Arpeggio

CS188: Scalable Internet Services

https://github.com/scalableinternetservices/Arpeggio

Ryan Hansberry	804018356
Karan Kajla	304136376
Marko Vojvodic	504158739
Qianwen Zhang	004401414

Table of Contents

I. Introduction

Arpeggio is a peer-to-peer musical instrument and equipment rental marketplace. It is very similar to services like Uber and Airbnb where a single resource or product is shared by many users one-at-a-time through rental over a period of days. Using Arpeggio, users looking to rent out their musical equipment to other musicians can do so by posting listings on the marketplace. Other users looking to rent this equipment can do so by browsing the listings on the marketplace, with options to filter by location and keyword search, and ordering the product for rental in a similar manner to any other online store.

II. Development

Our team utilized agile software development techniques throughout the course to implement Arpeggio. We used Pivotal Tracker as a collaboration tool for specifying application features and assigning team members stories to complete. Github and TravisCI were used for version control and to ensure all team members were up to date on the codebase, as well as to notify all members on the health of the most recent build. Unit tests were run through Rails built-in test fixtures, and load testing was executed through the Tsung load testing tool.

We used a mixture of pair programming and solo programming for feature implementation. Task assignment was done in an agile fashion using Pivotal Tracker, with each team member having a specific task (or story) assigned to them for the upcoming week on the Pivotal Tracker board. At the end of each week during the discussion section, we got together as a team to unblock anyone who was having trouble completing their weekly story, decide upon stories for the following week, and do some pair programming to implement key features that involved group decisions.

Any code written by a single member was reviewed by all members of the team to ensure code quality. Whenever a significant feature or change was to be implemented by a team member, he or she would create a new branch on which to implement this feature, implement whatever needed to be done, have other team members review the branch, ensure that it passed all unit tests, and finally merge it into the master branch. If tests were not passing because they did not properly cover new features, they were modified or new tests were added to achieve better test coverage. Developing using this technique gave us a lot of flexibility as to which of our production code was used. Buggy code was caught and fixed before it hit the master branch, and features that were initially wanted, but later became unwanted, were easily rolled back to a previous commit.

Coming from fairly experienced web-development backgrounds, we wrote the initial (before studying and performing scaling optimizations) version of Arpeggio with performance and efficiency in mind by default. This means we wrote fairly efficient SQL queries from the beginning, not making unnecessary iterative queries and including extra indexes on any models that would likely require them. On the front-end side this entailed adding AJAX functionality to improve the response time of the application in the user's eyes. However, these are just a few optimizations that we implemented as a default. We were still able to add numerous optimizations to increase performance and scale in the later stages of optimization. These are covered in Section V.

III. Application Architecture

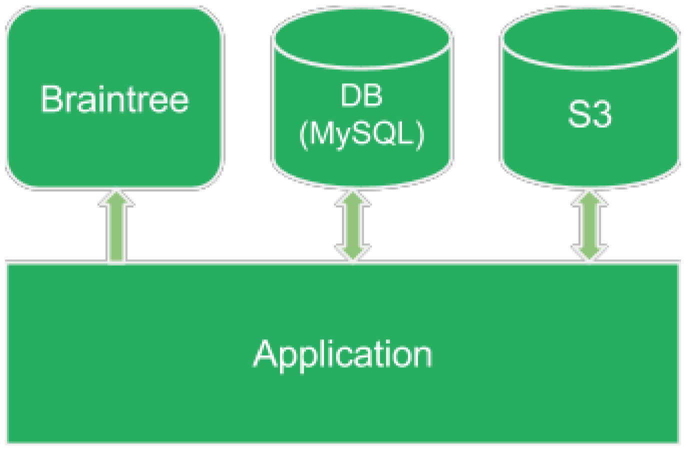


Figure 1: A high-level overview of the application architecture and interactions performed when serving a request

Arpeggio's overall application architecture consists of four primary entities: the database (MySQL) for persistence of user and application data, Amazon S3 for persistence of file upload data, Braintree's payment processing service (API) for handling checkout, and the application itself. All four of these units interact with each other frequently, with the application being the link between the other three. A high-level diagram of these four units and their interactions is shown in Figure 1.

The database and the application are constantly interacting back and forth between each other to signal when new objects are created, a query needs to be performed, and when data is ready to be used in controllers and in views, among many other things.

The application and Amazon S3 are also constantly interacting with each other in a similar fashion. Whenever a user uploads an image, the image is initially stored on disk on the application server (think of this as a staging area) and is eventually sent to S3 to be efficiently and effectively persisted so as to not use up the application server's disk space. Additionally, all views containing these uploaded images are served from S3, so the application server requests each file from S3 as the view is rendered, before being sent to the user.

The interaction between the application and Braintree is mostly one-way (which is why a one-sided arrow is drawn in the figure above) and occurs during user account creation and order creation (checkout). This is because the application utilizes API calls to send well-formatted messages containing things such as account creation information, payment information, or transaction information to Braintree's services where that information is used to create a payment account, record credit card or bank information, or create a transaction on behalf of the user. Braintree does not frequently interact back with the application, only under certain circumstances when information validation or a transaction fails.

IV. Critical Paths

In evaluating the performance of our application and determining the effectiveness of our performance optimization, we exercise three different critical paths through the application: the path followed by typical renters, the path followed by those who put their instruments up for rent lenders, and the path followed by users who do not immediately sign up, but frequently browse the product listings, browsers.

The critical path followed by renters involves account creation, filtering and searching through products, and cart manipulation and checkout. This entails several application actions per user: A user must be created. Several SQL queries for keywords, item classification, and location must be run. A cart must be created and manipulated. Finally, an order must be created, and payments must be processed using third party API calls.

In the second critical path, users are considered lenders, so they are creating accounts and creating listings for their musical equipment, in addition to filtering and searching through products and manipulating their cart before checkout. This critical path involves the following application actions per user: A user must be created. A product listing must be created, and its image must be processed and uploaded to Amazon S3. Additionally, the same actions as for the renter critical path are performed.

In the third critical path, users are simply browsing listings and not performing any writes. This is the read-only case, and is the most basic of the three critical paths. This critical path involves the following actions per user: the products index page will be requested by every user, and each user will request the show page for one or more products. This path is primarily used to test the effects of server-side-caching in isolation. This idea will be discussed further in Section V which covers scaling and optimizations.

The first critical path is the most realistic of the three as we estimate that, much like on other similar web applications, the typical user will be browsing listings and occasionally adding a product to their cart in order to rent it out. This is why we utilize the first critical path to get an idea of how the application will typically perform. However, the second critical path is not discounted even though it reflects what will, more than likely, not be the average situation for our application. It still represents a sanity check for the write heavy operations our application performs, primarily image processing and upload to Amazon S3. Finally, the browser path provides us with the case of many users who will simply be browsing product listings before signing up for the service. That is why, taking all three of these critical paths into account collectively gives a fairly accurate description of our application's performance under load, as it would be in production, given a large real user base and dataset.

