Scaling and Availability of RunSignUp.com to Support Large Registration Bursts

Stephen Sigwart
RunSignUp.com
info@runsignup.com

Tuesday 29th January, 2013

Abstract

A common problem many websites face is having a large burst of traffic. This paper discusses how RunSignUp solved this problem in detail so that other development teams can learn from our experiences to leverage cloud deployment to solve their own scaling problems.

For many of the larger road races, there is a limit on the total number of runners who can register. This leads to a massive influx of online registrations as soon as registration for the race has opened. For races that accept 30,000 to 50,000 participants, this requires a highly optimized and highly scalable infrastructure with hundreds of servers and a code base that takes advantage of multiple tiers of specialized load balancers, caching, queues, and other technologies to handle the traffic. Furthermore, the registration website needs to interact with external services, such as payment processors, to complete each registration.

It would be economically infeasible to build a static infrastructure capable of handling this traffic since servers would be idle on non-peak hours. Infrastructure-as-a-Service (IaaS) providers, such as Amazon Web Services (AWS) [1], provide a cost-effective solution by allowing customers to rent servers and use other cloud services on-demand. This allows an online registration service to rent extra servers to handle the burst in registrations, while running on a more modest infrastructure when the load is low.

While online registration systems should be able to scale to handle the current traffic, they also need to be highly available. To that end, IaaS providers can also provide physical separation of servers and services, allowing for an infrastructure that can withstand failures of multiple servers. Our registration system, RunSignUp.com, uses AWS to provide both scalability and availability to online race registration.

This paper describes the process of converting a single-server website into a scalable and highly-available multi-server architecture. It is written for a developer tasked with the job of converting an existing system to handle large traffic bursts and heavy write loads on common objects. It first introduces some applications and services that were used in our infrastructure and addresses the steps to move to a cloud architecture, including details on making it highly-available. Next, it discusses the use of caching and a few other techniques to gain performance and make the website more scalable. After this, the paper describes how we handle server management and a load testing framework to benchmark our system. Finally, we present the final performance that we achieved after making these updates.

1 Migrating to a Multi-Server Infrastructure using EC2

RunSignUp was founded in 2009 with the goal of providing a simple and easy experience for both race directors and runners. The first few years were spent building the business and incrementally adding features needed by large races. However, the website was not initially designed to be highly scalable, so the next steps were to build a robust infrastructure that not only could handle large numbers of transactions per second, but also was resilient to server failures.

The first task in our project was migrating the system from our existing single server architecture to a cloud architecture taking advantage of the services provided by AWS [1], primarily the Elastic Compute Cloud (EC2) [2]. We made use of Amazon Virtual Private Cloud (VPC) [3] to provide an isolated environment with our own private IP address allocation and subnets. Our infrastructure can be seen in Figure 1. The components of the infrastructure will be discussed in a mostly top-down fashion. First, though, a quick background on caching is required.

1.1 Caching Basics

One of the fundamental goals of our scalability project was to reduce the load on our database server. This is because the database server is restrained to a single server and therefore becomes a bottleneck. By caching data, fewer accesses to the database are needed.

This caching is achieved through memcached [4] and APC [5] caches. Both of these caches are memory-based, key-value systems that allow you to store and retrieve values by a key. This makes reads and writes very fast compared to the database, but are volatile, meaning that the data they store should be able to be rebuilt from the database if needed.

Memcached and APC cache differ primarily in size and accessibility. The memcached application runs on a separate server, where all web servers share the cache and can manipulate the data. APC cache is a per web server cache implemented as a shared memory segment that is much quicker since it is on the same server, but lacks the ability to share data with other web servers. This means that APC is faster, but much smaller than memcached. It also makes APC non-applicable for keys that need to be shared across web servers. Ideally, we would like to access data first in APC, then in memcached, and lastly in the database as shown in Figure 2.

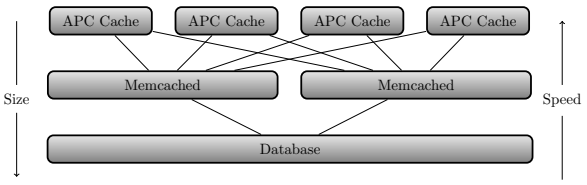

Figure 2: Caching Hierarchy

1.2 Domain Name Lookup - Route 53

When a user first accesses RunSignUp via our runsignup.com domain name, the user's browser contacts Amazon Route 53 [6] to determine where to send the request. Route 53 is the DNS service that handles the runsignup.com domain name. Route 53 is set up with weighted type-A records for runsignup.com, meaning that one of several IP addresses is returned in a round-robin fashion. Currently, there are six IP addresses associated with the runsignup.com type-A record.

1.3 Load Balancing - Nginx & NAT

Once the user has an IP address, their browser sends the request to the returned IP address, which will be one of multiple load balancers. Each load balancer runs Nginx [7], a web server that acts as a reverse proxy. Nginx simply proxies the request to the backend web servers which actually execute the request. Since the load balancers are the only servers in this infrastructure with a public IP address, the servers also acts as a NAT server [8] for all other servers. This helps to make the internal servers more secure while still allowing the internet access they require.

