Schedule.me Report
Winter 2021

CS 291A - Scalable Internet Services
Winter 2021

Akshay Heda
Andrew Zhang
Chinmay Sonar
Erin Woo
Rakshith Gopalakrishna
Vaishali Surianarayanan

Table of Contents

1. Introduction

In the world of constant online meetings, scheduling virtual meetings has become a new norm. Often this means an organizer reaching out to each individual to try to find a time that works. We have also seen the growth of scheduling tools like When2meet and Doodle polls to try to schedule asynchronously. These solutions, although significant, have incomplete feature sets.

We are introducing **schedule.me**, a site that allows users to schedule meetings while inputting how willing they are to give up the time slot. Our cutting-edge algorithm allows a group to find the most optimal times to meet. We have built an intuitive service that can scale to your team or even your entire company!

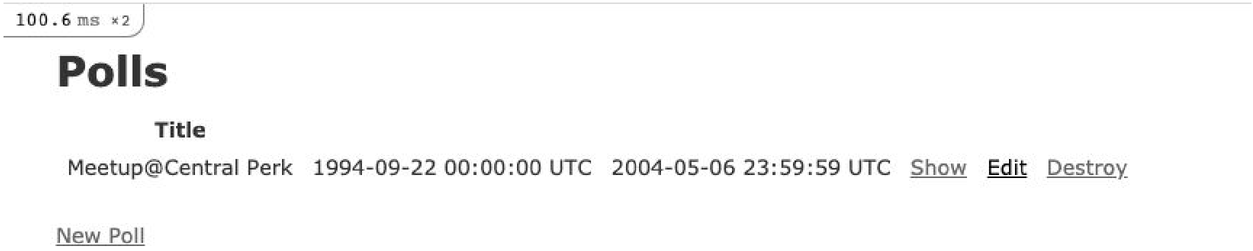
2. Features

Our application provides the functionality to create polls, which are composed of a timeframe for which users can input availability.

We describe our application's customers as two types: An **organizer** plans an event, creates a poll, and sends it out to their attendees. A **participant** is a user that is sent a link to a poll and is an attendee to the event. The distinction is muddled, as a user can be both an organizer and a participant, but this is defined as such for clarity as to their motivations.

2.1 Main polls page

The main polls page serves as the entry point into this application. In a real user scenario, the users would not view all the polls loaded from the database. However, for testing, all polls are accessible from this page. This is where users can create, edit, and view existing polls.



Polls

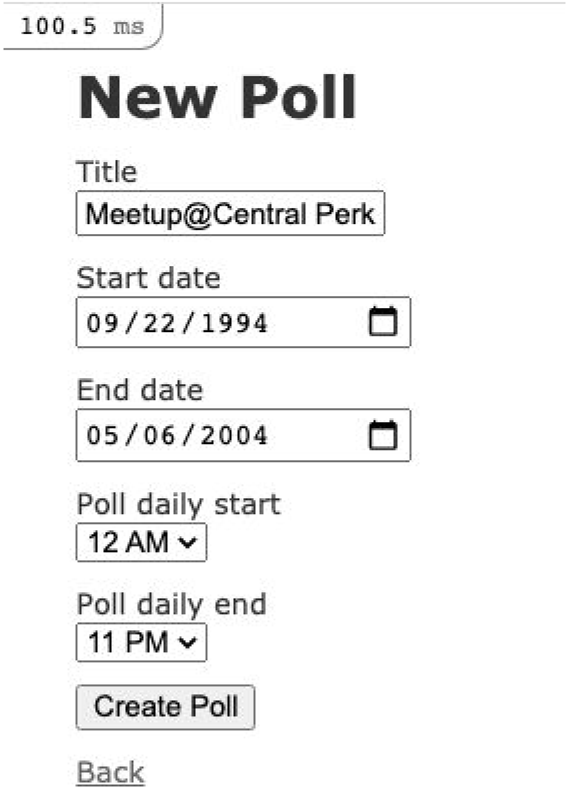
Title					
Meetup@Central Perk	1994-09-22 00:00:00 UTC	2004-05-06 23:59:59 UTC	Show	Edit	Destroy

New Poll

2.2 Creating a new poll

After clicking “New poll” from the main polls page, the user will be redirected to the new poll creation page. On this page, the user can indicate the poll title, polling start and end dates, and the daily polling timeframes (earliest and latest time an availability can be within a day).

After the user clicks “Create Poll”, the application will validate the user’s input before indicating successful or invalid submission. For a poll to be considered valid, the start date must be on or before the poll’s end date, and the poll must have a non-empty title. Additionally, the poll’s daily start time must be before the poll’s daily end time.



New Poll

Title
Meetup@Central Perk

Start date
09/22/1994

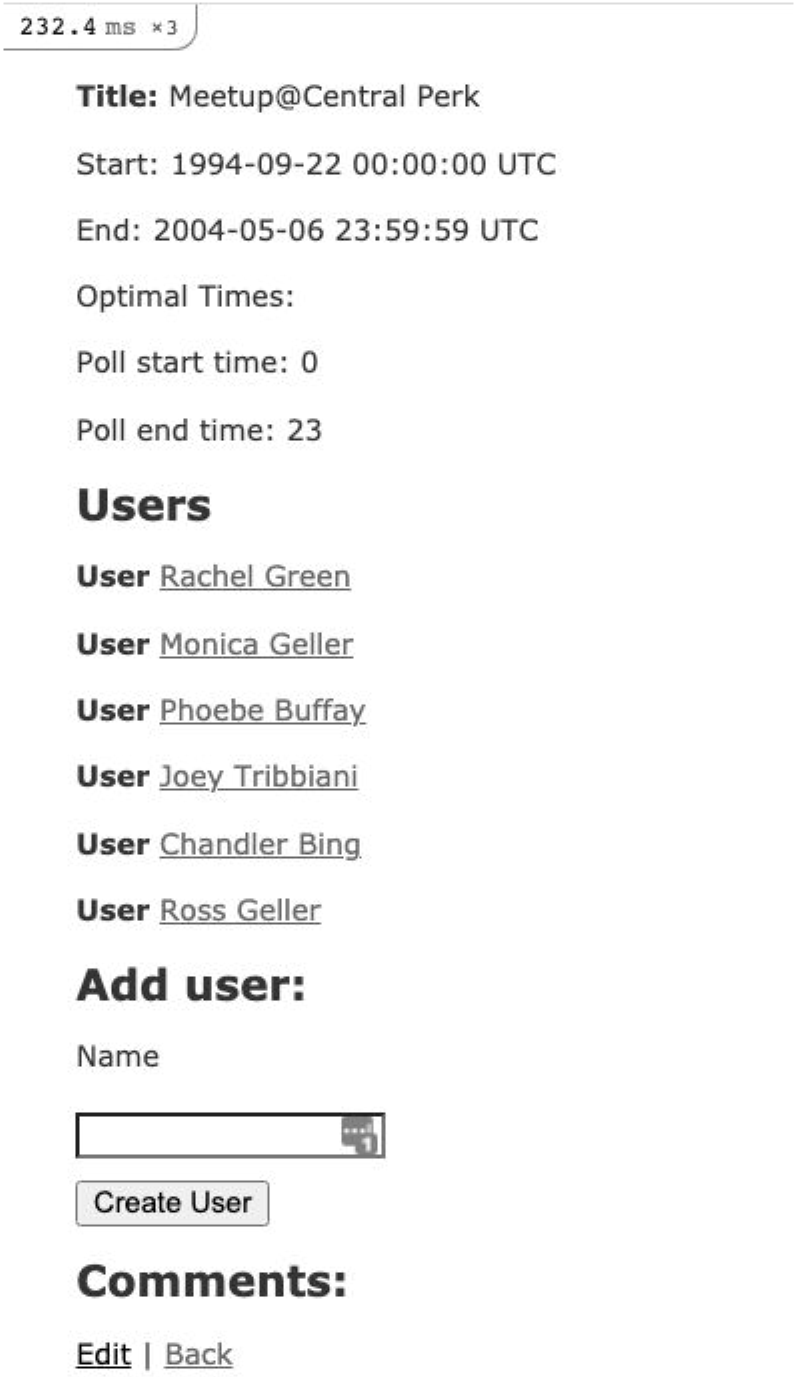
End date
05/06/2004

Poll daily start
12 AM

Poll daily end
11 PM

2.3 Poll details page

After creating a poll, the poll can be accessed with a unique ID. From the poll’s page, the user can see the poll details and its associated users and the users’ comments. This page also serves as an entry point for initializing new poll users.