In addition to evaluating our application's performance using load scripts following these critical paths, we document improvements made on individual page load times as a direct result of optimizations that the load scripts cannot effectively take into account. These optimizations mainly consist of caching on both the client-side and server-side and optimizing any unnecessary SQL queries. The reason we decide to do this in addition to load testing is because the load tests do not accurately account for everything like client-side-caching since users during the load test do not frequently visit the same page multiple times. Coupled with all three load tests, documenting individual page load times will give a better overall picture of our application's performance from two different perspectives.

V. Scalability

A. Optimizations

1. Seeding

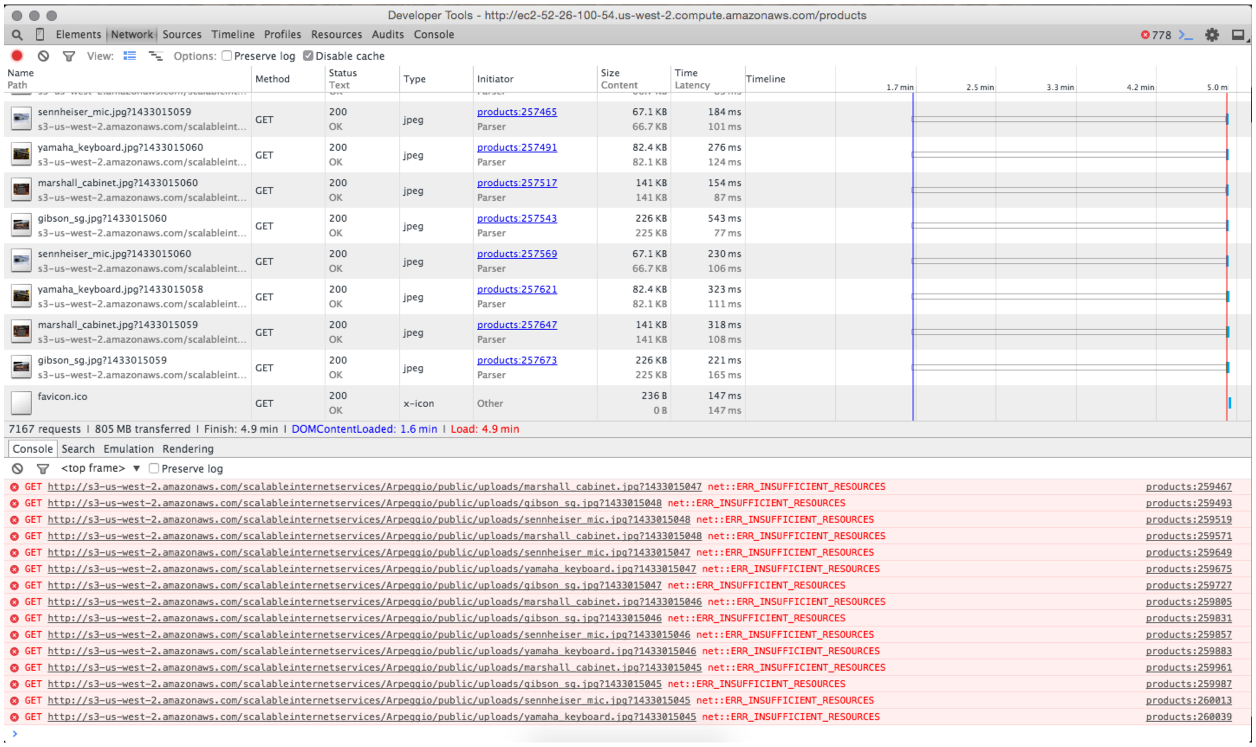
When testing required larger datasets, seeding the database or database instance became a time-consuming task of several hours. The delay resulted from thousands of images having to be uploaded to S3 from the application servers each time the seed script was run. Obviously, this delay was unacceptable for use of both human and technological resources. The solution was found in a gem called yaml_db. yaml_db allows for Rails application database dumping to and loading from .yml files. After integrating the gem into Arpeggio, database seed times were reduced to about 5 minutes for the 10,000 product dataset and about 20 minutes for the 100,000 product dataset. This optimization greatly improved the speed with which testing was executed. However, this method suffers from possible inconsistency; since the database is a copy, image URLs for files stored on S3 may be invalid at the time of loading the database. However, the tests run did not delete any objects, so the tradeoff of increased load time against decreased correctness clearly favored speeding up the testing procedures.

2. AWS S3

Arpeggio uses Amazon's S3 to provide storage for image uploads and static assets. S3 is reliable, fast, and cheap for file storage, and is easily integrated with EC2. Static assets, such as background images and icons, were manually uploaded to the S3 bucket. Image uploads are handed by the Paperclip and aws-sdk gems for Rails. Paperclip processes images uploaded to the application by caching them, transferring them to configured storage device, and adding image url and other information to a given product's database entry.

3. Pagination

Since the most visited page of the site is expected to be the products index, fast page response times for the page are vital for good user experience. The greatest optimization for the product index page was pagination. The effect of pagination was tested using the 10,000 product dataset.