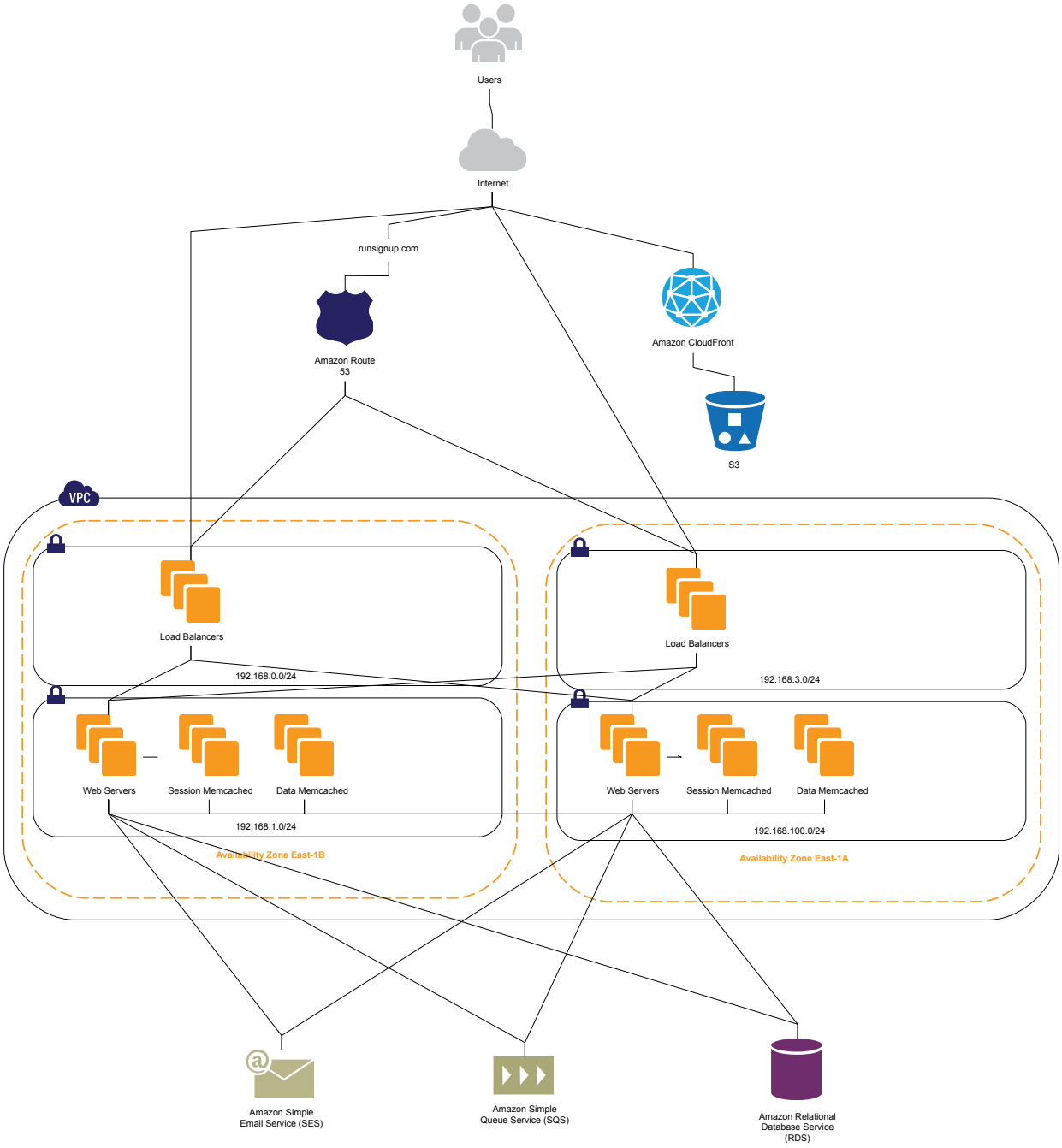

Figure 1: Final Infrastructure

1.4 Web Servers - Apache & PHP

The Nginx load balancers essentially forward requests to our Apache [9] backend web servers. These servers run PHP to process the requests and handle all of the logic associated with requests. While processing, the web server needs to communicate with many other servers and external services to satisfy the request.

1.5 Session Memcached

Since there are multiple web servers, the typical file-based session storage does not work. Instead, the session data needs to be stored in a shared location. This is achieved by using a custom session handler that stores the session data in memcached servers.

1.6 Data Memcached

To support data caching, we run multiple memcached servers reserved for application data. Frequently used data from the database is stored on these servers. This allows for quicker access to data and reduces the load on the database server. Data and sessions are stored on separate servers to eliminate risks associated with this setup. Firstly, it prevents data from accidentally overwriting sessions. Secondly, it allows the data caches to be flushed if needed without losing sessions.

1.7 Sending E-mail in Amazon Web Services - SES & SQS

The move to AWS required a change to the way e-mails were delivered. Instead of using SMTP, we switched to Amazon Simple Email Service (SES) [10]. However, SES places limits on both the total number of e-mails in a 24 hour period and the maximum number of e-mails per second. As such, all e-mails were added to a Simple Queue Service (SQS) queue [11]. SQS is a service to queue messages that will be processed by separate processes. The primary web server runs a PHP process that monitors this queue and sends e-mails without overstepping the e-mail limits.

1.8 MySQL Database - RDS

The MySQL database server is run using Amazon's Relational Database Service (RDS) [12]. This provides the benefits of the MySQL database with less overhead of configuration and maintenance, which is handled by Amazon. Furthermore, it simplifies the creation and maintenance of Read-Replicas, a MySQL feature that is very useful for scaling read operations.

1.9 Serving Static Resources - S3 & CloudFront

Most of the static resources on RunSignUp are stored in Amazon Simple Storage Service (S3) [13] and delivered to the user via Amazon CloudFront [14]. Both of these can be seen at the top right of Figure 1. S3 is a service for storing files in the Amazon cloud, providing redundant storage and essentially unlimited space. CloudFront is a Content Delivery Network (CDN) service that delivers additional resources to users via a globally distributed network of servers.

The static resources served via CloudFront include CSS stylesheets, images, and JavaScript used on the website along with resources uploaded by race directors, such as race logos, sponsor logos, and registration PDFs. By using CloudFront to serve these files, we drastically reduce the number of page requests to our servers. For some pages, this reduced the number of HTTP requests to our web servers from 25 requests to a single request, a 96% improvement.

To accomplish this, all static resources were placed in a specific directory. The code was updated to call a function whenever a static resource was required. For local development, this simply returned the location of the file, so local development is not hampered. However, for our production system, it mapped filenames to CloudFront URLs.

1.9.1 Generating the Static Resource Map

The map is generated during the deployment process, which checks for any updated resources and pushes them to S3. The filename that we use for S3 includes a few extra characters based on the modified time of

the file. This is for cache busting purposes, allowing the files to be locally cached by browsers while ensuring that stale versions of files will not be used when a file is updated. It then generates and stores the map for later use. For files that could not call the function to get the map, such as CSS stylesheets and JavaScript files, the deployment script scans the files and updates them to point directly to the CloudFront URLs.

1.10 Deployment

In order to quickly deploy new code to all of the web servers, there is a deployment script on our development server. In the deployment script, we utilized a GitHub [15] repository to store the code that should be loaded onto each web server. Once the initial processing is completed, such as updating static resources, the script starts a process on each of the web servers to do the deployment. Each web server updates its local copy of the same repository and runs the deployment script found in the repository. This setup allows for a quick deployment without putting a heavy load on our development server. Furthermore, the script ensures that existing resources (e.g. images, CSS stylesheets, and JavaScript) that will be updated are not deleted until after the new resources are in S3, ensuring that the deployment is transparent to users.

1.11 Typical Server Setup

Table 1 shows the typical infrastructure running on a daily basis. The server sizes listed are defined by EC2, with an m1.small server roughly 1/4th the size and cost of an m1.large server. During periods of high system load, the size and number of load balancers and web servers can be increased. While the number of memcached servers is not as easy to change frequently, the size of the servers can be easily and quickly changed.

1.12 Server Management

In order to keep our costs low, it is not practical to have the system set up to handle the high load all the time. All of the servers would be severely underutilized. Therefore, we take advantage of the cloud to use smaller and fewer instances the majority of the time. When we know that a large race will be opening registration, we update the infrastructure to handle the load. To do this, we built several scripts and user interfaces to do the majority of the work.

