WHISPR

Table of Contents

I. Introduction

Let's face it: healthcare stinks. The service designed to save lives is fraught with byzantine bureaucracy, frustrating inefficiencies, and a disturbing reliance on either paper, or insecure, third-rate software systems. Particularly for patients, the norm seems to be this: your healthcare history - including your prior surgeries, your medications, and family medical information - is scattered across a variety of healthcare providers, doctor's offices, and clinics across the places you've lived. If you keep your own records (which many Americans don't), it's either in a massive stack of paper in a filing cabinet, or an insecure spreadsheet sitting somewhere on your hard drive. In short, it's a pain to control your information. It's a pain to know who has it, a pain to give it out, and a pain to look it up to find out what it really contains.

Enter Whispr. We're a 100% online, 100% available, 100% easy solution to storing, sharing, and securing your healthcare information. Through our clean, modern web portal, you have the ability to coalesce all your information, all in one place. From there, it's easy to edit it as needed, anytime, anywhere. When you want to share it with a healthcare provider, it's as easy as searching their name, and selecting them. If you decide you don't want that provider to have access to your information anymore, it's just another button click to cut them off. From a healthcare provider's perspective, it's much easier to obtain up-to-date patient information. No more waiting for third-party hospitals and clinics. No more making patients fill out paint-by-number forms in the waiting room. Click a button, request their information, and bam: it's there to see.

The healthcare industry moves almost $8 trillion annually. It's about time it made it easy for patients and providers to share information safely and swiftly.

II. Overview

Features

In all, Whispr provides the following main features:

For Patients and Healthcare Providers

1. Sign-up and sign-in anywhere, anytime, with a website designed for desktop and mobile.
2. If Whispr's not right for you, delete your account easy peasy. No fuss, no emailing support, no waiting period - one click, and your information's wiped from our servers.

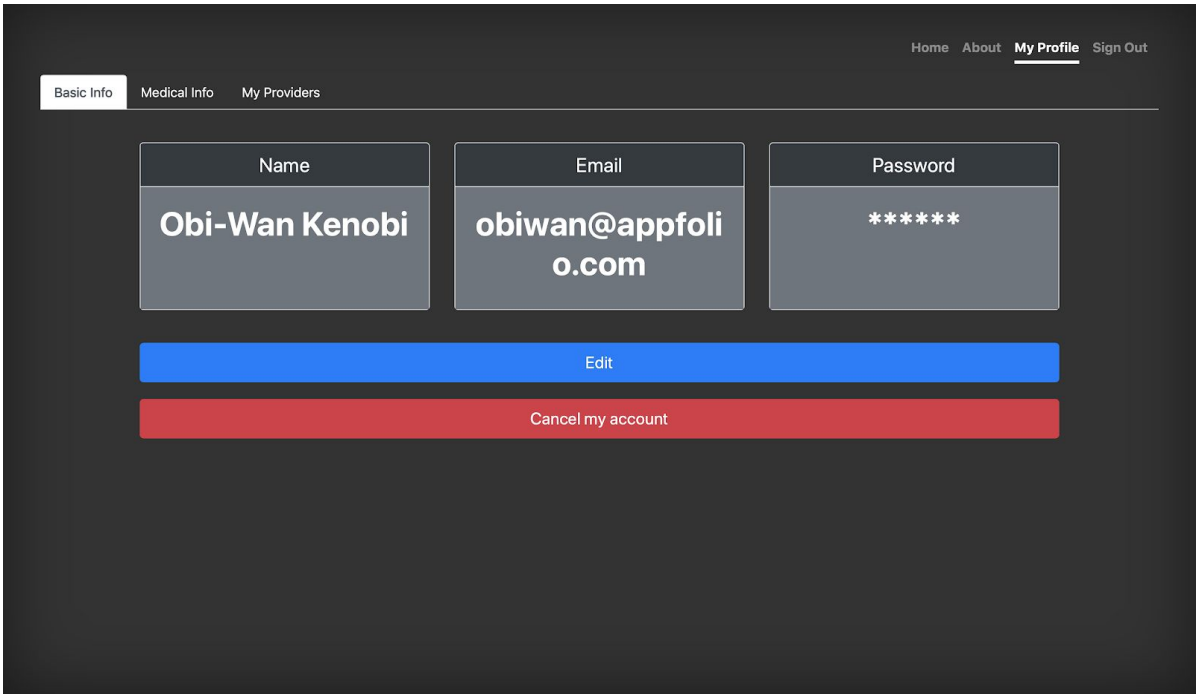


Figure II-A1: Whispr Profile Dashboard

For Patients Alone

1. Provide your initial information on sign-up, including name, medical history, and current medications.
2. Edit your information at any time with a simple button click.
3. Search for healthcare providers by name to find ones you're interested in.
4. Share your info with a provider with the press of a button.
5. Approve or decline provider's requests to see your information, as you see fit.
6. If you're done sharing information with a provider, cut them off with another button click.

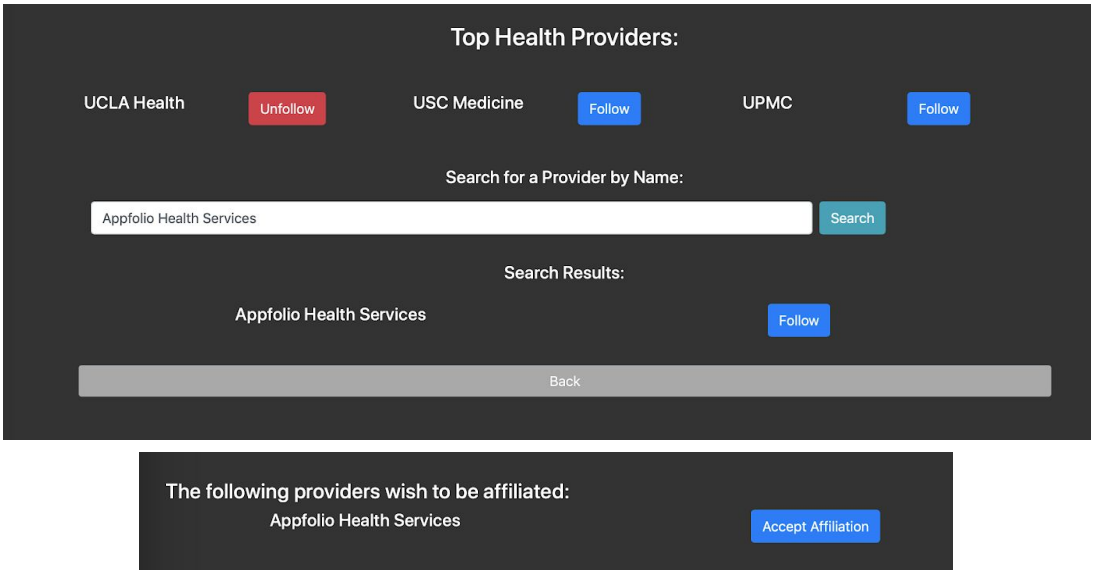


Figure II-A2: Patients Search for and Affiliate with Providers, and Can Accept or Ignore Requests

For Healthcare Providers Alone

1. Provide your corporate information on sign-up.
2. Search for patients by name to find ones you're interested in viewing the information of.
3. Request to see their information.
4. If approved, view that patient's individual information easily from the home dashboard.
5. Disassociate from a patient with a button click.

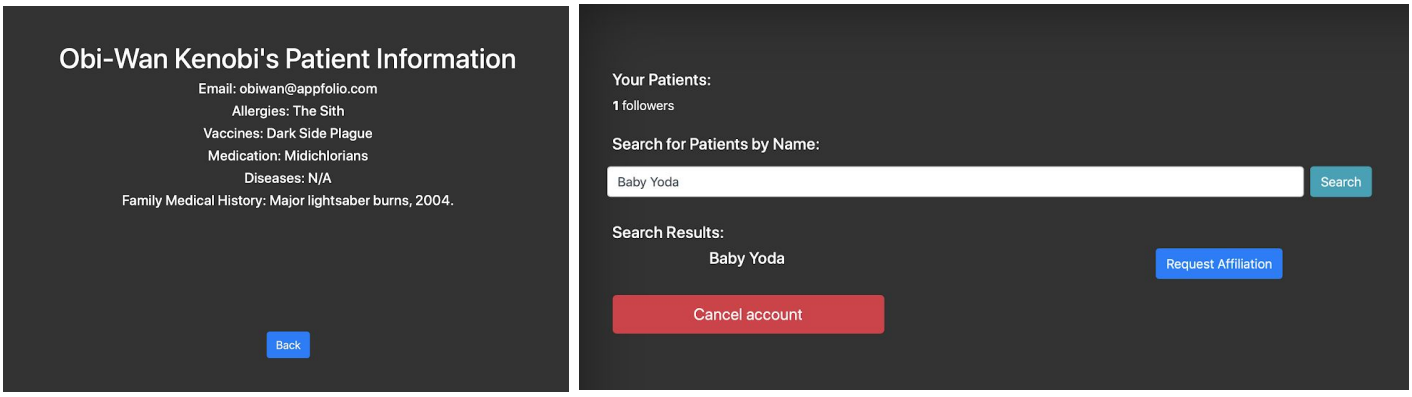


Figure II-A3: Providers Can Request Patient Info, and View Patients They're Affiliated With

Underlying Technology

Whispr was developed using Ruby on Rails 2.5, with a Postgresql 11.1 database, and a combination of Bootstrap UI and jQuery for the frontend. To host the app, we relied on AWS EC2 instances orchestrated by Elastic Beanstalk - effectively placing us under the umbrella of the widest-reaching, most-reliable cloud network in the world. In addition, to ensure patient information security, we used the widely-lauded Devise gem - middleware designed by experts to make user account management easy.

App Architecture

Since we used Ruby on Rails, we naturally developed Whispr along the Model-View-Controller (MVC) architecture. In short, MVC separates the concerns of web applications into three domains: the model, responsible for how data is stored, the view, responsible for how data is displayed, and the controller, responsible for accepting input and converting it to actions for the View and Model.

Specifically, we had three main controllers: an application controller to handle general tasks throughout Whispr, a users controller to manage the actions tied to both patients and providers, and a relationships controller to manage the relationships between providers and patients. We created a unique view for each page in the application, as well as several supporting, template views (to provide easy consistency to things such as button placement).

Further, our app is housed within an Amazon EC2 instance, which in turn communicates with a Relational Database Service (RDS). Users pass requests to the EC2 instance, which pulls data as needed from the RDS instance. Inside the EC2 instance, the MVC nature of the Ruby app handles the nitty-gritty: requests, data, and display. Finally, the results are passed back to the user. For a pictorial overview of this, see Figure II-C1.

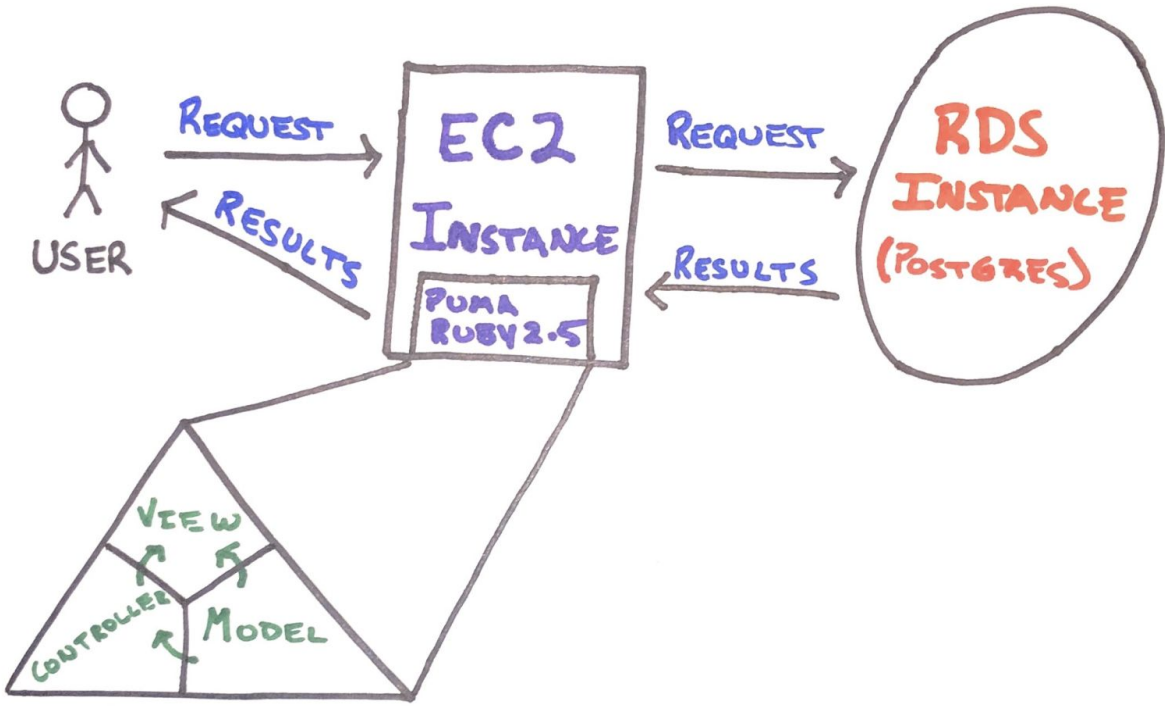


Figure II-C1: App Architecture Overview

Finally, we attempted to create the simplest database model possible. This was to aid in digestibility of the information, and to make scaling easier later on (e.g., fewer SQL optimizations

to worry about). From our perspective, we felt this was the best solution: it provided an easy to manage, simplistic design and it was easier to create relationships within one table than across multiple tables. Observe the two tables we used in Figure II-C2.