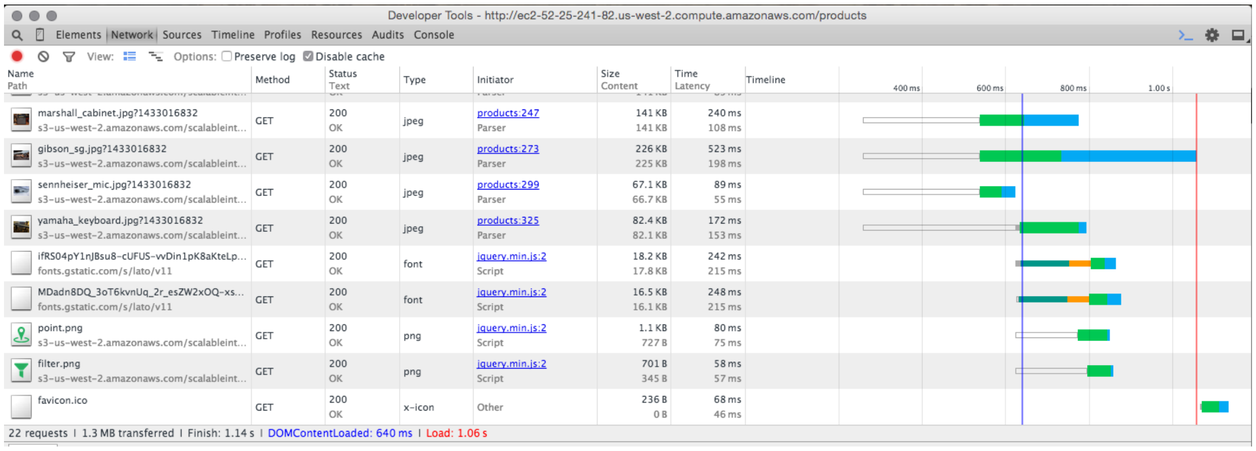
Name / Path	Method	Status / Text	Type	Initiator	Size / Content	Time / Latency	Timeline
sennheiser_mic.jpg?1433015059 / s3-us-west-2.amazonaws.com/scalableint...	GET	200 OK	jpeg	products:257465 / Parser	67.1 KB / 66.7 KB	184 ms / 101 ms	
yamaha_keyboard.jpg?1433015060 / s3-us-west-2.amazonaws.com/scalableint...	GET	200 OK	jpeg	products:257491 / Parser	82.4 KB / 82.1 KB	276 ms / 124 ms	
marshall_cabinet.jpg?1433015060 / s3-us-west-2.amazonaws.com/scalableint...	GET	200 OK	jpeg	products:257517 / Parser	141 KB / 141 KB	154 ms / 87 ms	
gibson_sg.jpg?1433015060 / s3-us-west-2.amazonaws.com/scalableint...	GET	200 OK	jpeg	products:257543 / Parser	226 KB / 225 KB	543 ms / 77 ms	
sennheiser_mic.jpg?1433015060 / s3-us-west-2.amazonaws.com/scalableint...	GET	200 OK	jpeg	products:257569 / Parser	67.1 KB / 66.7 KB	230 ms / 106 ms	
yamaha_keyboard.jpg?1433015058 / s3-us-west-2.amazonaws.com/scalableint...	GET	200 OK	jpeg	products:257621 / Parser	82.4 KB / 82.1 KB	323 ms / 111 ms	
marshall_cabinet.jpg?1433015059 / s3-us-west-2.amazonaws.com/scalableint...	GET	200 OK	jpeg	products:257647 / Parser	141 KB / 141 KB	318 ms / 108 ms	
gibson_sg.jpg?1433015059 / s3-us-west-2.amazonaws.com/scalableint...	GET	200 OK	jpeg	products:257673 / Parser	226 KB / 225 KB	221 ms / 165 ms	
favicon.ico	GET	200 OK	x-icon	Other	236 B / 0 B	147 ms / 147 ms	

7167 requests | 805 MB transferred | Finish: 4.9 min | DOMContentLoaded: 1.6 min | Load: 4.9 min

```
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/marshall_cabinet.jpg?1433015047 net::ERR_INSUFFICIENT_RESOURCES   products:259467
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/gibson_sg.jpg?1433015048 net::ERR_INSUFFICIENT_RESOURCES   products:259493
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/sennheiser_mic.jpg?1433015048 net::ERR_INSUFFICIENT_RESOURCES   products:259519
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/marshall_cabinet.jpg?1433015048 net::ERR_INSUFFICIENT_RESOURCES   products:259571
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/sennheiser_mic.jpg?1433015047 net::ERR_INSUFFICIENT_RESOURCES   products:259649
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/yamaha_keyboard.jpg?1433015047 net::ERR_INSUFFICIENT_RESOURCES   products:259675
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/gibson_sg.jpg?1433015047 net::ERR_INSUFFICIENT_RESOURCES   products:259727
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/marshall_cabinet.jpg?1433015046 net::ERR_INSUFFICIENT_RESOURCES   products:259805
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/gibson_sg.jpg?1433015046 net::ERR_INSUFFICIENT_RESOURCES   products:259831
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/sennheiser_mic.jpg?1433015046 net::ERR_INSUFFICIENT_RESOURCES   products:259857
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/yamaha_keyboard.jpg?1433015046 net::ERR_INSUFFICIENT_RESOURCES   products:259883
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/marshall_cabinet.jpg?1433015045 net::ERR_INSUFFICIENT_RESOURCES   products:259961
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/gibson_sg.jpg?1433015045 net::ERR_INSUFFICIENT_RESOURCES   products:259987
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/sennheiser_mic.jpg?1433015045 net::ERR_INSUFFICIENT_RESOURCES   products:260013
GET http://s3-us-west-2.amazonaws.com/scalableinternetservices/Arpeggio/public/uploads/yamaha_keyboard.jpg?1433015045 net::ERR_INSUFFICIENT_RESOURCES   products:260039
```

Figure 2: Product index page load time, pre-pagination

Figure 2 displays the page load time for the product index page from Chrome's developer tools. In addition the outstanding page load time of 4.9 minutes, the inspector reveals that the page needed over 7000 requests to be loaded and that the browser suffered from insufficient resource errors when attempting to render all the images on the page.


Developer Tools - http://ec2-52-25-241-82.us-west-2.compute.amazonaws.com/products

Elements | Network | Sources | Timeline | Profiles | Resources | Audits | Console

View: Options: ☐ Preserve log ☑ Disable cache

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	Timeline
marshall_cabinet.jpg?1433016832 s3-us-west-2.amazonaws.com/scalableint…	GET	200 OK	jpeg	products:247 Parser	141 KB 141 KB	240 ms 108 ms	
gibson_sg.jpg?1433016832 s3-us-west-2.amazonaws.com/scalableint…	GET	200 OK	jpeg	products:273 Parser	226 KB 225 KB	523 ms 198 ms	
sennheiser_mic.jpg?1433016832 s3-us-west-2.amazonaws.com/scalableint…	GET	200 OK	jpeg	products:299 Parser	67.1 KB 66.7 KB	89 ms 55 ms	
yamaha_keyboard.jpg?1433016832 s3-us-west-2.amazonaws.com/scalableint…	GET	200 OK	jpeg	products:325 Parser	82.4 KB 82.1 KB	172 ms 153 ms	
ifRS04pY1nJBsu8-cUFUS-vvDin1pK8aKteLp… fonts.gstatic.com/s/lato/v11	GET	200 OK	font	jquery.min.js:2 Script	18.2 KB 17.8 KB	242 ms 215 ms	
MDadn8DQ_3oT6kvnUq_2r_esZW2xOQ-xs… fonts.gstatic.com/s/lato/v11	GET	200 OK	font	jquery.min.js:2 Script	16.5 KB 16.1 KB	248 ms 215 ms	
point.png s3-us-west-2.amazonaws.com/scalableint…	GET	200 OK	png	jquery.min.js:2 Script	1.1 KB 727 B	80 ms 75 ms	
filter.png s3-us-west-2.amazonaws.com/scalableint…	GET	200 OK	png	jquery.min.js:2 Script	701 B 345 B	58 ms 57 ms	
favicon.ico	GET	200 OK	x-icon	Other	236 B 0 B	68 ms 46 ms	

22 requests | 1.3 MB transferred | Finish: 1.14 s | DOMContentLoaded: 640 ms | Load: 1.06 s

Figure 3: Product index page load time, post-pagination

Figure 3 displays the page load time after implementing pagination via the will_paginate gem. The product index paginates products into groups of 12. Note the significantly reduced load time of close to 1 second, and the manageable 22 requests for the page. Pagination improved page load times by requiring smaller files to be rendered by the Rails application.

