

# Performance-tuning a scalable Ruby-on-Rails web app

Course project

CS291: Scalable Internet Services

Michael Zhang, Sammy Guo, Sujaya Maiyya, Kyle Carson, Justin Pearson

December 8, 2017

## Abstract

For our course project for *CS291: Scalable Internet Services* at the University of California, Santa Barbara, we created a car-sharing app in Ruby on Rails, deployed it to Amazon AWS Elastic Beanstalk, and used Tsung to load-test it. We used various methods from the course to tune the site's performance, achieving a continuous simultaneous user flow of 128 users/second at a cost of \$3.46/hour.

## 1 Introduction

“Internet services” are a broad class of services in which a customer’s computer or mobile device interacts with server computers owned by the company providing the service. This definition encompasses essentially every company that conducts business over the Internet. The success of an Internet service crucially depends on its ability to scale up to serve an increasing number of clients and requests per second; high latency and unavailability are not acceptable. Scaling up an Internet service amounts to increasing server capacity intelligently, e.g., by weighing the costs of:

- implementing the service in a fast language like C++ or PHP,
- buying more powerful servers (“vertical” scaling),
- distributing the service across multiple servers (“horizontal” scaling),
- separating the service between an HTTP server and an App server,
- placing servers in key geographic locations,
- distributing the database (“sharding”),
- using an easily-distributed data model (e.g., distributed key-value store),
- caching at various levels,
- using and tweaking HTTP 2,
- and other optimizations.

Each approach must be balanced against the cost of the engineer-time required to implement it. For this reason, it’s best to keep things as simple and cheap as possible for the current and near-term customer load. However, it’s important to know the maximum capacity of a given configuration; this is the purpose of *load testing*, which simulates many simultaneous clients in order to discover performance bottlenecks before they occur in production.

This report describes the load-testing of a scalable web app we built for the course *CS 291: Scalable Internet Services* at the University of California at Santa Barbara. CS291 presented architectures for scalable systems and techniques for solving scalability problems. We exercised these techniques by building a car-sharing web app in Ruby-on-Rails, deploying it to the Internet via AWS Elastic Beanstalk, and load-testing it with Tsung.

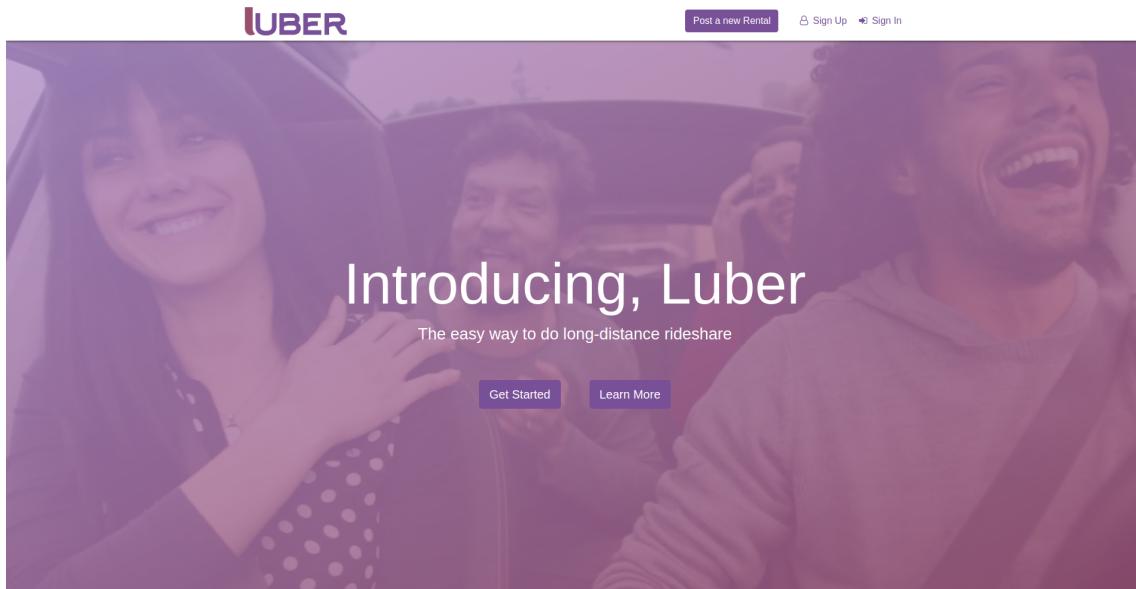
We proceed as follows. In Section 2 we describe our app. Section 3 describes our load-testing investigation. We conclude in Section 4.

## 2 The Luber web app

In this section we describe our car-sharing web app, “Luber” ([www.luber.fun](http://www.luber.fun)). We built Luber with Ruby on Rails and hosted it on the Internet via AWS Elastic Beanstalk. Luber allows people to rent out their cars that would otherwise be sitting idle. Instead of Alice’s car sitting in her driveway or her employer’s parking lot all day, she can make a little money by allowing Bob to rent it for an hour.

### 2.1 Site overview

The site uses the Bootstrap layout library; here is the home page:



Guests (non-signed-in clients) can only access static pages like “About”, “FAQ”, etc:

**Motivations**

(For a simple explanation of what Luber's function is, see the [FAQ](#) section)

With the services aforementioned above, only short distance rides in highly populated urban-cityscapes are truly feasible. Areas which are suburban, rural, or just less populated overall tend to have less demand for taxi-like services, thus making those rides that are available increasingly expensive. Furthermore, even in locations where these services are available, going any long distance becomes expensive quickly, so much so that it just stops making sense economically.

With Luber however, the renter has the freedom to take the owner's car wherever they please, as long as they meet the requirements set out by:

- The start and end locations
- The start and end times
- The terms explicitly set by the owner

This makes long distance travel much more viable, while simultaneously benefitting the car owner in that they need not be present for the duration of the rental, but only for the drop-off and pick-up of their vehicle.

**Sitemap**

- Home
- About
- FAQ
- Contact Us
- Privacy Policy

**Development**

- GitHub
- Issues

© 2017 Luber. All rights reserved.

Signed-in users can browse the cars available to rent:

**LUBER**

Post a new Rental    Rentals    user123    Sign Out

## Available Rentals

13 currently available

**Available** Glendale, CA → Garden Grove, CA **\$163**

From Tuesday, Dec 12th at 7:22 PM until Tuesday, Dec 12th at 8:22 PM  
 ↗ Terms: Dont try to impress everyone.  
 ↗ Owner: admin01  
 🚗 Mazda Fusion  
 ↗ Tags: sunroof 5-seater

**Available** Roseville, CA → Hayward, CA **\$163**

From Tuesday, Nov 28th at 7:22 PM until Tuesday, Nov 28th at 8:22 PM  
 ↗ Terms: When you quit, you fail.  
 ↗ Owner: admin01  
 🚗 Mazda Fusion  
 ↗ Tags: sunroof 5-seater

**Available** Corona, CA → Huntington Beach, CA **\$168**

From Tuesday, Nov 28th at 7:22 PM until Tuesday, Nov 28th at 8:22 PM  
 ↗ Terms: Make what is valuable important.