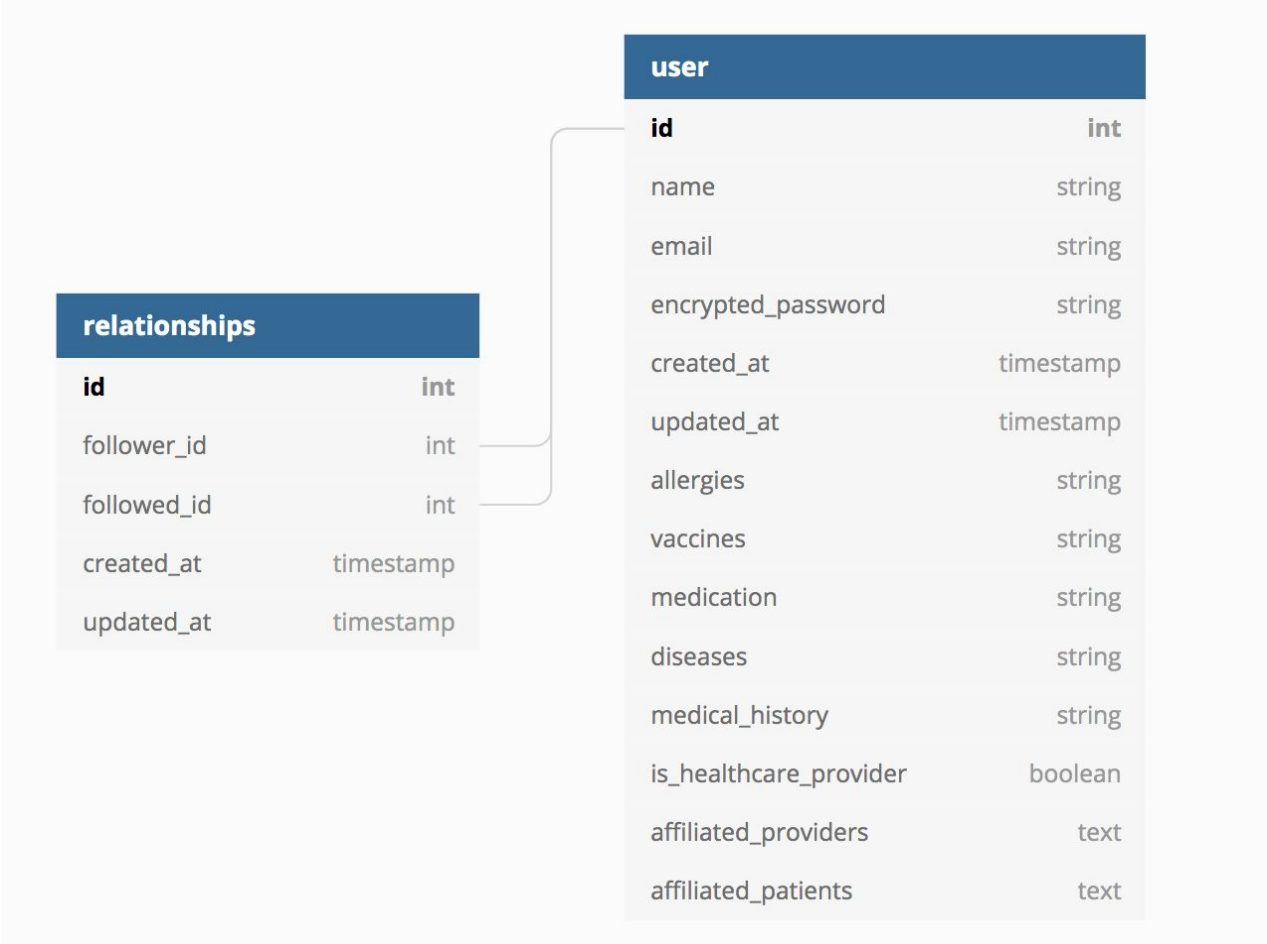


Figure II-C2: Whispr's Database Model

Ultimately, we were able to achieve all of our intended features with this model. Clearly, for anyone onboarding to the system, there isn't a large knowledge overhead: something we considered to be a huge plus.

III. Development Methodology

Initial Planning

To begin, we had an in-person brainstorming session. From here, we hatched various application ideas, ultimately settling on the idea of Whispr. We also took this time to sketch out the broad features we wanted to hit - including easily-accessible, easily-updatable medical information, the ability to share that information with providers, and providers' ability to request access to that information. This was largely the big picture meeting that helped guide our recurring planning goals, as discussed in the next subsection.

Recurring Planning

Whispr's development operated on an agile, weekly sprint schedule. Therefore, each week, we conducted a retrospective, followed by a planning meeting. The retrospective was a simple exercise in determining how well we had hit the previous week's goal, and moving unfinished tasks from the previous sprint into the new sprint. The planning phase involved accounting for unfinished tasks, and creating new tasks centered around a certain goal. These goals included things such as establishing a frontend skeleton, ensuring all aspects of backend capability, and establishing a baseline for scalability testing.

Once the tasks had been created (typically, 4-8 tasks, making for 1-2 per team member), we story pointed them - assigning values of difficulty based on the level of work a fully-caffeinated, fully-focused engineer could achieve in a single workday (without meetings, distractions, or other assignments). The actual values are rooted in the Fibonacci sequence. For example, a 0 indicates a task so easy it can be completed in under an hour. A 1 is meant to describe a day-long task, 2 is for a 2-day task, and 3 is for 3-day. In our development structure, anything we deemed to be worthy of a 5 needed to be broken up - we didn't want individual tasks to incorporate that much work.

Once tasks had been story pointed, we collectively assigned them to one another. We then held a question session: allowing each person a set of time to voice questions or ask for input on their tasks. Once that was completed, planning was completed.

Status Updates

Each week, we presented the progress of our application to either Andrew Mutz, or John Rothfels. This included a summary of the deliverables we had accomplished in that week, a projection for our future plans, and a demo of what we had accomplished. We took their feedback as offered, and incorporated that into our recurring planning sessions.

Software Development

We approached software development from a very traditional perspective. Once a person had been assigned their tasks, it was their job to manage them, and finish them, on their own. A few of our team members had had negative experiences with pair programming in the past, and we ultimately agreed to not conduct anything of the sort. Thus, software development was a one-on-one affair between the engineer and their laptop.

We also practiced test-driven development, utilizing some form of testing at most stages of production. Firstly, there are some basic tests to ensure the app's general infrastructure works as expected. Travis CI runs these tests on each pull request, to ensure it is not breaking something fundamental. Further, for any new pull request, we mandated not only code review, but user testing. Essentially, we developed a step-by-step set of user paths to follow. Before approving any pull request, the reviewer had to walk through these steps using the changes, ensuring no regression had been made. Though a somewhat primitive, manual form of testing, we found it to be quite useful.

Load Test Development

To test the scalability of our application, we used an evaluation tool called Tsung. We deployed a version of our application onto a set of AWS EC2 instances, sent various requests to the deployed instance, and analyzed the response. The biggest wins Tsung gave us were the abilities to observe response times and HTTP code rates, and deduce whether a given user was capable of utilizing the features that we implemented. We wanted a mixture of reads and writes to see how different `POST` and `GET` requests induced load on our app. Importantly, we arranged it so that not all of our synthetic users would have the same habits, introducing a level of randomness to the user action.

Additionally, we made sure to adapt the script as we went, responding to the results we were seeing. For instance, when testing vertical and horizontal scaling, we realized that our ordinary, 3-phase load test (detailed in Section IV) would be inadequate to thoroughly examine a broad range of EC2 instances. Therefore, we added up to 7 phases, increasing the intensity of the test to ensure we were able to accurately see how the various systems performed under load.

For nitty-gritty: we used Amazon CloudFormation (a service to manage other AWS services) to deploy instances containing Tsung. The Tsung test was initialized to hit the server endpoint at TCP Port 80. The global timeout for the TCP acknowledgement was set to 2 seconds. We made sure that every GET and POST request was HTTP URI encoded to properly send requests across the network. For further Tsung implementation details, please examine Section IV.

DevOps Technology

To assist in our development, we used several key pieces of technology: git for version control, VS Code for development, the Elastic Beanstalk CLI for deployment, and custom-made bash scripts. These bash scripts were used to ease tasks such as running the app, or deploying very specific types of EC2 instances. Initially, we placed a high priority on DevOps tooling, believing it would ease some pain further along down the line. However, unfortunately, it quickly became clear that the compressed 10-week timeline just did not allow that kind of time investment into tasks that were solely developer-focused. Thus, our plans to utilize things such as Docker and Kubernetes were scrapped.

IV. Performance Analysis Methodology

Critical User Paths

Our main load testing script, load_tests.xml (under the directory whispr/load_tests) performs the following critical user actions:

1. `HTTPS GET`: Visit the Home page
2. `HTTPS GET`: Visit the Sign Up page
3. `HTTPS POST`: Sign up as a healthcare patient
4. `HTTPS GET`: Visit the Medical Profile page
5. `HTTPS POST`: Fill out the Medical Profile page with random 15-character strings
6. `HTTPS GET`: Visit the user's (Healthcare) Providers page
7. `HTTPS GET`: Search for one of the 10,000 pre-initialized healthcare providers
8. `HTTPS POST`: Randomly follow (affiliate with) one of the healthcare providers
9. `HTTPS GET`: Visit the user's Profile
10. `HTTPS GET`: Visit the user's "My Providers" page (to see the affiliation made in #7 above)
11. `HTTPS POST`: Delete the user's account

Between each action, the user waits either up to 2 seconds (before simple actions, such as HTTP GET requests) or up to 5 seconds (before HTTP POST requests where the user needs to fill out forms).

Database State for Load Tests

Before launching each load test, we initialize the database to a certain state to mimic the real world, where the database would typically not be empty. Specifically, we used a SQL script (load_tests/database-setup/init-database.sql) to create 10,000 healthcare providers and 5,000 patients. Each of the 5,000 patients would follow one of the 10,000 healthcare providers, thus filling up the *relationships* table in our database in addition to the *users* table.

After finishing each load test, we cleaned up the database by truncating the *users* and *relationships* table using another SQL script (load_tests/database-setup/clear-database.sql). By doing this, we made sure that each load test was independent from the others, and was testing against the same database state each time.

Initial Baseline Load Testing

```
<arrivalphase phase="1" duration="30" unit="second">
  <users arrivalrate="1" unit="second"></users>
</arrivalphase>
<arrivalphase phase="2" duration="30" unit="second">
  <users arrivalrate="2" unit="second"></users>
</arrivalphase>
<arrivalphase phase="3" duration="30" unit="second">
  <users arrivalrate="4" unit="second"></users>
</arrivalphase>
```

Figure IV-A1: Load Test Arrival Phases

```
<request>
  <dyn_variable name="authenticity_token"></dyn_variable>
  <http url='/users/sign_up' version='1.1' method='GET'>
    <http_header name="Cookie" value="%%_cookie%%" />
  </http>
</request>

<setdynvars sourcetype="eval"
  code="fun({Pid,DynVars})->
  {ok,Val}=ts_dynvars:lookup(authenticity_token,DynVars),
  http_uri:encode(Val) end.">
  <var name="encoded_authenticity_token" />
</setdynvars>
```

Figure IV-A2: Getting an Authentication Token

We used Tsung to perform all of the load testing for our web application. We created a configuration file called load_tests.xml, which has 3 user arrival phases where the number of users that arrives per second is 1, then 2, then 4 (Figure IV-A1). Each arrival phase lasts 30 seconds. Each user performs a sequence of actions (see the section “Critical User Paths” above), consisting of both HTTP GET and POST requests to interact with the web app. For each POST request made, we needed to obtain the authentication token to have valid credentials for the transaction (Figure IV-A2). As mentioned above, the authentication token was also HTTP URI encoded to be in the correct URI format when sent across the network.

```
<!-- Assuming User Follows a Random Provider (ID)-->
<setdynvars sourcetype="random_number" start="10000" end="20000">
  <var name="random_provider_id" />
</setdynvars>

<!-- Wait for up to 2 seconds, user is searching for provider -->
<thinktime value="2" random="true"></thinktime>

<request subst="true">
  <http url='/providers?search=p%%_random_provider_id%%&amp;commit=Search'
</request>
```

Figure IV-A3: Simulate Database Search

```
<request subst="true">
  <http
    url='/users'
    version='1.1'
    method='POST'
    contents='_method=delete&amp;authenticity_token=
    content_type='application/x-www-form-urlencoded'>
  </http>
</request>
```

Figure IV-A4: User Account Deletion

Certain POST requests were set to simulate button presses and database queries. To test for database optimizations, the load test for multiple users to search for providers was added (Figure IV-A3). Finally, after all the critical paths were hit, we made sure to delete all the users created (Figure IV-A4) so as to have consistent state for successive tests.

To make sure that we had the correct parameters for the GET and POST requests, we used the Inspect Element feature in Google Chrome to determine how to generate the authentication token from the Cookies.

After we executed the script against a production Elastic Beanstalk instance (EC2 and RDS having both **M3.Medium** configuration), we plotted the Tsung-produced data for Mean Transaction Duration and HTTP Code Response Rates. This provided us with our benchmark, baseline performance. The graphs are shown below in Figure IV-A5.

Note: We wanted to have more arrival phases in the initial performance analysis, but a bug with Tsung caused the data points from the HTTP response rates graph to be truncated as soon as the server experienced significant load in the fourth and fifth arrival phases; therefore, we limited the number of phases to three, and this suffices for the purposes of a baseline performance analysis.

Figure IV-A5: Mean Transaction Times and HTTP Response Rates in Baseline Load Tests

As shown above, the web application seems to handle the load well until we are 50 seconds into the load testing script, which is just before the time that user arrival phase #3 (4 users/second) begins. At T=50 sec, the mean transaction duration spikes upward, and we also start to see the following HTTP responses: 404 (Not Found), 422 (Unprocessable Entity), and 500 (Internal Server Error). The goal is to implement optimizations that avoid or at least delay these breaking points in the server for the same amount of load.

V. Scaling Methods

Vertical Scaling

Overview

Vertical scaling is the usage of “bigger” and better hardware to achieve better performance for your application. For instance, utilizing a machine with a beefier processor, or a greater amount of RAM. The idea is: the better your hardware, the higher the load it can take. So, to vertically scale, you simply find a better piece of hardware to run your application.

AWS-Specific Approach

From an implementation complexity standpoint, vertical scaling is rather simplistic - particularly with AWS. In essence, you simply need to specify the proper instance types when creating an environment within Elastic Beanstalk. Amazon provides a wide variety of instance types.