4. Front End

Several other optimizations were introduced to improve page load times. Static image assets were crunched to reduce file size: before, combined static image asset size per usual request was 2.7KB, and afterwards was 1KB. External, third-party CSS files such as normalize.css were minified, saving a total of 1.7KB per request. Originally, the Roboto font was used for body text; however, Helvetica is similar enough and natively supported by browsers, so the stylesheet reference to Roboto was removed (doing so saved 1.5KB per request). Eventually, all possible references to Javascript files were modified to be served from CDNs in order to reduce load on Arpeggio servers, and take advantage of browsers' ability to make multiple parallel requests from different domains.

Finally, we used ImageMagick in conjunction with Paperclip to shrink uploaded images to a normal size (of width 500 pixels, JPG quality 60%) and thumbnail size (of width 250 pixels, JPG quality 30%). The "normal" sized asset is served on Product show pages, while "thumbnails" are shown on the Product index page. Average uploaded image size before this change was 131.8KB, and thus combined average image bandwidth per request to the index page was 131.8KB x 12 = 1.5MB. After the shift to ImageMagick, average uploaded image size is 27.7KB, and thus combined average image bandwidth per request to the index page is now 27.7KB x 12 = 321KB, roughly 4 to 5 times smaller.

5. Server-side Caching

In order to implement server-side-caching, the first thing to determine is which data is cacheable. In other words, which data is worth caching in memory for at least one (usually both) of the following reasons: it is very frequently requested by all or many users, and it is static enough that it can remain valid in the cache for some considerable period of time.

In analyzing Arpeggio, we determined that individual product data was worth caching because the same products are capable of being universally viewed by all users, and they change only when they were actually rented out or updated by their owners. In our minds, this makes them suitable for server-side-caching so that the server can quickly pull the frequently viewed product information from cache without having to run queries in the database.

To implement the server-side-caching, we employ fragment caching both regularly and in the Russian Doll style. We utilize basic fragment caching of products on user account pages as they are shown to all users viewing an account, whether it is their own account or another user's account. We utilize the Russian Doll fragment caching to cache individual products on the products index page, and wrap them into another cached object which stores each entire products index page (pagination).

The advantage of using the Russian Doll caching scheme is that the nested caching allows individual product data to become stale while still preserving the usability of the overall cached page. This maximizes the number of cache hits for a set of dynamic data such as the products on a page. Both caching schemes are implemented using auto-expiring cache keys so that cached data does not have to be manually invalidated whenever it becomes stale, instead, the keys are created using data corresponding to the state of the object (such as the "update" timestamp) and are evicted using an LRU scheme whenever the object is updated.

In addition to implementing conventional caching techniques as part of server-side optimizations we also serve all static pages (primarily forms and the home page) as static assets using the ActionPack-Page_Caching gem. This means that static pages like the home page and forms are served directly through the asset pipeline by the http server (in this case Nginx), effectively getting rid of the time it takes Nginx to tell Rails to serve up and render the page. This significantly reduces page load times on static pages, as will be discussed in the Vertical Scaling and Horizontal Scaling sections.

6. Database Query Optimization

Database optimizations were one of the harder optimizations for us to make, as we had already written some complex queries and designed our interaction with the database with efficiency in mind. However, we were still able to make a fair amount of optimizations in our interaction with the database which lead to the greatest boost in our application's performance as will be detailed in the results section.

All of the optimizations are implemented by performing eager loading of data that is known to be needed in the future so that it is fetched along with whatever data is explicitly asked for in the query. This is done using the `includes` method which lets the database know that certain relationships will be accessed using the current data soon. As a result, the database knows that it should pre-fetch this related data so as to prevent more queries being run in the future. Potentially many queries are avoided by simply performing this pre-fetching.

Much of this preloading technique is employed throughout Arpeggio wherever objects are being queried for, and later, objects related to them are being queried for. This happens particularly often when carts and orders are created and viewed because they are at the core of Arpeggio's model relations, having a relation to practically every other model. Thus, aggressive prefetching of products, orders, line items, and other objects is performed. The results of this optimization are prominent and will be discussed in detail in the Vertical and Horizontal Scaling sections below.

7. Client-side Caching

In implementing client-side-caching, many of the same guidelines as for server-side-caching are followed in order to ensure an effective caching scheme. However, since client-side-caching is specific to users, we are able to apply this type of caching much more ubiquitously as we no longer have to worry about data being too specific to a user and using up valuable cache space.

As such, we determined that all show pages (user, product, cart, order, etc.) could be cached by the user's web browser since most of the data for individual items would not be changing very frequently, if at all, in the near future. This caching scheme is implemented using the `stale?` and `fresh_when` methods which set HTTP caching headers in order to let web browsers know that a page or parts of it are cacheable. Thus, the web browser just asks the server if the data has changed the next time the same page is visited again. If the data has not changed, the web browser can simply display the same data it has cached. Otherwise, the regular request-response procedure is carried out.

The results of client-side-caching are not well evaluated by the lender and renter Tsung load tests as the simulated users do not typically visit the same pages again as part of the critical path. As a result, we use the browser (read-only) load test to evaluate this aspect of the optimizations and also showcase the results of these optimizations through improvement in individual page load times. These results are discussed later in the scaling sections.

8. Memcached

This branch of optimizations utilizes a memcached configuration in which a remote datastore is used as the key-value store implemented by memcached. This has the effect of relieving some of the cache and memory pressure locally on the application server since caching specific to the application can be performed on the remote data store. Server-side-caching will be used with and without a memcached configuration. Client-side-caching will be utilized strictly with a memcached configuration. We expect results to illustrate cache relief through an overall increase in performance.

B. External Bottlenecks

The performance of Arpeggio may be dictated by the availability and performance of hardware and external APIs. Large amounts of memory are vital to the application when many product image uploads are being processed. However, because memory must act as an intermediary before final storage on S3, the usage of memory by the application hogs space from other uses, such as caching database query results. In terms of infrastructure, the application currently relies heavily on Braintree for monetary transactions and AWS for EC2 and S3 services. Rate limiting on Braintree API calls may affect performance negatively; however, reaching the rate limit for Braintree is highly unlikely but could occur if Arpeggio generates enough users and transactions in a given period of time. Outage of service for Braintree can further hinder application functionality, as can outages for EC2 and S3.

C. Load Testing

We wrote three test scripts to evaluate the performance of our application from the aspect of renters, lenders, and browsers. Typical renter requests include creating an account, browsing the product index, viewing specific products, and going through the checkout procedure; these actions are simulated by the read-test script. Typical lender requests include creating an account, browsing products, creating new products with image upload, and checking out products of other merchants; these actions are simulated by the read-write-test. Finally, typical browser requests include requesting the products index page and many product show pages; these actions are simulated by the read-only script. Each script has several phases, in which the arrival rate of users doubles with respect to the previous stage. The performance of Arpeggio is measured by these three tests in order to influence and evaluate decisions regarding scaling.