Users can see an overview of their account:

**LUBER**

Post a new Rental    Rentals    user123    Sign Out

 **user123**  
 A member since December 5th, 2017

### My Info

user123  
 Bryce Boe  
 Goleta, CA  
 user123@gmail.com  
 Standard User

**About Me** **My Meetup Preferences**

Hi my name is Bryce! I teach CS291A at UCSB

Anywhere downtown, during the day

### Recent Rentals as an Owner

A user can monitor the progress of the rentals they are involved in:

 **user123**  
 A member since December 5th, 2017

**Upcoming** Vallejo, CA → Santa Maria, CA **\$172**

From Tuesday, Dec 12th at 10:35 PM until Tuesday, Dec 12th at 11:35 PM  
 ↗ Terms: You cant always get what you want.  
 ↗ Owner: user123 (You are the owner)  
 ☰ Renter: skater46  
 🚗 Kia Corolla  
 ↗ Tags: 5-seater sporty

**In Progress** Santa Clara, CA → El Cajon, CA **\$12**

From Wednesday, Dec 6th at 12:35 AM until Wednesday, Dec 6th at 1:35 AM  
 ↗ Terms: Be your best at all times.  
 ↗ Owner: user123 (You are the owner)  
 ☰ Renter: skater48  
 🚗 Kia Corolla  
 ↗ Tags: 5-seater sporty

A user can view the details of a specific rental that they are either the owner or renter of:

The screenshot shows a rental listing for 'Rental Post #6' created today at 10:35 PM. The listing is for an 'Upcoming' rental from Vallejo, CA to Santa Maria, CA, costing \$172. The rental period is from Tuesday, Dec 12th at 10:35 PM to Tuesday, Dec 12th at 11:35 PM. The terms are 'You cant always get what you want.' The owner is 'skater42' and the renter is 'user123 (You are the renter)'. The car is a 'Kia Corolla' from 2005, black, with license plate 4YMZ893 and tags for '5-seater' and 'sporty'. The owner is 'skater42' (Bob Jones, Goleta, CA, user2@boo.com, Standard User). The car is a 'Kia Corolla' (2005, black, 4YMZ893, 5-seater, sporty). The start location is Vallejo, CA and the end location is Santa Maria, CA. Maps are provided for both locations.

We used the `geocoding` gem to convert a rental's "Start" and "End" location strings into coordinates, and used the Google Maps API to display the maps. However, we were careful to disable these features when load-testing, in order to focus on our own app's performance.

## 2.2 Data Model

There are six main models in our app. **Users** can own **Cars** and create **Rentals** for those cars. A Car has many **Tags** describing its amenities: "sun-roof", "stereo", "off-road", etc. A join model called **Taggings** holds (car,tag) pairs that store each car's tags. A user can view his or her recent activity through **Log** entries.

A Rental has a state to track its progress: "Available", "Upcoming", "In Progress", etc. Figure 1 illustrates a Rental's progression through its states:

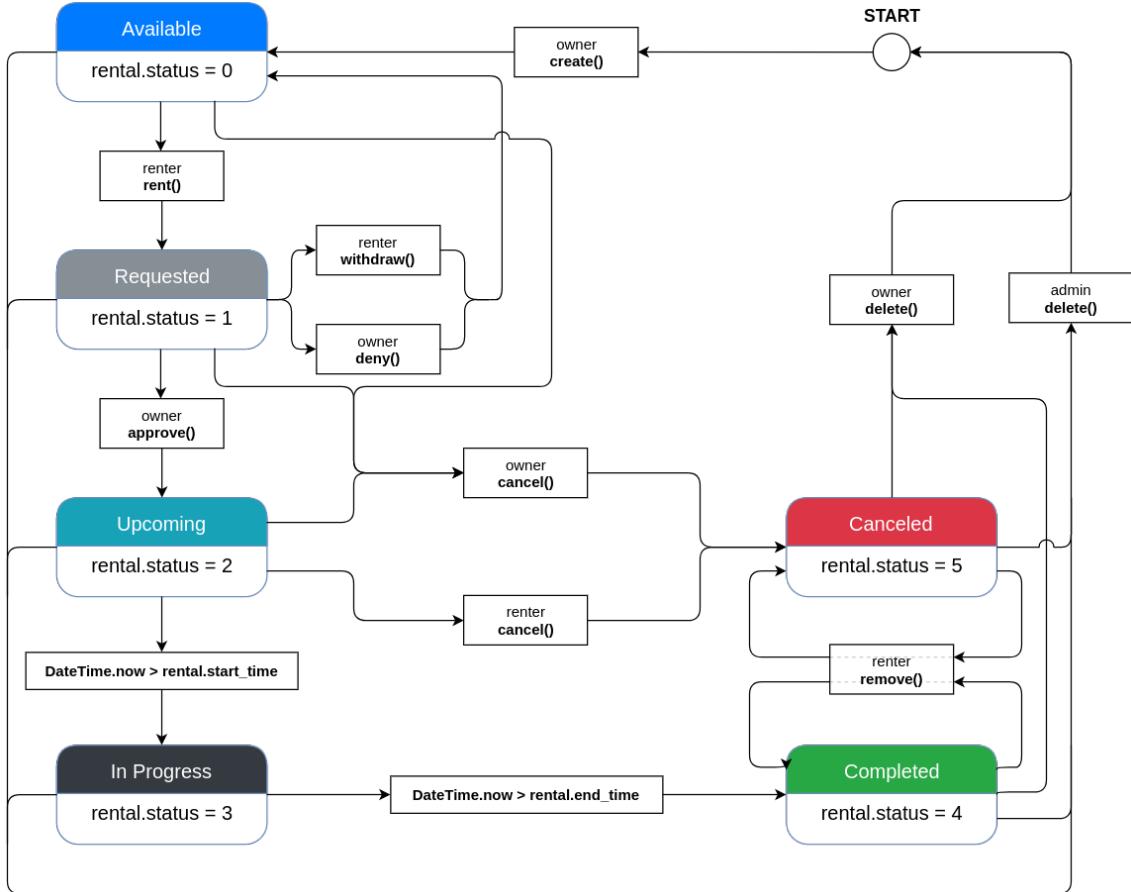


Figure 1: A Rental's lifetime.

Rails applications use the model-view-controller design pattern. The entity relation diagram for Luber's data model appears in Figure 2. A few aspects of the data model are worth mentioning. The `geocoder` gem automatically converts `start_location` and `end_location` varchars into their corresponding latitude and longitude floating-point values. The `bcrypt` gem and Rails `has_secure_password` method are used to store password digests for security, as advised in [railstutorial.org](http://railstutorial.org). Rails validations are used to verify the length and format of nearly every field, even ensuring a Rental's license plate adheres to DMV regulations.