Title: Meetup@Central Perk

Start: 1994-09-22 00:00:00 UTC

End: 2004-05-06 23:59:59 UTC

Optimal Times:

Poll start time: 0

Poll end time: 23

Users

User Rachel Green

User Monica Geller

User Phoebe Buffay

User Joey Tribbiani

User Chandler Bing

User Ross Geller

Add user:

Name

Comments:

2.4 Adding a user to a poll

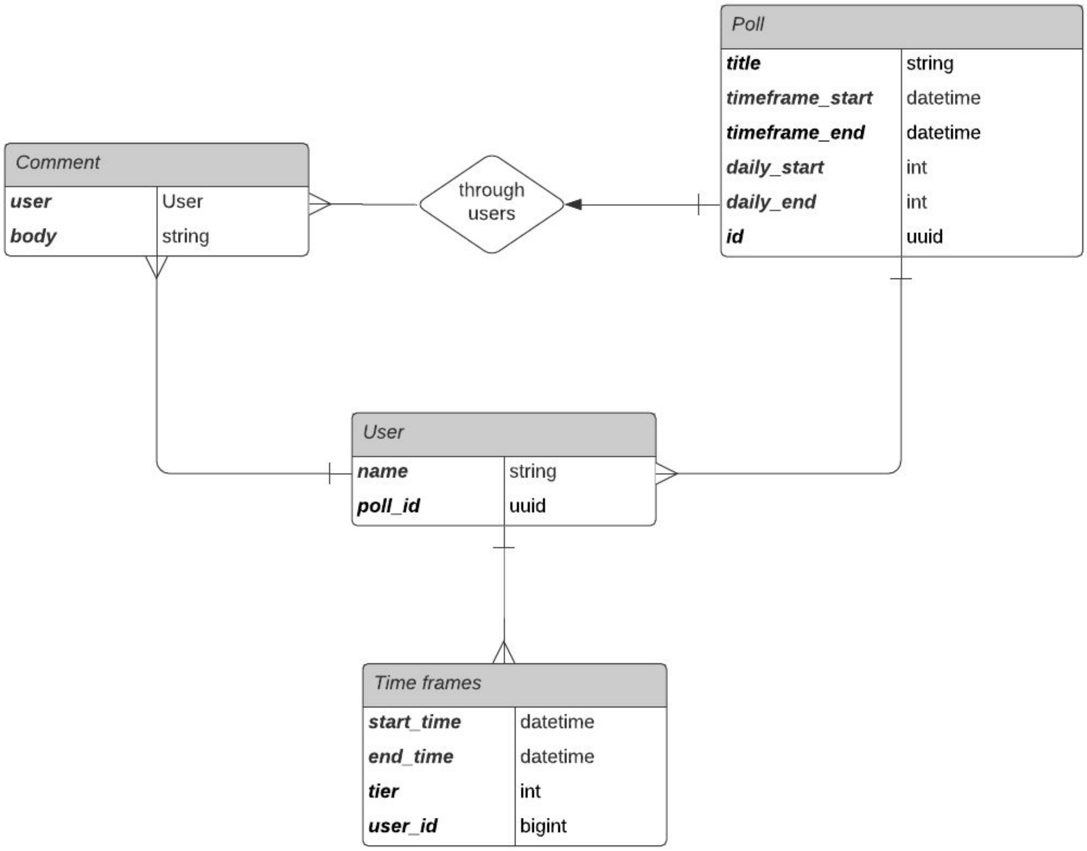
The entry point for initializing new poll users begins with the user entering their name in the "Add user" section of a poll's page. After clicking "Create User", the application will redirect them to the new user's details. Here, they can enter their availability as timeframes, with each timeframe having a specific "tier" or priority: 1 being the most preferred timeframe and 3 being the least preferred. For a timeframe to be valid, the timeframe must be within the poll's specified times. Every timeframe must have an indicated tier value.

This page also allows users to write comments that will be visible from the given user page and its associated poll page.

2.5 Optimal timeframe selection

Our proprietary algorithm uses cutting-edge state-of-the-art technology built by leading algorithms researchers. Based on the users' inputted timeframes, we find which 15-minute time slots are most ideal to hold a meeting. We iterate over each user's timeframes to count which time slots have the greatest number of users available. If multiple time slots have the same number of users available, they are ordered by preference. A 15-minute time slot's preference is calculated by summing the squares of each available user's "tiers". This places more weight on avoiding undesirable time slots.

3. Data Models



Poll	
title	string
timeframe_start	datetime
timeframe_end	datetime
daily_start	int
daily_end	int
id	uuid

Comment	
user	User
body	string

User	
name	string
poll_id	uuid

Time frames	
start_time	datetime
end_time	datetime
tier	int
user_id	bigint

3.1 Timeframe Model

A timeframe object represents a start and end time in UTC. This object refers to a single timeframe of availability for a specific user within a poll. It also has an indicated tier value ranging from 1-3 (1 being the most prioritized) that represents the user’s willingness to give up that timeframe.

3.2 User Model

A user object represents a participant in a poll. Anyone with a link to the poll can add users, whether that be the poll organizer or external participants. Each user belongs to a poll.

3.3 Comment Model

A comment object represents additional information that a user can specify regarding their availability. It is created by a user in the user's specific page. It can then be viewed either in the user's page or in the poll's page, where all comments by all users of the poll are visible.

3.4 Poll Model

A poll object is the base model for this application and is referenced by it's poll id. Upon creation, it's initialized with a name and start/end days that represent the time that the organizer would like to poll from. The daily start and end times represent the window of time each day that users can indicate their availability from.

4. Load Testing

4.1 Setup

We will start by defining a **transaction** as it is in Tsung: a single action against the application. We then define a **session** as a set of transactions that model a series of actions a user would commonly perform. We set up our testing script to have transactions be associated with specific sessions to compare transaction time across sessions.

To make the sessions more realistic, we set up random access for workflows involving accessing polls/users. We added routes and controllers that map a random number generated in tsung to a valid pre-seeded poll uuid or user id.

4.2 Seeding

Every test is initialized with a seeded database. As part of seeding, we create 1000 polls, 10000 users, 20000 timeframes, and 10000 comments. This was to make sure that our database already had data and that we could randomly access polls and users to test response times.

We leveraged the bulk insert feature of Rails to accomplish faster insertion. Seeding the development database was easy and could be achieved by the rake db:seed command. For the production database, Elastic Beanstalk provides an interface to run commands before the application is deployed in the staging phase. However, adding the db:seed command to this interface resulted in a failed deployment. We were able to work around this by running the db:seed command inside the EC2 instances of the deployed application using required environment variables. We note that we were able to automate this process with the help of a script.

4.3 Transactions

For our workflows, we defined the following transactions representing primitive requests:

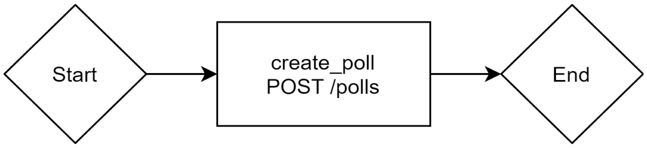
create_poll: POST request to create a poll
create_user: POST request to create a user associated with a poll
create_timeframe: POST request to create a timeframe associated with a user
create_comment: POST request to create a comment associated with a user
view_poll: GET request to view a specific poll page
view_user: GET request to view a specific user page

These are the transactions that are contained within the sessions described in the next section.

4.3 Sessions

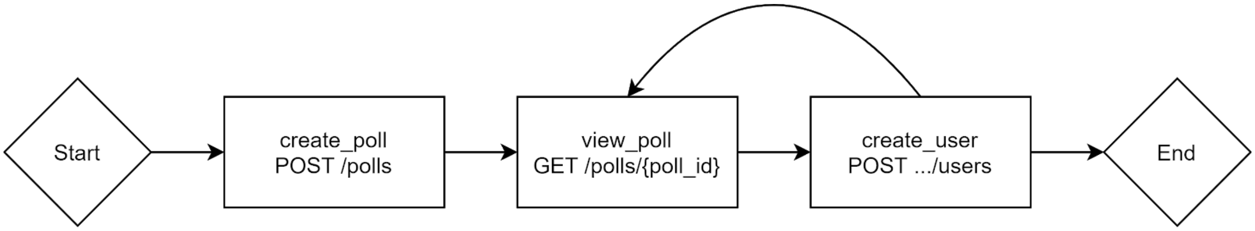
We defined the following sessions:

Session 1: Create Poll



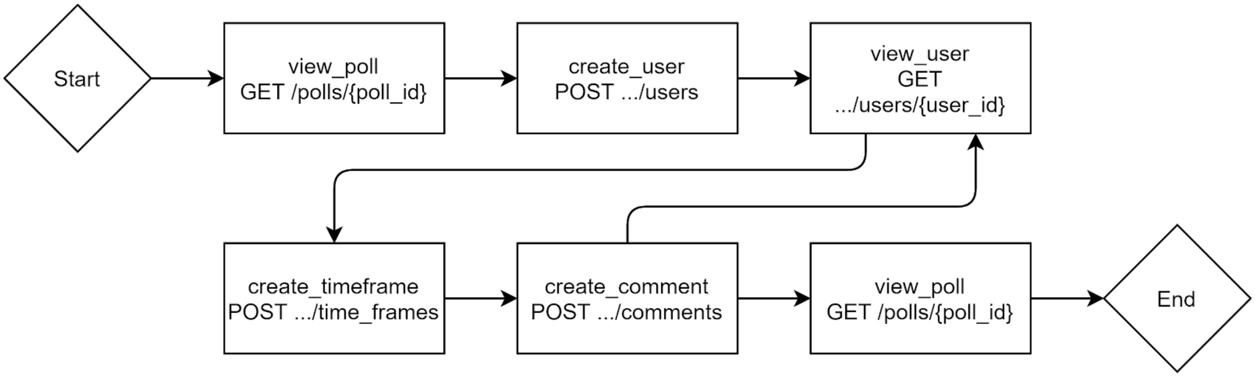
This flow represents a user's entry point when using the application. After creating a poll with their event details, the user is redirected to the poll's page. Using the link to the poll's page, the organizer can send the link to participants who can add themselves as users and indicate their availability.

