

Linksln

Tinder for gamers



Amy Tu
Junhong Wang
Omar Tleimat
Cheuk Yin Phipson Lee

1. Introduction	6
1.1 Matches	7
1.2 Conversations	9
1.3 Events	10
1.4 Profiles	12
1.5 Settings	13
2. Development	15
3. Application Architecture	20
4. Load Testing and Scaling	23
4.1 Vertical Scaling	24
4.1.1 t3.micro	25
4.1.2 t3.large	27
4.1.2 t3.xlarge	28
4.1.4 t3.2xlarge	30
4.1.5 m4.4xlarge	31
4.1.6 m4.16xlarge	33
4.1.7 Summary	34
4.2 Horizontal Scaling	35
4.2.1 t3.micro x 1	36
4.2.2 t3.micro x 2	37
4.2.3 t3.micro x 4	39
4.2.4 t3.micro x 8	40
4.2.5 t3.micro x 16	42
4.2.6 t3.micro x 32	43
4.2.7 Summary	45
4.3 Pagination	45
4.4 Processes/Threads	49
4.4.1 Process x 1 & Thread x 1	50
4.4.2 Process x 8 & Thread x 1	52
4.4.3 Process x 1 & Thread x 8	53
4.4.5 Summary	54
4.5 SQL Optimization	54

4.6 Client-side Caching	58
4.7 Server-side Caching	61
4.8 Read-Slaves	66
4.8.1 Master database x 1	69
4.8.2 master database x 1 & read-slave x 1	70
4.8.3 Summary	72
5. Further Development	72
5.1 Database Sharding	72
5.2 Improving Events	73
5.3 Real-time Messaging	73
5.4 Push Notifications	73
5.5 Linking External Accounts	73
5.6 Combining All Optimizations	74
6. Conclusion	74

1. Introduction

The onset of gaming and eSports has risen over the past decade. Yet, with the influx in users playing games ranging from Fortnite to League of Legends, many still struggle to build friendships or romantic relations with people in the same gaming ecosystem. With Linksin, we bridge the voice between social networking and gaming, to offer gamers a platform where they can easily match, converse, and play with fellow gamers they never would have met otherwise. Using a Tinder-styled card-swiping mechanism, Linksin, which was built using Ruby on Rails and later deployed on Amazon Web Services, is the networking app of the future.

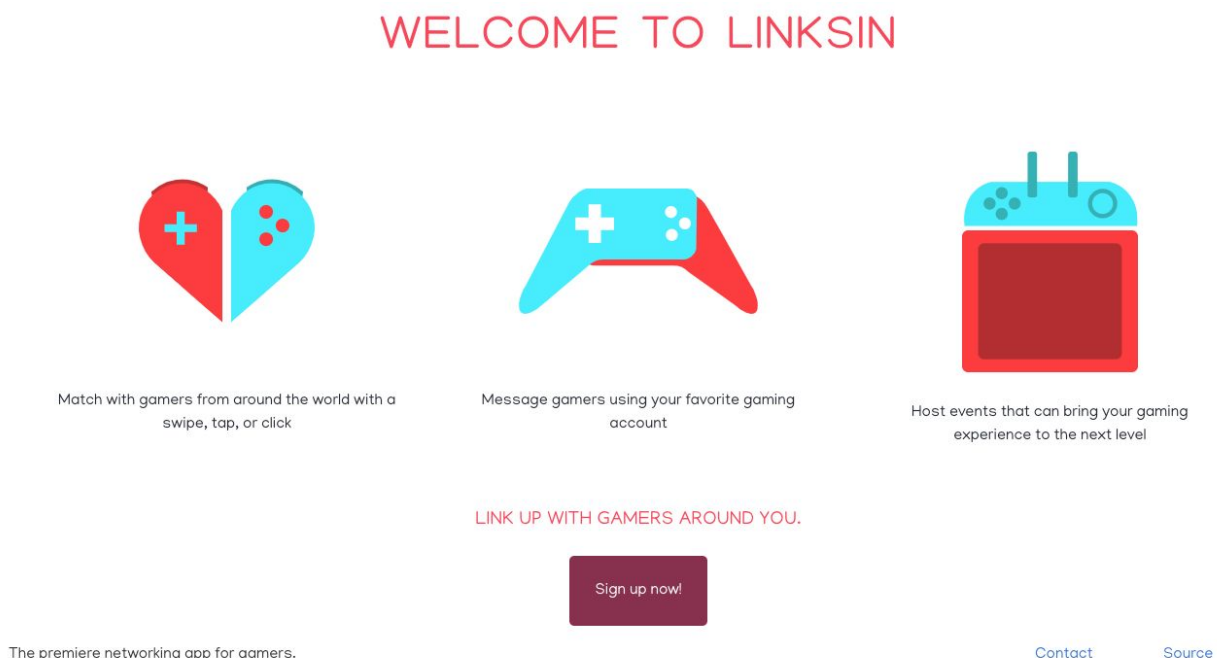


Figure 1.0.0. Process of swiping and matching user profiles on Linksin

1.1 Matches

Much like Tinder, Linksin focuses on matching gamers through a card-swiping interface. Upon signing up and logging in, gamers are immediately directed to the home page, where they are given a series of

cards, which feature profiles of other gamers who have signed up on the platform. By swiping left and right, or tapping on the “heart” and “cross” buttons below the app, users are given different methods of expressing their interest in a gamer (indicated by a smiley face or heart), or rejecting a gamer (indicated by a sad face or cross). The process of interacting with the cards is illustrated in Figure 1.1.0.