Server Type	Server Size	Servers
Load Balancers (Nginx)	m1.large	2
Web Servers (Apache)	m1.large	2
Session Memcached	m1.small	8
Data Memcached	m1.small	8

Table 1: Infrastructure on a Typical Day

1.12.1 Load Balancers

Typically we run two m1.large load balancers that double as NAT servers for other servers on our private network. The runsignup.com domain name currently resolves to one of 6 IP addresses. Hence, each load balancer is assigned 3 of these IP addresses.

There is a UI that lists the running load balancers and allows us to start more load balancers or stop running load balancers. When a load balancer is added, the scripts will wait until the server is ready and then assigns it an IP address and set up route tables to route NAT packets though it. When a load balancer is stopped, the script first assigns its IP address and route table to another load balancer. Furthermore, the UI allows us to change the instance type of a running server to use a smaller or larger server. Since the server must be shutdown to do this, it will follow the steps to stop it, change the instance type, then follow the steps to start it again.

1.12.2 Web Servers

During normal workload, there are two m1.large web servers running in separate availability zones. The load balancers evenly distribute incoming web traffic to these servers. They have no public IP address, which makes them less susceptible to attack. However, it also means that they need to use the load balancers as a NAT server to reach external services,

such as other Amazon services and our payment gateway.

Prior to a large race opening, the number of web servers needs to be increased to handle the increased load. We have a UI that lists our running and stopped web servers, allowing us to easily select specific instances or groups of instances to start or stop. If we need more instances than are currently created in EC2, there is a field specifying how many new instances need to be created and the script will handle the cloning and set up of new web servers. When web servers are added or removed, we need to notify the load balancers so they can adjust their list of web servers. This is accomplished using Amazon's Simple Notification Service (SNS) [16], which reliably sends notifications of added and removed IP addresses to each of the load balancers. These scripts allow us to quickly and easily go from two web servers to hundreds of web servers in a matter of minutes.

Furthermore, there are times when we may want to adjust the instance type of the web servers to be a more powerful server. There is a command line script that helps us to quickly convert the instance type of all stopped web server instances. This came in handy when doing our load testing by allowing us to experiment with various instance types. This script is available at `https://github.com/RunSignUp-Team/OpenSource/blob/master/serverManagement/changeWebServerInstanceTypes.php`.

1.12.3 Memcached Servers

Our infrastructure uses two sets of memcached servers, one to store user sessions and the other to store arbitrary data. As seen in Table 1, the infrastructure uses 8 m1.small servers of each type instead of 2 m1.large servers of each type. There are two big advantages to this setup. First, the network traffic to each server should be around a factor of four less than with the initial setup. Secondly, it allows for more scaling possibilities since each server can be upgraded to larger instance types.

The problem was to figure out how to automatically switch these instance types without losing data and without disrupting users. For the data servers, data loss is acceptable since the information can be rebuilt from the database. However, for the session servers, data loss is unacceptable. For this, we wrote a command line script that does the conversion. This script can be found at `https://github.com/RunSignUp-Team/OpenSource/blob/master/serverManagement/changeMemcachedInstanceTypes.php`.

The conversion process first locates all the memcached instances of a given type that need to be updated, which currently is 8 instances. For every running memcached server, there is another stopped instance that can be used as its replacement. The process for changing the instance type of a memcached server is as follows:

1. For each stopped memcached instances of this type.
   (a) If needed, change the instance type.
   (b) Start the server.
2. For each running memcached server of this type.
   (a) Reassign the private IP addresses to one of the newly started servers.
   (b) Shut down the server.

For the session storage servers, we add in a couple of extra steps. First, the `memcached-tool` utility is used to dump the contents of memcached just as the private IP address is switched. This dump file is copied to the new server and used to rebuild memcached with the session data from moments beforehand. Even though there is a very small chance that a few pieces of this data could be outdated, our custom session handler will recognize this and use the up-to-date data from another server since session data is duplicated. Therefore, on the next request for a given session, the server will be updated to the current session information. This process helps to maintain session information so that user disruption is minimized.

2 Availability

To provide high-availability, our infrastructure is set up to run in multiple availability zones (AZs) in EC2. Each AZ is a data center in EC2 that is expected to be isolated from failures in another AZ. As Figure 1 illustrates, our system always runs servers in at least two availability zones.

2.1 Load Balancers

At all times, there is a load balancer running in at least two availability zones. This means that if one load balancer fails, the other load balancer will take over for it and process the requests. This is done using custom scripts written for the Nagios [17] infrastructure monitoring software. When the software detects that a load balancer is down, it moves the IP address of the failed server to the running server. Furthermore, it updates the packet routing for any subnets that use the load balancer as a NAT server. All future requests will be routed to the running load balancer until the failed server is restored. This script can be found at `https://github.com/RunSignUp-Team/OpenSource/tree/master/serverManagement/nagios`.

2.2 Web Servers

The Apache web servers are also evenly distributed across AZs to ensure that server failures are not visible to users. The load balancers constantly monitor each web server and will stop proxying traffic to a server if it detects an error. This means that a failed server will simple result in traffic being routed to the web servers that are still running. Once the web server is running again, it will be automatically detected by the load balancers and added back to the server pool.

2.3 Memcached Servers

The memcached servers are distributed across AZs to ensure that failure of one server does not eliminate all caching in the system. PHP will detect failed servers and remove them from the pool in the event of failure, allowing caching to continue working. The Nagios software is set up to send E-mail notifications in the event of memcached failures. When we receive this notification, we can fix the failed server or simply start up a new server and associate the IP address with this new instance.

2.4 Session Storage

We wrote a custom session handler in PHP that uses memcached to store session information by a key based on the user's session ID. The session handler stores the information, with a timestamp on two memcached servers, both in different availability zones. If one server fails, the user's session is not lost since it is still available on at least one server. If the failed server becomes available again, but has out-of-date session information, the timestamp is used to decide on the proper session to use. This ensures that session data is not lost in the event of server failure or maintenance.

2.5 Database

The database uses Amazon's Relational Database Service with the Multi-AZ option. This ensures that we have access to our data in the event of a failure in one availability zone. The service is set up to quickly fail over to a different availability zone in case the primary server fails. This feature is available as a simple option in RDS, with all the complexity handled by Amazon, meaning more time can be spent on the application instead of database configuration.

2.6 Amazon's High-Availability Services

Parts of our availability are based on the high-availability provided by AWS. The combination of S3 and CloudFront allows the static resources on RunSignUp to be delivered to users from various servers based on the user's geographic location. Other critical services, such as SQS [11], are designed to be highly-available, allowing us to focus more on RunSignUp than managing a custom queueing service.

3 Performance Via Caching

The primary goal in making RunSignUp scalable was to reduce the load on the database server. While we are limited to a single database server, we can have a large number of caching servers. We used two levels of caching, memcached and APC cache. In addition to application specific caching, PHP using APC opcode caching, which caches PHP files to make requests faster.