The M* series are the most well-balanced between RAM and CPU, while T* focuses on burstable CPU performance, and A* focuses on economical ARM-focused environments. Increasing the value of the number attached to a letter series indicates the same performance focus, but with higher-tier hardware overall. Thus, M3 should usually perform better than M2, and so on (though, of course, there are exceptions to this). To vertically scale, you could launch better and better versions of M3 instances, jump from M3 to M4, etcetera, depending on costs and needs.

Launching such instances can be done in many ways. For CLI junkies, the `eb create` command can be used along with the `--instance_type` and `--database.instance` flags[1]. Proper values for these flags can be found within the AWS CLI documentation[2].

Our Approach

To demonstrate the results vertical scaling could have for our application, we utilized six different instance types. These were: M3.Medium, M3.Large, M3.XLarge, M3.2XLarge, M4.Large, M4.XLarge, M4.2XLarge, M4.4XLarge and M4.10XLarge. We felt that this provided a great range of hardware, allowing us to truly see how great of an impact beefier instances could have.

As an aside, to keep things simpler, we ensured the EC2 instances’ types were the same as the associated RDS instances. For example, an Elastic Beanstalk Environment launched with the flag `--instance_type m3.medium` would also have the flag `--database.instance db.m3.medium`.

---

[1] "eb create - AWS Elastic Beanstalk - AWS Documentation." https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb3-create.html. Accessed 22 Nov. 2019.
[2] "describe-ec2-instance-limits — AWS CLI 1.16.282 Command Reference." https://docs.aws.amazon.com/cli/latest/reference/gamelift/describe-ec2-instance-limits.html. Accessed 22 Nov. 2019.

Keeping the types the same reduced overall testing variability. In all, the instance types we used, along with a brief rundown of their hardware specs and costs, are displayed in Table V-A1.

To perform our load tests, we expanded upon the baseline tests established in Section IV. Instead of just 3 phases, we utilized 7: each phase lasting 30 seconds, and doubling the number of users per second. Hence, phase 1 had 1 user arrive per second, phase 2 had 2, phase 3 had 4, and so on, up to 64 users a second. This was possible due to a discovered (yet time-consuming) work-around to the Tsung bug that truncated HTTP code results. Since this load test was of a more heavy-duty variety than the baseline used elsewhere in this report, we re-ran the test with the M3.Medium setup as well, to ensure our results had proper context.

Instance Type	vCPU Count	RAM (GiB)	Storage (SSD)	Price/Hour
M3.Medium	1	3.75	1 x 4 GB	$0.1113
M3.Large	2	7	1 x 32 GB	$0.225
M3.XLarge	4	15	2 x 40 GB	$0.450
M3.2XLarge	8	30	2 x 80 GB	$0.900
M4.Large	2	8	EBS-only	$0.100
M4.XLarge	4	16	EBS-only	$0.200
M4.2XLarge	8	32	EBS-only	$0.400
M4.4XLarge	16	64	EBS-only	$0.800
M4.10XLarge	40	160	EBS-only	$2.000

Table V-A1: AWS Instance Types Used for Vertical Scaling[3] [4] [5] [6]

Results

As expected, vertical scaling produced positive results, including lowering mean transaction time and HTTP 503 code responses, and increasing the number of HTTP 200 code response. For

[3] "AWS Update – New M3 Sizes & Features + Reduced EBS ...." 21 Jan. 2014, https://aws.amazon.com/blogs/aws/aws-update-new-m3-features-reduced-ebs-prices-reduced-s3-prices/. Accessed 22 Nov. 2019.
[4] "EC2 Instance Pricing – Amazon Web Services (AWS)." https://aws.amazon.com/ec2/pricing/on-demand/. Accessed 1 Dec. 2019.
[5] "The New M4 Instance Type (Bonus: Price Reduction on M3 ...." 11 Jun. 2015, https://aws.amazon.com/blogs/aws/the-new-m4-instance-type-bonus-price-reduction-on-m3-c4/. Accessed 1 Dec. 2019.
[6] "Previous Generation Instances - Amazon Web Services." https://aws.amazon.com/ec2/previous-generation/. Accessed 1 Dec. 2019.

reference: mean transaction time refers to the average time it took to complete an individual "request" to the website (such as loading the profile page), HTTP 200 codes refer to the *OK* status, and 503's refer to the *Service Unavailable* status.

Specifically, for the M3.Medium instance, you can see in *Figure V-A1* that mean transaction time reached an incredibly high peak of about 6,000 milliseconds. This is by far the worst performer, with a huge degradation in performance only 50 seconds into the load test (just as in our initial baseline load test from Section IV). All other EC2 instance types took at least 120 seconds (until phase 4) to reach performance-degrading points.

In general, we can see that M3 instances scale as expected - M3.Large is, overall, better than M3.Medium, peaking approximately 120 seconds into the test, at almost 2,000 milliseconds per request. Similarly, M3.XLarge is better than M3.Large, peaking about 160 seconds into the test, at about 1,500 milliseconds per request. Finally, M3.2XLarge peaks approximately 190 seconds in, at about 1,000 milliseconds per request.

For comparison's sake, this means that, in general, when bumping up each rung of the M3 ladder, you achieve about a 100% increase in cost, in exchange for an approximate 50% decrease in transaction time. However, these numbers still appear to be inadequate. Even the best of the M3 - M3.2XLarge - results in worst-case response times of approximately 1,000 milliseconds.

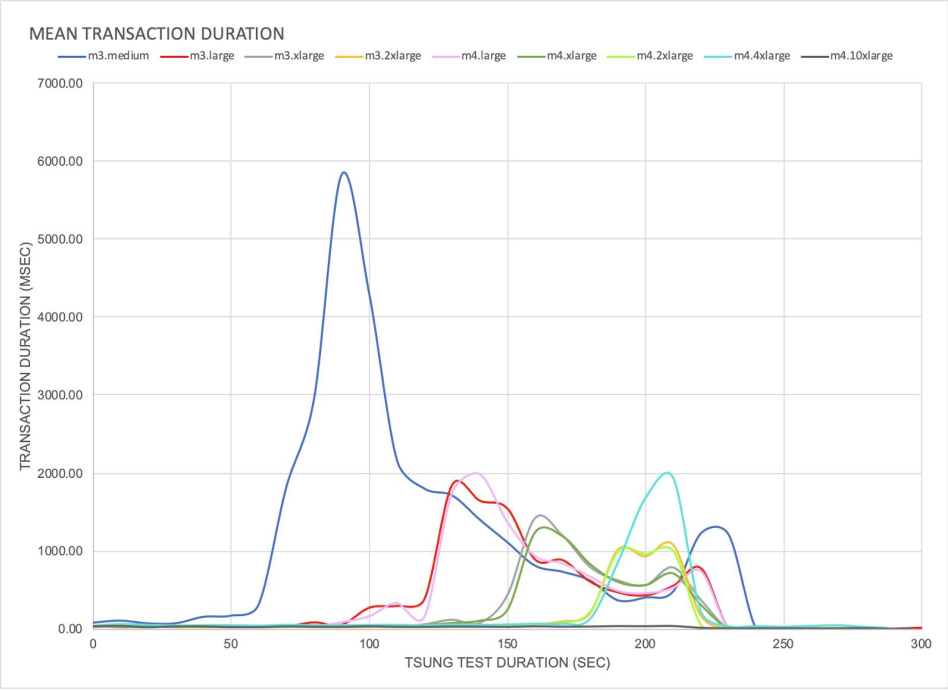


Figure V-A1: Mean Transaction Duration Over Time For all Instance Types

To that end, we now consider only the M4-series instances. For better viewing, take a look at Figure V-A2, which shows only the M4 transaction durations on a single graph.

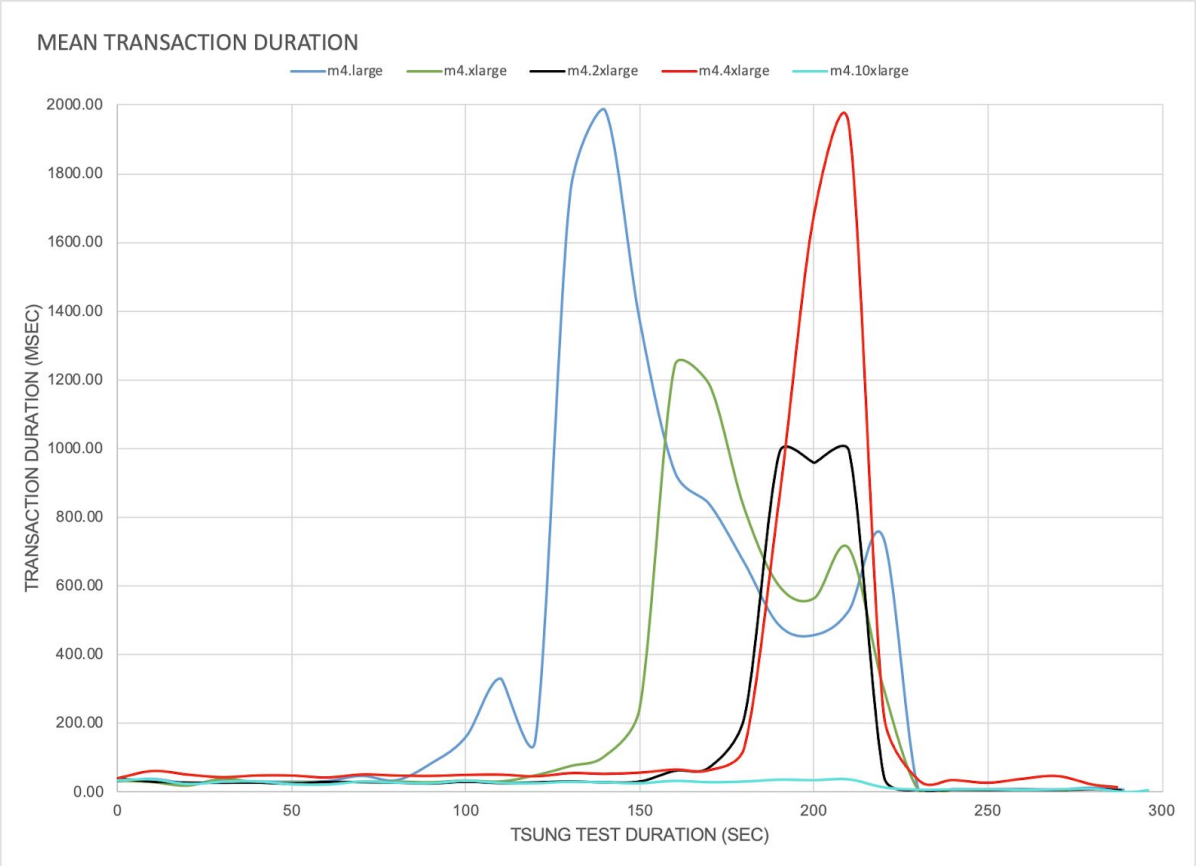


Figure V-A2: Mean Transaction Duration Over Time For M4 Instances Only

As expected, M4.Large, M4.XLarge and M4.2XLarge each provide incremental improvements over their predecessors - decreasing the time at which the spike in transaction time occurs (typically by about 30 seconds). Curiously, M4.4XLarge spikes at about the same time the M4.2XLarge did, with worse mean transaction time. On the face of it, this seems to indicate that M4.2XLarge vastly outperforms M4.4XLarge. However, Figure V-A3 reveals that this isn't really the case.

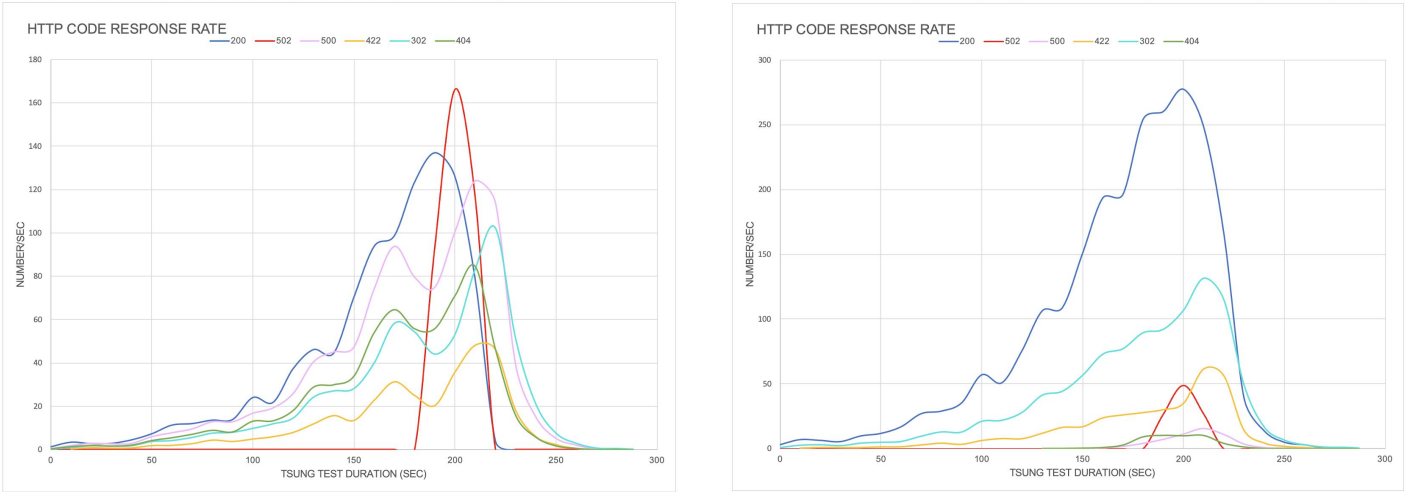


Figure V-A3: M4.2XLarge Response Code Rate (left) and M4.4XLarge Response Code Rate (right)

That shows that, though the mean transaction time for requests remains lower in an M4.2XLarge than an M4.4XLarge, that's more a byproduct of the fact that the system is throwing out 502 codes instead of meaningful results. Sure, an M4.2XLarge isn't completely stalling transactions like its smaller siblings, but it still isn't properly serving a large number of requests that come in - peaking at over 160 502 codes per second. Meanwhile, M4.4XLarge never surpasses 50 502 codes per second.