Session 2: Populating a poll with users



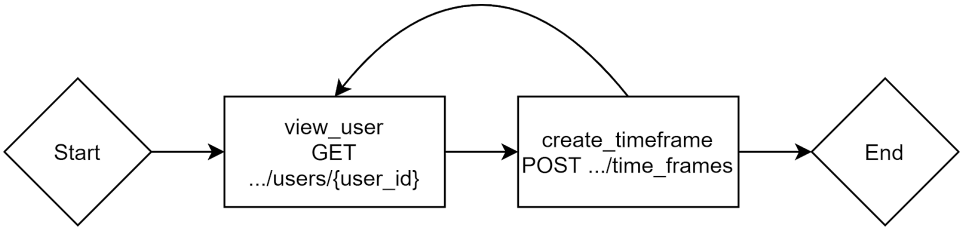
The scenario represents an organizer who created individualized pages for each participant so that they can send a specific link to each participant. This also tests a similar kind of load that involves many users adding themselves to a poll right after a link is sent out.

Session 3: Create a user within a poll and populate their availability



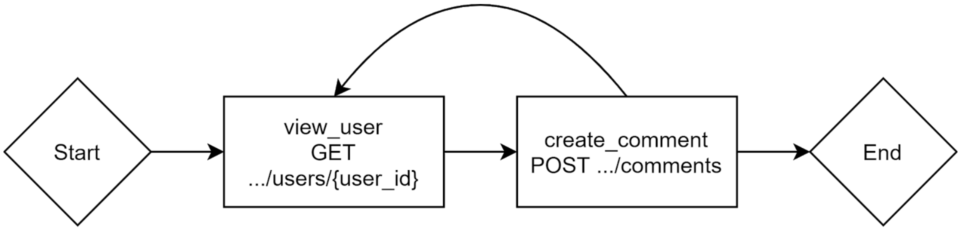
This scenario represents the most common way a user is driven to our application. They are sent a link to a poll by a friend or peer who is organizing an event, and asked to enter their availability. They will start on the page for a poll, and will add themselves to it and provide information about when they are free.

Session 4: An existing user enters their availability



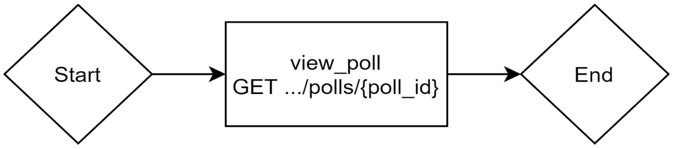
This scenario represents an edge case, where availability has changed and a user must amend their information. Another way this scenario might occur is if an organizer created user pages for everyone they are inviting.

Session 5: A user enters comments



Comments allow users to enter details about their availability. Users are able to view the comments they’ve entered from their user page. All the associated comments for a poll are visible from the poll’s main page.

Session 6: A user loads a poll



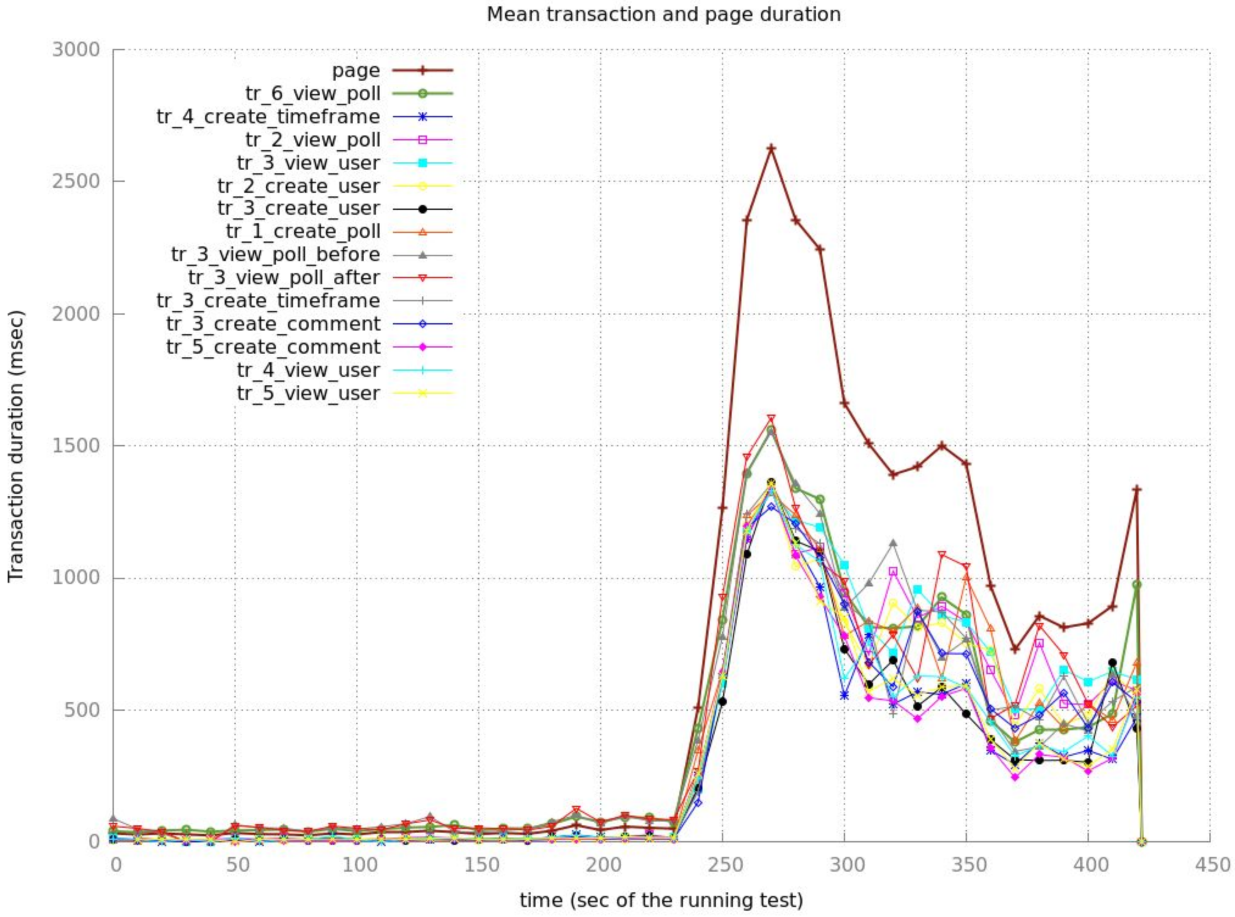
This is one of the most common scenarios, which involves a participant or organizer viewing a poll to find the best times to have an event. They can also view other participants' availability. When running a poll, an organizer will regularly check how many people have filled out the poll and consider which times are the most optimal.

4.4 Baseline:

We ran baselines against ec2's m5.large single instances for both the application and database servers on AWS ElasticBeanstalk. The database was also seeded as described in the Seeding section.

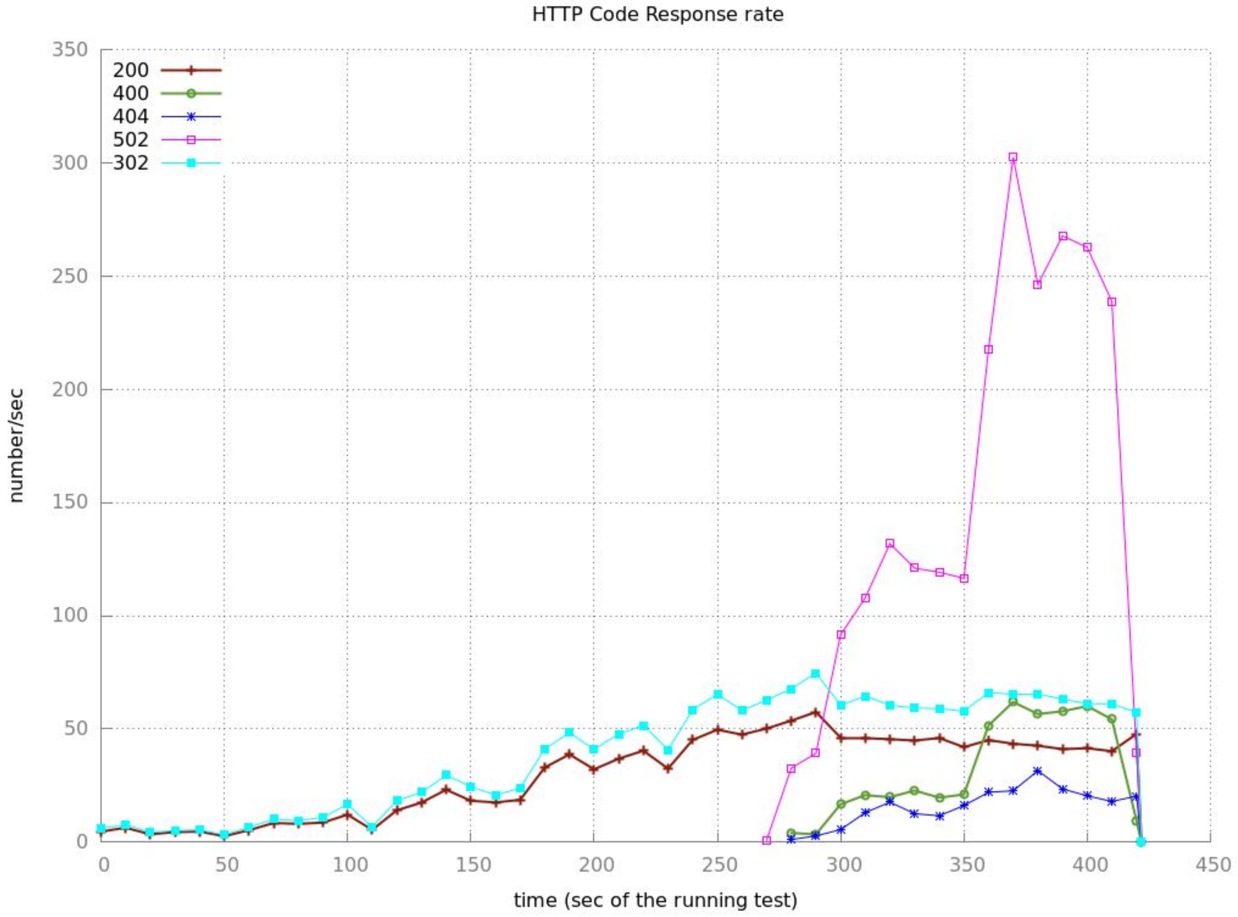
We test with 7 arrivals phases, each phase lasting for 60 seconds. We start with 2 users per second in the first phase and double the number of users in each subsequent phase; hence, ending with 128 users per second at the end of the test. In each phase, an arbitrary user performs one of the six sessions described in the previous section. The exact session performed among the six sessions is sampled randomly through a predefined weighting distribution on the sessions.