D. Vertical Scaling

The first, and simpler, scaling strategy to implement is that of vertical scaling. This is because it does not involve any significant change of the codebase or deployment workflow to accommodate for running on an application server cluster (among other configuration changes). It is simply about adding computational power, memory, and other resources so that more computation can be performed on a single machine and more requests can be handled. Initially, it seems like the cheaper and more viable option to the naive engineer. Nonetheless, this type of scaling is often justified when an application does not have that much demand, and the need for a highly scalable cluster of application servers is not present. This section will discuss the results of the lender test in order to showcase the effects of vertical scaling performed on Arpeggio.

1. t1.micro

In order to illustrate the results of vertical scaling we ran the two major load tests (renter and lender) on a t1.micro instance, then an m3.medium instance, and finally an m3.large instance. Each load test was run on several phases until the point when the application could no longer handle the next phase of load. This was done to show how an increase in computational power affected the ability of the application to take on greater load. We will analyze the results of the lender test to understand these trends because it is the most computationally intensive.

The load test running on the t1.micro instance did not get very far as it began producing the 5xx series of server errors in phase 1, which generates 1 user per second for 30 seconds, and recorded outrageous page response times of over 30 seconds. It is clear that a t1.micro instance will degrade in performance very quickly. This test was simply conducted as a base point to compare with better deployment configurations. Some of the results of running the lender load test are shown below in Figure 4: The response times are extremely high and fluctuate a lot, indicating overload of the server and inability to server requests early.

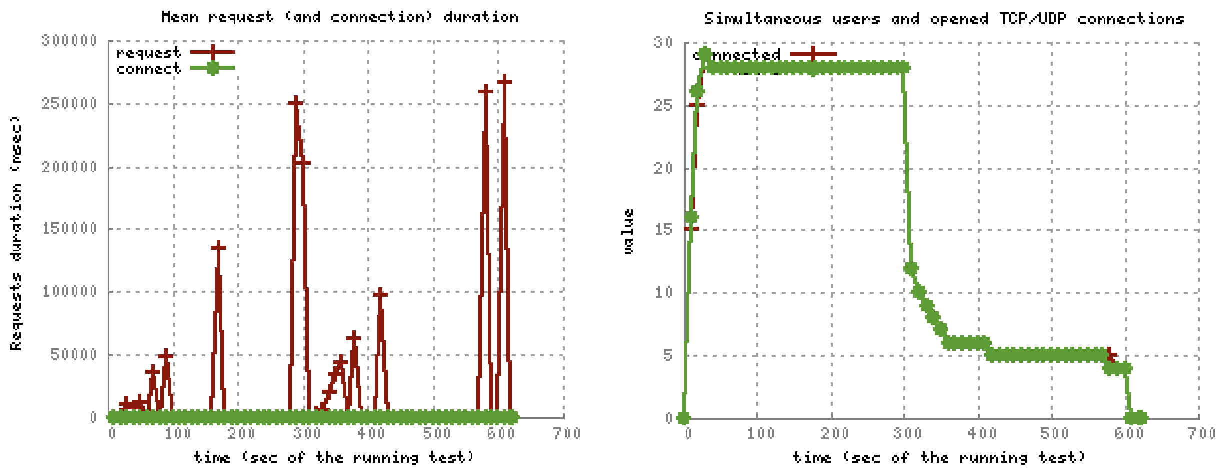


Figure 4: Graphs of the response time and number of simultaneous users over the course of the lender load test on the t1.micro instance.

2. m3.medium

Next, we upgraded our application to an m3.medium instance which boasts a more powerful CPU (still a single core) and more than three times as much memory (3.75 GB as opposed to 0.613 GB). This increase in resources, particularly the increase in memory, helps the application server handle more requests as it can store more data in memory, reducing writes to disk. Testing conditions for this instance remain the same.

This time, the application was able to handle more load from the lender test as it was able to serve nearly 50 concurrent users and complete phase 2 before becoming overloaded and unable to serve subsequent requests in phase 3. The results are promising if there is no intention of the application scaling beyond the 50 concurrent users it is able to support. However, if anything past this figure is expected, this solution will fail to perform just as the micro instance did. The results of the lender test on the m3.medium instance are shown below in Figure 5. More compute capability and memory are needed to increase performance and scale further.

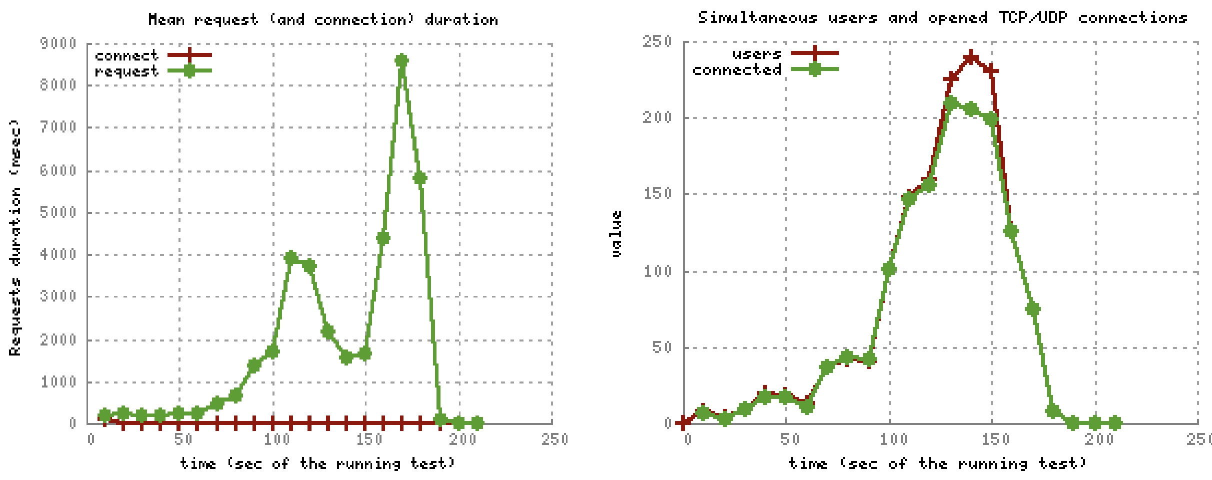


Figure 5: Graphs of the response time and number of simultaneous users over the course of the lender load test on the m3.medium instance.

3. <u>m3.large</u>

As our final test of vertical scalability, we fired up our application on an m3.large instance. Armed with 2 CPU cores and twice as much memory (7.5 GB vs. 3.75 GB) as the m3.medium, this transition effectively doubles the resources available to the application. In (naive) theory, this should translate to a doubling in performance. Once again, the testing conditions remain the same.

As expected, the application was able to handle significantly more load from the lender test and was able to serve about 75 concurrent users, finishing phase 3 comfortably before getting overloaded in phase 4. This is a good performance increase; however, it does not correspond to the doubling in computational resources. The m3.large instance has 2 CPUs and twice the amount of memory of the m3.medium instance, but we did not achieve the logically corresponding 2x increase. This is largely due to the proportional improvement constraints described by Amdahl's Law and is one of the many pitfalls of vertical scaling that make it only viable as a solution in addition to horizontal scaling. The results of the load test on the m3.large instance are shown in Figure 6 below.