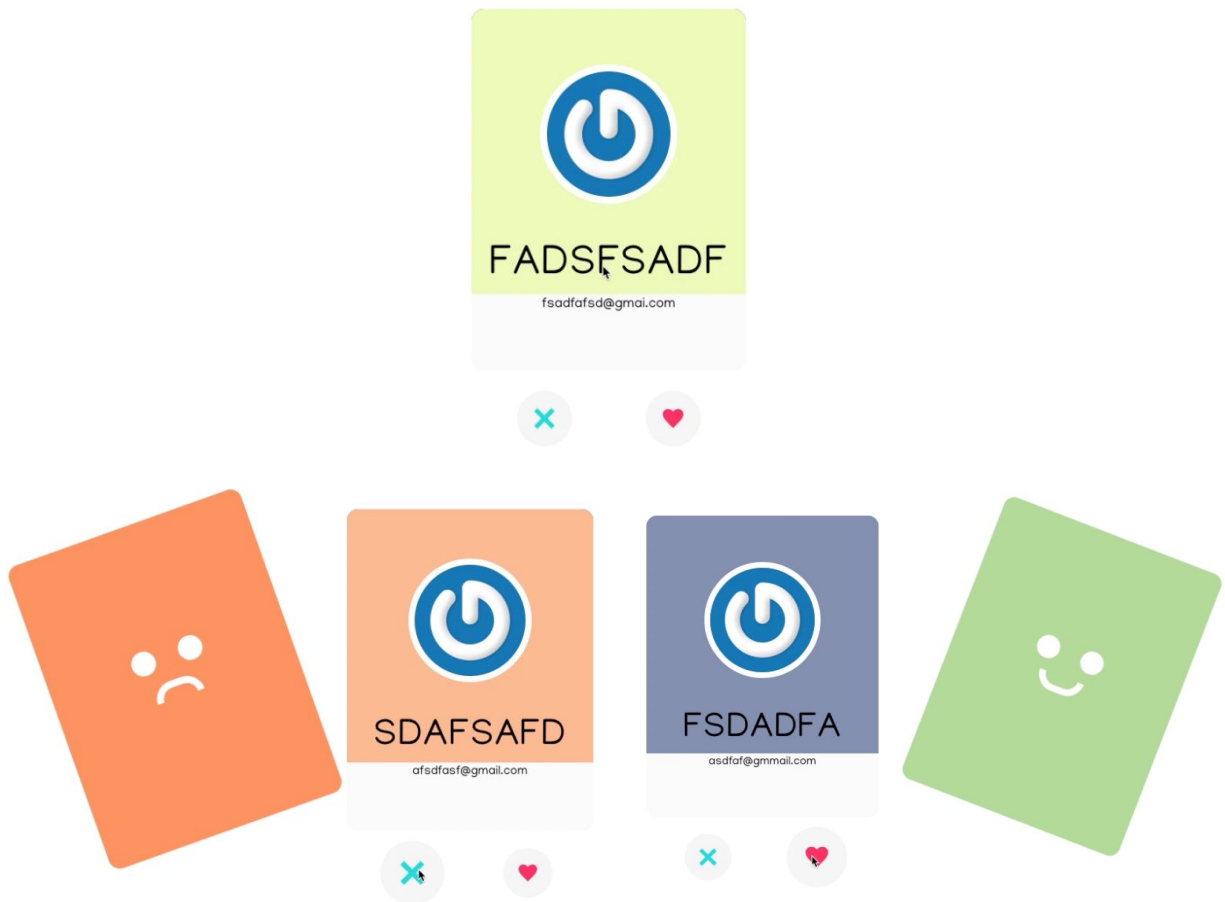


Figure 1.1.0 Process of swiping and matching user profiles on Linksln

Once a user has successfully swiped a card, data is sent to the app server indicating that the user has liked or disliked another gamer’s profile. If two users both ‘like’ each other, indicated by swiping right, then a ‘match’ event is detected. Matching enables users to message each other on Linksln, or schedule gaming events with other gamers. Users can only converse with or invite other gamers that they have matched with.

A high-level overview of the user control flow is illustrated in Figure 1.1.1. When a user first enters the page, he or she will be asked to create an account, or log into an existing account. If the account credentials are correct and valid, the user will then be directed to the primary card swiping page shown in Figure 1.1.0, where he or she will swipe cards containing gamer profiles. Once there is a match, the user will gain access to messaging and events for the matched user.

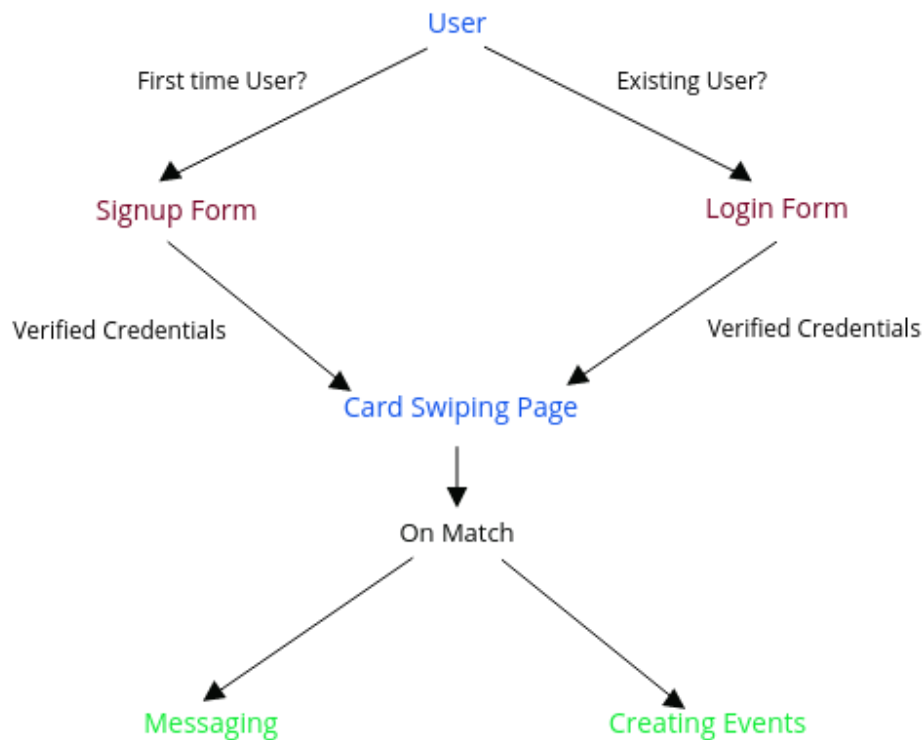


Figure 1.1.1. User Control Flow in LinkedIn

User can view who they matched with in Matches tab, where they can see details of who they matched with as shown in Figure 1.1.2.

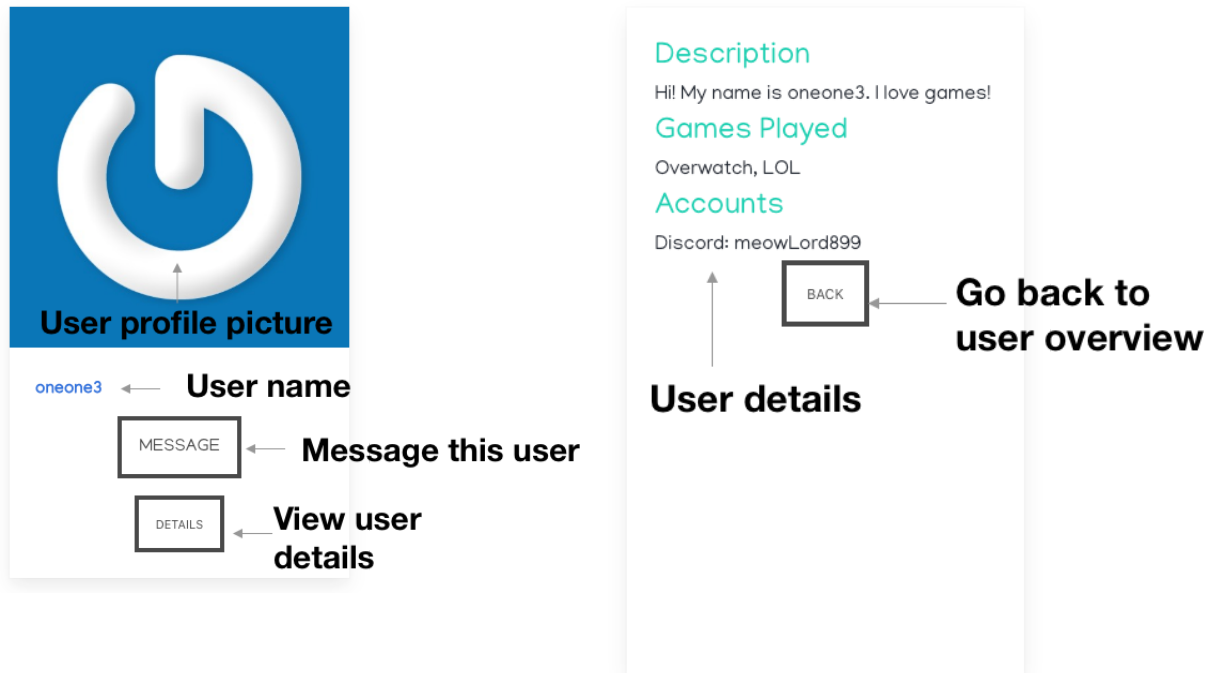


Figure 1.1.2. Annotated diagram of matches page

1.2 Conversations

Once two users have matched with each other, LinkedIn immediately directs them to a conversations page, where they can send messages to each other. Similar to pervasive messaging applications such as WhatsApp and Facebook Messenger, users must first type their message in the text input shown in Figure 1.2.0. They then send the message by pressing the “Send” button. All messages are updated and shared between the two users. For each conversation, the most recent message and its corresponding sender as displayed under the name of the match in the list of all matches.