Still, the only true "winner" of a performer was M4.10XLarge. This beast of an instance never struggled with even the most intense portion of the load script, peaking at a measly 36.03 millisecond transaction duration. Obviously, this is a product of its insane 40-core, 160 gigabyte-of-RAM setup. Obviously, that setup comes at a hefty price tag - with $2.00 an hour being 122% more expensive than the second-most expensive M3.2XLarge.

Of course, we could craft a tweaked load script which would, eventually, break this system down, too. At this juncture, however, we think the point is clear: vertical scaling works, to a point. For a quick-and-dirty, in-a-bind solution to scaling, it works wonders. However, it has its limits. Ultimately, the cost to fielding a single machine capable of handling millions of users becomes incredibly prohibitive. Add cost to the danger of having a single point of failure for you entire service, and vertical scaling's niche starts to define itself: beef up hardware to a point, but seek alternative methods to achieve true scalability.

As a point of education, we've provided the graphs showing the HTTP code response rates for all instances below (aside from the M4.2XLarge and M4.4XLarge, as they are found in Figure V-A3). Note that they prove as we stated earlier: beefier instances provide better results, with more 200s and 300s, and fewer 502s and 404s.

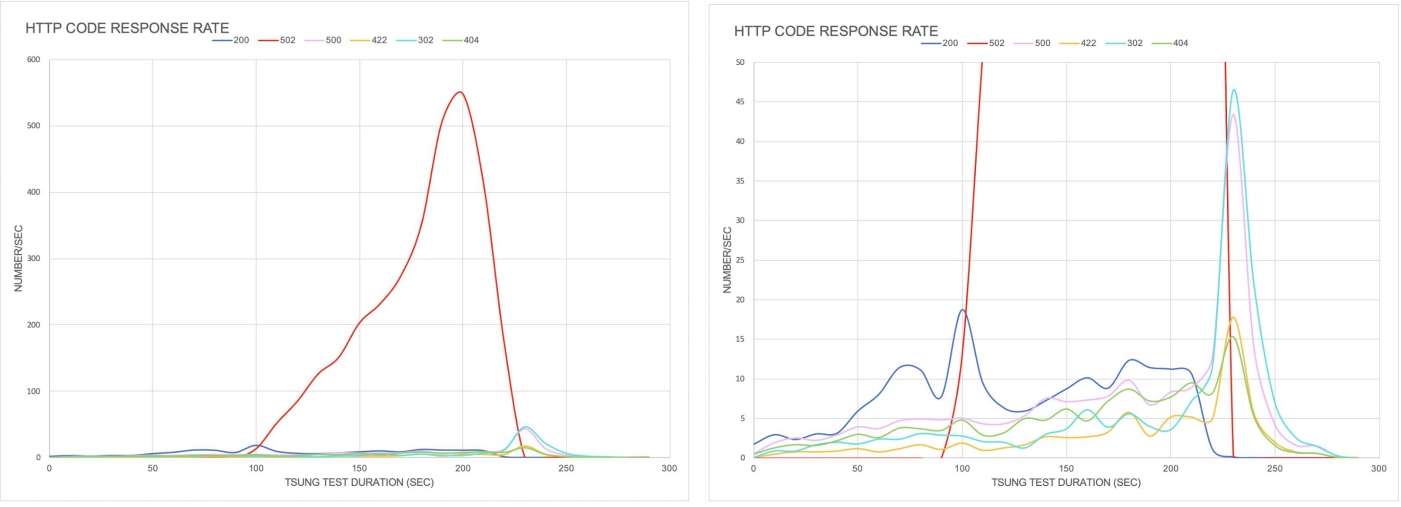


Figure V-A4: M3.Medium HTTP Response Code Rate; Zoomed-Out (left), Zoomed-In (right)

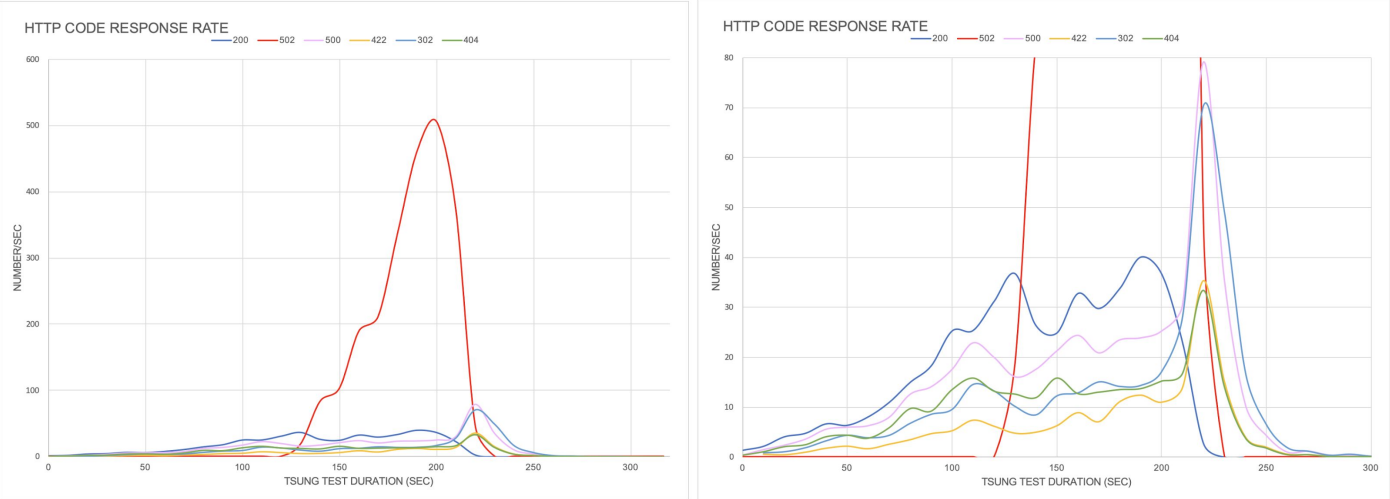


Figure V-A5: M3.Large HTTP Response Code Rate; Zoomed-Out (left), Zoomed-In (right)

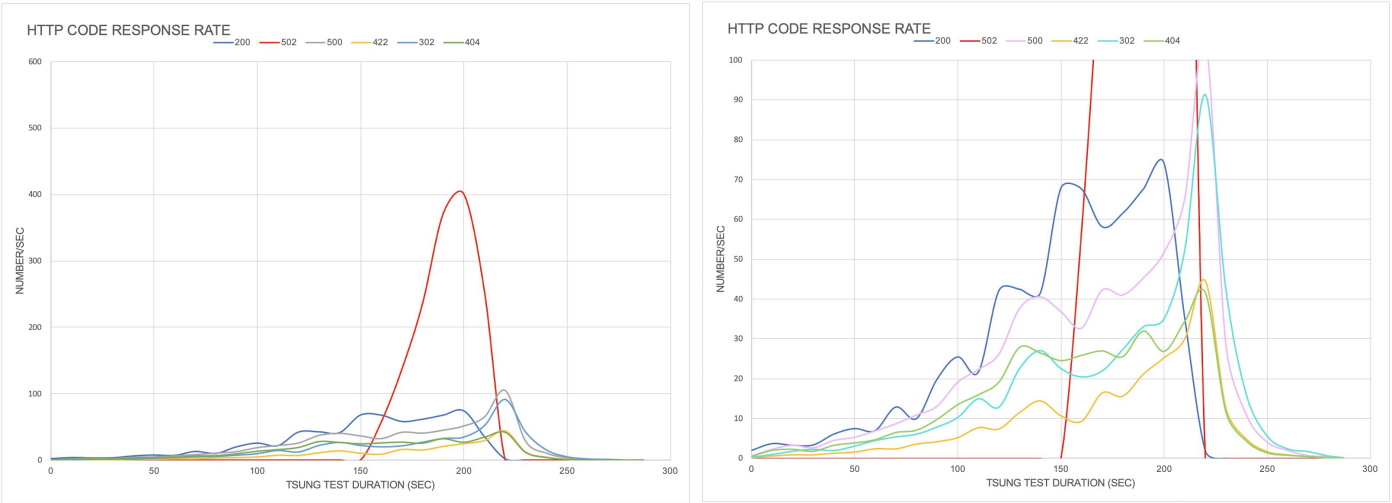
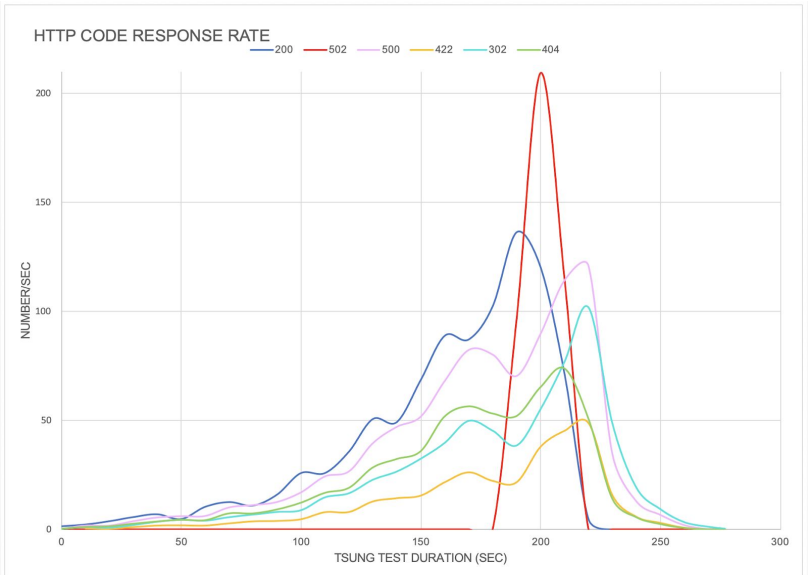


Figure V-A6 (above): M3.XLarge HTTP Response Code Rate; Zoomed-Out (left), Zoomed-In (right)

Figure V-A7 (below): M3.2XLarge Response Code Rate



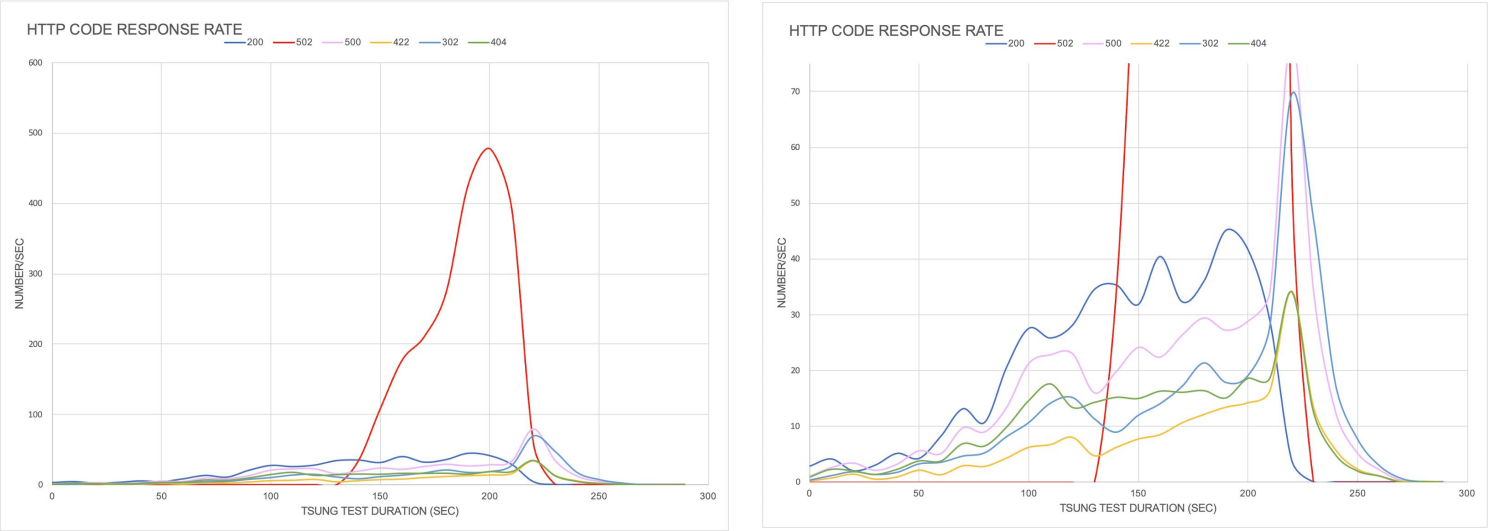


Figure V-A8: M4.Large Response Code Rates; Zoomed-Out (left), Zoomed-In (right)

Horizontal Scaling

Overview

Horizontal scaling is the usage of more hardware to achieve better performance for your application. For instance, coupling together three different machines, and having some external app (called a load balancer) route traffic to these machines based on their individual load. It's the same concept as splitting jobs up amongst people: the more people you have to divide work up, the easier it gets. Each person can only handle so much, but, together, they can handle much more. Similarly, each piece of hardware can only handle so much traffic - but together, they can handle more than they could alone.

AWS-Specific Approach

As with vertical scaling, AWS and Elastic Beanstalk make horizontal scaling trivial. The same instance types as described in vertical scaling are available. To make them horizontally scaled - meaning, to have multiple instantiated and arranged behind a load balancer - is as easy as using a flag on the Elastic Beanstalk CLI, or going through a step-by-step menu within the AWS web console. Specifically, for the CLI, the proper flag is `--scale {count}`, to be used instead of `--single`. AWS will take care of orchestrating the rest.

Our Approach

As with vertical scaling, we wanted to see the results for a good variety of hardware configurations. To choose these instances and their total count, we used an experimental approach. Essentially: start with 3 M3.Medium instances. If the app fails under load, double the number of instances. If it still fails, double it again. Once there is an arrangement with 9 of an

instance type, if the app is still failing, move back down to 3, but use the next, more powerful type of instance - going from M3.Medium, to M3.Large, for example. Repeat the entire procedure again until the app no longer fails. Once it no longer fails, if the current, non-failing arrangement uses more than 3 instances, see if 3 instances of the next, more-powerful type of instances could be equally successful.