We defined weighting based on our expectations of how a real user would use this application. When using similar applications, such as when2meet, we usually fill out our availability once and check back multiple times for updates to schedule a meeting. So, we expect a user to more frequently view a poll (Session 6) to see optimal times rather than creating or modifying a poll. As such, we weighted this read-heavy workflow with 10x the weight of all other flows.



After running our baseline load test, we obtained the following results. The transactions in the graph are labeled with the session number described in the previous section. Looking at the mean transaction times, we saw that the *view_poll* transactions consistently took longer than other transactions. It is difficult to see in the graph due to the scale, but for the first 200 seconds, the *view_poll* transactions took ~40 ms while all other transactions stayed around 10-20 ms.

At ~240 seconds, the start of phase 5 (arrival rate 32), we saw a large increase in mean transaction time. This is quickly followed by a decrease in mean transaction time most likely due to the large increase in HTTP error codes shown below.



When looking at the HTTP response codes over time, we can see mostly 200 OK and 302 Redirect codes up until 270 seconds. We have a mix of 302s and 200s because several of our workflows create a resource and are directly redirected to it (ex. Creating a user).

At ~270 seconds, in phase 5 (arrival rate 32), we start seeing a sudden upshot in 502 errors. This is shortly followed by an increase in 4XX errors. Finally, at the data points from 360 seconds, when phase 7 (arrival rate 128) begins, we see huge increases in 502 and 4XX errors.

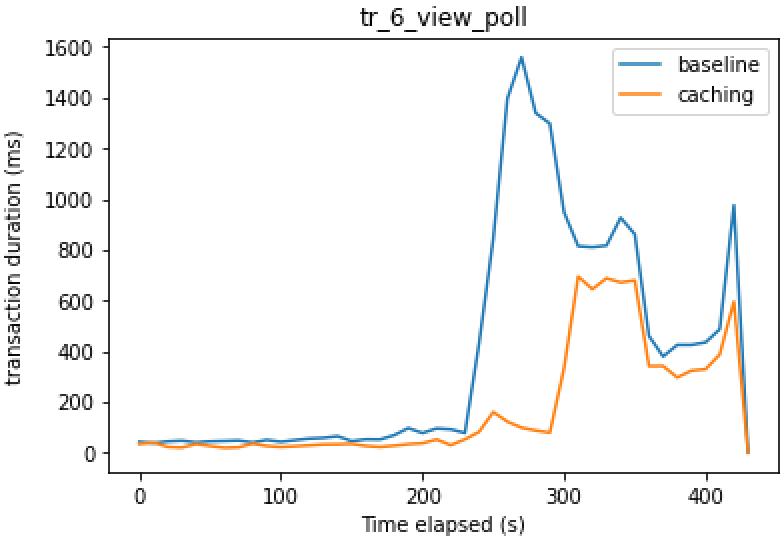
We believe that our 4XX errors are mostly a result of the 5XX errors, as we are trying to redirect to resources that the database does not have due to failed creations. As such, this data tells us that we are mostly being database throttled. At higher throughputs, our database is not able to keep up, and is returning 502 upstream server errors to the client.

5. Optimizations:

Based on the baseline results from our load tests, we can see that loading the poll data takes significantly longer compared to other requests. We believed that these results were due to bottlenecks when making requests to the database server. In order to address this potential issue, we implemented server-side query optimizations that would reduce the overall load on the database.

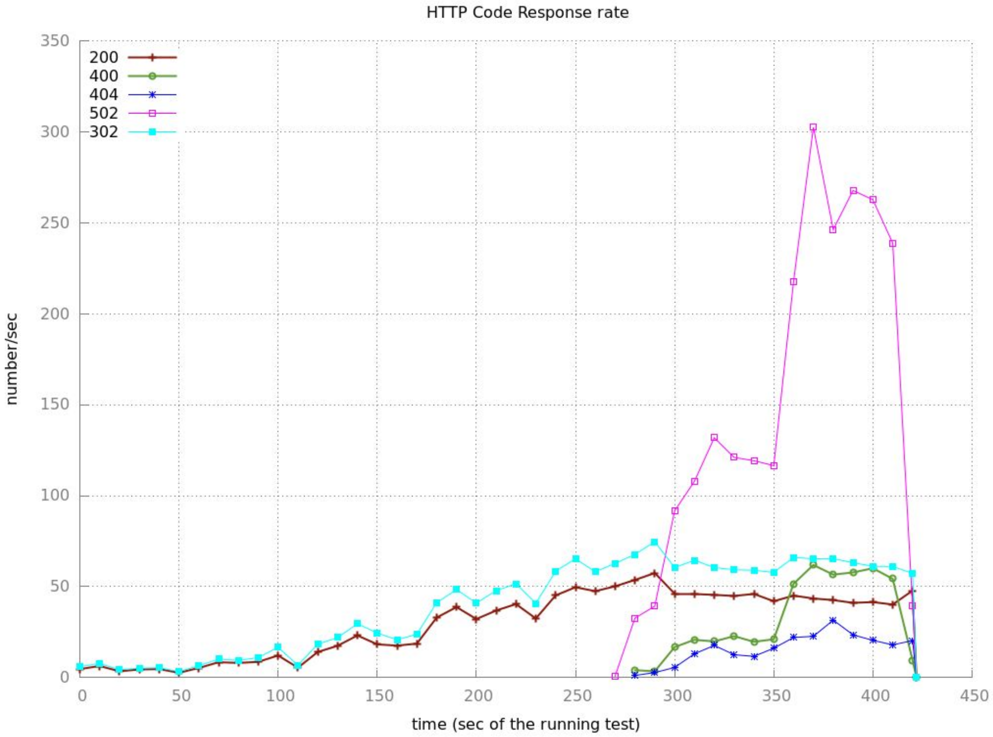
5.1 Caching

One of the first optimization strategies we looked into was server-side caching. The logic for doing so was straightforward, as caching the poll views and associated partials would reduce the number of necessary requests. Additionally, because our optimal time algorithm requires expensive computation, those results were cached as well. This all occurs on the view poll page. Because of the structure of our models, we used Russian Doll caching to store the view data in an in-memory cache on the application server. The size of our in-memory cache was 1024 megabytes.

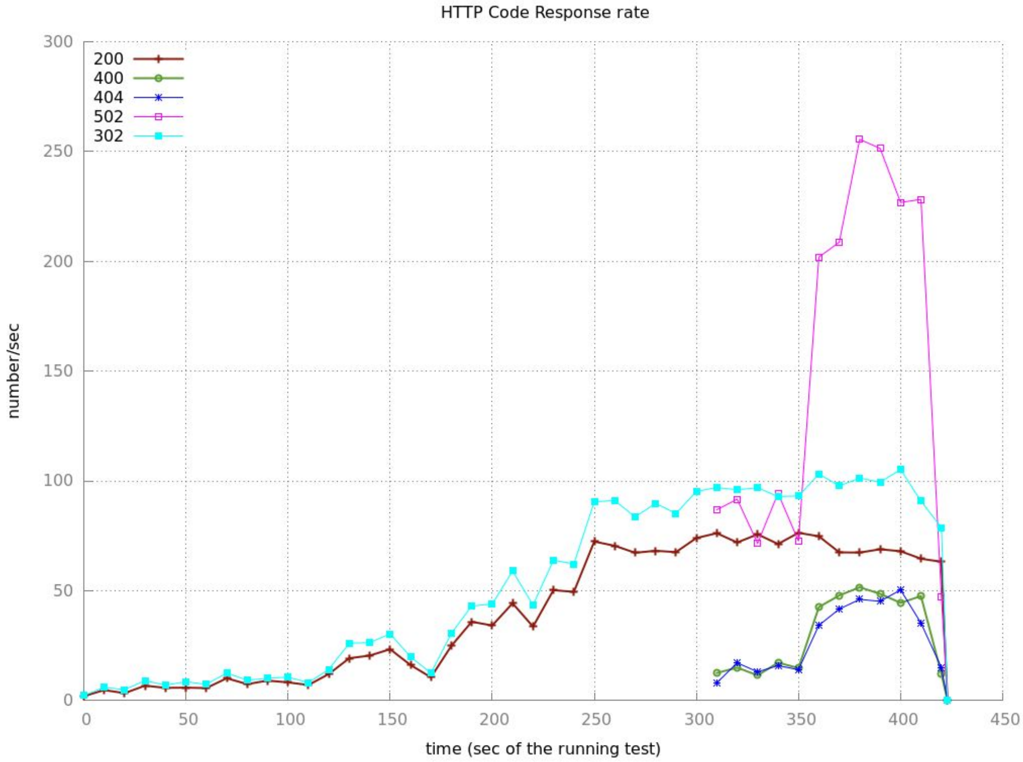


Initially, the maximum duration for a transaction occurred at around ~270 seconds into running the tests with a transaction time of around 1600 ms. The following test shows the results of caching the poll data, view partials, and optimal timeframes. The transaction times significantly improved by nearly two-fold, with the view poll operation reaching a maximum transaction time of ~1000 ms. Additionally, it took another 55 seconds until the server began to overload. Because the poll view page is where most of the caching occurs, these results match our goals of reducing the load at this point in the user flow.