3.1 Caching Framework

To implement the data access hierarchy of Figure 2, we developed classes to provide uniform access to the caches. The `getValue` function is the cornerstone of these classes. This function first attempts to get a value from APC cache. In the event of a cache miss, a request to memcached is made. If the key is found, the function will decide whether the key should be stored in APC cache, using a technique we refer to as frequency-based cache promotion. To do this, it uses memcached to increment a counter (once per page load) with a few minute expiration time. If this counter reaches a threshold parameter, the value will be stored in APC for the next request. This allows the code to use the faster APC cache for races that are receiving more traffic.

Furthermore, the `getValue` function has options that will limit database calls during bursts, preventing an issue know as "data stampede". Suppose a key is not present in any of the caches, but several processes attempt to get the key simultaneously. They will all fail and go to the database get the value and then store it in memcached. However, only one process really needed to query the database. The other processes could have waited for the first process to store the value in memcached. The `getValue` function can do this by attempting to set a lock in memcached if the key was not found. The process that gets the lock will go to the database while the other processes will do a quick sleep before trying the cache again.

This caching is a big win for performance, but it would be unacceptable for the race data to be out of date. To solve this, we added functions to store values with a timestamp. Other functions to get the value from the cache check the value against a timestamp to see if the value is valid. This allows us to quickly update a timestamp for a race which will essentially invalidate cache items for a race. For example, if a race name is edited, we simply update the timestamp in memcached. If we find a cache entry with a timestamp less than the updated time, it is discarded and the new data is retrieved from the database.

Furthermore, the memcached class implements a `getOnce` function, which will retrieve a value from the cache only once, with all future requests returning the same value. The class implements a cap on the total amount of memory used for this purpose. This function is particularly useful for timestamps, which will be needed many times in a single page request. For instance, there can be around 50 cache items for a race, each of which might need to check the timestamp. Instead of fetching the timestamp from memcached 50 times, it is fetched once and reused on later calls.

3.2 Caching Race Information

The core race information in RunSignUp is created and modified through a five step wizard. This information includes basic race information such as name and location; event information such as start times, registration periods, pricing, and participant caps; and any text content the race director wishes to include. All of this information is needed on the race page, the entry point of most registrations for the race. As shown in Figure 3, this page can contain a large amount of data that must be retrieved from the database. This particular page required over 50 database queries to build, which would not scale well to heavy traffic loads.

In order to reduce the database calls, we built a profiling system that allowed us to see exactly which database queries were needed for each page. Furthermore, it detailed the amount of time spent in various portions of the code. Using this data, we systematically cache race information and optimized SQL queries. The goal was to completely eliminate the need to connect to the database for most users, thus reducing the load on our database server.

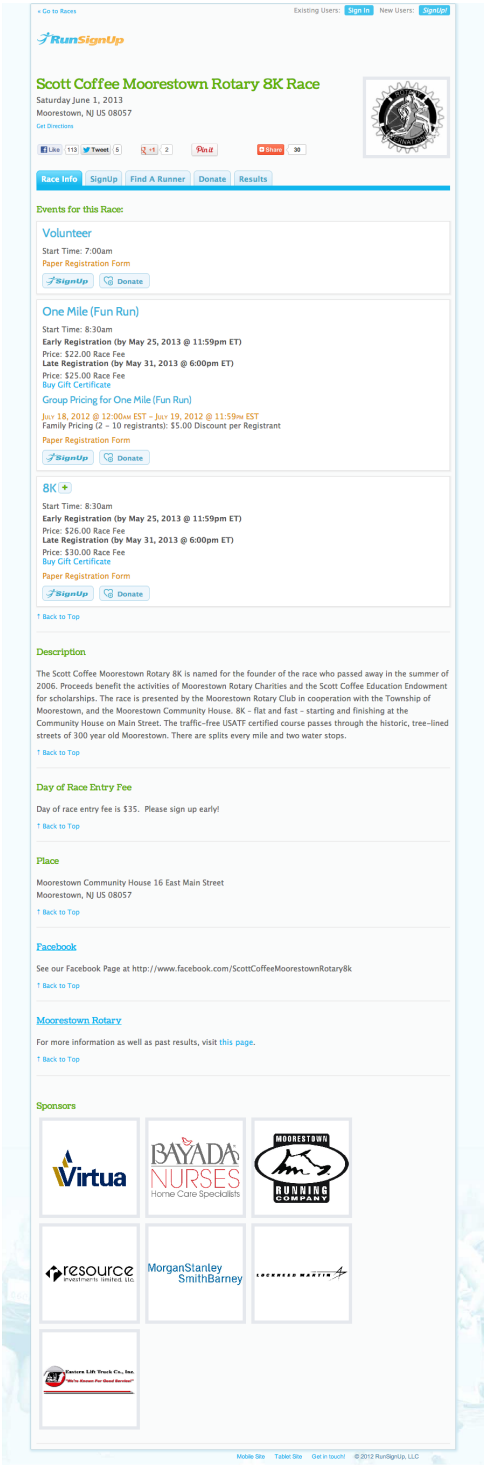

Figure 3: Sample Race Page

Much of this information was static race information that is modified by the race director and rarely changes. This information was simple to cache by just storing the data and using the race timestamp as a cache invalidator. However, other pieces of data were dynamic, such as the number of spots remaining for races with participant caps. Before caching, this information was retrieved by using a `SELECT COUNT(*)` query and subtracting the result from the participant cap. To eliminate this database call, we stored a counter in memcached for each race event. Anywhere that added or removed a registrant from an event updated this value using the `increment` and `decrement` memcached operations.

While an accurate number of participants is critical in determining if a user can register for a race, it is not so critical in other areas. For example, this information is simply displayed on the race page and it is not critical that it be 100% accurate at all times. Therefore, we used APC to cache the count (for viewing only) for 5 seconds. This helped to reduce the load on the memcached servers, freeing them up for access to more critical data.

Once we removed all database calls from the race page, our tests revealed that we were still making unnecessary connections to the database even though no queries were executed. This was because our database wrapper class was derived from the `mysqli` PHP class. The constructor for `mysqli` established a connection to the database immediately regardless of whether the connection was required. To rectify this, we updated our wrapper such that it is no longer derived from `mysqli`, but instead used PHP's `__call` method overloading function [18] as shown in Listing 1 in Appendix B. If a method is undefined, PHP will call the `__call` of the `Database` class, which checks if the method is part of the `mysqli` class. If so, a database connection is established when needed. Otherwise, the code checks for the function in other modules that we have loaded into the wrapper, making it easier to separate our code.

3.3 Caching Registration Related Information

While the race page contains mostly static information which is relatively easy to cache, the registration process is more write oriented and deals with constantly changing data, making caching more difficult. Although much of this information was cached from the race page, there was additional information needed for the registration process, such as teams, fundraisers, donations, and store items. Some of this information, such as donation details and store items, was fairly static and easily cached. However, the lists of teams and fundraisers are frequently updated as users register for races as a team or create a new fundraiser. These lists were cached in a similar way to all the other data, except that a very short (30 seconds to 1 minute) expiration was set on the cache items. By doing this and using the locking scheme previously mentioned, we were able to handle large traffic bursts without putting a large strain on the database. Furthermore, limits on store items was achieved through cached counters, much like the registration counts previously discussed.