We used this experimental approach because we felt it was a good way to determine a cost-effective, performant arrangement for horizontal scaling. After all, scalability always comes at a price. So, instead of jumping to utilizing the most performant instance type from the vertical scaling results, we started with our baseline M3.Medium, and proceeded upward as needed. This way, we could ensure we were only doing as much scaling as necessary for our particular application.

To that end, we utilized the following arrangements: 3 M3.Medium instances, 6 M3.Medium instances, 9 M3.Medium instances, 3 M3.Large instances, 6 M3.Large instances, and 3 M3.XLarge instances. We felt this not only accomplished our goal of experimental, step-by-step progression of horizontal scaling, but also showed a nice variety of hardware. We were able to observe both a wide range in the amount of EC2 instances put behind the load balancer, and the power of each individual EC2 instance.

Results

As expected, horizontal scaling was successful. Generally, increasing the number of instances behind the load balancer increased the app's performance, in terms of HTTP code rate. In addition, utilizing better instances typically worked as you'd expect: 3-pronged M3.Large instances would outperform 9-pronged M3.Medium, and so on.

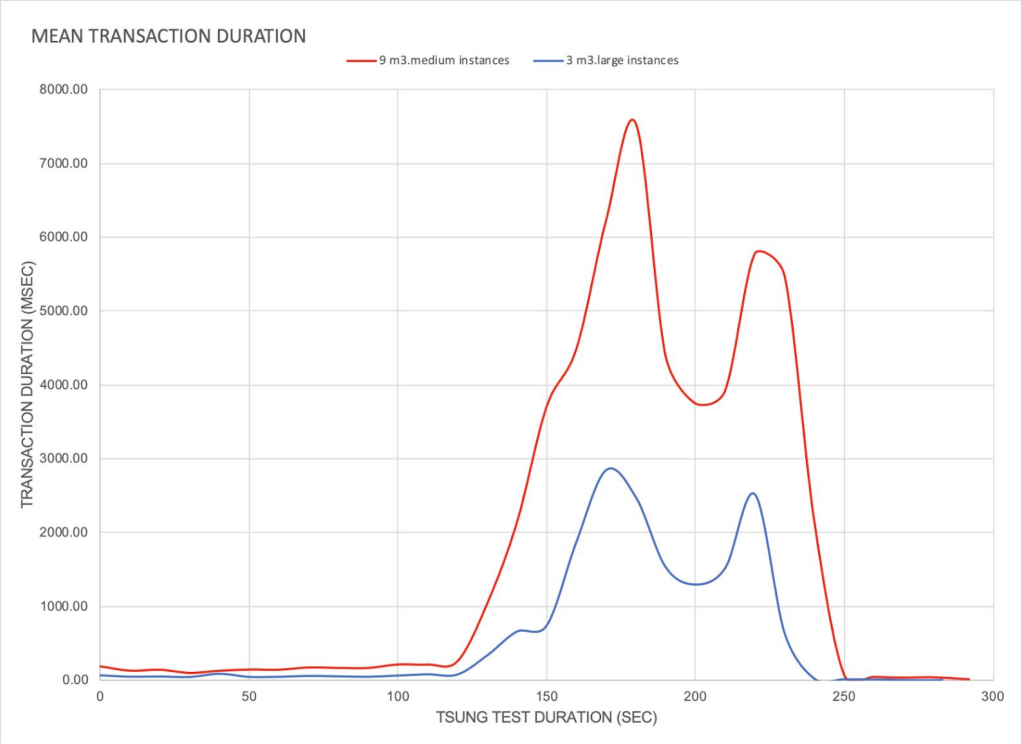


Figure V-B1: Average Response Time for a 9-pronged M3.Medium Instance and a 3-Pronged M3.Large Instance

Curiously, response times actually increased in their worst-case when comparing scaling within an M3 bracket. For example, look at Figure V-B2. Clearly, you can see that the 9-instance M3.Medium peaks with longer reaction times than the 6-instance version, which in turn has a higher peak than the 3-instance version.

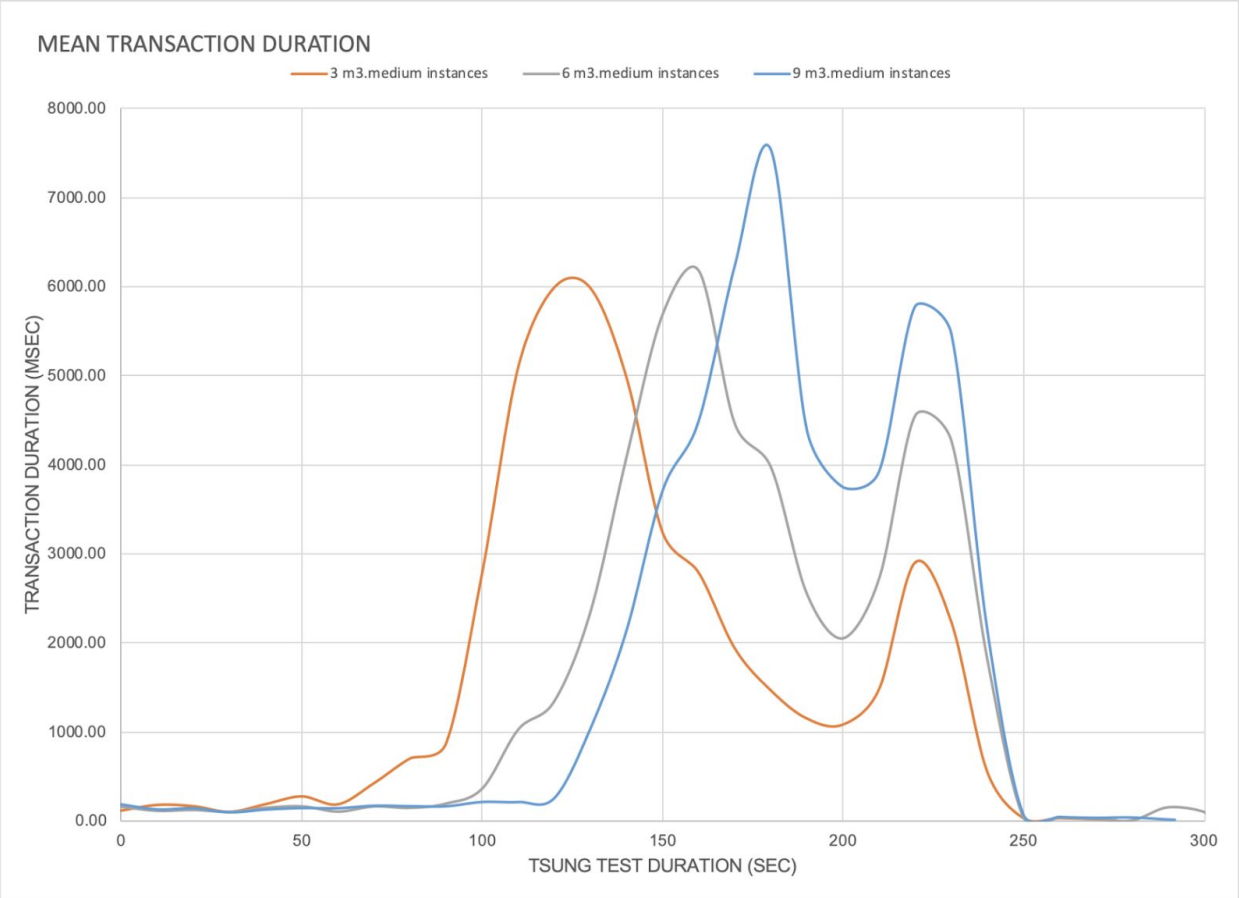


Figure V-B2: Mean Transaction Duration for all Variations of M3.Medium Instances

Though perhaps initially perplexing, these results make sense: each request must be routed through the load balancer first, then to the appropriate instance. Checks must be done to ensure the chosen instance is, in fact, the optimal one to route to based on current load. Obviously, as there are more instances, more of these status checks must be done, slowing the time it takes to route to the appropriate instance, therefore slowing individual response times. Ultimately, all horizontally scaled instances performed better than the baseline - peaking well after 50 seconds.

That reinforces the idea that scaling in this way is not a failure. Look at Figure V-B3. From that, we see that, despite the lower reaction times for arrangements with more M3.Medium instances behind the load balancer, the number of requests adequately served actually increases! This manifests in the lower number of 502 response codes, and the higher number of 200s.

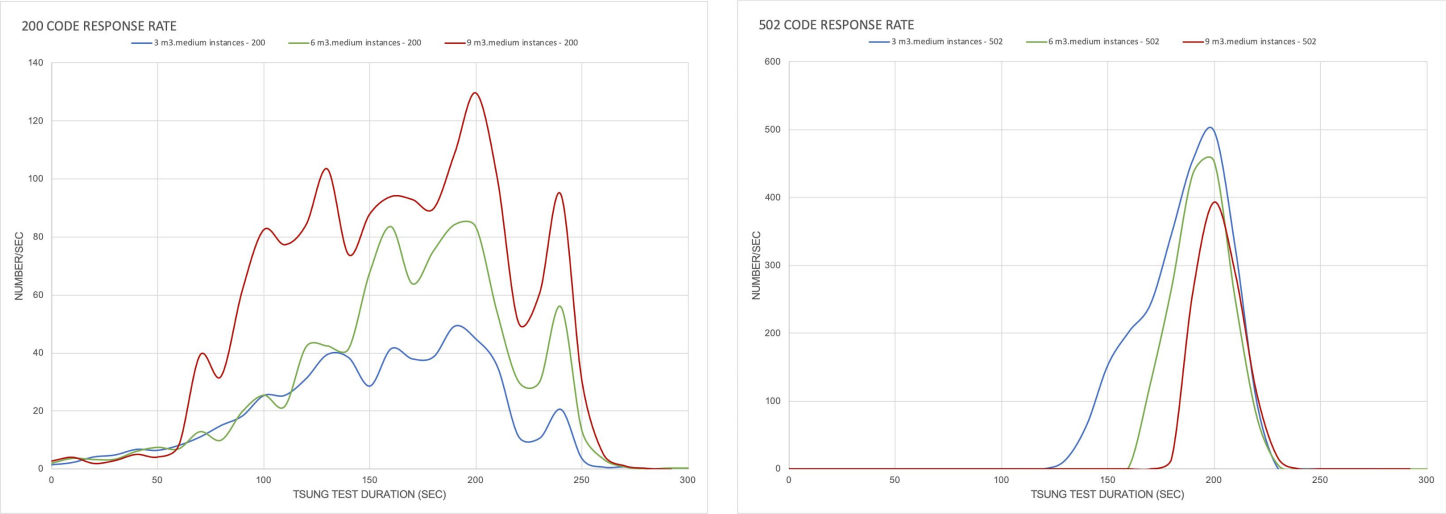


Figure V-B3: HTTP Code Rates for Horizontally-Scaled M3.Mediums; 200 (left), 502 (right)

As mentioned, the general trend of higher availability continued as expected - with higher instance-counted, better instance-laden arrangements performing better than their fewer-counted, worse-hardware siblings. One remaining point of interest, though: as can be seen in Figure V-B4, 6 M3.Large instances outperforms 3 M3.XLarge instances.

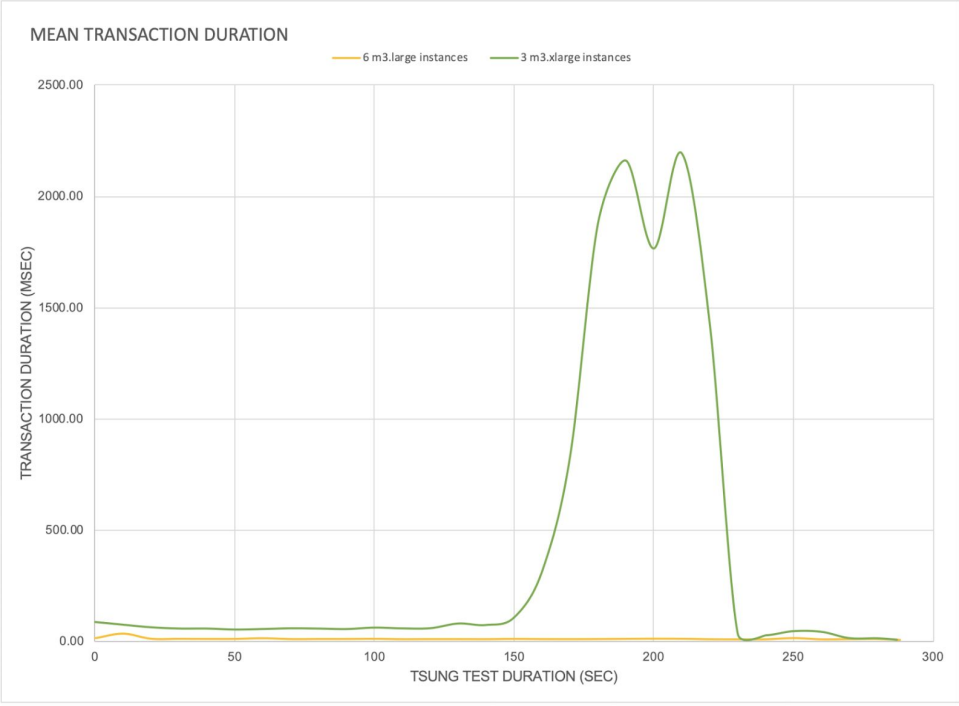


Figure V-B4: 6-Instance M3.Large Arrangement versus 3-Instance M3.XLarge Arrangement

This is a cautionary tale. The general advice that better EC2 instances will make for better results in horizontal scaling is not a hard-and-fast rule. Consider: based on the results above, 6 M3.Large instances is better than 3 M3.XLarge. However, based on Table A-V1, they cost the same, each coming in at $1.35 an hour. The conclusion is to be judicious - understand the needs of scaling your application, and the costs associated. It's not quite as simple as slapping on bigger and better instances.

Ultimately, as we stated at the start of this section, horizontal scaling was successful. Whispr demonstrated less catastrophic failure under load when rigged with horizontal scaling methods. For educational purposes, we've included Figure V-B5 below to provide more information about how all of our horizontal scaling arrangements performed.