Baseline - HTTP Response Codes



Cache Optimized - HTTP Response Codes

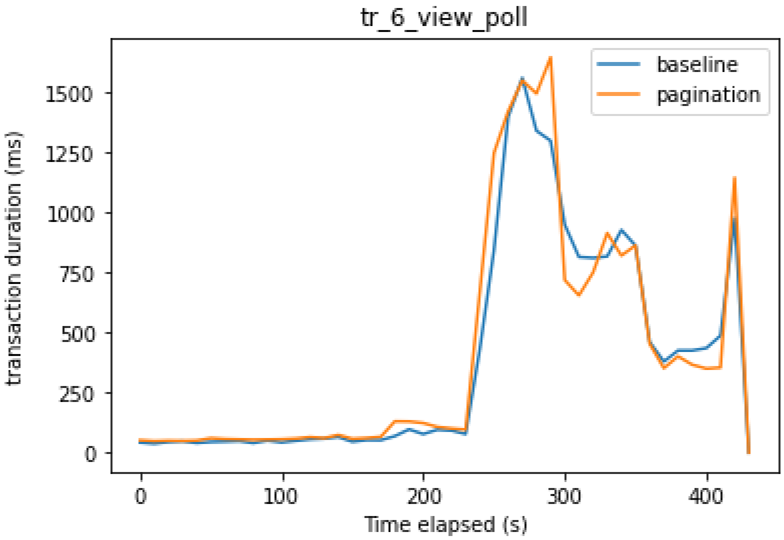


When comparing response codes across baseline versus cache optimized, we can see significant improvement. In the cache optimized chart, we don't see 502 errors until 310 seconds, a full 40 seconds and one phase further along than the same spike in the baseline. We do still see the same spike at 360 seconds when phase 7 of arrival begins. Even amongst the spikes though, the rate of errors is ~20% lower for cache optimized. We see a max error rate of ~300 5XX errors in our baseline, compared to ~255 after optimizing.

Caching had the largest effect on the *view_poll* transactions, as they are able to utilize the benefits caching provides. We saw a significant decrease in mean time for this transaction in the earlier phases of our load test, as it dropped from ~40 ms to ~25 ms. In the latter phases, decreasing these transaction times contributed to lower transaction times and error rates overall. Because of these improvements, we believe caching is an effective means of optimization for our service.

5.2 Pagination

The aim in pagination is to decrease the volume of data queried from the database in an individual query. Additionally, pagination also decreases the amount of time required to render long lists fetched from the database. We paginate the users and the comments, which are displayed on the poll view page to show five users and five polls per page.



When comparing the baseline versus pagination optimized transaction durations, we see little to no improvement. Both graphs show spikes in transaction time starting at ~240 seconds. We see similar spikes to around 1500ms for the most expensive transactions.

We can see that improvements in both transaction times and time into failure is minimal when using pagination. The volume of data returned upon requesting users and comments for a poll is not heavy enough for pagination to yield any improvements since realistically we expect about 10 users per poll and 10 timeframes per user.

5.3 N+1 Query Optimizations

N+1 query optimization is a common optimization applied to ORMs to mitigate the N+1 select problem. This problem refers to the naive approach the ORMs take when selecting from tables that have a one-many relationship. For instance, in our web application, a poll can have many users. Each user can have multiple timeframes that they are available for within the poll and multiple comments about the poll. In the page that displays a particular poll's details, the users belonging to a poll along with their timeframes and comments are retrieved from the database.

A naive way to do this would be to first fetch all users belonging to a poll, fetch the timeframes for each such user, and then fetch comments for each such user. This requires N+1 select queries for each user depending on the number of timeframes they add to a poll. Two such scenarios for our application are shown below.

Original queries

Scenario 1:

Accessing timeframes for each user in a given poll while (i) computing optimal times and (ii) rendering the polls page. This occurs at '/polls/<poll-id>'.

```
SELECT "polls".* FROM "polls" WHERE "polls"."id" = $1 LIMIT $2
SELECT "users".* FROM "users" WHERE "users"."poll_id" = $1
for user_id:
      SELECT "time_frames".* FROM "time_frames" WHERE
"time_frames"."user_id" = $1
```

Scenario 2:

Accessing the users corresponding to each comment in a given poll while rendering the polls page. This also occurs at '/polls/<poll-id>'.

```
SELECT "comments".* FROM "comments" INNER JOIN "users" ON
"comments"."user_id" = "users"."id" WHERE "users"."poll_id" = $1
```

```
for user_id:
      SELECT "users".* FROM "users" WHERE "users"."id" = $1 LIMIT $2
```

Optimized queries

An optimization here would be to reduce the number of select statements to 1 by making use of the SQL IN operator. This reduces the number of database queries. In Rails this is achieved by using the ‘includes’ query method. The optimized SQL queries for both the scenarios are shown below.

<u>Scenario 1:</u>

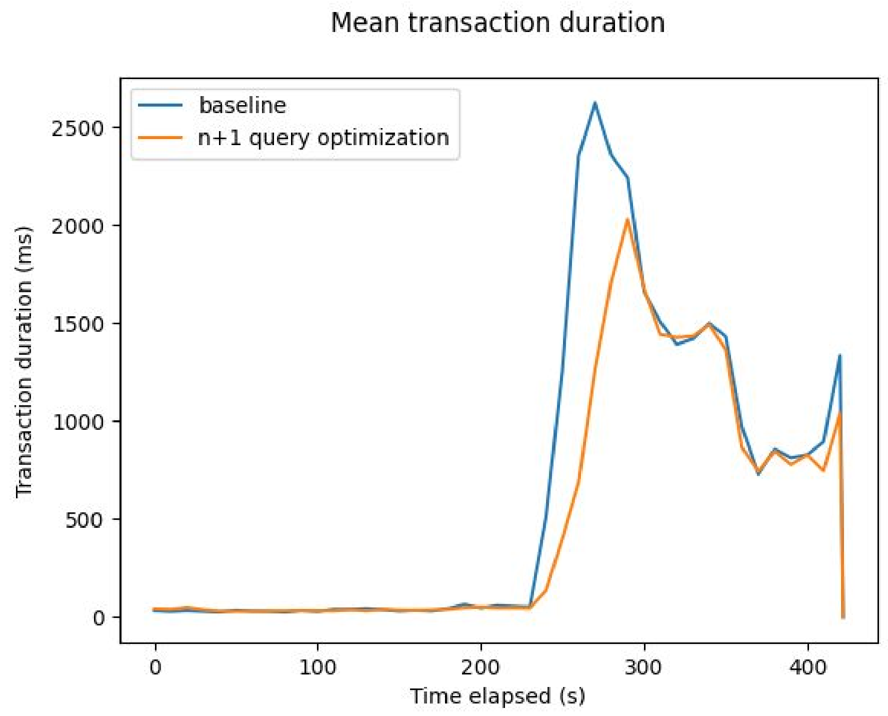
For both (i) and (ii).

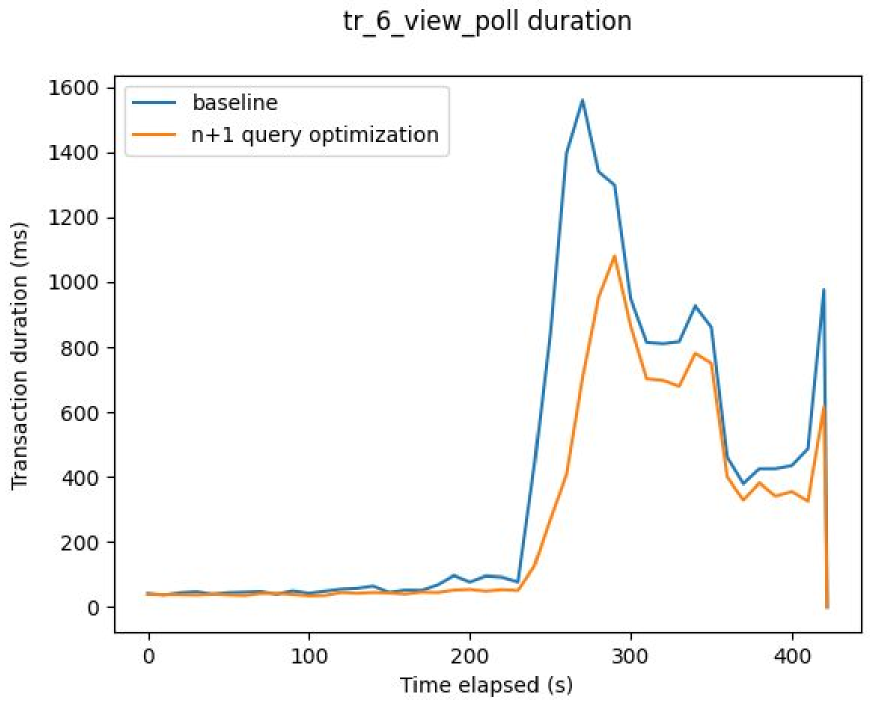
```
SELECT "polls".* FROM "polls" WHRE "polls"."id" = $1 LIMIT $2
SELECT "users".* FROM "users" WHERE "users"."poll_id" = $1
SELECT "time_frames".* FROM "time_frames" WHERE "time_frames"."user_id" in
($1, $2, ...)
```

<u>Scenario 2:</u>

```
SELECT "comments".* FROM "comments" INNER JOIN "users" ON
"comments"."user_id" = "users"."id" WHERE "users"."poll_id" = $1
SELECT "users".* FROM "users" WHERE "users"."id" in ($1, $2, ...)
```

We now present our results by means of two plots. Both help us differentiate between mean transaction duration times before and after the optimization. The first one refers to the mean transaction time for all transactions combined and the other one refers to the mean transaction time for just the view_poll transaction.