Models diagram  
 Date: Dec 04 2017 - 21:05  
 Migration version: 20171119063345  
 Generated by RailRoady 1.5.3  
<http://railroadystatic.prestonlee.com>

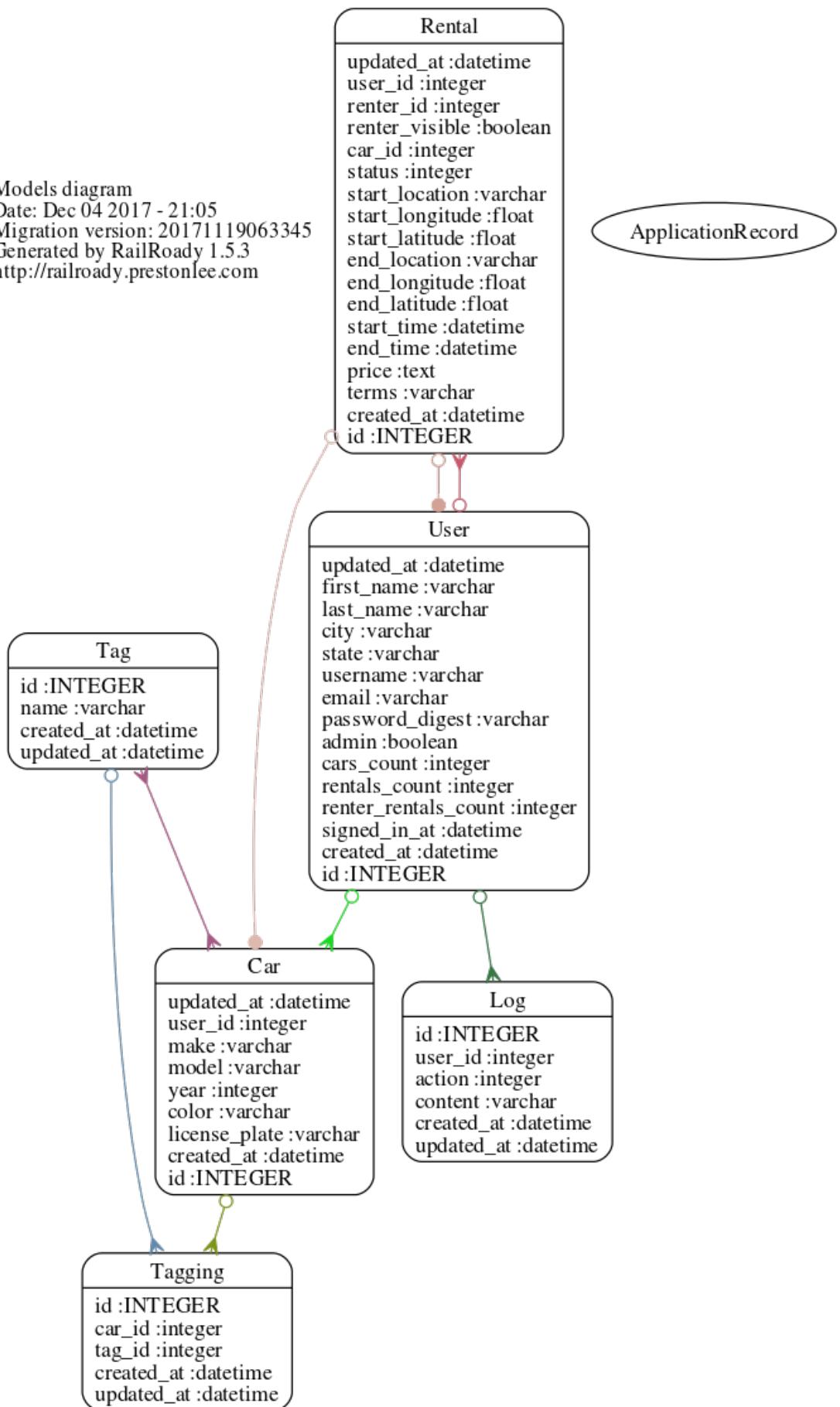


Figure 2: The complete ER diagram of data model.

Luber's controllers relation diagram is depicted in Figure 3. The Rails routes file was configured so that URLs are user-friendly, e.g., `luber.fun/users/bob/cars`.