3.4 Checking E-mail Address on New Accounts

While this eliminated or reduced the number of database calls for most of the information needed as the user steps through the registration process, there is other user-specific data that is needed from the database. First, an unregistered user needs to create a new account when registering for a race. This means that we need to ensure that the e-mail address provided is not already used on an existing account. That means we need to make a database call, and worse, we needed to use MySQL's `FOR UPDATE` [19] clause to ensure that no other user has registered the same e-mail address between the time we checked the database and inserted the new record. This type of SQL statement results in row locks that could hurt performance.

Our solution was to use a read-replica of the database to check each individual address and use the cache to store whether or not the e-mail address was used. Future requests could use the cache, ensuring that the read-replica is only accessed once for each e-mail address. The cache is updated to reflect when the e-mail address is actually used to create an account.

Using read-replicas has the disadvantage of having lag between when the master database is updated and when the replica is updated. In this situation, the lag does not cause issues because we update memcached as we update the database. If an e-mail address is not in the database, it will not be in the master database. When a user registers with that e-mail address, it is recorded in the master database and memcached. Suppose another user attempts to create an account with the same e-mail address prior to the update reaching the read-replica. In this case, memcached will indicate that the e-mail address is used and the lag has not caused any issues.

3.5 Checking For Existing Registrations

The second user related detail that we needed to address was the possibility that an existing user already registered for the race. Therefore, we needed to do another `FOR UPDATE` database call on each registration to check this, again resulting in unwanted database locks. We followed a similar read-replica and cache approach for this, storing and updating flags indicating whether or not a specific user registered for an event. Again, this limits the number of accesses to the database and is safe despite the lag present in read-replicas.

4 Additional Performance Gains

While caching contributed to a large chunk of the performance gains, other areas needed some attention to achieve our scalability goals. These included queuing registration requests, optimizing SQL statements, tuning the web servers, and tuning Linux TCP parameters.

4.1 Database Queuing

With a target of 50,000 registration in 10 minutes, that places a huge load on the database to actually save the registration. That averages out to 83 database connections a second, but it is more likely that the initial minutes would be spend filling out forms, meaning that over 1,000 connections a second is by no means unrealistic. Therefore, the registration process was updated to queue registrations instead of having each process save its own registration. After a user's payment is accepted by the payment processor, their registration information is stored in an SQS queue. This limits the number of database connections that will be established.

Multiple background processes consume messages from the queue to save the registrations to the database and report registration IDs back to the user. This also has the advantage of reducing the overhead of database connections and allows for prepared statements to be shared across multiple registrations. The number of background processes is customizable, with each server automatically spawning up to 10 processes to handle registrations. Each process reads up to 10 messages from the queue at a time, which further limits the number of processes spawned. All this helps to reduce database contention and allows the system to scale to large number of registrations.

4.2 Optimizing SQL Statements

By analyzing all of the SQL statements used on critical pages, we were able to identify statements that could be optimized. Rewriting statements to do more efficient JOINs and adding indices to tables produced a few benefits. It helped to reduce the reduce average page load times, particularly under high load. This also reduced the load on the database, which also helped other statements run faster under heavy load since there was less contention. Furthermore, some database statements were updated to use the read-replica database if the statements could do so without issues related to replica lag.

4.3 Tuning Nginx & Apache

By default, the Nginx version we were running used a new, very strong SSL cipher that was very slow. The Nginx configuration was updated to use a different cipher that was much quicker, but still PCI compliant [20]. On both the Nginx and Apache servers, the configuration files were tuned to change the maximum number of requests served, update the socket backlog, and enable compression, along with other updates. Some of the key portions of the configuration files for the Nginx load balancers and Apache web servers can be seen in Appendix A.1 and Appendix A.2 respectively.

4.4 Linux TCP Tuning

With increasing demand on the site during testing, we ran across web server connection issues and NAT connectivity issues. We experimented with many different settings, primarily focusing on the socket backlog size, the number of Nginx worker processes, and Linux TCP parameters. This tuning continued throughout the entire load testing process. The final settings that we found to be adequate for high load are listed in Figure A.3 of Appendix A.

5 Building a Test Framework

In order to load test our infrastructure, we developed a test framework with the following benefits:

- It provided a consistent, automated, and repeatable way to run load tests against our system.
- It rented large numbers of test servers to allow for massive numbers of simulated registrations.
- It kept track of previous tests to monitor progress and compare results.

Used in conjunction with New Relic [21], an infrastructure monitoring server, we were able to profile slow transactions and pinpoint some bottlenecks that deserved more attention. Portions of this code are available at `https://github.com/RunSignUp-Team/OpenSource/tree/master/loadTesting`.

5.1 Simulating Registrations

The first part of this entailed being able to programmatically simulate a registration. We looked into multiple online cloud-based testing frameworks, but decided they did not suit our needs and were too expensive. We needed something more functional than Bees with Machine Guns [22] and more cloud native than JMeter [23]. Since we needed a test framework with session support and it was fairly simple to fulfill our needs with a few custom PHP classes, we decided to develop our own framework. We used PHP and `curl` to create a class that simulated a session on RunSignUp, including accepting cookies, simulating user delays, outputting debugging information, and loading resources needed on the web pages. Additionally, it implemented a simple browser cache using the `If-Modified-Since` and `If-None-Match` HTTP headers. Furthermore, it was capable of establishing Keep-Alive connections for loading page resources. However, it did not use persistent connections for the page loads since real-world connections would typically expire between page loads. Lastly, it included a function that was used to automatically fill in field names such as name, address, credit card information, etc. Another PHP class parsed the returned HTML and produced a PHP `DOMDocument` [24]. It also provided other functions to easily check for page errors; get form fields; and get the URLs of images, stylesheets, and JavaScript that should be loaded.

Using these base classes, we derived a class that handled the logic involved in race registrations. The class simulated a user registration by loading the URL of the race page. From here, the code searched the page for the link to sign up for the race. Once the link was found, the URL was used to start the registration. The code then entered a loop that processes the current page to determine which action to take next. This was done by simply checking for certain form elements that were unique to the different pages. We implemented the code this way since the registration process is not identical for all races. For example, some races may enable teams, donations, or stores while others may not.

When a registration was simulated, some information was sent to another process that collects stats that were used to display real-time data as a load test was running. Furthermore, the framework stored the raw HTML and `curl` connection information for each page load. This information was analyzed after the test to summarize the test and stored in S3 for later manual evaluation.