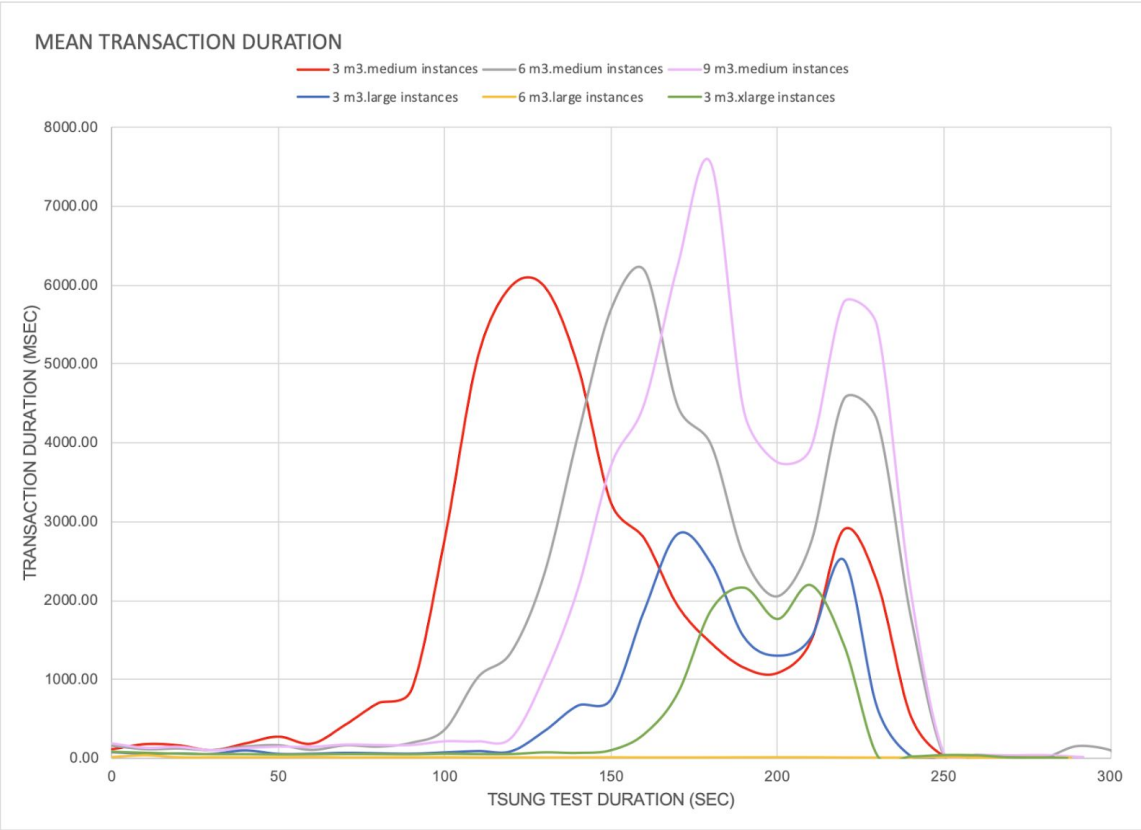


Figure V-B5: Mean Transaction Durations for All Horizontal Scaling Arrangements

SQL Optimizations

SQL Optimization #1: Adding Indices

After analyzing the underlying SQL queries in each controller action of our critical user paths, we found that most of the queries were already using the default **id** index that Rails automatically created for our *users* and *relationships* table. However, some controller actions performed

sequential scans. For example, one of our critical user actions is to randomly search for one of the 10,000 pre-initialized healthcare providers in the database by **name**, which we did not have an index for initially, so a sequential scan would be used to find the provider. After adding an index for the user's name (in addition to other indices, such as an index for the follower_id column in the relationships table), we measured the performance shown below. As you can see, the mean transaction times noticeably decreased from a peak of about 25,000 to about 22,000 msec. In addition, the 4xx and 5xx HTTP responses started occurring 80 seconds into the script rather than at 50 seconds. Therefore, our app was able to handle 4 users/sec rather than 2 users/sec.

Baseline

Adding Indices

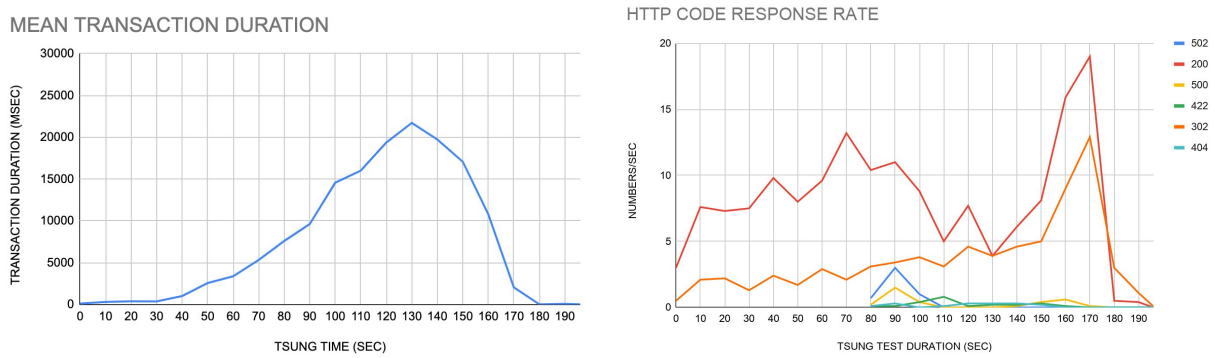


Figure V-C1: Baseline vs. Adding Indices

SQL Optimization #2: Preloading

We also found that one of our controller actions (*/providers*) was very inefficiently interacting with the database and exhibiting the N+1 problem *with joins*. Specifically, our */providers* endpoint creates a variable `@users` that is set as the first 5 providers in the database. Then, our `providers.html.erb` file loops through these 5 providers and checks whether each of them is already affiliated with the currently signed-in user like so:

`<% if @current_user.following.include?(provider) %>`.

Initially, this code causes N+1 SQL queries like below:

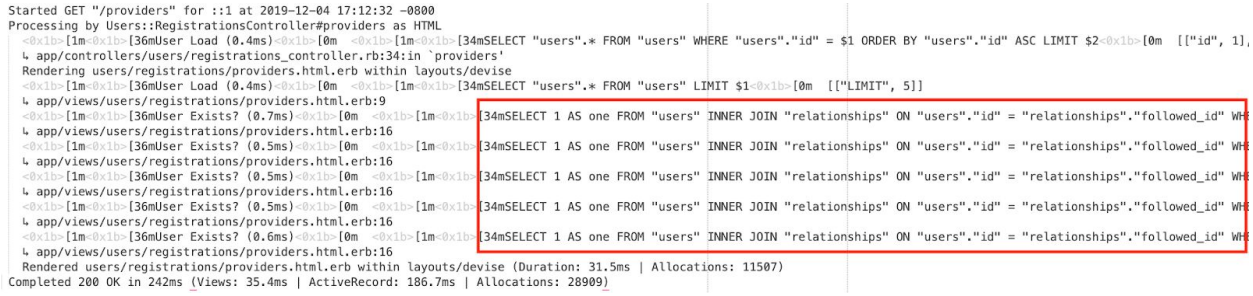


Figure V-C2: Problematic N+1 Relationships Queries with /providers Endpoint

The problem is that every time `<% if @current_user.following.include?(provider) %>` is executed, it is performing a SQL query with an inner join between the *users* and *relationships* table. To optimize this, we used the *includes* method in the controller for /providers to preload the current user's list of affiliated health providers (aka those in its "*following*" list) like so:

`@current_user = User.includes(:following).find_by_id(current_user.id)`.

Now we don't have the N+1 problem since the repetitive queries are replaced by a single query to just read all the relationships corresponding to the current user (whose id is 1 in the picture below):

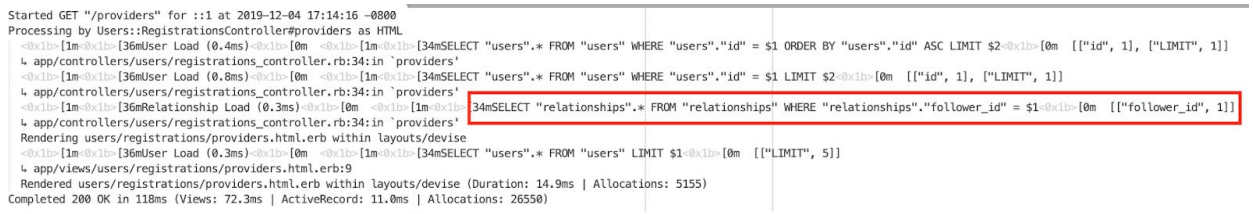
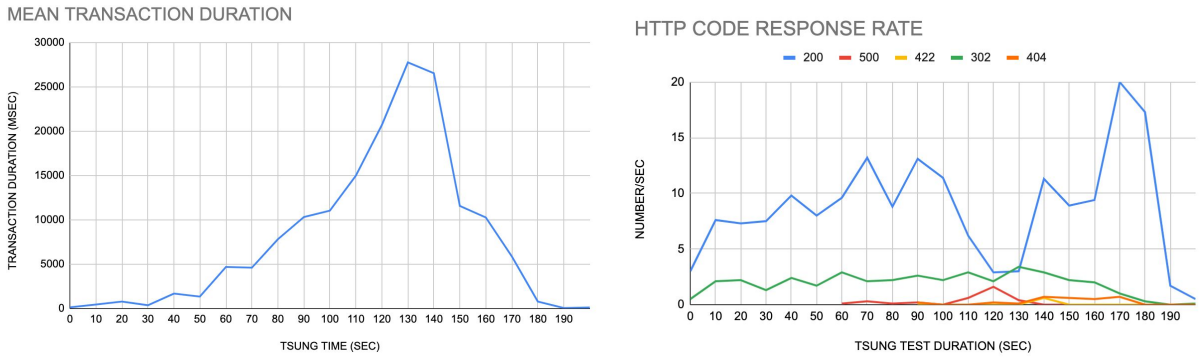


Figure V-C3: A Single Relationships Query with /providers Endpoint

We measured the resulting performance but found that the performance did not change much **initially** because visiting the /providers endpoint was only one small part of the load testing script we described earlier. **However**, after changing the load testing script to have more calls to this endpoint (five instead of one) and removing calls to some other endpoints, we saw noticeable improvements in performance, as shown below. As you can see, the mean transaction times noticeably decreased from a peak of about 27,500 to about 22,000 msec. In addition, the 4xx and 5xx HTTP responses started occurring 70 seconds into the script rather than at 60 seconds.

Baseline



Preloading

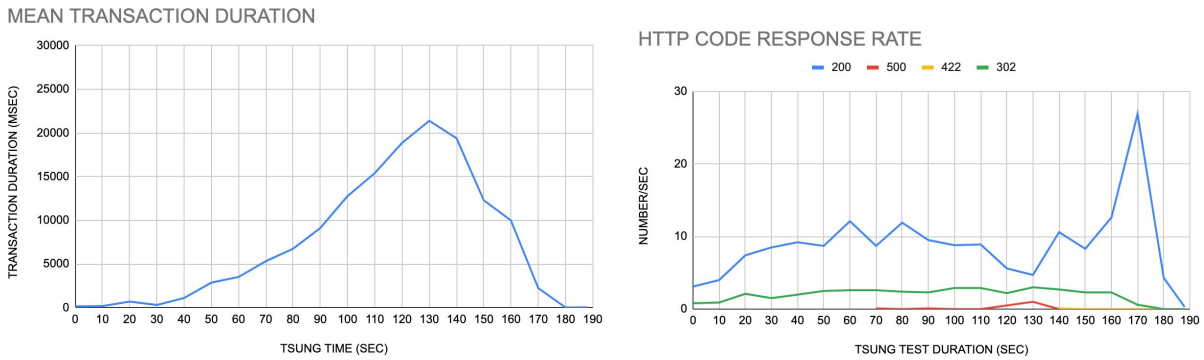


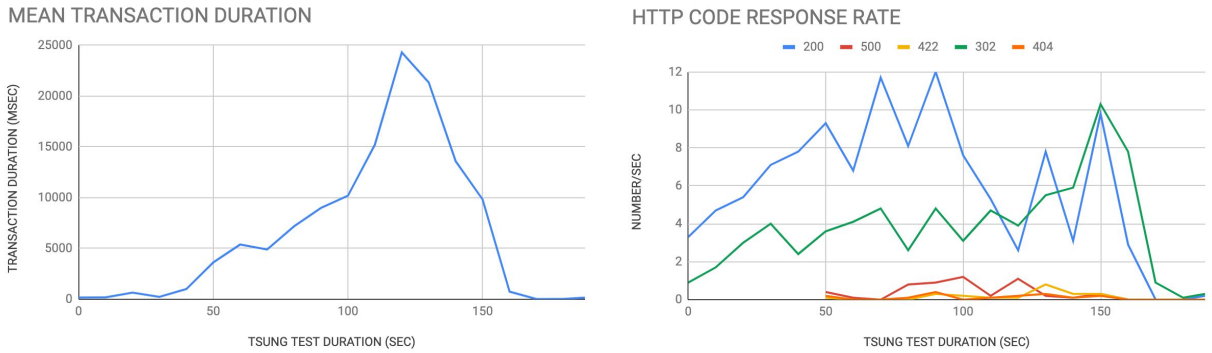
Figure V-C4: Baseline vs. Preloading

SQL Optimization #3: Database Table Changes

A final SQL optimization that we tried was changing the column types in our *users* and *relationships* tables in hopes of reducing the overall row sizes and thus increasing performance in reads and writes to the database. We changed the *bigint* id columns in both tables to the plain *integer* type. We also changed several variable-length string types to fixed-length string types of max length 20 (e.g. for the user's name, email, allergies, diseases, etc.).

However, the performance did not change by much, as shown in Figure V-C5 (next page). We believe that our initial baseline tests did not insert rows into the tables whose column values were long enough for this optimization to make a significant difference. Still, we see that there is a minor decrease in peak transaction duration as well as a decrease in the number of 4xx and 5xx HTTP responses, which combine to be a relatively small improvement as compared to the previous two SQL optimizations.