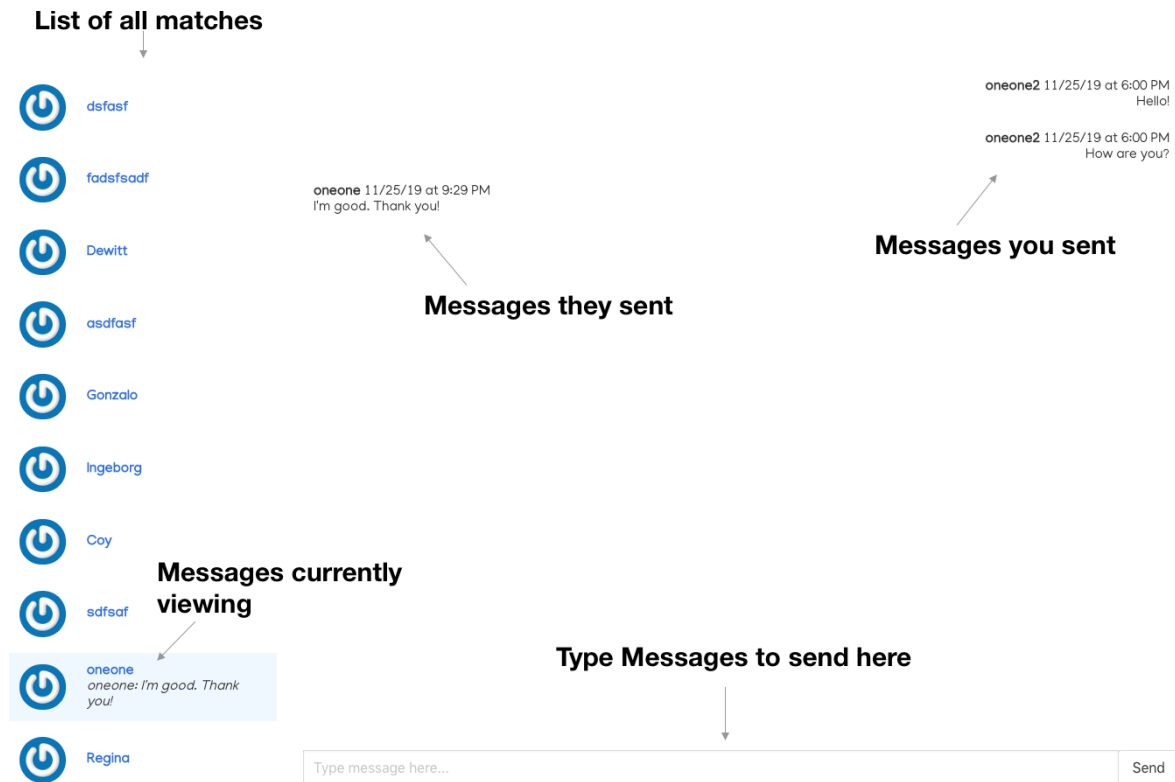


Figure 1.2.0. Annotated diagram of LinkedIn's messaging feature.

1.3 Events

In addition to messaging other users, LinkedIn also offers gamers the opportunity to schedule and setup events, so that they can play games with their new friends. In the events tab, shown in Figure 1.3.0, users can view currently registered events.

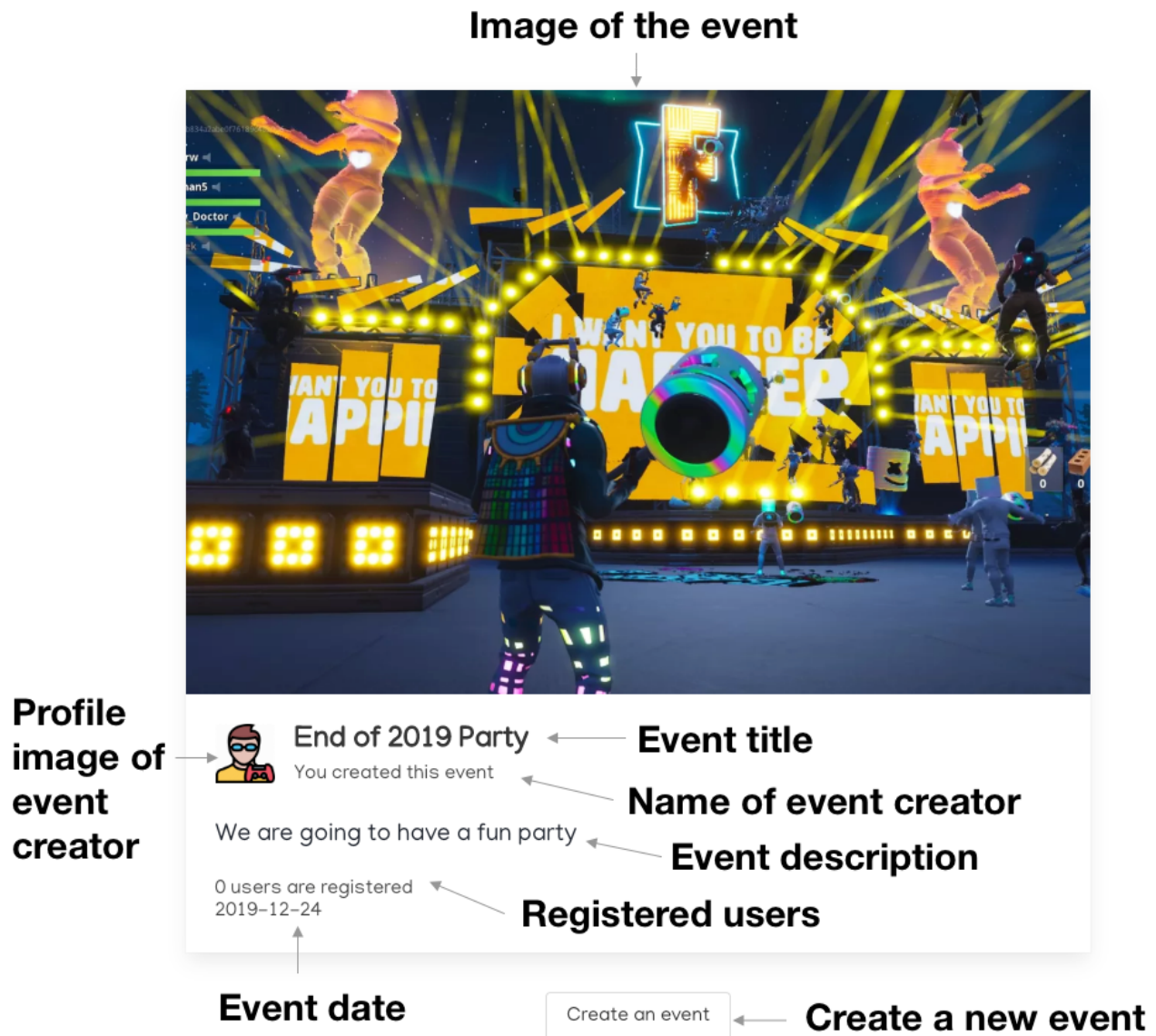


Figure 1.3.0. Annotated diagram of LinkedIn's event list feature.

When users click the "Create an event" button, they are redirected to an event creation page, and they simply have to input the name, date, time, and description of the event to create it, as shown in Figure 1.3.1. They can also invite gamers they have matched with to their event, by inputting the gamer's email.

CREATE EVENT


Name

Date

Time

Description

Invite friends


oneone3

← **Select friends to invite here**

Create Event

Figure 1.3.1. Annotated diagram of Linkn's event creation feature

1.4 Profiles

In addition to creating events, users can also update their profiles, shown in Figure 1.4.0. Upon signup, a user is redirected to the profile editing page to enter their information. By changing their profile description and profile picture, users can change how other gamers see them on a card. This also gives users more information to determine whether they like a gamer or not enough to play with him or her.

UPDATE YOUR PROFILE

Description

Games

Accounts

Save changes



change

Figure 1.4.0. Annotated diagram of Linkn's profile modification feature.