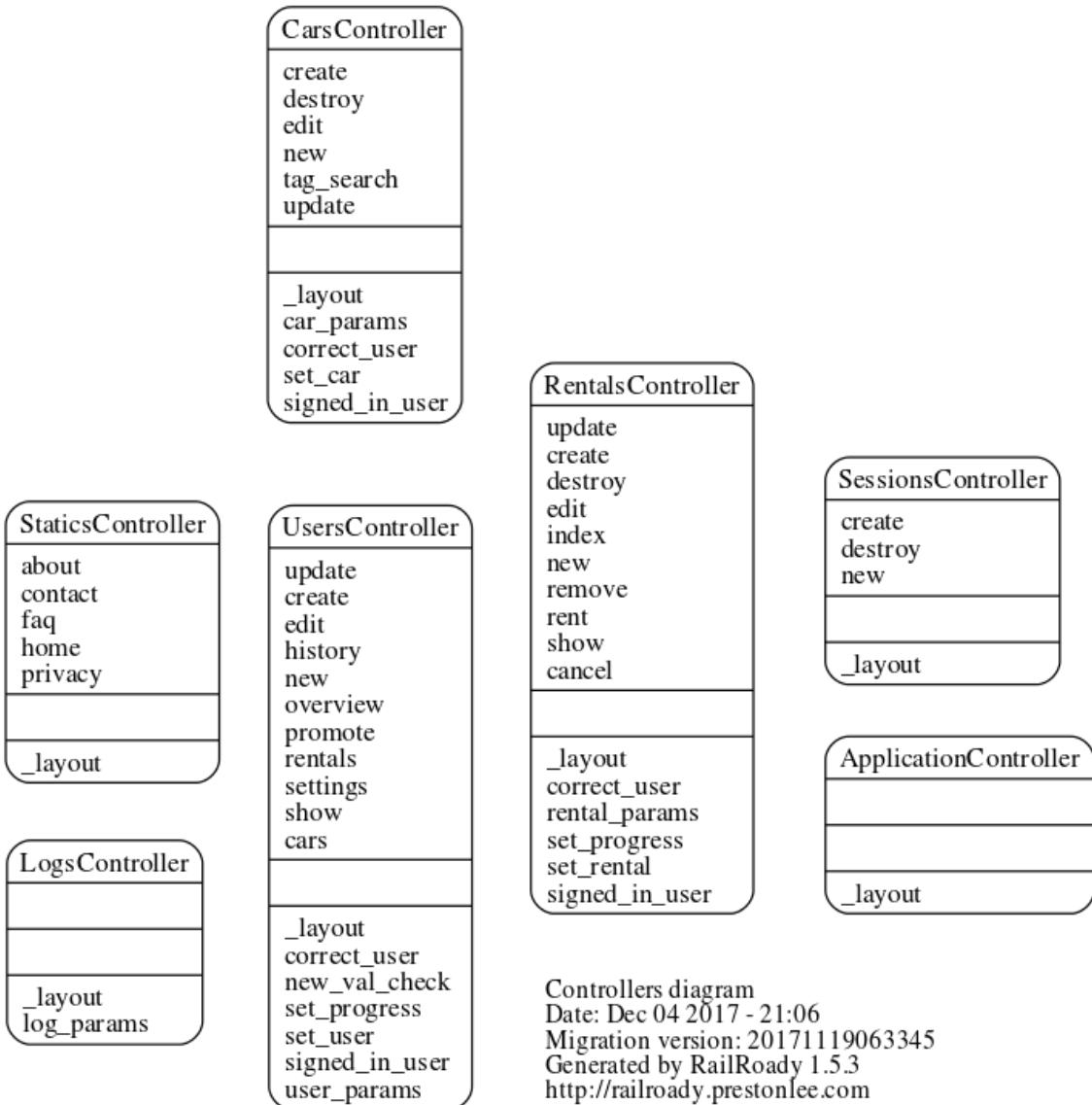


Figure 3: The complete ER diagram of controllers

### 3 Load testing

In this section we describe how we load-tested the Luber app with Tsung. We first describe the “typical user workflow” that Tsung simulated. Then we describe how our app performed under several different hardware configurations. We also explored optimizing the site using pagination, caching, and increasing the maximum number of concurrent connections on the Nginx HTTP server.

#### 3.1 Workflow for a “Typical User”

We created a Tsung XML file to simulate a “typical user” on the Luber site, executing these actions (see Figure 4):

1. User login

2. Add a car
3. Add a rental
4. Edit the rental
5. Delete the rental
6. Delete the car
7. User logout

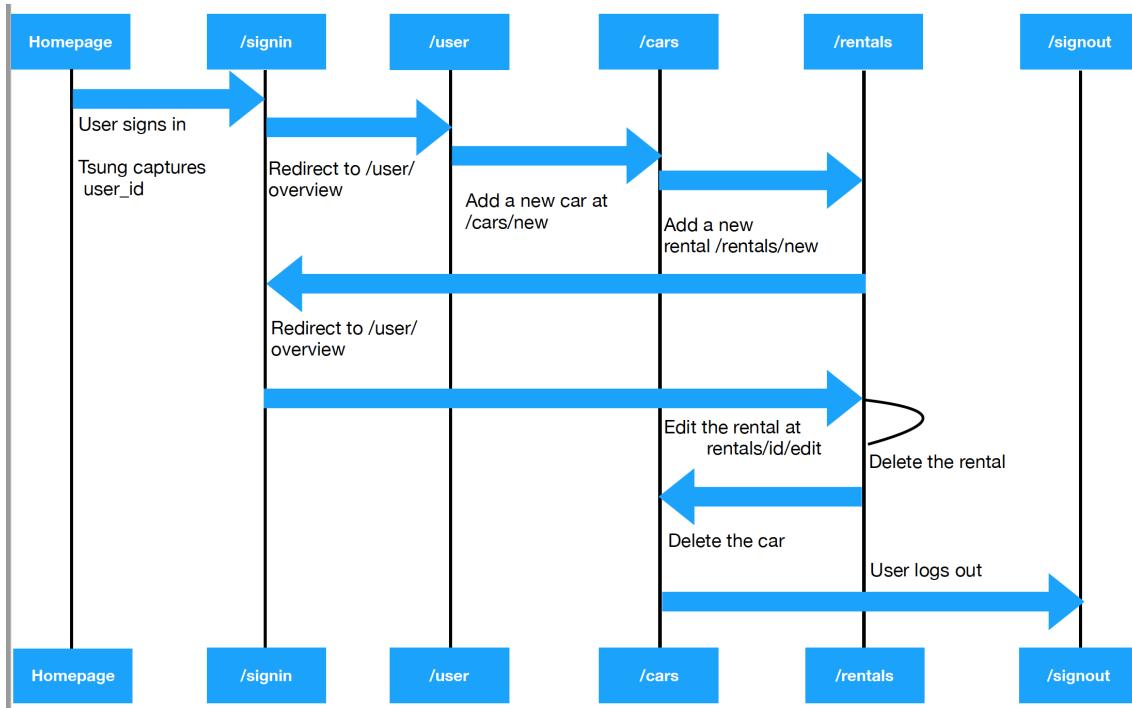


Figure 4: Sequence diagram for user workflow

For simplicity, we encoded this behavior as a single Tsung session, instead of separating each action into its own session and assigning a probability to it. It would have been better to have one Tsung session dedicated to creating rentals and have another session dedicated to renting them at an equal rate. However, it turned out to be difficult to deal with the concurrency problems that arise when two Tsung sessions attempt to rent the same Rental simultaneously. Therefore it seemed best to simulate a large set of idempotent sessions, each defining a user who acts in an isolated fashion. One minor snag with this scheme was that a user cannot rent his own Rental, so we chose instead for the user to simply *edit* his own Rental, which we supposed would impose a load on the database similar to a user renting a Rental; editing and Renting each result in a PATCH request, so this seemed like a good assumption.

The Tsung XML file defined 9 phases, differing only in the number of sessions (users) spawned per second. It started with 1 session spawned per second, and doubled that rate in each successive phase. Each phase lasted 60 seconds. We configured the phases to wait until the previous phase finished; this delineated the data nicely. Under this scheme, the number of “simultaneous users on the site” as a function of time should appear as successively larger humps. Indeed, such a plot appears in Figure 5.

