

Figure 5 shows CPU utilization for arrival rates from 1 to 550 requests per second, using 20 back-end Apache servers.

**Table 3: Dynamical server allocation experiments results**

| measurement | Auto Scale | Optimal | Always on |
|---|---|---|---|
| avg. response time | 144.6 ms | 116.0 ms | 101.8 ms |
| avg. CPU usage | 44.62 % | 37.22 % | 23.84 % |
| instance hours * | 312 | 347 | 480 |
| cost of servers * | 49.92 $ | 55.52 $ | 76.80 $ |
| requests lost | 343763 | 32288 | 2148 |
| successful requests | 98.44 % | 99.85 % | 99.99 % |

* only Apache servers are counted; instance charge is 0.16 $ per hour (us-east-1 region)

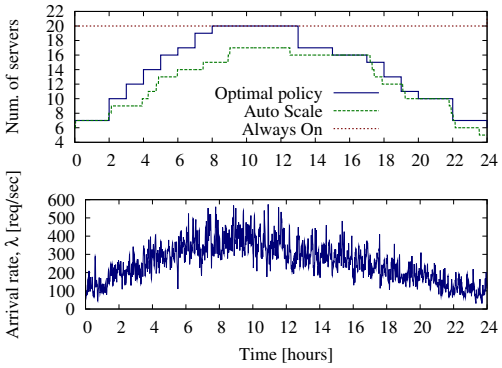


**Figure 4: 24-hour experiment of server allocation for different policies**

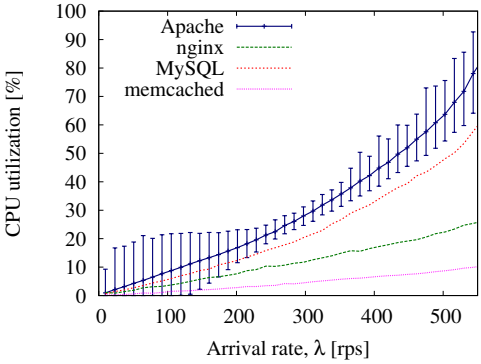


**Figure 5: CPU utilization for different servers compared with arrival rate**

Variation between Apache server CPU utilization is caused by using least round-robin algorithm, where requests are passed to the most idle server. Because at small load, couple of first servers in the server pool are capable of serving all the requests faster than they are arriving into system, making them idle again and next request is sent to the same servers. MySQL server is moderately loaded and additional or faster servers are needed to cope with larger arrival rates.

## 6. CONCLUSION AND FUTURE WORK

Testing the QoS of web applications is a cumbersome task. Cloud computing opens significant opportunities in this respect by allowing engineers to cost-effectively achieve various configurations and loads. While cloud computing provides a suitable platform for conducting performance experiments, significant manual effort is required to set-up and monitor server farms on the cloud in order to answer questions such as how jobs are entering into the system and how they are divided between different resources, etc. This paper presented a framework that helps in monitoring and testing scalability of web applications on the cloud. The framework eases the task of finding possible bottlenecks in the system, which helps to optimize the configuration of the web applications. The framework also helps in managing the cloud resources like turning on/off virtual servers.

MediaWiki, a popular web application supporting the Wikipedia service, was used to thoroughly analyze the framework. Several load curves were simulated on Amazon Cloud to test the usage of the framework. Extensive benchmarking of MediaWiki application has given a picture of the possible bottlenecks and how the system works under heavy load. We tried three different instance types to accommodate MediaWiki service in the cloud and found that `c1.medium` instance provides a good tradeoff as it provides enough CPU power to serve a reasonable amount of requests per server.

We also tested dynamic server allocation policies on the MediaWiki application. The established approach for dynamic resource allocation on the Amazon Cloud is Amazon Auto-Scale, which provides a robust way of scaling web applications using CPU thresholds for adding and removing instances. Customers using Auto Scale do not need to fully understand the performance of virtual instance or web applications, resulting in easier configuration and at the same time meet QoS requirements, while using reasonable parameters. Our experiments showed that using CPU upper threshold as 70% and breach time set to 15 minutes starts to lose traffic, because additional servers are added too late. Accordingly the paper also discussed the optimal policy based auto scaling, which is shown to be better than the Amazon Auto Scale mechanism in terms of handled successful requests. The framework incorporated its own methods for predicting the arrival rates and algorithms for adding and removing instances. Future work will involve testing Amazon Auto Scale with different parameters and conducting experiments with other instance types to see, if using more CPU power per instance will be more cost saving option than currently used instance type `c1.medium`.

## 7. ACKNOWLEDGMENTS

This research is supported by European Commission via the REMICS project (FP7-257793), the Estonian Science Foundation grant ETF9287, and ESF via the Mobilitas program. The authors are grateful to Michele Mazzucco for his valuable input and comments.

## 8. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[2] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 119–128. IEEE, 2007.

[3] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, page 12. USENIX Association, 2009.

[4] E. Cecchet, V. Udayabhanu, T. Wood, and P. Shenoy. Benchlab: an open testbed for realistic benchmarking of web applications. In *2nd USENIX Conference on Web Application Development*, page 37, 2011.

[5] J. Dejun, G. Pierre, and C. Chi. Resource provisioning of web applications in heterogeneous clouds. In *Proceedings of the 2nd USENIX conference on Web Application development (WebAppsŠ 11)*, 2011.

[6] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. Technical Report CMU-CS-12-109, School of Computer Science, Carnegie Mellon University, 2012.

[7] S. Godard. Sysstat: Performance monitoring tools for linux, 1999.

[8] H. Lim, S. Babu, J. Chase, and S. Parekh. Automated control in cloud computing: challenges and opportunities. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 13–18. ACM, 2009.

[9] M. Mazzucco and M. Dumas. Achieving performance and availability guarantees with spot instances. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 296–303. IEEE, 2011.

[10] M. Mazzucco, M. Vasar, and M. Dumas. Squeezing out the cloud via profit-maximizing resource allocation policies. In *20th IEEE Int. Symp. of Modeling, Analysis and Simulation of Computer and Telecommunciation (MASCOTS)*, 2012.

[11] I. Mitrani. Management of server farms for performance and profit. *The Computer Journal*, 53(7):1038–1044, 2010.

[12] S. Srirama, O. Batrashev, and E. Vainikko. Scicloud: Scientific computing on the cloud. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 579–580. IEEE Computer Society, 2010.

[13] M. Vasar. A framework for verifying scalability and performance of cloud based web applications. Master's thesis, University of Tartu, 2012.