1.5 Settings

We allow users to update their account information including the password as shown in Figure 1.5.0.

UPDATE USER INFORMATION

Name

Email

Password

Password confirmation

Figure 1.5.0. Annotated diagram of LinkedIn's settings page

2. Development

LinkedIn was built using version control on git, agile development features with GitHub Issues, and travis CI for testing. Pair programming was also used at times due to individual computer issues, and for tackling more difficult issues. Figure 2.0. shows a more high-level timeline that we followed in order to create this working application.

LinkedIn Application Development Timeline

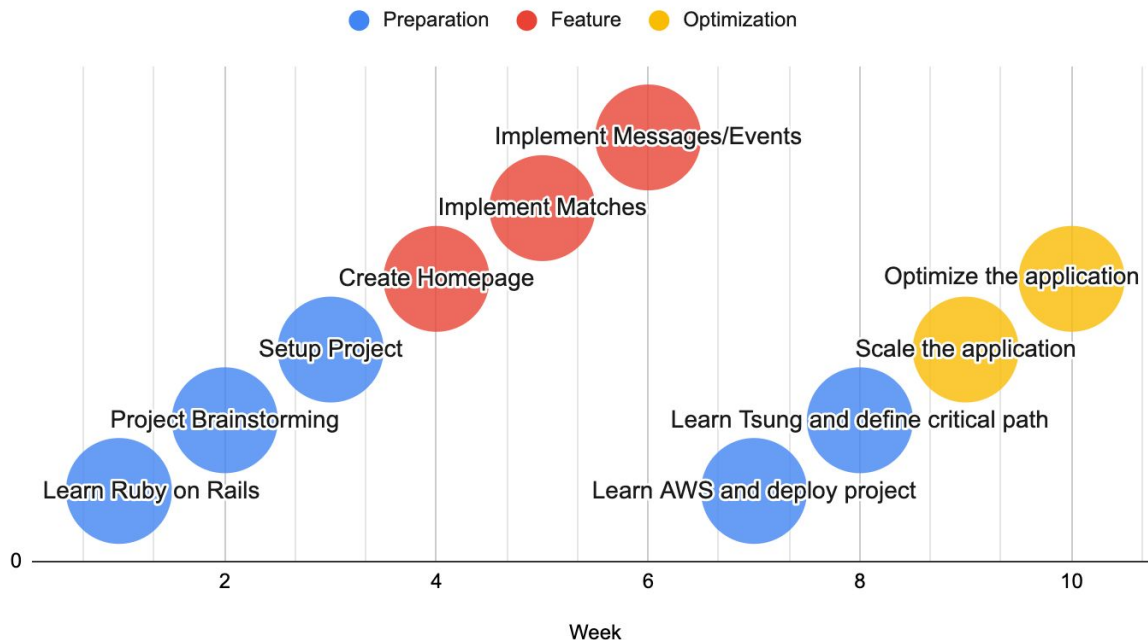


Figure 2.0. Timeline of the application development process

With version controlling on git, we were able to separate each of the features shown in Section 1 to a discrete branch. From there, we would make our own modifications and evaluate our progress in weekly meetings before merging our changes into the master branch. It also helped us help other group members out with aspects of the project they were struggling with, thus isolating feature-specific problems and keeping the project organized. Some branches of our project is shown in Figure 2.1.

All branches					
master	Updated 21 hours ago by loneone	✓	Default	Change default branch	
sharding-opt	Updated 13 hours ago by Phipson	✓	16 35	New pull request	
frag-client-server-cache	Updated 14 hours ago by Phipson	✓	17 38	New pull request	
sql-optimization	Updated 15 hours ago by ayemetoo	✓	16 28	New pull request	
memcache-new	Updated 18 hours ago by loneone	✓	17 40	New pull request	
load-testing	Updated 20 hours ago by loneone	✓	16 29	New pull request	
multi-process	Updated 21 hours ago by loneone	✓	16 50	New pull request	
read-replica	Updated 2 days ago by loneone	✓	16 33	New pull request	
pagination	Updated 3 days ago by EC2 Default User	✓	16 27	New pull request	
fragcache-clientcache-all	Updated 3 days ago by Phipson	✓	17 33	New pull request	
fragcache-clientcache	Updated 4 days ago by Phipson	✓	17 31	New pull request	
fragcache-all-n-user	Updated 4 days ago by Phipson	✓	17 23	New pull request	
fragcache-n-user	Updated 4 days ago by Phipson	✓	17 23	New pull request	
fragcache-conv	Updated 4 days ago by Phipson	✓	17 22	New pull request	
fragcache-events	Updated 4 days ago by Phipson	✓	17 21	New pull request	

Figure 2.1. Various branches of LinkIn

In addition, TravisCI was used to ensure that new commits or merges from feature branches to master would not cause our application to break. Whenever a pull request was opened to merge a feature into master, Travis would be used to make sure that the build would remain passing after doing so. The status of build by TravisCI is shown in Figure 2.2.

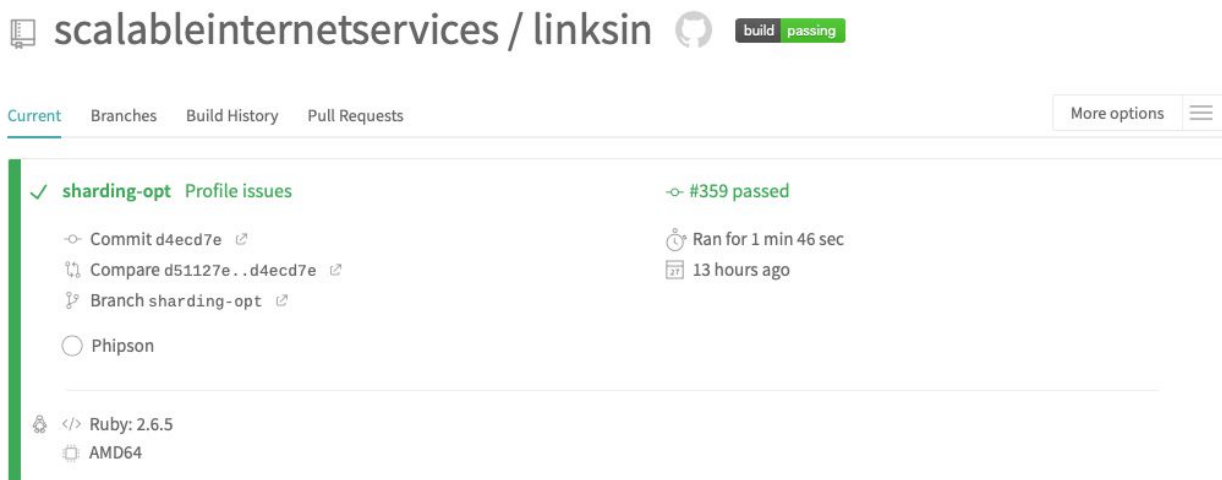


Figure 2.2. Build status of Linksin by TravisCI

To ensure that all members were held accountable and kept updated about group progress, we utilized the agile development methodology, which was implemented on GitHub Issues. Each issue we made contained a Scrum Story, which was later assigned to a specific group member(s), and was later updated the following week. A screenshot of the agile development process is shown in Figure 2.3.