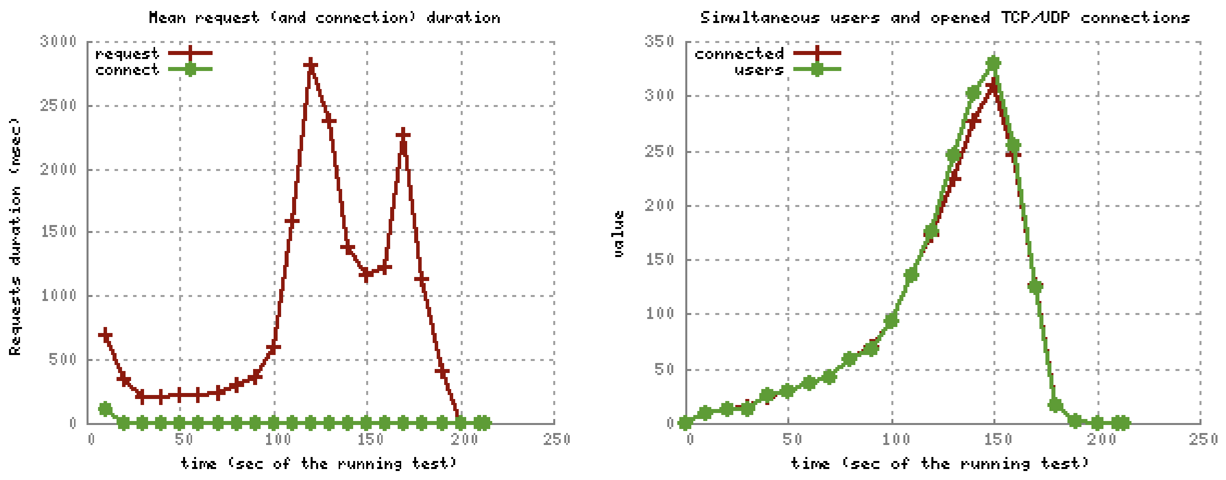


Figure 6: Graphs of the response time and number of simultaneous users over the course of the lender load test on the m3.large instance.

E. Horizontal Scaling

Since the effects of vertical scaling were limited fairly quickly, the next step in scaling was expanding horizontally. Horizontal scaling entails adding more machines as processing resources rather than increasing computing power and memory size. Unlike vertical scaling, it is easy to dynamically scale by adding machines when necessary and service is not solely reliant on a single machine.

As with most marketplace or online store websites, Arpeggio expects more “read” traffic than “write” traffic; that is, users will more likely be browsing products and managing their accounts than completing rental transactions or creating product listings and uploading images. As such, horizontal scaling was tested using a higher quantity of less powerful machines. The following tests were run on a load-balanced configuration with a single db.m3.medium instance connected to 8 m3.medium instances. Each test was executed using a 100,000 product dataset present in the database.

1. Browser

The results from running the browser test scripts provide interesting insight into the performance of the application if only GET requests are being issued from clients. This is most evident during 5-stage testing.

Table 1: 5 stage Browser Test

Branch	Mean Connection Time	Mean Page Time	Mean Request Time	Max Network Throughput (Received)	Max Concurrent Users
Master	4.05 ms	18.15 s	18.15 s	940.35 Kbps	541
Server-side Caching	3.96 ms	25.92 s	25.92 s	834.48 Kbps	622
Database Optimizations	3.96 ms	7.20 s	7.20 s	1.47 Mbps	440
Client-side Caching	5.24 ms	90.76 ms	90.76 ms	2.99 Mbps	124
Memcached	4.19 ms	17.71 s	17.71 s	1.04 Mbps	554

Client-side caching clearly outperforms any of the other configurations, keeping page load times in the millisecond range as opposed to several seconds. Although the mean connection time is slightly higher, client-side caching was able to provide more than twice the network throughput of any other configuration and keep concurrent users low. At high load, less concurrent users means the server is able to quickly handle requests and free sockets for other incoming requests. Since the cache is valid, files can immediately be served to client requests. On the other hand, high concurrent users at this load level means that clients are connected and actively waiting for pages to be rendered.

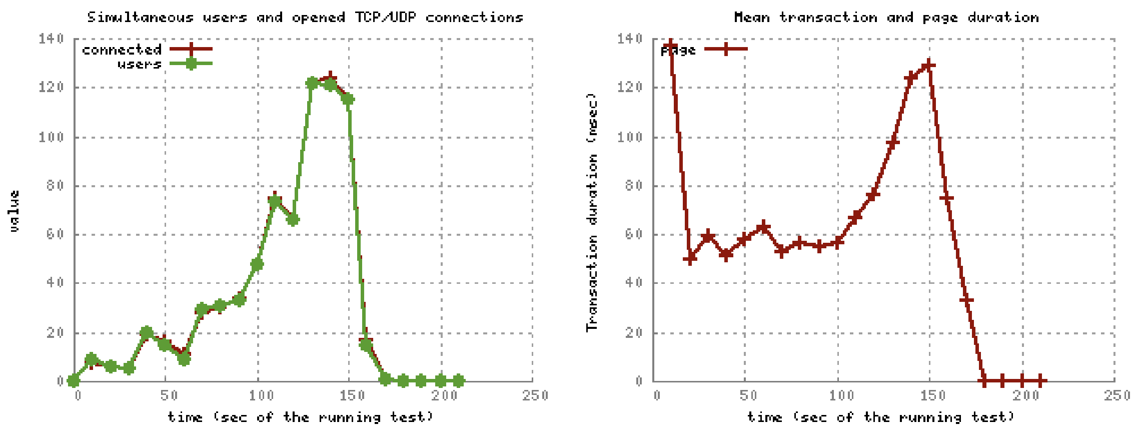


Figure 7: Simultaneous Users vs. Mean Transaction and Page Duration for Client-side Caching

Note in Figure 7 that the mean transaction time remains fairly constant until the 5th stage of user arrivals, when transaction time rises. The rise in transaction times indicates that this configuration can handle 16 users arriving per second, but that extended durations of equal or greater activity may overload the server and significantly increase response times. Compare the mean transaction and page duration graph from Figure 7 to Figure 8 below:

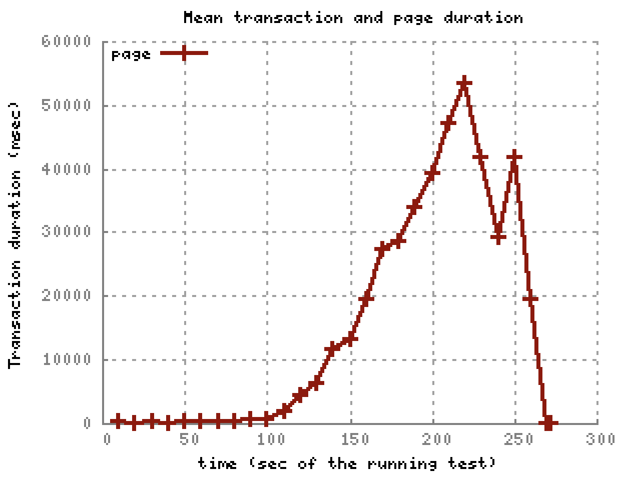


Figure 8: Mean Transaction and Page Duration for Master

Notice that the master configuration begins to falter at the same point as client-side caching, but is unable to respond to all requests quickly and thus displays an extreme increase in response time. The master configuration provides response times that are, on average, 200 times slower than the client-side caching configuration. The next fastest configuration, database optimizations, is 80 times slower than utilizing client-side caching. Clearly, caching is beneficial when cache blocks are rarely or never being invalidated by database writes.