5.2 User Interface

We designed a simple user interface, as shown in Figure 4, which allows quick setup of load tests. It gives the option of registering for a particular event in the race or having each simulated user randomly select an event. The form includes the number of registrations that should be simulated and delays that should be simulated. These are specified in terms of minimum and maximum times that are used to randomly produce a delay. Using these parameters, we can simulate the initial burst of people hitting the website when registration opens, but allow for longer delays between form submissions to accurately simulate people entering their information. There is a setting to specify whether or not external resources, such as images, stylesheets, and JavaScript, should be loaded. The test only downloads resources from the runsignup.com domain since we want to load test our setup, not other proven external services such as CloudFront and Google's Hosted Libraries [25].

Furthermore, we have settings for the number of web servers and test servers to use. The number of web servers is just for future reference as new servers will not be automatically started. However, the number of test servers determines how many test servers the load test will use.

After submitting and confirming the test, the website redirects to a page that shows the progress of the test. When completed, the results of the test are displayed along with graphs of the data, as shown in Figure 5. Initially, only the gauges and bottom text box are visible and update in real-time as the test runs to display progress messages. A progress bar is also shown and automatically updated during the test. The page also has a comments section where we can document things such as the exact infrastructure setup, CPU usage information we gathered during the test, changes that we ran the test to check, error

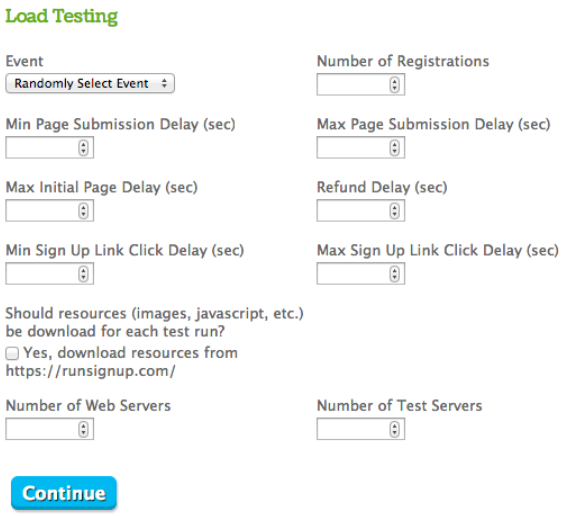

Figure 4: Setting up a registration load test.

analysis, or any other information we want to store for later reference.

5.3 Background Load Test Process

When a load test is set up and submitted, a background process is spawned on the web server that the request came in on. This background process runs throughout the duration of the test to set up and control the test servers and report data back to the UI. Since this can be quite resource intensive as the number of test servers increases, the script first removes this web server from the server pool so no requests are sent to it.

Next the script tries to start up enough test servers to run the test. The test servers that we use are EC2 spot instances [26], which are EC2 instances that are typically available at a lower price, but may be preempted at any time if another EC2 customer bids a higher amount. This allows us to get powerful test servers for much cheaper. Furthermore, once an instance is started, we pay by the hour, so an instance is left running after the test in case it is needed again. The next test that is done can use the same instances to save money as well as setup time. There are cron jobs that take care of shutting down the servers when they approach the the start of a new hour. The cron jobs simply uses the database to find servers that should be shutdown, then uses SSH calls to shut down each server.

Once the test servers are available, a setup script is run to install the required packages (e.g. php), raise kernel limits on the number of processes and open files, and update other settings. The background process then creates an archive of all the files that will be needed on the test server for running the tests and

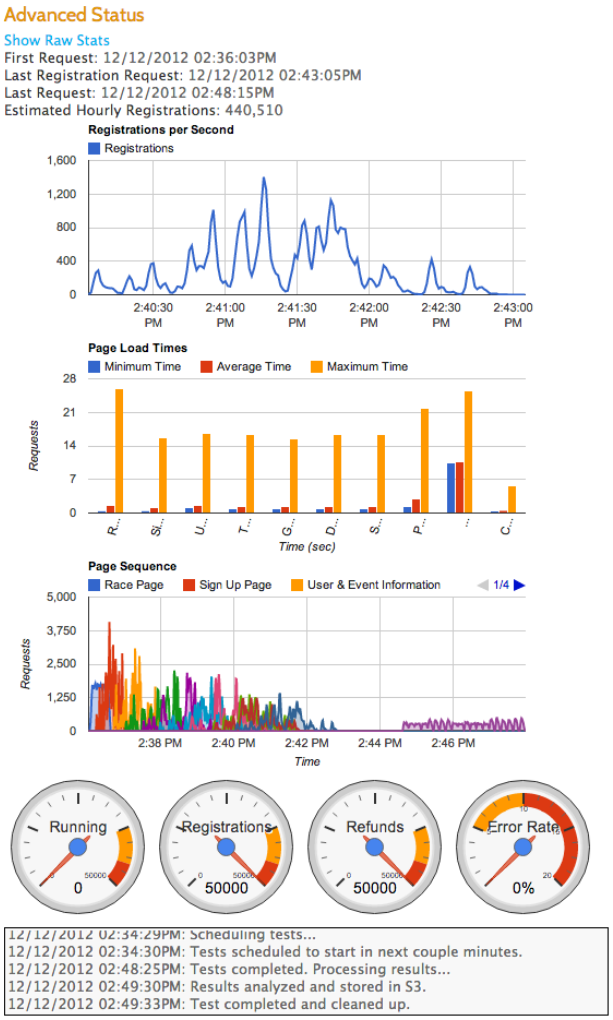

Figure 5: Setting up a registration load test.

analyzing the data. This archive is uploaded to S3 and a command is sent to each test server to download the file and unpack the files.

When all of the setup of the servers is completed, an initial registration is tested. There are two reasons for doing this. First, it ensures that if there is an issue that would prevent registration, such as registration being closed, we don't waste time running all the simulated users. Secondly, it allows the script to figure out how many pages it took to complete the registration. This data is then used during the test to provide a fairly accurate progress bar.

After the initial test has successfully completed, several cron jobs are set up on each test server. Each cron job starts up to 25 PHP processes that will each simulate a registration. By using multiple cron jobs, more processes can be started in a shorter period of time. Cron jobs also allow all of the NTP synced test servers to start the test simultaneously. An additional cron job is set up to collect statistics from the PHP processes.

The background process running the load test now just continuously requests data from the stats collectors on each test server. This information is used to update the gauges and progress bar in the UI. When the test is completed, which can be determined by looking at the stats, this is reported to the UI while it requests each test server to analyze its data and archive the raw HTML and logs to S3. The analyzed data that was returned is then merged together and stored in the database. This data is then sent to the UI to build the graphs.

One piece of data that we generate is an estimated hourly registration rate. This is a conservative estimate that takes the total number of registrations and divides by the time from the first request to the last registration completion. Assuming a continuous stream of new users, the system could likely handle a much higher load than is reported.

5.4 Open Source Framework

We intent to come back to the test framework we developed and create a open source, general purpose load testing tool. The major pieces of code that went in to the load testing framework can be found at `https://github.com/RunSignUp-Team/OpenSource`.