Baseline



Database Table Edits

Figure V-C5: Baseline vs. Database Table Edits

Fragment Caching

Overview

As a form of server-side caching, fragment caching allows dynamic applications to cache individual, or fragment, components of a view. The advantages of fragment caching lie in its ability to reduce server load and increase view generation. Essentially, fragment caching is conducted by obtaining a unique cache key for a desired view fragment. This view fragment is then rendered by retrieving the cache corresponding to the generated cache key; if the cache key is expired or does not exist, then the server queries and caches the desired information from the database.

Our Approach

We can categorize the requests our application receives into two buckets: shared and user-specific. Both shared and user-specific view fragments are cacheable, but require different cache keys and attention.

Shared Fragments

Shared view fragments are generally simple in terms of determining efficient cache keys. However, there are some instances when these keys could potentially cause problems. We can further categorize shared view fragments into two categories: static and form pages.

Static pages can be implemented naively with a simple cache key - nothing! For example, our home page and about page are completely static and aren't dependent on factors beyond the contents within the the `html.erb` file. Therefore, the caching for these pages is as simple as caching the fragment with an `etag` upon first load. Upon a subsequent load of the page, we check to see if the `etag` matches the cached `etag` (if the contents of the `html.erb` file were changed, the `etag` will be different from the cached `etag`). If they match then we render the cache; if not, we cache the fragment corresponding to the new `etag` and then we render the fragment. Such a caching code block is seen below.

```
<%cache do%>
  <!-- static information >
<%end%>
```

We can see this in action locally through the output.

```
Started GET "/about" for ::1 at 2019-12-04 23:27:42 -0800
Processing by ApplicationController#about as HTML
  Rendering application/about.html.erb within layouts/application
Read fragment views/application/about:f90bfc9433f9dbbfb002e4d45dfbc6c5/localhost:3000/about (0.1ms)
Write fragment views/application/about:f90bfc9433f9dbbfb002e4d45dfbc6c5/localhost:3000/about (0.3ms)
  Rendered application/about.html.erb within layouts/application (Duration: 8.3ms | Allocations: 2659)
Completed 200 OK in 37ms (Views: 36.2ms | ActiveRecord: 0.0ms | Allocations: 7819)


Started GET "/about" for ::1 at 2019-12-04 23:27:54 -0800
Processing by ApplicationController#about as HTML
  Rendering application/about.html.erb within layouts/application
Read fragment views/application/about:f90bfc9433f9dbbfb002e4d45dfbc6c5/localhost:3000/about (0.1ms)
  Rendered application/about.html.erb within layouts/application (Duration: 0.5ms | Allocations: 168)
Completed 200 OK in 7ms (Views: 6.3ms | ActiveRecord: 0.0ms | Allocations: 5404)
```

As we can see upon the first load, there is no cached fragment corresponding to the `etag`: `f90bf...6c5`. So our application creates a new fragment cache with that same `etag` and writes it to our server. Upon our subsequent load, the application calls the server for a fragment from the about page with the `etag` - `f90bf...6c5` (same as the previous one because the content hasn't changed). The server actually does indeed have that `etag` so the application now renders that cached view fragment. In the case of the above example, the actual rendering of the about page

took a duration of 8.3 ms without the cache and 0.5 ms with the cache. Overall, the cache allowed us to shave 30 ms total. While these numbers aren't considerably large, they still contribute to having a "faster" user experience.

Form pages are very similar in terms of implementation, except for one catch: we always want to make sure our submission of the form will update the database with the current information, not the cached information. How can we ensure this? We only want to cache the form itself - we NEVER want to cache the submission. This results in a cache format that looks as follows:

```
<%= form_for(resource, as: resource_name, url: registration_path(resource_name)) do |f| %>
  <% cache(resource) do %>
    <!-- form information stuff >
    <!-- form information stuff >
  <%end%>
  <div class="actions">
    <%= f.submit "Sign up", class: 'btn btn-primary m-2'  %>
  </div>
<% end %>
```

Run locally, this code outputs the following results:

```
Started GET "/users/sign_up" for ::1 at 2019-12-04 23:28:44 -0800
Processing by Users::RegistrationsController#new as HTML
  Rendering users/registrations/new.html.erb within layouts/devise
Read fragment views/users/registrations/new:49984ed8f15a88679bbb06635ae79374/users/new (0.2ms)
  Rendered users/shared/_error_messages.html.erb (Duration: 0.8ms | Allocations: 213)
Write fragment views/users/registrations/new:49984ed8f15a88679bbb06635ae79374/users/new (0.9ms)
  Rendered users/shared/_links.html.erb (Duration: 1.0ms | Allocations: 136)
  Rendered users/registrations/new.html.erb within layouts/devise (Duration: 13.4ms | Allocations: 3287)
Completed 200 OK in 34ms (Views: 32.3ms | ActiveRecord: 0.0ms | Allocations: 9739)

Started GET "/users/sign_up" for ::1 at 2019-12-04 23:31:45 -0800
Processing by Users::RegistrationsController#new as HTML
  Rendering users/registrations/new.html.erb within layouts/devise
Read fragment views/users/registrations/new:49984ed8f15a88679bbb06635ae79374/users/new (0.3ms)
  Rendered users/shared/_links.html.erb (Duration: 0.4ms | Allocations: 96)
  Rendered users/registrations/new.html.erb within layouts/devise (Duration: 2.5ms | Allocations: 475)
Completed 200 OK in 16ms (Views: 14.7ms | ActiveRecord: 0.0ms | Allocations: 5813)
```

As predicted, the output is similar to our static page; we reduced our rendering process by 18 ms. Obviously, this is not detectable by the human eye at such small delta, but it is proof that our caching works as expected and that it does indeed shed some fat from our application.

User-Specific Fragments

Now to hit more interesting fragment caching, we must recognize that our application is extremely user specific. For data unique to the current user, information cannot be naively cached as with shared fragments. Instead, we should be caching information based upon user

specific information: `user_id`, `updated_at`, or even `user_follower_count`. For example, our profile page displays the name and email for the current user. We can use fragment caching to save the most updated user information.

We defined the cache key in `application_helper.rb` as follows.

```
def cache_key_for_profile_info
   "user_#{current_user.id}_#{current_user.updated_at}"
end
```

We then defined our cache fragment for our profile page as follows.

```
<% cache(cache_key_for_profile_info) do %>
  <!-- user information stuff >
  <!-- user information stuff >
<%end%>
```

Beyond just the profile page, we fragment cached other pages as well to reduce server load and boost performance. Please see the table on the next page.

Page	Fragment Cache Approach
Home Page	Static - cached entire page
About Page	Static - cached entire page
Sign Up Page	Form - cached form + resource
Edit Profile Page	Form - cached form + resource
Edit Medical Page	Form - cached form + resource
Profile Page	User-Specific - cached profile info (i.e. name)
Medical (information) Page	User-Specific - cached med info (i.e. allergies)
Affiliation Page	User-Specific - cached affiliation between users

In order to test the validity of our caching, we seeded our database with 5000 users and adjusted our tsung scripts to hit pages multiple times in order to discern improvement. See Figure V-D1.

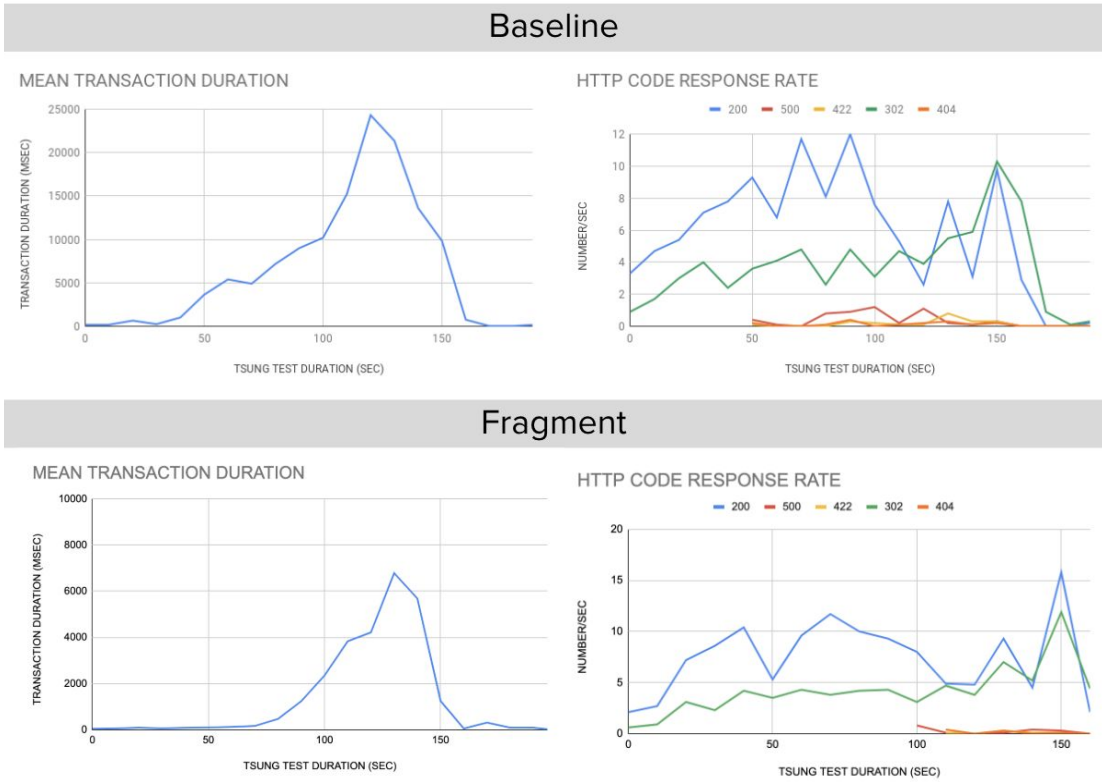


Figure V-D1: Fragment Caching vs Baseline

From our results we can see that our fragment cache is indeed able to allow our application to increase performance and reduce response time. Compared to the baseline, fragment begins to fail almost 2 whole phases after in terms of transaction duration. We can also see a similar phenomenon in the HTTP responses. Our error codes, in red and orange, between fragment and baseline are also separated almost 2 whole phases apart. Additionally, the mean duration of transactions between baseline and fragment is decreased by more than 3 fold. These results indicate higher rate server performance and an overall better user experience.

Low-level Caching

There are times when an application needs to cache query results instead of an entire view fragment. These cases are generally caused by computationally expensive queries that are relatively stable. However, our application doesn't have many instances of these cases because the most expensive query we perform is a search - which doesn't happen that frequently.

We implemented low-level caching anyway, to see its effect. One such example can be seen in our Affiliation Page, where healthcare providers can search for patients by name. Rather than constant search queries for the same patients, it stores patient information in memory for subsequent calls. In our Application Controller, we added the following line of code.

```
@patients = Rails.cache.fetch("/user/#{params[:search]}", :expires_in => 5.minutes)
```

In this case, we search for all patients with the name given by a parameter search. Thus, repeated calls of a specific search can be retrieved directly in memory, rather than in the database. However, since searching isn't a massive part of our application, the addition of the low-level caching doesn't really help that much. In fact, when we were running load testing on just low-level caching, we found that our application was slower! We think this is caused by two things. One, memory is limited to a certain size. Thus, caching results dependent on name could result in new caches overwriting old caches, depending on name length. Second, our Tsung scripts create names so that they are random. Because the names are random, there isn't much overlap between names to take advantage of. This results in an increased rate of unique search queries and cached results, which also results in a reduced cache hit rate because old caches are being overwritten. This could explain the increase in transaction time as search queries would now contain an extra step to cache information - despite that caching offering little to no performance boost.

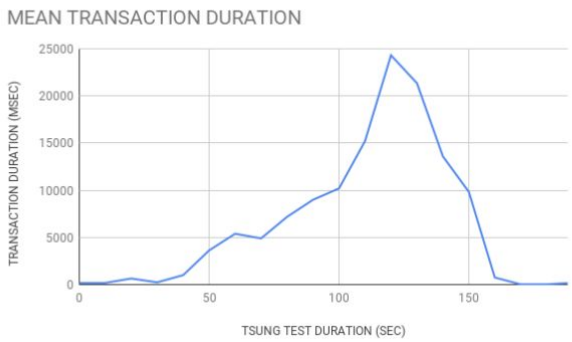
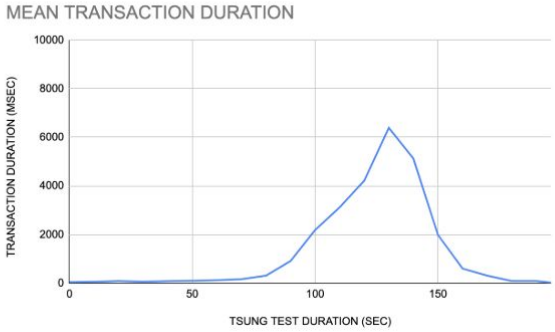
Memcaching

By default, rails caches information in memory.

```
config.cache_store = :memory_store
```

This can be problematic: the cache keeps entries in the same Ruby process and is bounded, by default, to a size of 32 megabytes. When the cache stores has exceeded the allocated size, it will automatically clean up and remove older caches. If running large applications or ones that contain multiple Ruby processes, this cache store would be insufficient because the rails server instances wouldn't be able to share cache data between each other. This is where memcaching comes into play.

Memcaching provides a centralized cache for your rails application. It is used to provide a singular, centralized, shared cache cluster with boosted performance and reduced redundancy. We used the *dalli* gem along with ElastiCache to create our centralized cache cluster. From there, we continued using our fragment caching from above to utilize the cache store.

Baseline	Fragment (memcache)
	

The output as we can see of both fragmentation and memcaching results in higher performance that is very similar to fragment caching by itself. Our application lasts almost 2 phases longer than before and peak response time is dramatically reduced. One conclusion we can make from the similarity between our fragment caching and memcaching output is that our application isn't large enough to see considerable improvement through memcaching, nor does it run multiple server instances.