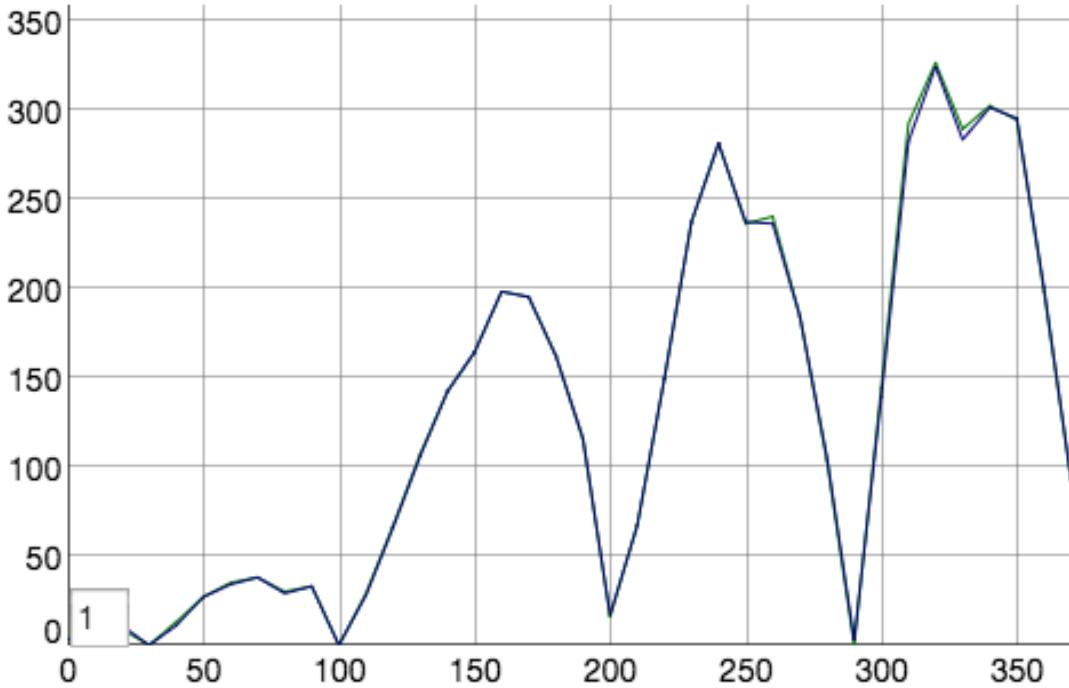


Figure 5: Number of simultaneous users spawned by our Tsung XML file as a function of time (sec). Note that in this figure, the first phase — spawning 1 user/sec — lasts only 30 sec; the others are 60 sec long.

### 3.2 Varying instance number and type on Elastic Beanstalk

To explore Luber’s performance, we ran our Tsung load-test against the Luber app running on several different hardware configurations in AWS Elastic Beanstalk. Specifically, we varied the number of instances (App servers), the instance types (m3.medium, etc), and the instance type of the database server. Table 1 shows the various configurations. Before each test, we deleted all database records and re-seeded it with 11000 users, 2000 cars, 10000 rentals, 20 tags, and 4000 taggings. We ran the same Tsung XML file for each configuration.

Instance type	Num. instances	DB instance type
m3.medium	1	m3.medium
m3.medium	1	m4.2xlarge
m3.medium	4	m3.medium
m3.medium	4	m4.2xlarge
c5.2xlarge	4	m4.2xlarge
c3.4xlarge	1	r3.2xlarge
c3.4xlarge	2	r3.2xlarge
c3.4xlarge	4	r3.2xlarge

Table 1: Hardware configurations we load-tested against.

#### 3.2.1 Results

We divided the various steps within our single Tsung session into transactions, allowing us to assess how long it takes a user to sign in, rent a car, delete it, etc. A one-second thinktime was added to each of these transactions to simulate human users. Figure 6 shows a healthy run, in which each transaction takes 1000 to 1200 ms, which amounts to 0–200ms transaction time after subtracting the 1-second thinktime. The transaction time stays constant during this run, indicating that some degree of steady-state was achieved. On the other hand, an unhealthy run (256 users/sec), appears in Figure 7. Here, the transaction times skyrocket unpredictably to between 2 and 8 seconds.

## Transactions

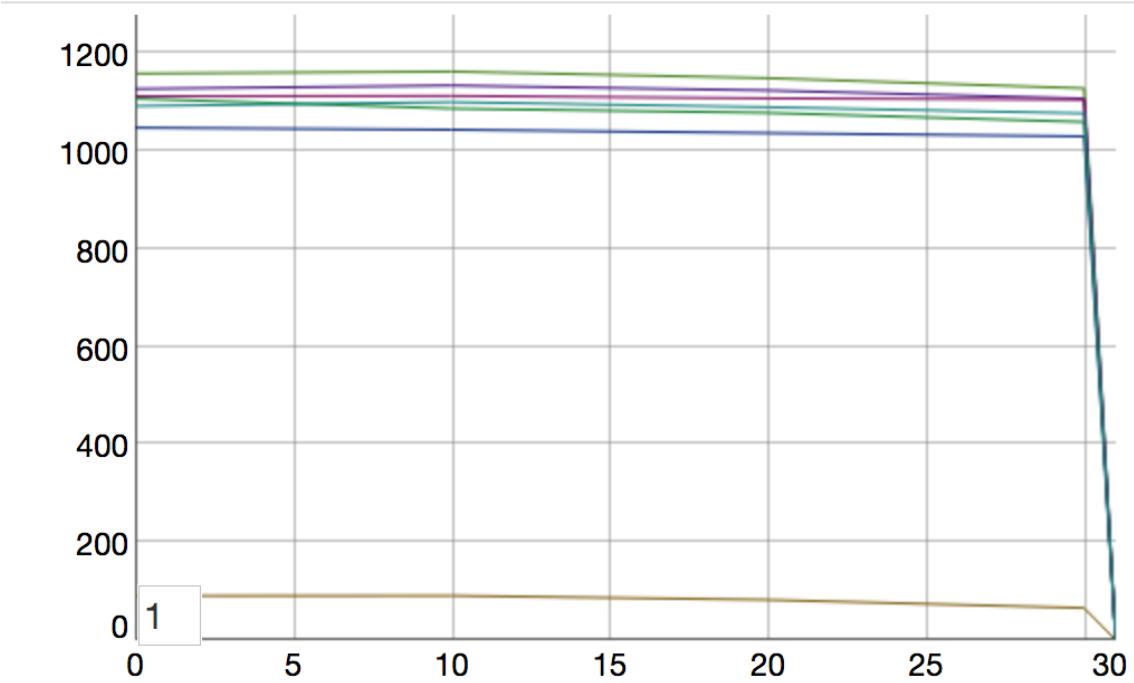


Figure 6: Normal response time for transactions is 1000–1200 ms (1 second of which is the think-time).