2. Renter

The renter test provides insight into more realistic behavior of the application. As with any marketplace site, it is highly unusual for a service to only receive GET requests; new users will be joining the site and products will be rented from existing users. The 4-stage renter test provides a good example of application behavior when users are signing up and renting products.

Table 2: 4 stage Renter Test

Branch	Mean Connection Time	Mean Page Time	Mean Request Time	Max Network Throughput (Received)	Max Concurrent Users
Master	9.60 ms	2.27 s	2.27 s	1.08 Mbps	190
Server-side Caching	4.37 ms	4.20 s	4.20 s	909.77 Kbps	240
Database Optimizations	5.86 ms	4.42 s	4.42 s	840.16 Kbps	245
Client-side Caching	5.66 ms	3.35 s	3.35 s	1.02 Mbps	212
Memcached	9.29 ms	3.08 s	3.08 s	1018.30 Kbps	229

Notice that client-side caching is no longer the dominating configuration. In fact, the master branch appears to perform best under such conditions. This behavior may be explained by the introduction of product rental; upon finalizing a transaction, the status of the product is updated to false. Any update to the database will invalidate a cache block and degrade read performance. Another factor that may influence the performance is the number of concurrent connections accepted by the application server.

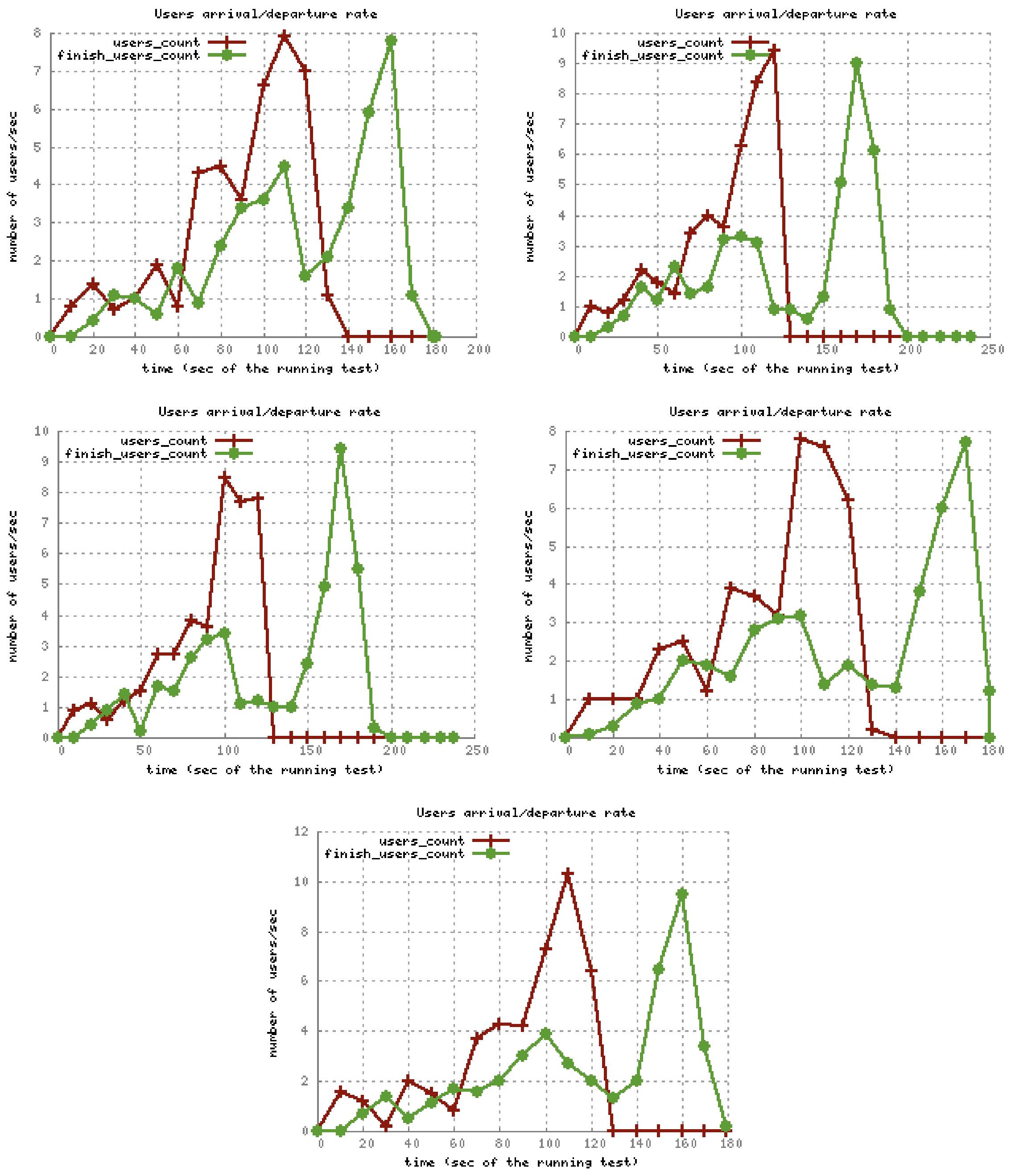


Figure 9: Comparison of User Arrival & Departure Rate for Configurations, in Table 2 order

Note that the results in Figure 9 were generated by running the same test script against all configurations. At stage 4, new users are supposed to arrive at a rate of 8 users per second; however, the arrival rate ranges anywhere from 8 to 10 users per second between the 5 tests. Although the discrepancy in load appears small, it skews the test results in favor of those configurations which received less load. Those with less load generated by Tsung have to handle less concurrent users, and thus the application servers can respond more efficiently to their current connections. Taking these conditions into consideration, it is important to observe that the optimized instances trade page load performance for more concurrent users, but the overall performance benefits are minimal due to cache invalidation.

3. Lender

The final test script examines the performance of Arpeggio under conditions where many users are creating products, uploading images, and renting products. Factors such as file upload, ImageMagick crunching, and transferring files from the Rails application to S3 storage must be considered when evaluating response times and throughput.