6 Load Testing Results

We did two types of load tests on the system. The primary focus was on scaling for surges in registrations. The secondary focus was on scaling our results capabilities.

6.1 Race Registrations

Listed in Table 2 are a few of the hundreds of load tests that were performed on our infrastructure. The first test is an easy test with only a few of the performance updates to give a reference point for other results. The other tests attempted to simulate a realistic user experience, with simulated delays of 15-60 seconds for filling out the forms. However, they both required that all 30,000 or 50,000 registrations started within the first 30 seconds of the test. During this traffic burst, there was a peak of 4,904 requests per second for the 50,000 registrations test. With 45 web servers, this equates to about 109 requests (all HTTPS) for each server. With an average response time of around 1.5 seconds, this means that each server would likely have an even larger number of concurrent requests.

This explains why 45 web servers were needed. After the initial burst, the load on each server was lower and the average number of requests per second per server became quite low. For example, the 50,000 registration test goes through 10 pages in 7 minutes and 2 seconds to complete each registration. This is 500,000 page loads, averaging out to only about 26 page loads per second per server. However, the need to handle the initial page burst necessitates the extra capacity.

One also needs to take into account the redirect that happens on 6 of the 10 page requests. The redirect happens when the user submits a form POST request. The initial POST request processes the information, then redirects to prevent the user from submitting the same form twice. The redirected page must then be requested from the user, resulting in

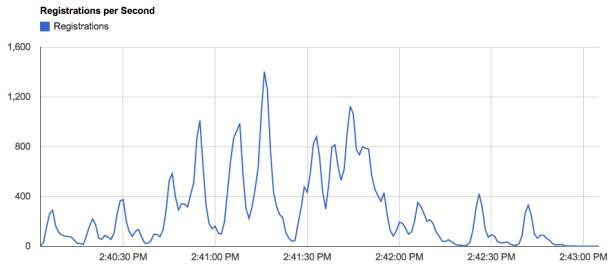

Figure 6: Registration per second during 50,000 user test.

2 actual HTTP requests for each of these 6 pages. Therefore, for are 16 HTTP requests per registration. For the test with 50,000 registrations, this means 800,000 HTTP requests and an average of 42 HTTP requests per second per server. While this is still low compared to the initial burst, it is something that needs to be considered when scaling the system.

During our testing, the system spiked at over 1,500 registration completing per second as seen in Figure 6. This figure shows the number of registrations per second that reached the confirmation screen. The jaggedness of this graph is due to the waiting screen the user is on prior to reaching the confirmation screen. This screen always waits at least 10 seconds before checking if the registration is completed. The waiting follows an exponential backoff pattern, with the initial wait of 10 seconds, followed by a wait of 20 seconds if the registration did not complete, followed by a 40 second wait, and any further wait times will be 1 minute. This helps to limit page requests if the database queue gets too backed up.

Table 3 shows a couple of tests on the typical daily infrastructure described in Table 1. The delays are set to simulate semi-realistic user interaction. As is evident, the more modest infrastructure with smaller servers struggles a little with the demand, with high CPU load on the web servers. However, all other servers in the infrastructure, including the load balancers, stayed below 10% CPU usage. This is a good sign, since a couple extra web servers can be added very quickly to handle the spike. This has potential to be completed automated since the process is simple and not likely to run into issues that require human intervention.

6.2 Viewing Race Results

With large races, the influx of runners wanting to view their race results online can easily take down or severely limit the functionality of the system. Our system displays searchable results that are displayed in pages, with 10 results per page by default. We also have an auto-complete search box using AJAX. By utilizing one or two database read replicas and setting up short-lived items in the cache, we handled peaks of over 3,000 result page requests per second. This includes both searches and normal results viewing and is in addition to the simulated auto-completes, which top out near 5,000 auto-completes a second.

Figure 7 shows the graph of the number of pages per second experienced during a load test. The top line shows the total across all pages. Our tests showed sustained periods hovering around 2,000 result page requests per second, which equates to 120,000 requests per minute and over 7 million requests an hour We also saw over 7,500 SSL requests per second at some points during our testing. With 45 web servers running, this equates to about 167 requests per server.

7 Conclusions

Switching from a single server website to a cloud-based, multi-server architecture with extensive caching greatly improved the scalability of RunSignUp. We scaled from struggling to handle 1,000 registration bursts to handling 50,000 registrations in under 10 minutes. Similar steps and methods can be used to scale other websites as well. The key to such a process is systematically finding and eliminating bottlenecks. Much of the time, the bottleneck is a single database server. By implementing a caching hierarchy using APC and memcached, the database load can be greatly reduced. For write heavy operations, queuing the tasks can provide huge benefits by reducing database contention. The use of IaaS and PaaS systems provided by Amazon Web Services or similar services allows for scalability in a

No. Registrations	1,000	30,000	50,000
No. Load Balancers	2 (m1.large)	4 (m3.2xlarge)	6 (m3.2xlarge)
No. Web Servers	10 (m1.large)	25 (m3.2xlarge)	45 (m3.2xlarge)
Success Rate	94%	100%	100%
Test Duration	15:21	5:43	7:02
Avg. Response Time	∼32 sec	∼1 sec	∼1.5 sec
Peak Req. Per Sec	N/A	N/A	4,904
Est. Hourly Registrations	3,659	318,599	440,510

Table 2: Load Testing Results For Full Registration

No. Registrations	250	1,000
Page Submission Delay	5 - 10 sec	5 - 40 sec
Initial Page Delay	0 - 30 sec	0 - 120 sec
Success Rate	100%	100%
Test Duration	2:47	8:10
Avg. Response Time	∼2.2 sec	∼7.9 sec
Max. CPU Usage on Web Servers	81.9% & 74.1%	86.2% & 86.4%
Max. Response Time	14.6 sec	45.9 sec
Peak Req. Per Sec	46	61
Est. Hourly Registrations	6,522	7,347

Table 3: Load Testing Results For Full Registration On Typical Daily Infrastructure