## Transactions

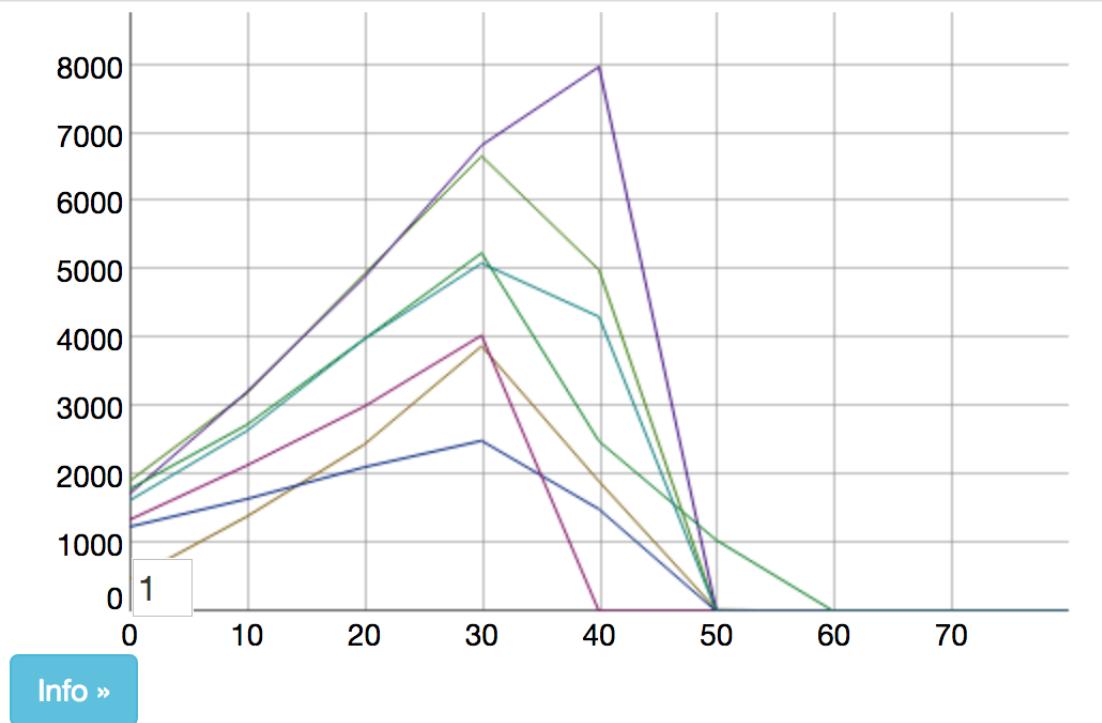


Figure 7: Transaction response time in a heavily-loaded system.

Figure 8 compares (1) the number of users that can be supported by a given hardware configuration versus (2) the cost of the hardware configuration. We consider the maximum number of users for a hardware configuration to be simply the largest user-spawn rate that does not produce 400- or 500-level HTTP responses or Nginx web server errors. It would have been better to define more Tsung phases with various user-spawn rates — not just powers of 2 — because it would have provided better x-axis resolution. Nevertheless, Figure 8 allows us to easily compute the cost of supporting a given number of users per second. For example, if usage data reveals that at peak times there are 128 users on the site, then we should change the Elastic Beanstalk configuration to run four c3.4xlarge instances and a r3.2xlarge database, which would cost \$3.36 per hour. Each user during that time would be costing \$0.0004375 per user.<sup>1</sup>

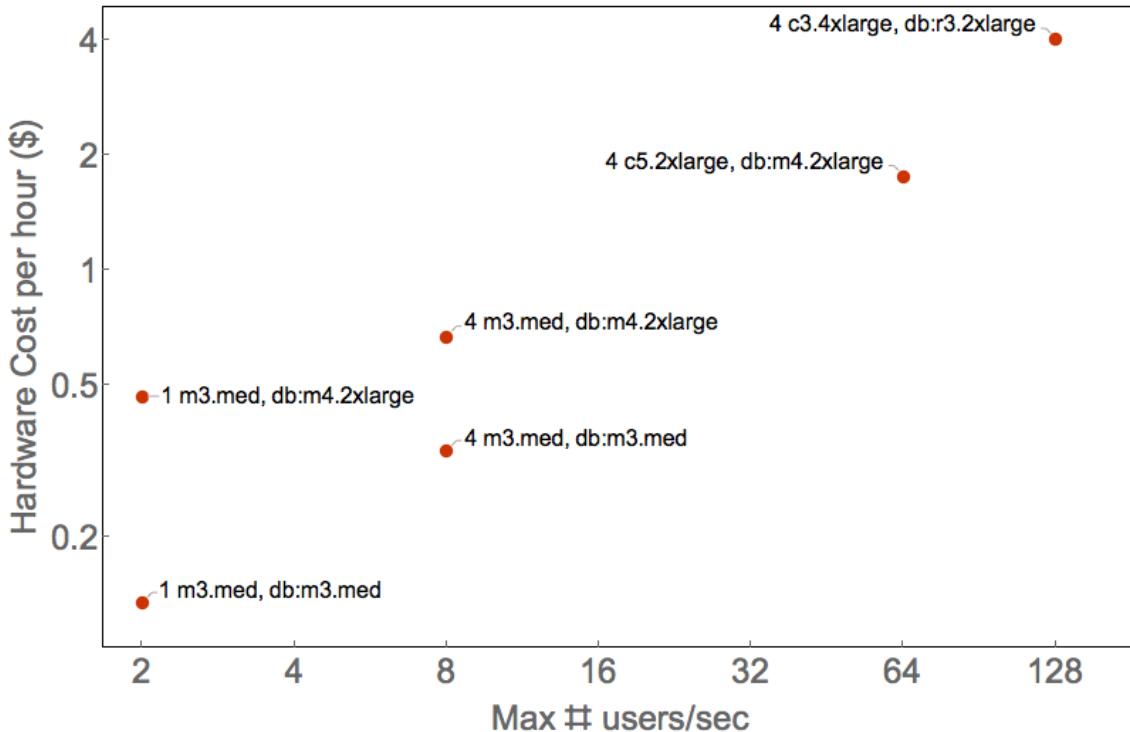


Figure 8: Hardware cost versus user capacity for several AWS hardware configurations.

### 3.3 Pagination

When there are large numbers of rental posts (i.e.  $> 1000$ ) our application will try to query, render, and display all 1000 rental posts in our rental post view. Thus one of the first optimizations we applied was to paginate the view so that the application will not constantly overload the database and the Ruby process.

As shown in Figure 9 the performance of our application with no pagination was abysmally slow; we could only add a user every 20 seconds across a long period of time to avoid crashing (i.e. HTTP 500 errors) Rails. Many requests took between 15 to 20 seconds to query, render, and display while our beanstalk instance spiked to over 60 percent CPU usage. We had to configure Tsung to add a new user every 20 seconds for 3 minutes.

Figure 9 shows the reduced application response time thanks to pagination. Our rental post view now shows only 8 posts per page instead of the entire 1000 in our database. Thus the application's response time stays below 10 seconds.

While 10 seconds may not be the best response time for a practical application, we are pushing the limits of our given elastic beanstalk infrastructure (m3 medium instances for app and db server). Our Tsung configurations for user arrival rates are derived through trial-and-error; we

<sup>1</sup>Since not every user is using the site at once, the total number of users may be many times the peak number of users loading the site.