<input type="checkbox"/>	3 Open <input checked="" type="checkbox"/> 61 Closed	Author ▾	Labels ▾	Projects ▾	Milestones ▾	Assignee ▾	Sort ▾
<input type="checkbox"/>	As a user, I would like to be able to edit my account details, view other user profiles, and not have others access things private to me #16 by Phipson was closed on Oct 25						1
<input type="checkbox"/>	As a back-end developer, I want to create an "Events" model so that users can view and join events #15 by omarTI was closed on Oct 28						
<input type="checkbox"/>	Add temporary favicon #13 by ioneone was closed on Oct 18						
<input type="checkbox"/>	Fix Header when user is logged in #11 by ioneone was closed on Oct 18						
<input type="checkbox"/>	Resolve security vulnerability in bootstrap-sass version #9 by ioneone was closed on Oct 18						
<input type="checkbox"/>	Logo and Customized CSS #7 by omarTI was closed on Oct 25						1
<input type="checkbox"/>	travis CI #6 by omarTI was closed on Oct 18						1
<input type="checkbox"/>	As a Github README developer, I want to be able to add my photo onto the Github repo for our README.md file #5 by omarTI was closed on Oct 25						1
<input type="checkbox"/>	As a Front-End engineer, I want to be able to create the Tinder card-swiping mechanism, so that users can select and pick the profiles I want to connect with enhancement #4 by omarTI was closed on Nov 1						2
<input type="checkbox"/>	User login/registration enhancement #3 by omarTI was closed on Oct 21						2
<input type="checkbox"/>	As a UI designer, I also want to be able to integrate the Greensock JS library onto Ruby on Rails so we can animate the UI for polish. enhancement #2 by omarTI was closed on Oct 25						1

Figure 2.3. Agile Development Process

In some cases, it was also productive to work together and fix issues with the code. Hence, in order to resolve certain feature issues, we would perform pair programming by having multiple members share 1 laptop. This capability allowed us to give input and advice in areas related to front-end development, load testing, and database management.

In addition, information on various aspects of development, as well as project structure, were documented on our GitHub repository's README file. This can be seen in Figure 2.4 below.

Deployment

1. SSH into AWS server

```
ssh -i ~/.ssh/linksin.pem linksin@ec2-34-209-211-32.us-west-2.compute.amazonaws.com
```

2. Clone this repository

```
git clone https://github.com/scalableinternetservices/linksin.git
```

3. Change directory to linksin

```
cd ./linksin
```

4. Initialize AWS Elastic Beanstalk instance

```
eb init
```

5. Deploy

```
eb create -db.engine postgres -db.user u -db.pass password --envvars SECRET_KEY_BASE=linksin --instance
```

6. Go to [AWS console](#) to check deployment status

Figure 2.4. Helpful documentation included in README.

3. Application Architecture

Our application uses a model-view-controller (MVC) architectural pattern. This design decision was influenced by our framework choice, Ruby on Rails, which follows a MVC architecture in order to provide better maintainability of applications. In a traditional MVC model, the roles of the models, views, and controllers are described as:

- ❖ **Models** centralize the business logic. The information in the database is represented by models
- ❖ **Views** render data in specific formats and represent the front-end portion of the application through user interfaces
- ❖ **Controllers** interact with models and views. They deal with application flow by parsing requests and passing data to the correct view

We used six models to structure our data; the important properties and associations between the primary models are:

User

- ☐ Has one profile
- ☐ Has many events, through members
- ☐ Has many conversations and matches

Profile

- ☐ Belongs to one user

Event

- ☐ Has many users, through members
- ☐ Belongs to one host, a user

Conversation

- ☐ Has many messages
- ☐ Belongs to a sender and receiver, users

Users and events have a many-to-many relationship. To represent this association, we allowed users to have several events (and vice versa) by proceeding through a third model, members.

Users and profiles have a one-to-one relationship in which a profile is dependent on user; each user has one, and only one, profile that is created

when the user is created and destroyed along with it if or when the user is removed from the system.

Lastly, users and conversations have a many-to-many relationship. Users can have many conversations, and conversations belong to a sender and a recipient, both of which are references to users. Conversations also have a one-to-many relationship with messages, which are nested in and dependent on their parent conversation. Figure 3.0 gives a high-level idea of our database architecture while figure 3.1 visualizes our full entity-relationship (ER) model.