Table 3: 4 stage Lender Test

Branch	Mean Connection Time	Mean Page Time	Mean Request Time	Max Network Throughput (Received)	Max Concurrent Users
Master	5.05 ms	8.63 s	8.63 s	1020.63 Kbps	336
Server-side Caching	7.11 ms	11.89 s	11.89 s	864.04 Kbps	354
Database Optimizations	4.33 ms	11.27 s	11.27 s	819.10 Kbps	334
Client-side Caching	7.17 ms	12.29 s	12.29 s	919.66 Kbps	363
Memcached	6.90 ms	8.24 s	8.24 s	1.14 Mbps	334

One of the biggest issues with handling file upload is blocking IO. This slows down performance by requiring processes to wait until writes are complete. When handling dozens or hundreds of file uploads, the application servers are bottlenecked by the speed at which writes can be executed. Clearly, the configurations that received less load, such as master and memcached, performed better as they spent less time blocked on IO and more time responding to client requests. Because this test was write-heavy, the effect of caching was nullified by the constant invalidation of cache blocks.

4. Results

The results previously presented provide excellent insight into the optimizations Arpeggio requires to perform well. One overarching issue that degrades performance arises from cache invalidation. Under write-heavy load, in particular for product creation, writes invalidate cached data too quickly and introduce overhead for pre-fetching product data that is then immediately invalidated. Following such considerations, it is sensical that the master configuration outperforms configurations with caching; the master executes a database query only when the data is necessary, whereas a caching configuration may have to execute the same query twice to account for invalidated pre-fetched data. In an actual production environment, it would be wise to analyze the average traffic to the application servers and tweak the adjust the application code to optimize common activity and load patterns.

As with any project, it is interesting to see what upper bound of performance can be achieved with a set of given resources. Consequently, the renter and lender tests were run against a load balanced configuration of 8 m3.2xlarge application instances, 1 db.m3.2xlarge instance, and 1 m3.2xlarge memcached instance.

Table 4: 6-stage Test Results, Upper Bound Testing

	Renter	Lender
Mean Connection Time	1.94 ms	2.00 ms
Mean Page Time	0.52 s	1.21 s
Mean Request Time	0.52 s	1.21 s
Max Network Throughput (Received)	6.21 Mbps	7.16 Mbps
Max Concurrent Users	621	819

At stage 6, 32 users arrive per second to the application servers. The tests that ran with stage 7 included failed, responding with thousands of 5xx errors; thus, the comfortable threshold for new users arriving to our application lies between 32 and 64 users per second. It is clear from Table 4 that performance under large load is excellent, with response times close to 500 ms without file upload and 1.2 s with file upload. Throughput was high, with over 6 Mbps outgoing from servers. Interestingly, the lender test generated more concurrent users connected to the application servers; this is most likely due to the connections being held open during blocking IO from file upload. Although these results are strong, we expected the results to show more improvement using these powerful resources. While Arpeggio is able to comfortably handle several hundred concurrent connections, the application's scalability barely scratches the surface of handling millions of users that major sites such as Google and Facebook must deal with daily.

VI. Future Development

Arpeggio was built over the course of ten weeks; consequently, it is not perfect and lacks several features that are non-vital to the application but may certainly provide better user experience and performance benefits.

From a user experience perspective, some additional features that may be useful are implementing multiple images per product, email notifications, payment options, rental statistics, and collaborative filtering. Allowing lenders to upload multiple images per product provides renters with better information about the product they wish to rent without requiring additional communication between the two parties. Thus, page visits are reduced and users may be more inclined to pursue a transaction if all information is contained on one page. Email notifications may be useful in for user registration and confirmation, bookkeeping on transactions in case of discrepancies in payment, and keeping users up to date on the features of the application or policy changes. Adding support for payment methods such as Apple Pay, Venmo, or credit card transactions would simplify the rental process for many users.

Rental statistics include analysis tools for lenders as well as a feedback system for all users. Analytics can be used to inform lenders of their rental sales, most popular items, and comparisons against similar items in order to allow lenders to make adjustments to pricing. As with any peer-to-peer service, user ratings are important; feedback for both renters and lenders can better inform users on their decisions in renting or lending products, as well as provide an incentive for quality transactions. Collaborative filtering is a data mining and machine learning technique used for recommendation systems. By implementing a recommendation system based on user behavior and location, Arpeggio will be able to tailor recommendation of instruments and equipment to each user, thus maximizing the probability a renter will find the instrument they are looking for.

From a technical perspective, PostgreSQL was and still is being considered for the application database instead of MySQL. The original motivation for Postgres stemmed from a desire to use the PostGIS adapter for geospatial data since the geospatial adapters for MySQL were poorly maintained and incompatible with the newer versions of Rails. Postgres provides excellent features for handling JSON as well as offering geospatial support through PostGIS. The integration of Postgres with Arpeggio was successful; however, deploying to EC2 via CloudFormation templates was challenging, and modifying the templates to support Postgres proved to be an inefficient use of time. Thus, Postgres and geospatial database features in general were abandoned in favor of storing latitude and longitude in the products table and executing optimized SQL queries to filter products by location. It would be nice to use optimized geospatial database features in the future to further speed up the product index queries.

VII. Conclusion

Overall, this is an attempt to implement, understand, and develop reasoning behind many popular scaling decisions, and also to understand the detriments of incorrect decisions through trial and error. There are many aspects of this project and paper that may not seem very conducive to high performance scalable architectures. For example, utilizing Ruby on Rails to implement a scalable system is not necessarily the best technical design decision, but it allows teams to develop a reliable application quickly so that they can rapidly move towards optimization and scaling, much like we were able to do in the short span of ten weeks. Additionally, some of the results produced are not easily or comprehensively explained due to possible error. As mentioned in section V.E.2, Tsung would sometimes generate different load for the same tests and thus offered minorly skewed performance results between different configurations. Even so, our project introduced us to important design decisions taken and development techniques used when building and testing scalable application architectures.

Building scalable Internet services is no easy task. However, using web frameworks such as Ruby on Rails and hosting services such as Amazon AWS make the task much more manageable and allow the developers to focus on application features and optimization. Although most of us were familiar with developing web applications and the optimizations that are necessary, learning Rails and more advanced features of AWS such as horizontal scaling configurations was exciting. As evident from our data, the optimizations we attempted to introduce on top of the original application provided minimal improvement in the common case. However, from our development process, we learned that query optimization, caching, lightweight HTML rendering (such as through pagination), and serving static assets from CDNs can greatly improve the performance of a web application. Most importantly, we learned that even optimal code can fail under load; more hardware is the inevitable solution to larger scaling problems.

When taking into account characteristics such as reliability and availability as part of the scaling problem, vertical scaling just does not provide the appropriate architecture or infrastructure to handle these issues. On the other hand, horizontal scaling naturally provides redundancy to increase an application’s reliability and availability. Although vertical scaling provides a more immediate solution to the scaling problem, this solution is short lived as the problem quickly creeps up again. For this, and the many other reasons discussed above, horizontal scaling is the right way to tackle large scaling problems and architect an infrastructure that is built for scale and is easier to expand when the need to improve performance and availability arises again.