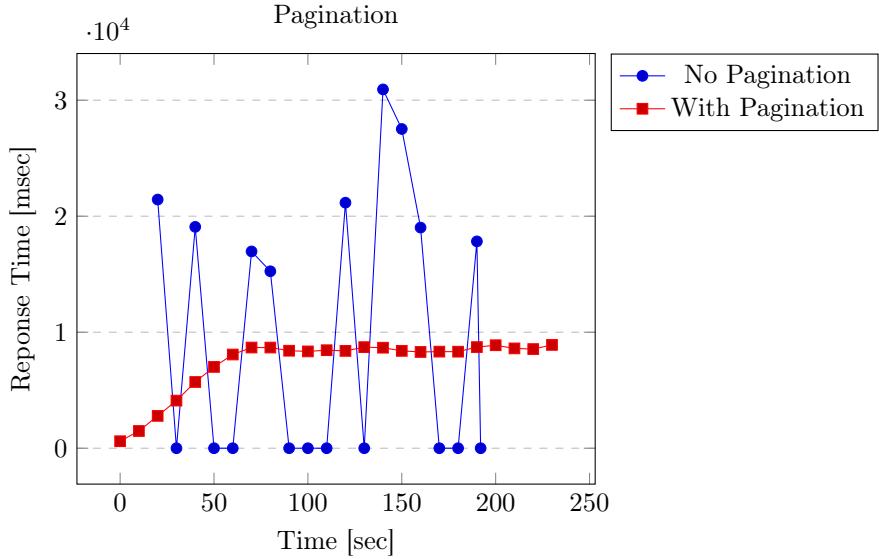


Figure 9: Pagination optimizations halved response times for rental listings. Paginated version tested with 2 new users per second for 60 seconds.

kept increasing the user rate until the application throws HTTP 500 errors. The point at which beanstalk stops throwing 500s are where we report this new performance capability.

In other words, our optimizations gave us better performance under higher stress.

### 3.4 Caching

After implementing pagination, we explored caching as another avenue for performance boosting. We switched to a beefier c3.4xlarge instance since we anticipated caching to be most effective on larger scale systems.

We applied Tsung tests to the same rental listing pages and established a new baseline for our c3.4xlarge instance. Figure 11 shows a new average response time around 2 seconds with 2 new users per second for 60 seconds.

We'd like to point out the performance difference of using a larger EC2 instance compared to the m3.medium we used for the pagination tests. Turns out that a machine with over 10 times more memory (3 GB vs 30 GB) and 16 times more processors (1 vs 16 cores) results in over 4 times application speedup (see Figure 10).

We postulate several reasons why our Russian-doll fragment caching didn't work:

- The view rendering was not the performance bottleneck; since we did not observe a major speedup in our results.
- For each page view of the rental listing, our application still had to query the database for details on the rental posts. This is where we think the majority of our application runtime was spent.

In retrospect we would have focused our efforts on caching database queries to reduce the overall transaction load on our db instance. One idea we thought may work would be to analyze the mean time between all requests that modify/create any rental post. We'll use that value to set the expiration time for an in-memory cache of as many rental posts as we can. The idea is to cache the rental for the average period it will likely change. Thus, if there are large numbers of users listing rentals, most of the requests will hit the memory cache instead of querying the database assuming users are more likely to browse rentals than modify them at any given time. As long as that's the case, we suspect this proposed caching scheme will improve the overall performance of the application.

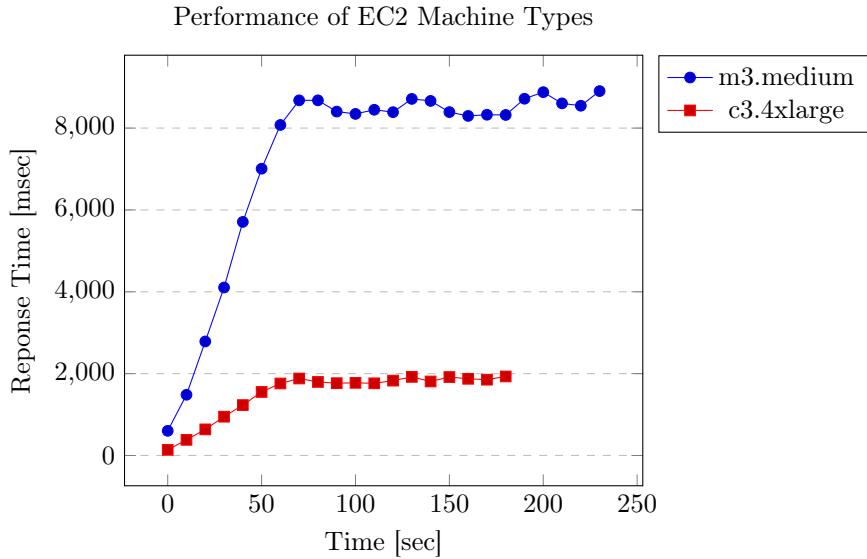


Figure 10: Simply switching our EC2 instance to a more powerful type gave us over 4 times speedup.

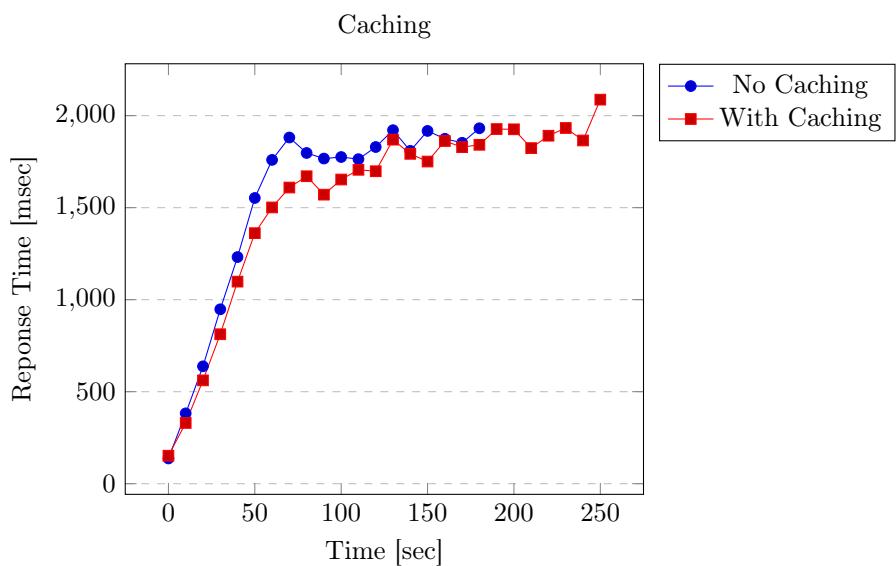


Figure 11: While we did observe minor speedups after implementing Russian-doll fragment caching, we've decided that the performance bottleneck may lie in the database transaction speeds.

### 3.5 Number of concurrent connections

Each of the elastic beanstalk instances runs Nginx web server. Web server limitations can be one of the biggest bottlenecks while trying to scale an application. During load testing, one of the bottlenecks we found was the number of concurrent connections allowed by Nginx. The default values in nginx.conf for all elastic bean instances was 1024. This implies that at any point if the number of connected users increase more than 1024, there are high chances that the server will start throwing 5xx errors for the http connections. This may not be a deterministic behavior as web servers may allow more connections, depending on the resources available.