Client-Side Caching

We implemented Client-Side Caching in our information view pages such as Profile and Medical. Instead of having our server cache view fragments, we can have our browser conditionally make GET requests dependent on whether the contents of a page have changed or not. We implemented the **fresh_when()** method in our Application Controller as follows.

```
fresh_when([current_user.id, User.all, Relationship.all])
```

The fresh_when method sets the `etag` and modified headers of the last updated timestamp of the user and relationship table as well as the current users id. The server can now determine whether or not a GET request is necessary based on whether the computed `etag` matches the `etag` computed upon request.

When we tested our caching locally, we got the following output in the network portion of our inspect.

Name	Status	Type	Initiator	Size	Time
providers	200	xhr	turbolinks.js:154	3.3 KB	88 ms
providers	304	xhr	turbolinks.js:154	979 B	46 ms
medical	200	xhr	turbolinks.js:154	3.8 KB	158 ms
medical	304	docum…	Other	979 B	25 ms

With our client-side caching implementation, upon the first request of our profile pages, we do a full GET request to receive information. As long as the user doesn't edit their information, all subsequent requests resulting in a complete bypass of view rendering and a status code of 304. As we can see from the response time and size, we can reduce our response time by 125 ms in the case of the medical page.

NGINX Processor and Thread Count Modifications

Overview

The next area of scaling we tackled was the number of threads and processes employed by the instance we used. A thread is a path of execution within a process, and can basically do anything a process can do. A process is an executing instance of an application and can contain multiple threads.

The main difference, however, is that threads within the same process share the same address space, while processes do not share address space. Multithreading is a feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU. By having multiple threads, we could allow one thread to start handling requests from a user while another thread is waiting on database I/O.

To handle multiple requests, we can use a multiple process format: spawning a process per request. For example, we can create a certain amount of processes in a process pool, eating the costs of startup and cleanup up front. Each process can then go and handle a different user requests as they become available, without waiting for any sort of initialization.

Our Approach

We wanted to focus on a wide variety of processor and thread count combinations. The baseline for this section (and for all other tests) was conducted with the default puma configuration shipped with Elastic Beanstalk: 8 threads minimum, 32 threads maximum, and 1 worker. Note: to modify these values, we could not change the `config/puma.rb` file in our Rails application, we had to change `.ebextensions/11_puma.rb` to change the Elastic Beanstalk configuration for threads and processes.

In total, four tests were conducted: 1 Thread/8 Processes, 1 Thread/1 Process, 15 Threads/8 Process, and 15 Threads/1 Process.

Results

The results were not surprising. The worst performing test was 1 Thread and 1 Process, which had high transaction duration early on (approximately 30 seconds in, compared to the other tests at around 60 seconds in). See Figure V-E1 (next page) for a visual representation of this. Further, the tests utilizing greater processor and thread counts shifted the time of the peak to the right - indicating that the app was able to handle more of the increasingly-difficult phases before keeling over.

One downside was that each increased processor and thread count instance had relatively similar graphs in regards to mean transaction duration, peaking at 140 seconds. In other words, though they were improvements over the baseline, they were not improving relative to each other. This is most likely due to the fact that the tests were run on a single core processor. Theoretically, if we were to able to use a multicore system to compare, such as M3.XLarge or M3.2XLarge (4 and 8 cores respectively), the results would be much clearer.

However, we wanted to maintain a simple baseline on an M3.Medium machine. We also did not want to conduct a combination of scaling methods (in this case, using vertical scaling to go to an M3.XLarge or 2XLarge, along with the Processor/Thread manipulations). From that perspective, we knew the theoretical results that were achievable, but did not necessarily see them in practice due to this focus on simplicity.

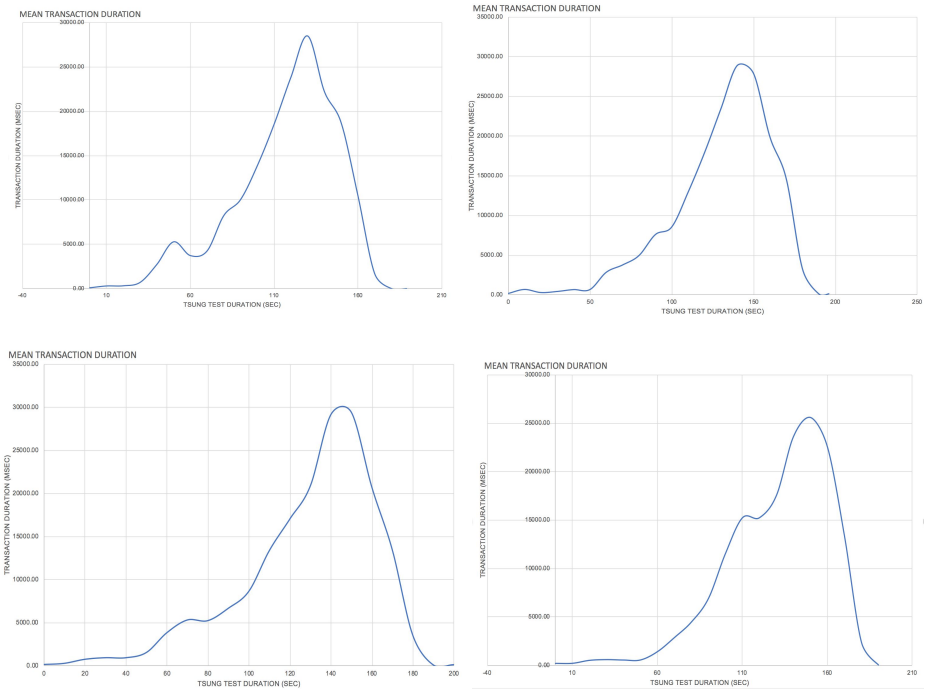


Figure V-E1: Mean Transaction Duration for 1 Thread 1 Process (top-left), 1 Thread 8 Process (top-right), 15 Threads 8 Processes (bottom-left), and 15 Threads 1 Process (bottom-right)

On a more positive note: in regards to HTTP error code rates, the results were still very clear. The test with 1 thread and 1 process had noticeably more 404 and 500 errors than the tests with increased thread and processor counts. This result makes sense. One process and one thread is simply inadequate to handle the flood of requests the load test introduces, resulting in request timeouts. As can be seen in Figure V-E2, increasing processor and thread count drastically reduced the number of 500 and 404 return codes - indicating an overall success in tweaking those figures.

Figure V-E2: HTTP Code Response Rate for 1 Thread 1 Process (top-left), 1 Thread 8 Process (top-right), 15 Threads 8 Process (bottom-left), and 15 Threads 1 Process (bottom-right)

Thus, we see that multithreading and having multiple processes do show better results in terms of latency and error code response rates.

Read Slaves

Overview

As we finished the likely areas of load testing and scaling, we tried to implement more difficult areas of scaling, particularly read slaves. Broadly speaking, read slaves offer enhanced performance and durability for the main database - providing a supplementary, read-only database. This comes in handy for read-heavy query sequences. Thus, with a read slave available, we can serve read requests even when the primary database is locked for write queries. These read replicas can even be promoted to master status when the primary is down! This means increased **availability**. In essence, any downtime experienced by the primary will have the replica ready to serve! Load is reduced from the primary database!

Our Approach

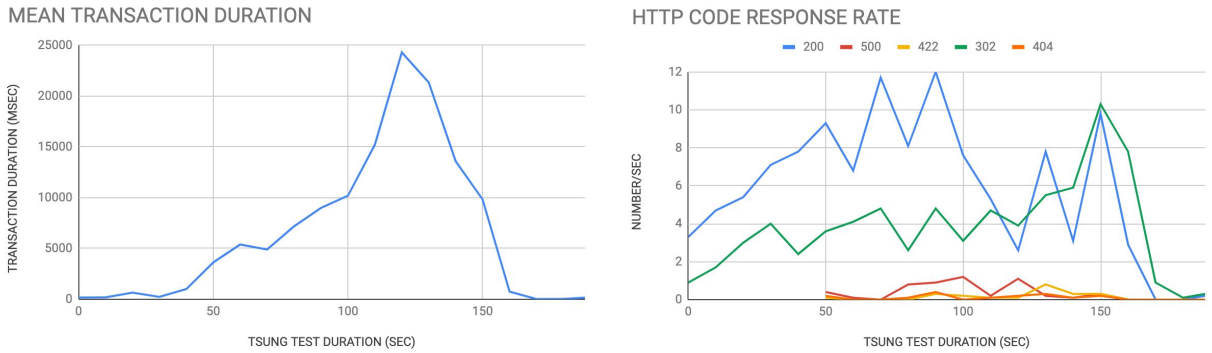
Originally, we were going to use the Octopus gem to configure read slaves, but the gem was not supported on Rails 6. Thus, we turned to Professor John Rothfels' advice: creating a read replica manually on AWS' RDS. This read replica acted as a direct copy of our primary database; all we needed to do was configure our application to direct read queries to the replica. The *config/database.yml* file was changed to configure support for the replica database by specifying the replica hostname and environment variables. The activation record was made aware of our replica by updating to specify writing to the primary database, and to use the replicated primary for strictly reading. The concept of 'automatic connection switching' was added in the application config, allowing the application to switch access to the primary or replica based on the HTTP VERB. POST will switch to primary, and GET requests will switch to replica—unless there was a recent write. Basically, if there was a recent write, the client will read from the primary while the replica is updated within our specified delay of two seconds. Otherwise, most if not all the reads requests will be routed to the replica—offloading the primary.

Results

As you can see below, with read slaves the mean transaction duration peaks at about 20,000 msec, which is about 5,000 msec less than the peak in the baseline test. With this optimization, there were also no 404 and 422 HTTP errors, and the 500 errors that did appear only appeared in minimal amounts. Clearly, this improvement is quite significant!

Note that our baseline load testing script has a balanced mix of reads and writes to the database; if a larger percentage of our critical user actions were reads, then we would see an even greater performance boost because the read slaves optimization would help us avoid situations where one write stops a very large amount of reads.

Baseline



Read Slaves

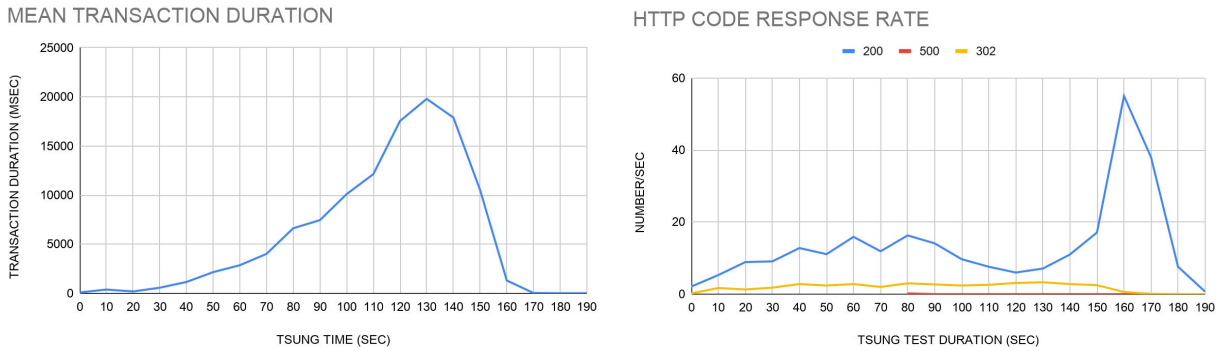


Figure V-E3: Baseline vs. Read Slaves

VI. Future Development

Given an expanded timeline, there are several features we'd like to either implement, or polish. Firstly, we believe adding a profile picture system to the application would be vital to its long-term viability. The ability to more easily distinguish same-named users via picture would be invaluable. We'd also like to expand the level of detail available in the health forms - marking out specific fields into various subsections, to make finding the individual bit of information easier.

From a provider perspective, we'd add the ability to view aggregated statistics about the patients they had subscribed to. This would let them get a greater level of information on the patients they were trying to treat, hopefully providing them with better data about the care they were giving. These statistics would be able to be filtered at will, restricting it to certain age ranges, stated gender, and so on.

Additionally, we would like to turn our attention to a family friendly feature - essentially parental controls - for parents whose children have medical data. Parents would be able to manage their children's data and have access from their account page. Thus the relationship between the family members, even spouses, would be acknowledged.

Finally, the largest ticket item: we would turn our attention to an iOS app. To truly make Whispr fulfill its goal of convenient access to medical information, we'd need to ensure users have the best possible experience when on the go. To that end, we'd focus on an iPhone app first, then an Android one.

VII. Conclusion

Ultimately, all of the scaling methods we tried were successful. Horizontal and vertical scaling gave us the largest impact for the least effort, while well-targeted attempts at caching and SQL optimizations definitely pulled their weight. We were easily able to scale the app to handle ever-increasing user loads, ensuring that - whatever the real world might throw at us - we'd be ready to defend Whispr from it.

However, we learned that ensuring even a relatively simple app will work with an explosion in the user base is no small task. It requires thoughtful planning, careful analysis and targeted improvements to really account for the stress millions of users can put on a system. One of the largest difficulties is that - despite knowing a large variety of methods to ensure scalability - it is hard to determine their eventual value. For instance, horizontal scaling typically resulted in a tradeoff between response times, and adequately-served requests. For Whispr, that might be acceptable: allowing requests to take a few milliseconds longer in exchange for greater reliability fits in with our mission. For another app, where speed might be of the essence (a financial tracking app, perhaps), this might reduce the effectiveness of horizontal scaling entirely - reducing the number of ways to achieve true scalability, making the job harder.

Difficulty ought not be a deterrent, though. From our perspective, scalability is perhaps the most important thing to consider when initially designing an application. Aesthetic choices can be easily reversed at any point in development. The infrastructure, architecture and technology choices made at the beginning of a project greatly impact the ability of scaling methods to have a noticeable impact. Changing those aspects after they have been set for a period of development is essentially akin to a from-scratch rewrite - which, from any perspective, is a herculean task.

In the end, we believe that we have demonstrated the importance of scaling, and the effectiveness of each of the methods used on Whispr. A blanket, no-holds barred approach to scaling will certainly be successful, but truly impactful scaling needs to be targeted. Regardless, if you want to build a meaningful app today, you better be ready to scale it.