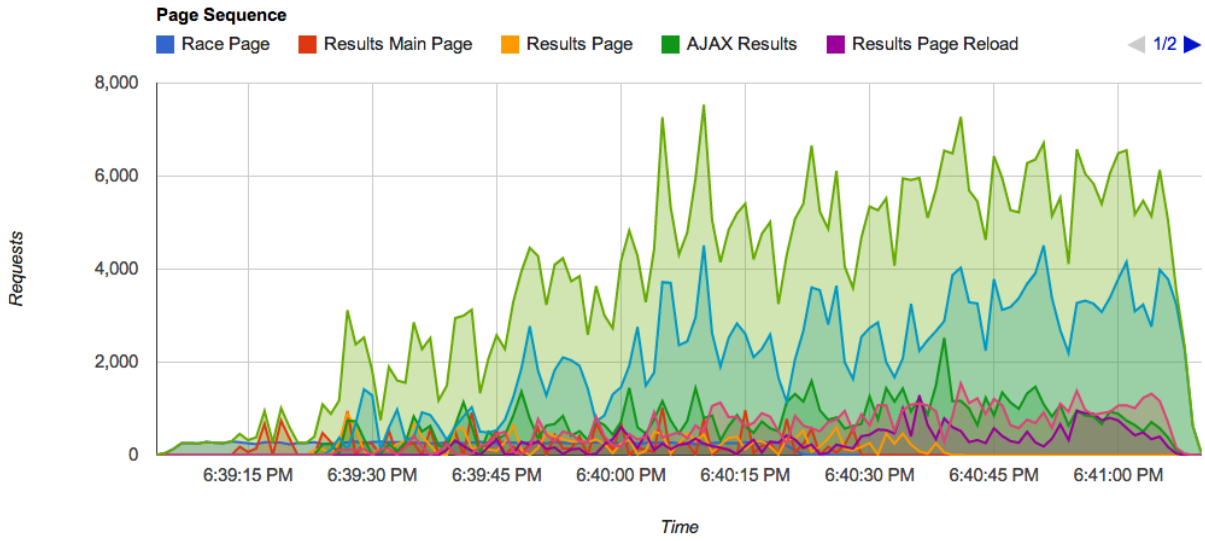

Figure 7: Graph of Number of Page Requests per Second During Results Load Test

cost effective way, which can be passed on to the customer. Furthermore, it allows for a highly-available system that is resilient to many network disruptions and server failures. Lastly, having a test framework and infrastructure monitoring software is extremely beneficial to this process, helping bottlenecks to be spotted quicker and changes to be tested without too many other variables.

8 Thank You

We would like to thank Rich Friedman for his time and advice in helping to develop our infrastructure and caching framework.

References

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

[2] "Amazon Elastic Compute Cloud (Amazon EC2)." `http://aws.amazon.com/ec2/`.

[3] "Amazon Virtual Private Cloud (Amazon VPC)." `http://aws.amazon.com/vpc/`.

[4] "memcached - a distributed memory object caching system." `http://memcached.org/`.

[5] "PHP: APC - Manual." `http://php.net/manual/en/book.apc.php`.

[6] "Amazon Route 53." `http://aws.amazon.com/route53/`.

[7] "NGINX." `http://nginx.org/`.

[8] P. Srisuresh and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)." RFC 3022 (Informational), Jan. 2001.

[9] "Apache HTTP Server Project." `http://httpd.apache.org/`.

[10] "Amazon Simple Email Service (Amazon SES)." `http://aws.amazon.com/ses/`.

[11] "Amazon Simple Queue Service (Amazon SQS)." `http://aws.amazon.com/sqs/`.

[12] "Amazon Relational Database Service (Amazon RDS)." http://aws.amazon.com/rds/.

[13] "Amazon Simple Storage Service (Amazon S3)." http://aws.amazon.com/s3/.

[14] "Amazon CloudFront." http://aws.amazon.com/cloudfront/.

[15] "Github." https://github.com/.

[16] "Amazon Simple Notification Service (Amazon SNS)." http://aws.amazon.com/sns/.

[17] "Nagios." http://www.nagios.org/.

[18] "Php: Overloading - manual." http://php.net/manual/en/language.oop5.overloading.php#object.call.

[19] "MySQL :: MySQL 5.0 Reference Manual :: 14.2.8.3 SELECT ... FOR UPDATE and SELECT ... LOCK IN SHARE MODE Locking Reads." http://dev.mysql.com/doc/refman/5.0/en/innodb-locking-reads.html.

[20] "Official PCI Security Standards Council Site." https://www.pcisecuritystandards.org/.

[21] "New Relic : Web Application Performance Management (APM) & Monitoring." http://newrelic.com/.

[22] "Bees with Machine Guns." https://github.com/newsapps/beeswithmachineguns.

[23] "Apache JMetere." http://jmeter.apache.org/.

[24] "The DOMDocument class." http://php.net/manual/en/class.domdocument.php.

[25] "Google Hosted Libraries." https://developers.google.com/speed/libraries/devguide.

[26] "Amazon EC2 Spot Instances." http://aws.amazon.com/ec2/spot-instances/.

A Configuration Files

A.1 Key Configuration for Nginx Load Balancers.

```
worker_processes          64;
worker_rlimit_nofile      65536;

events {
  worker_connections      16192;
  multi_accept on;
}

http {
  # Buffer access log
  access_log  /var/log/nginx/access.log  main buffer=64K;

  # Sendfile on
  sendfile        on;

  # 15 second keep alives
  keepalive_timeout  15;

  # Enable gzip compression
  gzip  on;
}

server {
  listen       443 backlog=16384;

  # Set PCI complient ciphers
  ssl_protocols SSLv3 TLSv1 TLSv1.1 TLSv1.2;
  ssl_ciphers RC4:HIGH:!aNULL:!MD5:!kEDH;
  ssl_prefer_server_ciphers   on;

  location / {
    # Enable HSTS
    add_header Strict-Transport-Security max-age=15768000;

    # Set proxied headers
    proxy_set_header    Host $host;
    proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;

    # Proxy request
    proxy_pass          https://runsignup;
  }
}
```

A.2 Key Apache Configuration for Web Servers

```
StartServers              75
MinSpareServers           50
MaxSpareServers           100
ServerLimit               256
MaxRequestWorkers         256
MaxConnectionsPerChild    4000
ListenBacklog             8192
```

A.3 Linux TCP Parameters for Nginx Load Balancers

```
net.ipv4.tcp_max_syn_backlog = 65536
net.ipv4.ip_local_port_range = 2048 64000
net.core.somaxconn = 65536
net.ipv4.ip_conntrack_max = 262144
net.nf_conntrack_max = 262144
net.netfilter.nf_conntrack_max = 1048576
net.core.netdev_max_backlog = 4000
net.ipv4.tcp_max_tw_buckets = 1048576
```

B Code

```php
<?php

public function __call($method, $params)
{
  // Check if this is a mysqli method
  static $mysqliMethods = null;
  if ($mysqliMethods == null)
    $mysqliMethods = get_class_methods('mysqli');
  if (in_array($method, $mysqliMethods))
  {
    // Set up MySQLi
    if (!$this->mysqli)
      $this->setupMySQLi();

    // Call function
    return call_user_func_array(array($this->mysqli, $method), $params);
  }

  // Try to find the method in the modules
  foreach ($this->modules as &$mod)
  {
    // Check if the method exists
    if (method_exists($mod, $method))
      //return call_user_method_array($method, $mod, $params);
      return call_user_func_array(array($mod, $method), $params);
  }

  // Raise error
  $trace = debug_backtrace();
  trigger_error(
    'Undefined method ' . $method,
    E_USER_NOTICE);
  return null;
}

?>
```

Listing 1: Deferred database connections and external module loading in RunSignUp database wrapper.