In order to load test our application with respect to number of concurrent users it can handle, we used Tsung. The test case had 2 sessions with an arrival rate of 128 users/second for one minute:

- Session 1: A user logs in, rents a car, waits for a few seconds, returns the rented car, signs out. 70% of Tsung tests go through this session.
- Session 2: A user logs in, adds a car, creates a rental from it, waits for a few seconds, removes the rental, removes the car, signs out. 30% of the Tsung test go through this session.

In the figure 12, at time 50th second, we can see that the number of simultaneous users are highest (with a value of 1616). Figure 13 plots the number of different HTTP responses sent by the server over time. In 13, we see that at the 50th second, the server starts throwing different errors, the point at which the number of concurrent connections was maximum. And the kind of error we faced was `error_connection_closed` as seen in 14.

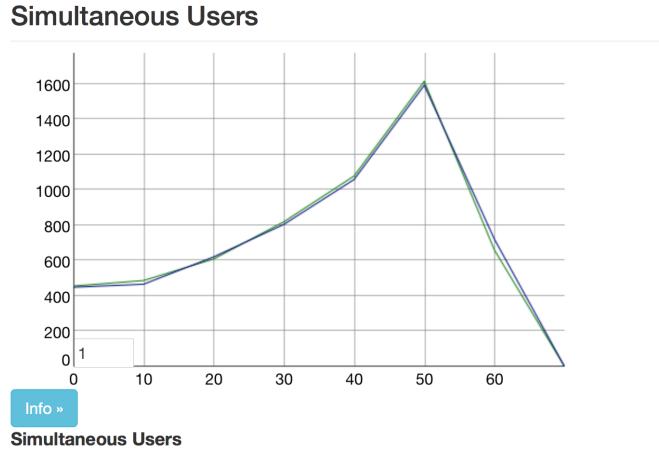


Figure 12: Number of simultaneous users

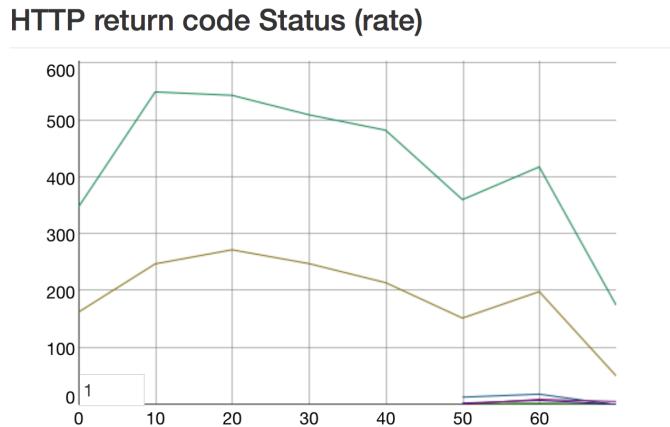


Figure 13: HTTP responses

## Errors

Name	Highest Rate	Total number
error_connection_closed	4.2 / sec	64

Figure 14: Errors

In contrast to the above plots, consider the figures 15 and 16. In this situation, the maximum number of users is 519, at 40th second. This number is significantly lower than the previous case and we also notice that there are no HTTP errors in this case.

## Simultaneous Users

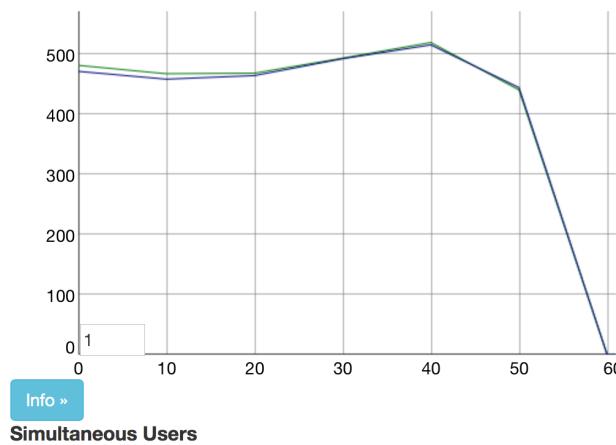


Figure 15: Number of simultaneous users

## HTTP return code Status (rate)

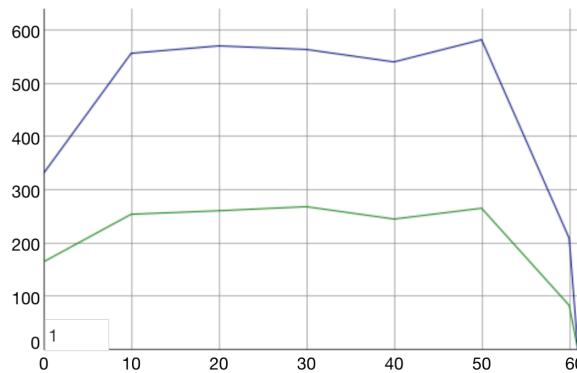


Figure 16: HTTP responses

From the above plots it was evident that the web server starts misbehaving when the number of concurrent connections increase more than 1024, which is the default limit of Elastic Beanstalk instances. The Nginx configurations are stored in `/etc/nginx/nginx.conf`. The field `worker_connections` holds the number of allowed concurrent connections. We tried different approaches to increase the limit by adding a proxy config file and by manually ssh-ing into the eb instance and updating the `nginx.conf` file. But none of it seemed to actually update the configuration. Some of the other optimizations we tried was to update the `Network Tier` configurations. If load balancing is enabled, which is the case in our application, the default setting is to drain the connections after 20 seconds. We disabled the connection draining, as most of the errors faced

during Tsung test was `error_connection_closed`. We also enabled Sessions stickiness and increased the Health check interval to 30 seconds, as each health check request establishes a TCP connection, adding to the number of concurrent connections. This reduced the average time taken for by each session from 6.8 seconds to 4.79 seconds (the session time also includes thinktime used for simulating actual user behavior). This reduction in session time in turn reduced the number of concurrent connections and thus our application could support an arrival rate of 128 users per second without any errors.

## 4 Conclusion

This paper described a scalable web app — Luber — and how its performance was measured and improved using the Tsung load tester, Amazon AWS Elastic Beanstalk, and concepts from the UCSB Computer Science course *CS291: Scalable Internet Services*.

In this paper, we introduced the Luber website and described its data model and controllers. We defined the workflow of a “typical user” and implemented it in Tsung load-tests. We ran these load tests against a variety of AWS hardware configurations — both vertical and horizontal scaling — to get a sense of the tradeoff between peak use and cost. Then, we explored additional optimizations — pagination, caching, and boosting the max number of concurrent Nginx connections — and measured their effect on performance.

We are confident that our Tsung tests represent a typical user workflow, and that Luber can handle a decent amount of traffic based on our load testing results.