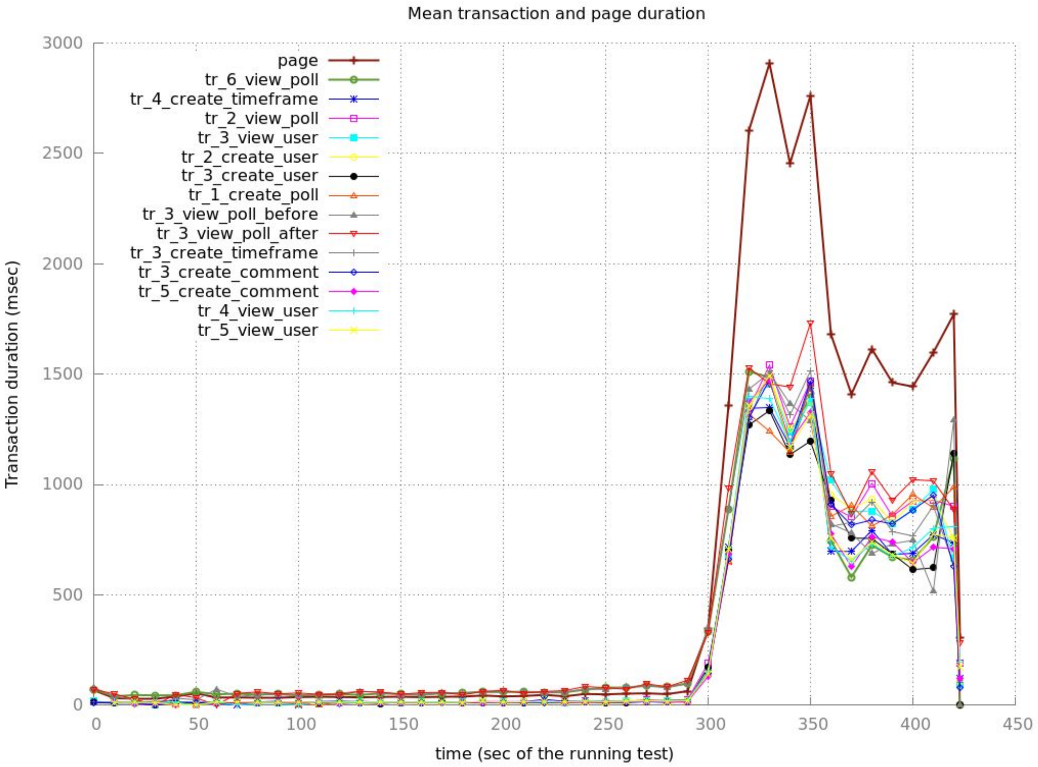
We observed that n+1-query optimization helped reduce our peak transaction times by nearly a factor of 20%. It can be observed that the view_poll transaction has higher gains because all the 3-places that we applied this optimization were relevant to this transaction. The gains were as expected because Rails by default uses naive queries that could be optimized.

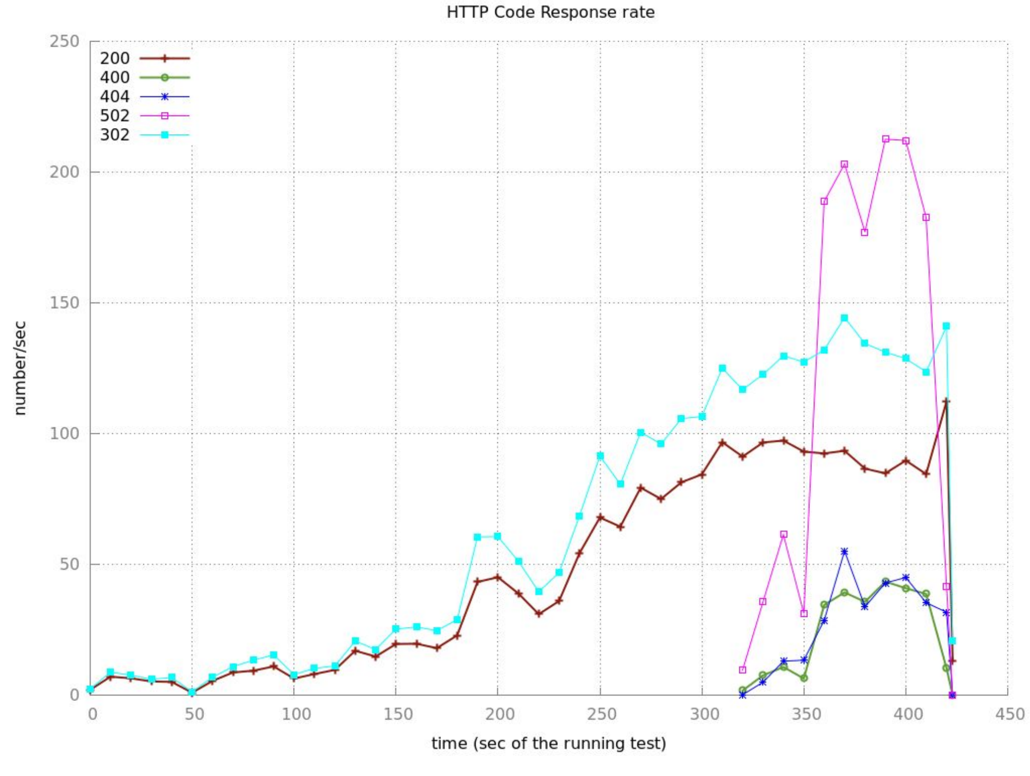
Indexing

We observed that our web application runs SELECT and JOIN queries when we want to view the details of a particular poll. In order to minimize the cost of these queries, we realized that we might have to index the columns being used in the queries. In particular, the columns used were foreign keys. However, Rails adds an index to the foreign keys by default.