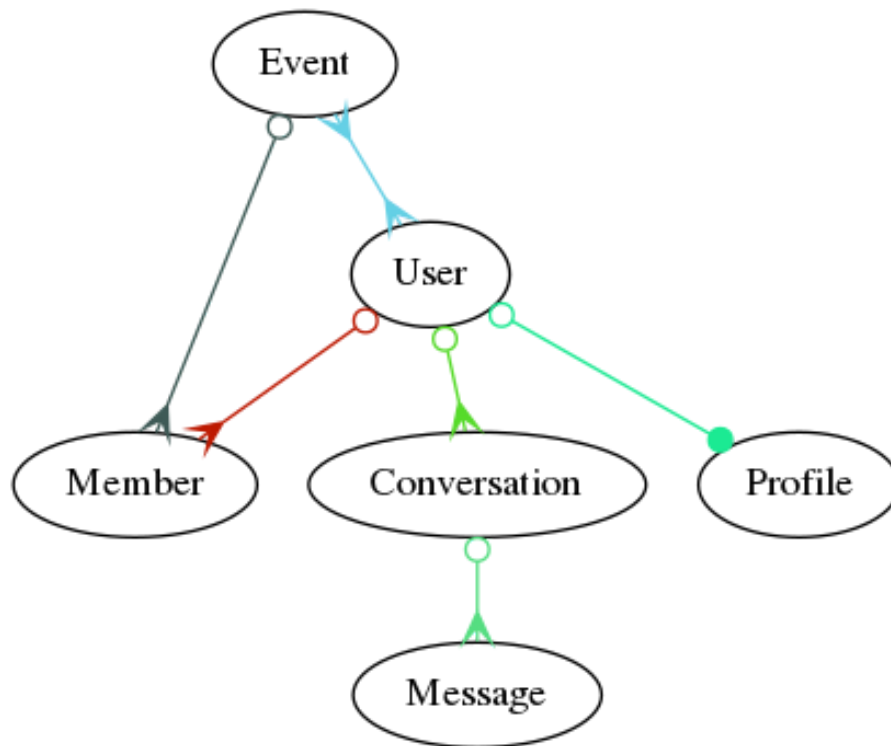


Figure 3.0. High-level DB architecture

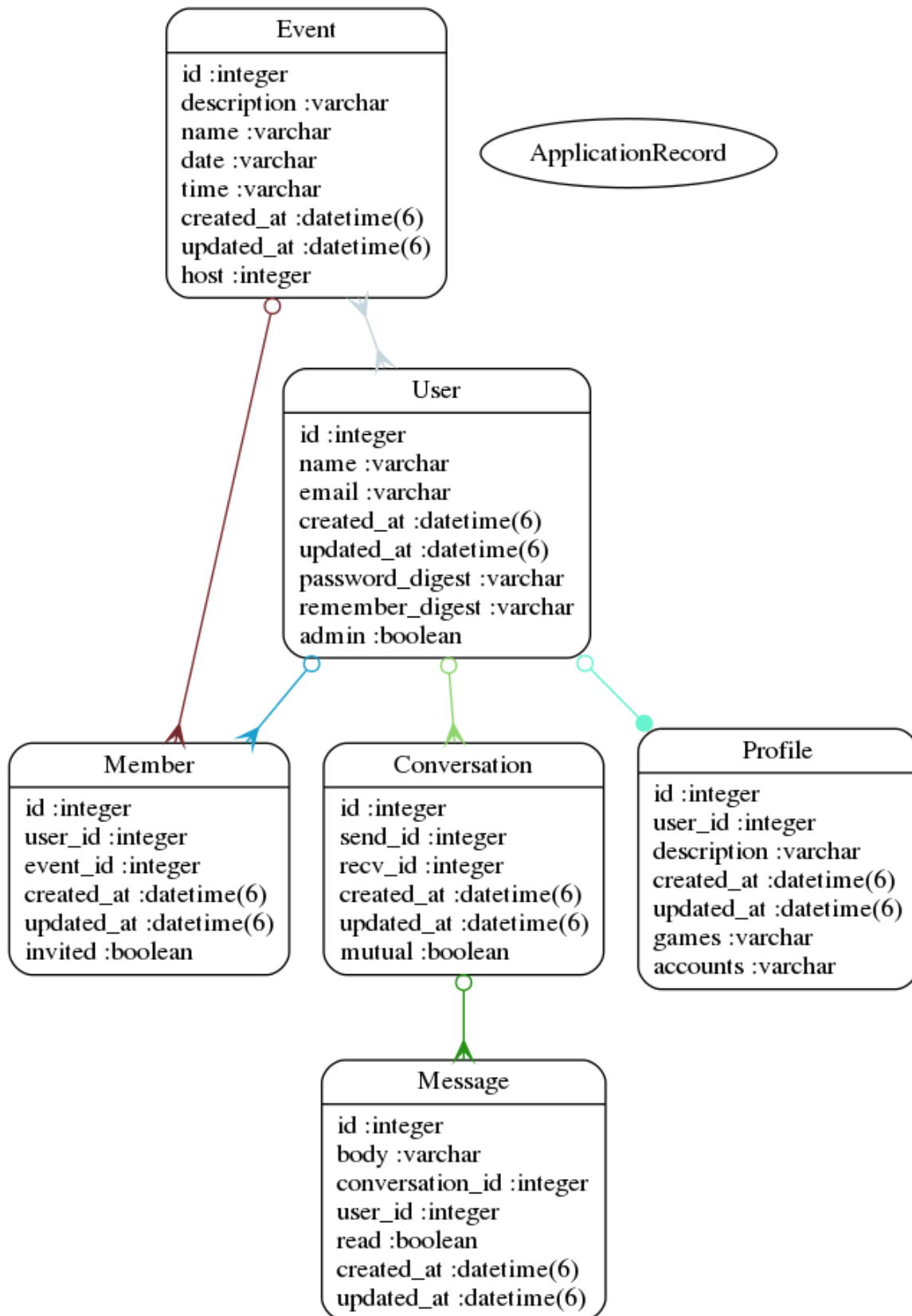


Figure 3.1. LinkedIn Entity-Relationship (ER) Model

4. Load Testing and Scaling

For testing, we have 12 phrases with each phase running for 60 seconds, as shown in Figure 4.0.1. Upon each successive phase, the arrival rate of users is increased.

Phases	Users/sec
1	1
2	1.5
3	2
4	4
5	6
6	10
7	16
8	20
9	25
10	35
11	45
12	55

Figure 4.0.1. Test phases

Every user performs a sequence of predefined operations as follows:

Step	Operation
1	Go to home page
2	Go to login page
3	Fill out login form
4	Log in and redirected to home page
5	Swipe right on user 1 and redirect to messages
6	Messages to user 1
7	Go to home page
8	Repeat step 5 - 7 for user 2, 3, 4, and 5
9	Go to events page

10	Go to event creation page
11	Create an event

Figure 4.0.2. Critical Path

To effectively simulate a real user, random ‘think times’ were added for actions such as filling in forms and navigation of the website. From the results found in testing, we concluded that the application fails to handle the load if the response time is greater than 200ms.

4.1 Vertical Scaling

Vertical scaling refers to the idea of scaling application by using a more powerful server (i.e. more memory, more/faster cores, higher bandwidth).

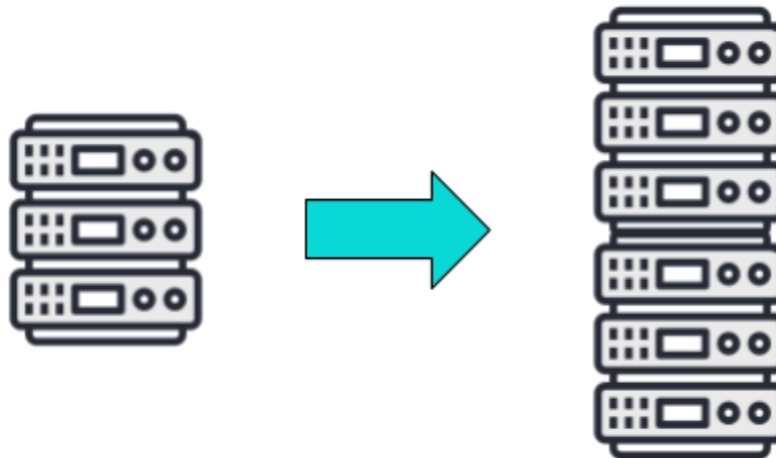


Figure 4.1.0 Diagram of Vertical Scaling

This section explores the performance of the t3.micro, t3.large, m4.4xlarge, and m4.16xlarge instances. Detailed specification of each instance is shown in Figure 4.1.1.

Instance	vCPU	Memory (GiB)	Price
t3.micro	2	1	\$7.4/month
t3.large	2	8	\$60.0/month

t3.xlarge	4	16	\$119.8/month
t3.2xlarge	8	32	\$239.6/month
m4.4xlarge	16	64	\$576/month
m4.16xlarge	64	256	\$2304/month

Figure 4.1.1. Instance comparisons

4.1.1 t3.micro

We used the following command to deploy the application on t3.micro instance.

```
eb create -db.engine postgres -db.user u -db.pass password --envvars
SECRET_KEY_BASE=linksin --instance_type t3.micro --single
```

Response time of the test using t3.micro instance is shown in Figure 4.1.1.

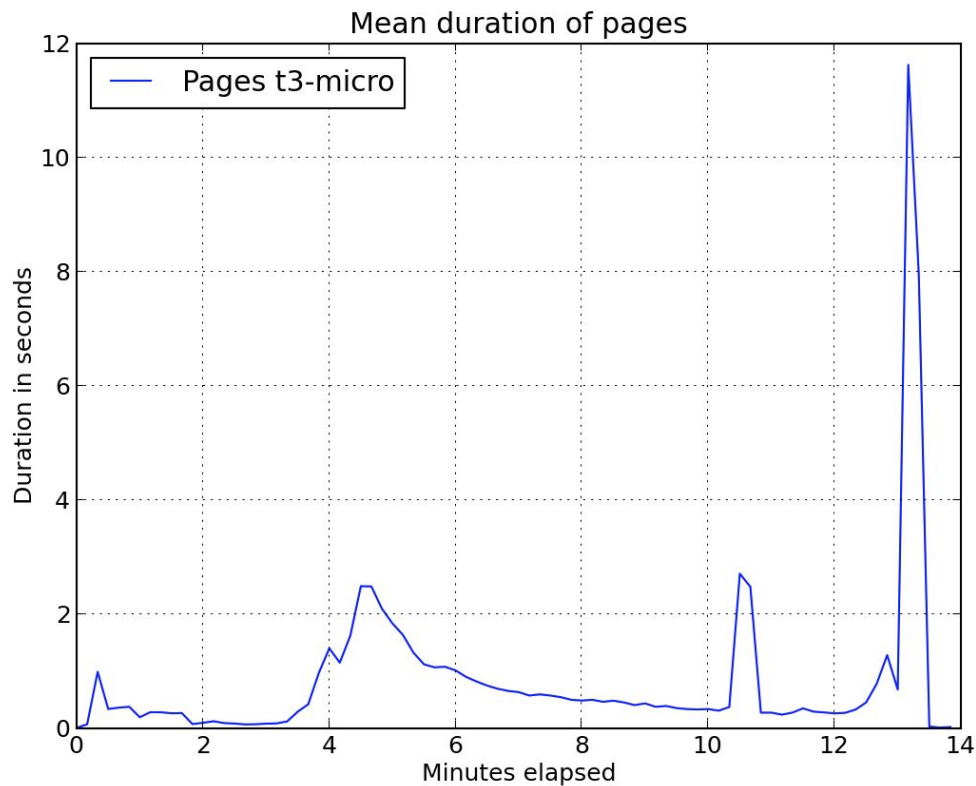


Figure 4.1.2. Response time for t3.micro

The response time around 210 seconds is shown in Figure 4.1.3.

Time (sec)	Response (ms)
190	87.73
200	122.05
210	292.58
220	421.96
230	967.51

Figure 4.1.3. Response time around 210 seconds for t3.micro

From the results, we see that the response time becomes significantly slow when it reaches 210 seconds. Thus, the application was able to handle phase 3 but fails at phase 4.

4.1.2 t3.large

We used the following command to deploy the application on t3.large instance.

```
eb create -db.engine postgres -db.user u -db.pass password --envvars  
SECRET_KEY_BASE=linksin --instance_type t3.large --single
```

Response time of the test using t3.large instance is shown in Figure 4.1.4.

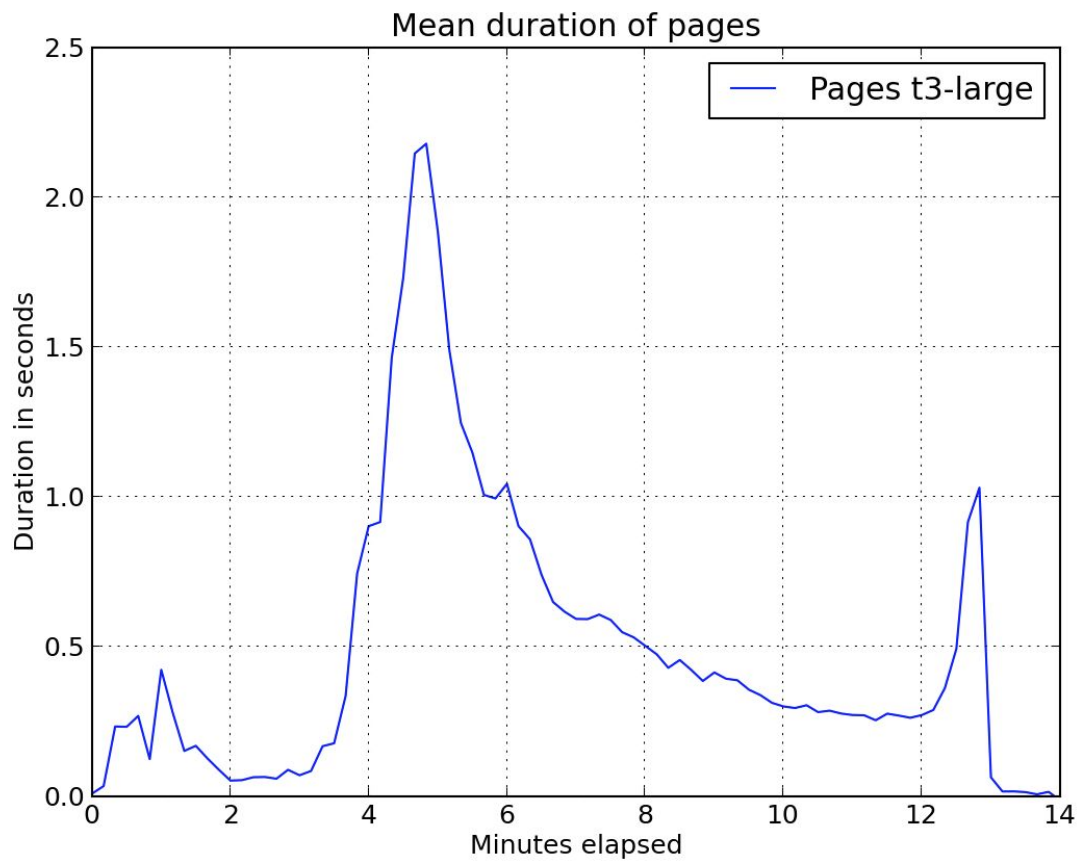


Figure 4.1.4. Response time for t3.large

The response time around 210 seconds is shown in Figure 4.1.5.

Time (sec)	Response (ms)
190	84.90
200	167.50
210	177.36
220	335.29
230	744.30

Figure 4.1.5. Response time around 210 seconds for t3.large

From the results, we see that the response time gets significantly slow at 220 seconds. Just like with t3.micro, the application fails to respond at phase 4.

4.1.2 t3.xlarge

We used the following command to deploy the application on t3.xlarge instance.

```
eb create -db.engine postgres -db.user u -db.pass password --envvars  
SECRET_KEY_BASE=linksin --instance_type t3.xlarge --single
```

Response time of the test using t3.xlarge instance is shown in Figure 4.1.6.

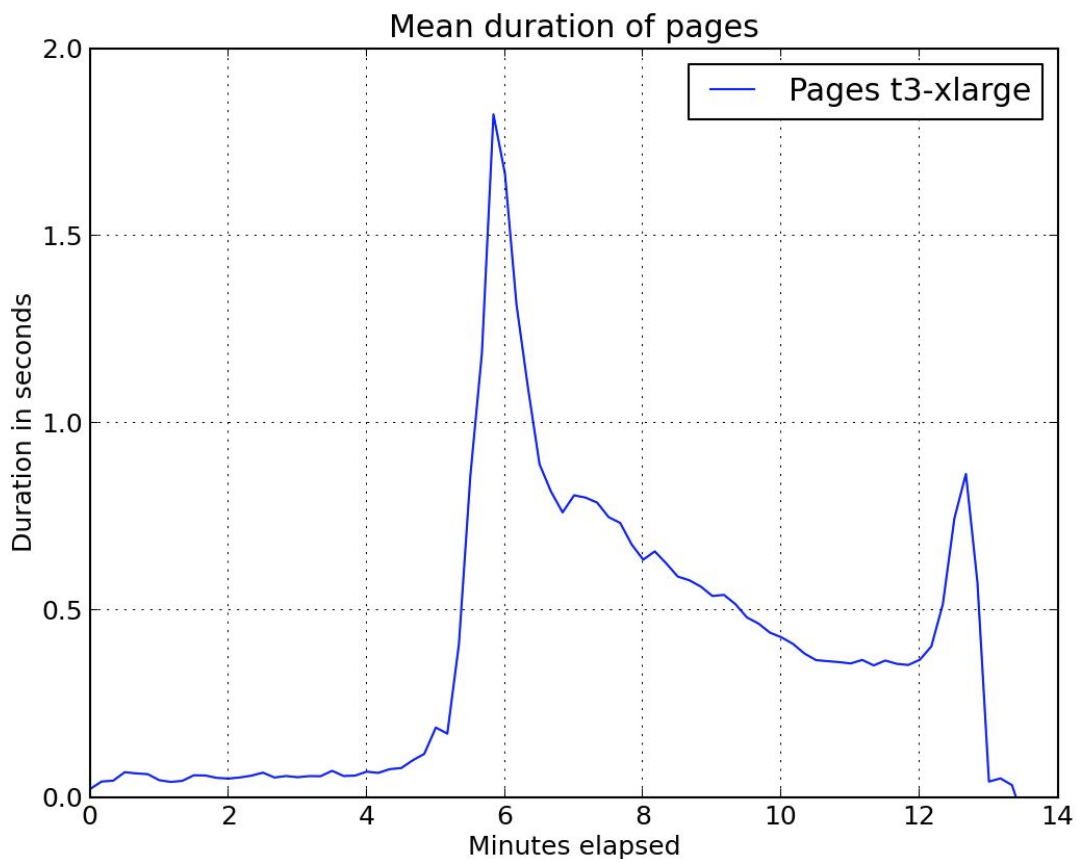


Figure 4.1.6. Response time for t3.xlarge

The response time around 310 seconds is shown in Figure 4.1.7.

Time (sec)	Response (ms)
290	116.28
300	186.64
310	170.26
320	207.83
330	859.99

Figure 4.1.7. Response time around 310 seconds for t3.xlarge

From the results, we see that the response time exceeds 200ms at 320 seconds. So the application was able to handle phase 5, but fails to manage phase 6.