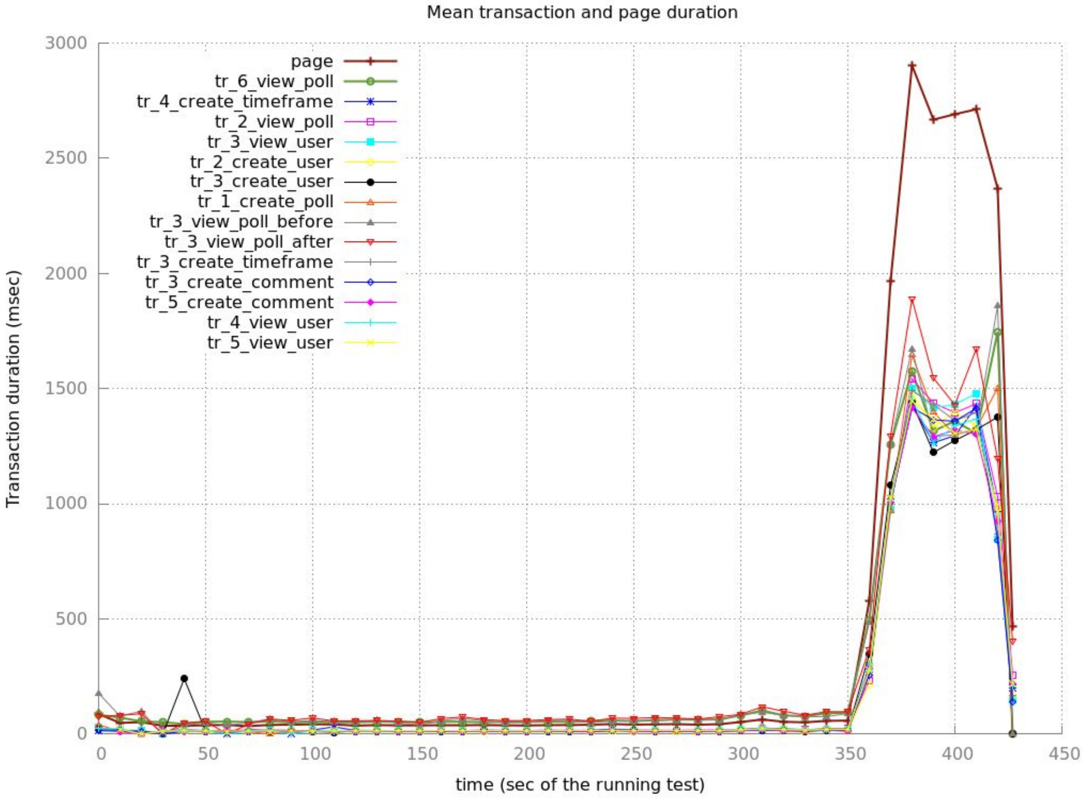
5.4 Horizontal Scaling

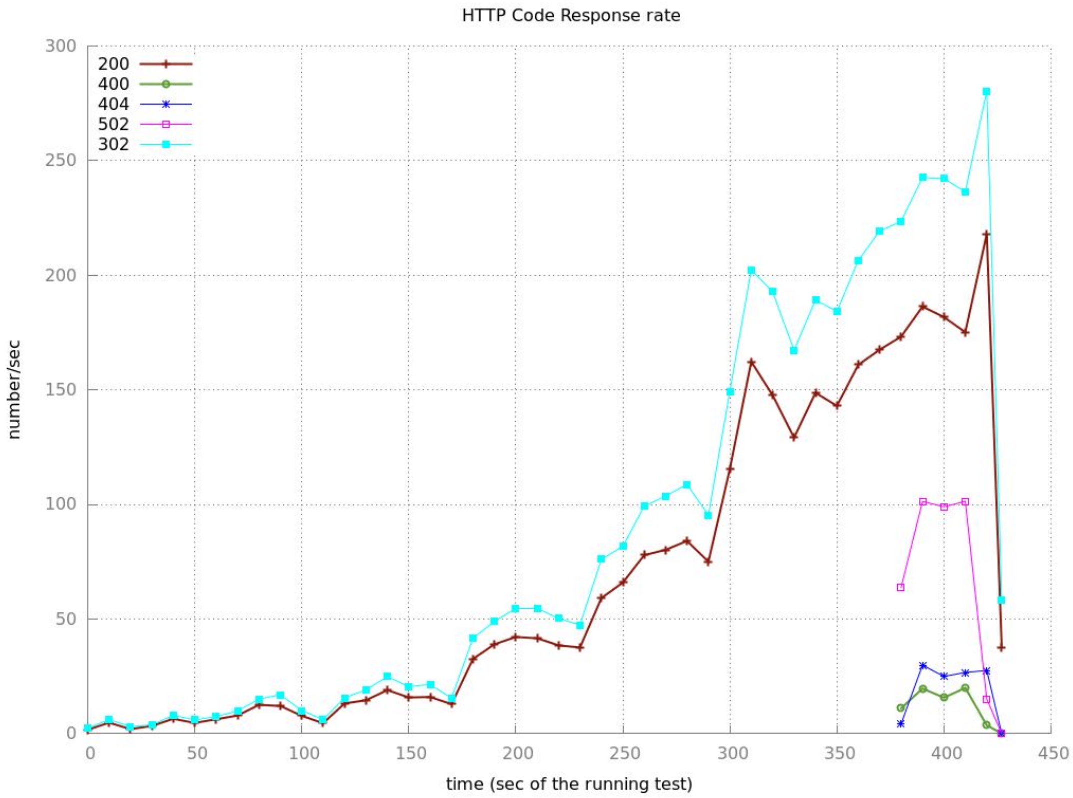
Two Servers:



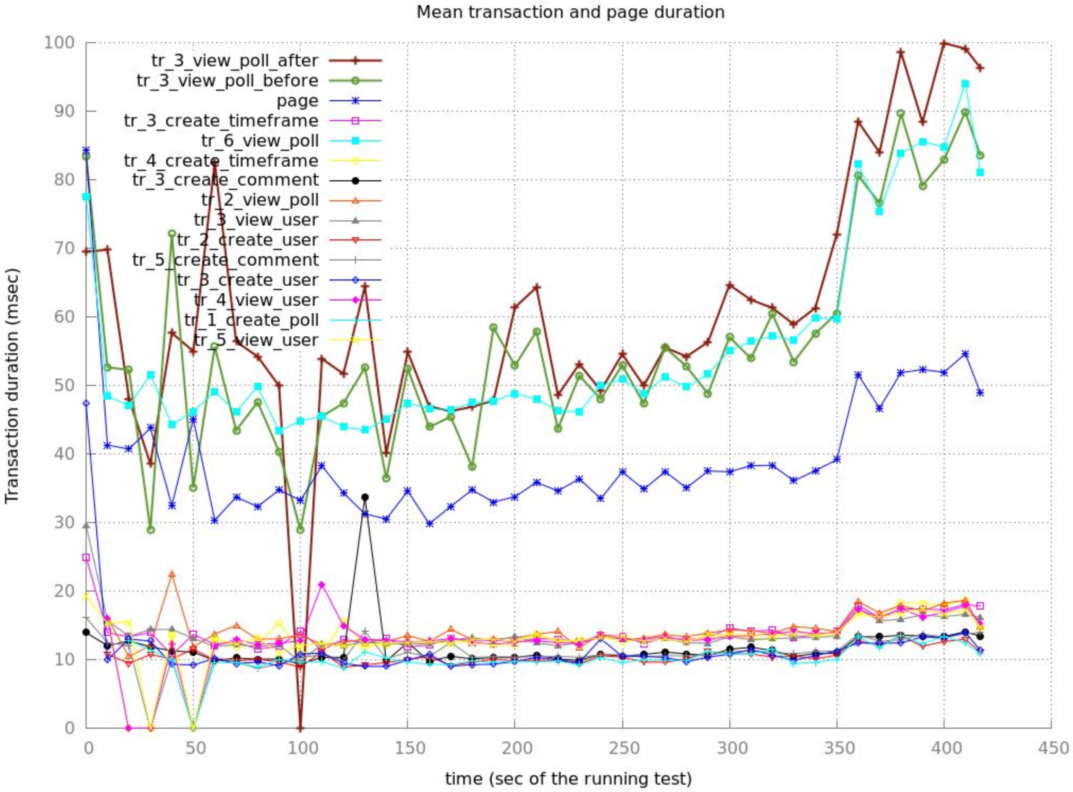


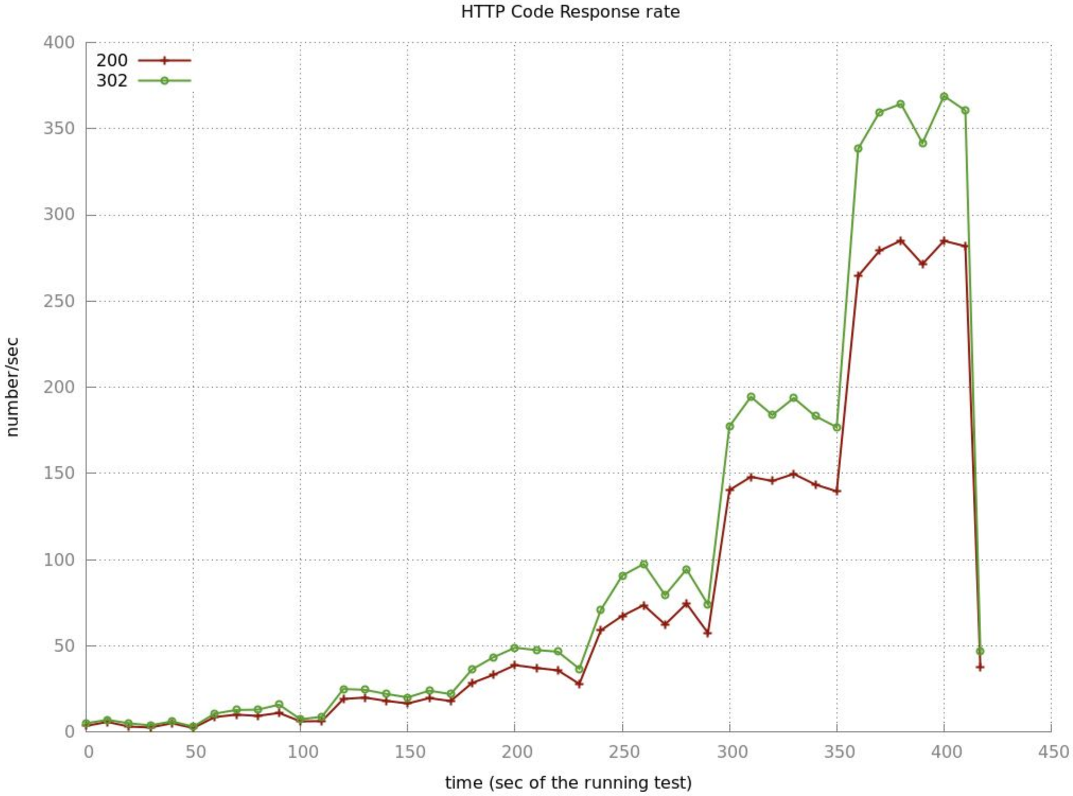
Four Servers:



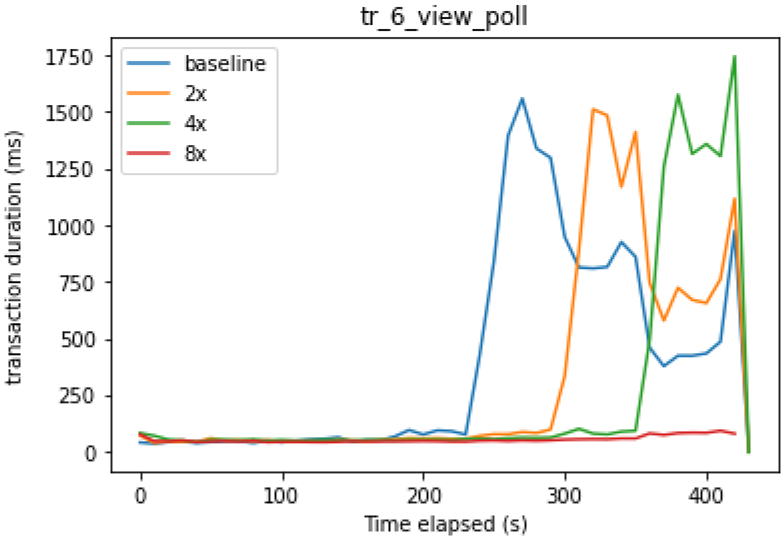


Eight Servers:





Summary graph:



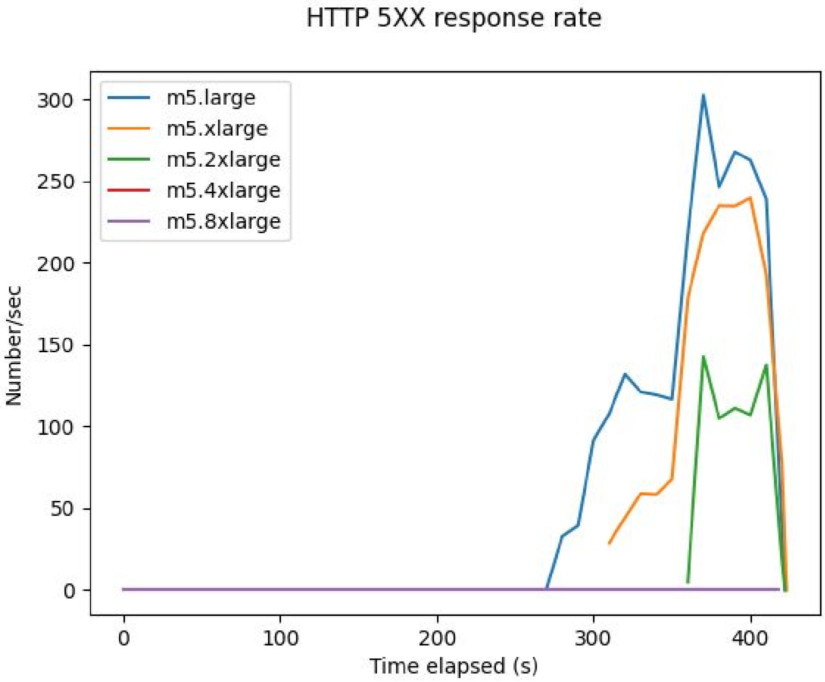
As a final measure, we spent some time looking at horizontal scaling on m5 instances, as this reduces overall bottlenecking. This was one of the most effective “optimizations”. We see at two servers that we spike in transaction time at 300 seconds, which would by phase 6. By four servers, we see the similar spike at 360 seconds, or phase 7. Finally at eight servers, we see some small increases in response time at 360 seconds, but significantly smaller than previous. We also saw a reducing number of 4XX and 5XX response times as we increased the number of servers.

5.5 Vertical Scaling

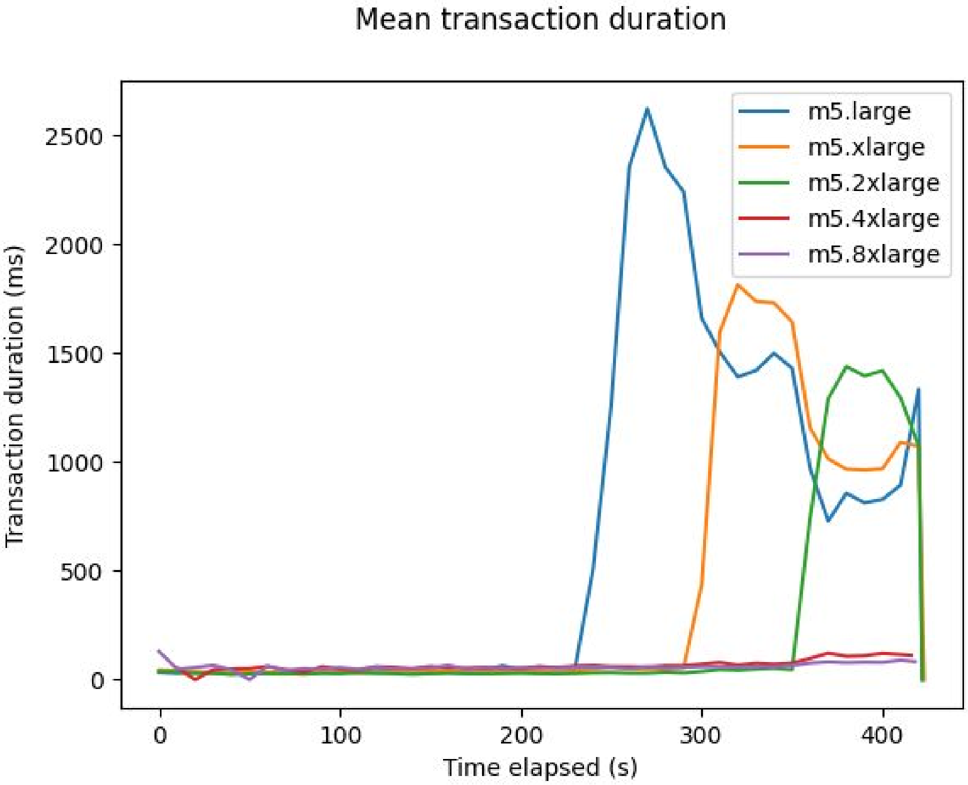
Cloud Vertical Scaling refers to adding more CPU, memory, or I/O resources to an existing server, or replacing one server with a more powerful server. Amazon Web Services (AWS) vertical scaling can be accomplished by changing instance sizes. AWS cloud services have many different instance sizes, so scaling vertically is possible for everything from EC2 instances to RDS databases. For this project, we chose to go with m5 instances.

The instances that we chose are m5.large, m5.xlarge, m5.2xlarge, m5.4xlarge, and m5.8xlarge for both the EC2 instances and the database.

Initially there are many HTTP 5XX responses but as we increase the instance sizes, the 5XXs decrease each time and finally disappear in m5.4xlarge and m5.8xlarge. This can be observed in the figure below.



We also observed that as we increased instance sizes, the peak mean transaction duration time for all transactions decreased substantially. Also, using m5.4xlarge and m5.8xlarge instances stabilized response times. This can be observed in the following figure:



We note that just using an m5.4xlarge instance helps us get nearly 100% speedup with zero HTTP 500 responses. Also observe that the mean transaction duration times of m5.4xlarge and m5.8xlarge instances are very similar and the HTTP 500 responses are absent. This helps us conclude that using m5.4xlarge instances are sufficient and we need not scale more vertically (for the load we tested). In conclusion, we can view m5.4xlarge instances as a cut off point for vertically scaling our application.

5.6 Cost Analysis

As a service, we must consider the cost per user of running our service. To begin, our baseline of a m5.large instance costs $0.096 per hour.

We define failure as a sizable increase in response times on transactions. Costs are based on current ec2 pricing.

Type	Failure Time	Phase/Arrival Rate	Cost
m5.large	240	5/32	$0.096
2x m5.large	300	6/64	$0.192
4x m5.large	360	7/128	$0.384
8x m5.large	360ish?	7/128+	$0.768
m5.xlarge	300	6/64	$0.192
m5.2xlarge	360	7/128	$0.384
m5.4xlarge	360ish?	7/128	$0.768
m5.8xlarge	N/A	7/128++	$1.536

Based on response times, it seems that horizontally scaling (at least to the point we tested), works slightly better than vertically scaling. This is most obvious in comparing the 8x m5.large versus m5.4xlarge, where the horizontal scaled instances provide max response times of 100ms, compared to 120 ms for vertically scaled. Up until that point, we see that vertically scaled servers have better response times during failure, but as a whole, both horizontal and vertical scaling have similar metrics per price point.

6. Conclusion

In this project, we gained the skills to build [cs291] **scalable internet services**. This project required us to build a full-stack web application using the Ruby on Rails framework. We designed models, built a basic user interface, architected a service that would provide value to users like any other fully-scaled application would. Learning how to load test in Tsung gave us insight into how larger-scaled applications would require reiteration and refinement in order to adapt to a growing user base and evolving user needs. By using modern tooling such as AWS ec2 and Elastic Beanstalk, we were able to deploy and test in an environment similar to that of industry.

To approach optimization strategies, we increased the load on our application in increments in order to find which parts of the user flow bottlenecked our overall performance. After analyzing our tests and implementing optimization techniques, we found that caching and N+1 optimizations provided significant gains while pagination offered minimal improvements.

For the future, we would love to clean up our application's user experience and build flows that would more accurately model use cases and user flows. We would also love to try building this setup on a non-relational database to allow for easier horizontal scaling. Finally, we would optimize our database by archiving old user data to reduce strain.

Thank you for coming to our TED talk.

∞ Fun Facts!

- Ruby has a date class, a time class, and a datetime class. I am still not sure what the difference is.
- The average age of our team is slightly over 22. We have one member that is legally unable to buy alcohol
- The pagination library we used, named Kaminari, means thunder in japanese.
- Our application was almost named when3meet! What a travesty that would have been.
- I was disallowed from submitting this report in Comic Sans
- Our team mascot is Erin