4.1.4 t3.2xlarge

We used the following command to deploy the application on t3.2xlarge instance.

```
eb create -db.engine postgres -db.user u -db.pass password --envvars  
SECRET_KEY_BASE=linksin --instance_type t3.2xlarge --single
```

Response time of the test using t3.2xlarge instance is shown in Figure 4.1.8.

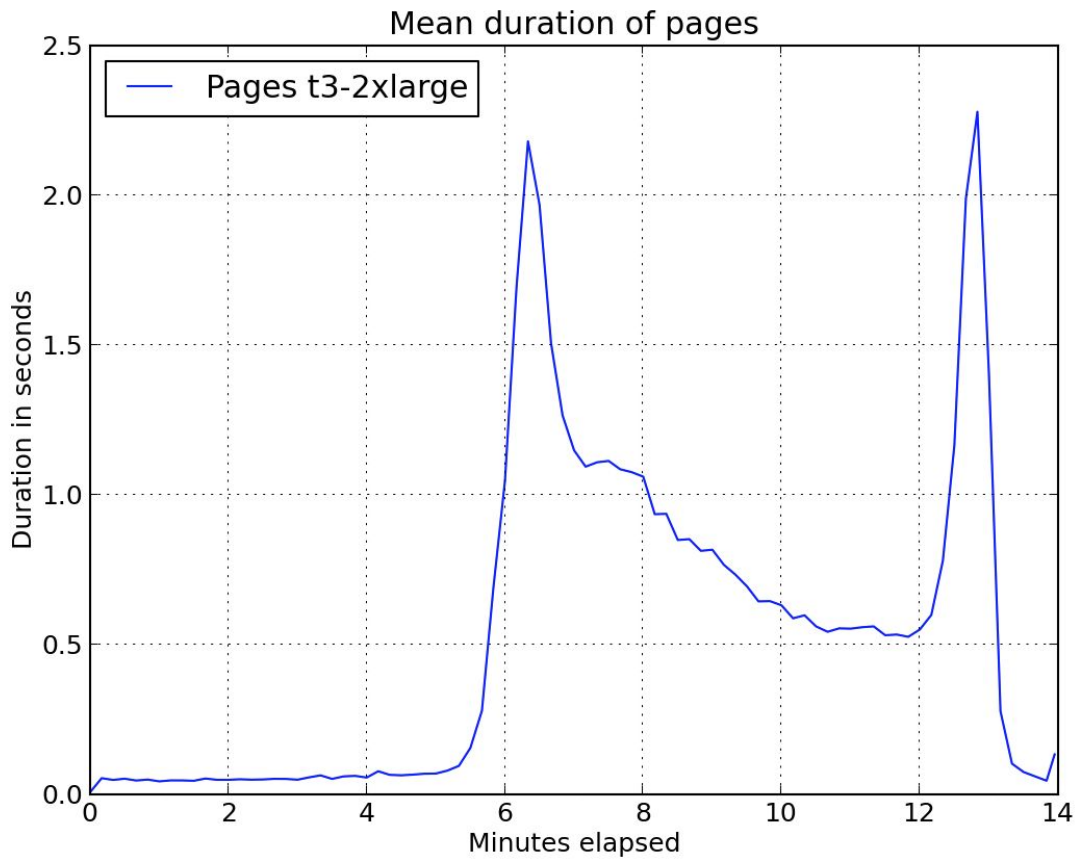


Figure 4.1.8. Response time for t3.2xlarge

The response time around 330 seconds is shown in Figure 4.1.9.

Time (sec)	Response (ms)
310	79.30
320	95.15
330	155.28
340	279.05
350	691.36

Figure 4.1.9. Response time around 330 seconds for t3.2xlarge

As in t3.xlarge, the application was able to handle phase 5 but fails at phase 6.

4.1.5 m4.4xlarge

We used the following command to deploy the application on m4.4xlarge instance.

```
eb create -db.engine postgres -db.user u -db.pass password --envvars  
SECRET_KEY_BASE=linksin --instance_type m4.4xlarge --single
```

Response time of the test using m4.4xlarge instance is shown in Figure 4.1.10.

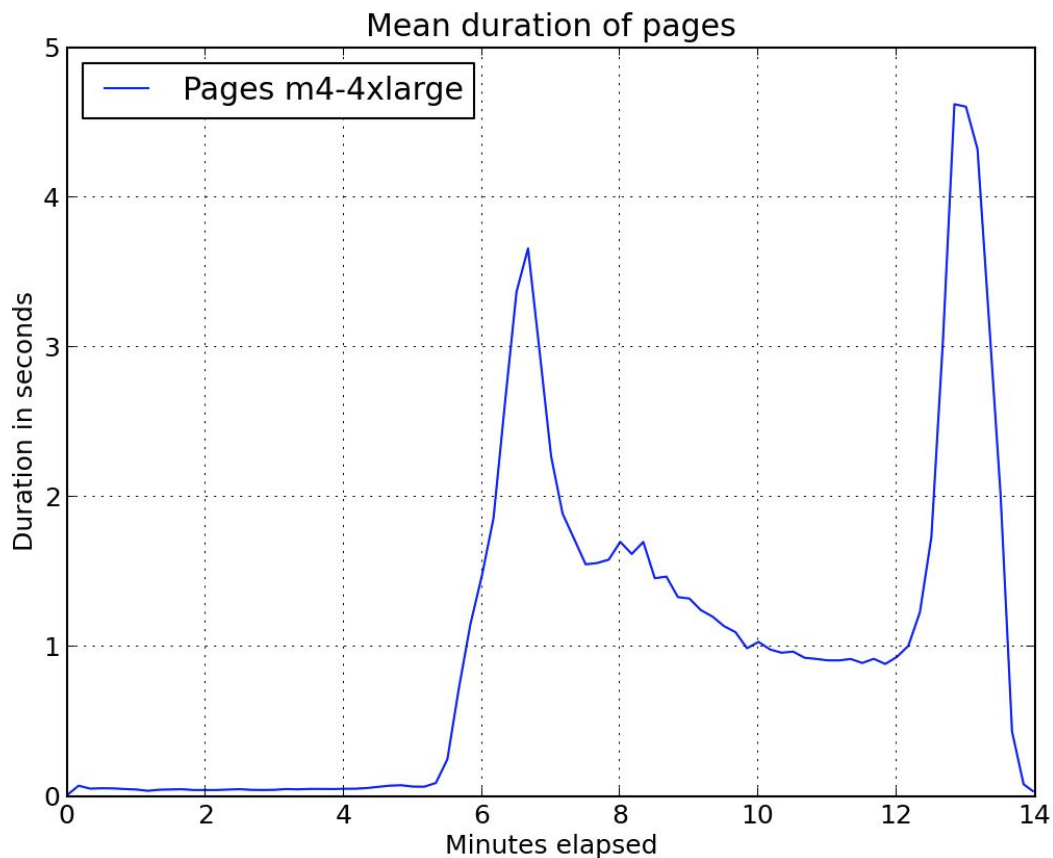


Figure 4.1.10. Response time for m4.4xlarge

The response time around 320 seconds is shown in Figure 4.1.11.

Time (sec)	Response (ms)
300	65.65
310	64.84
320	89.74
330	248.16
340	723.47

Figure 4.1.11. Response time around 320 seconds for m4.4xlarge

The application still fails to handle the loads at phase 6.

4.1.6 m4.16xlarge

We used the following command to deploy the application on m4.16xlarge instance.

```
eb create -db.engine postgres -db.user u -db.pass password --envvars  
SECRET_KEY_BASE=linksin --instance_type m4.16xlarge --single
```

Response time of the test using m4.16xlarge instance is shown in Figure 4.1.12.

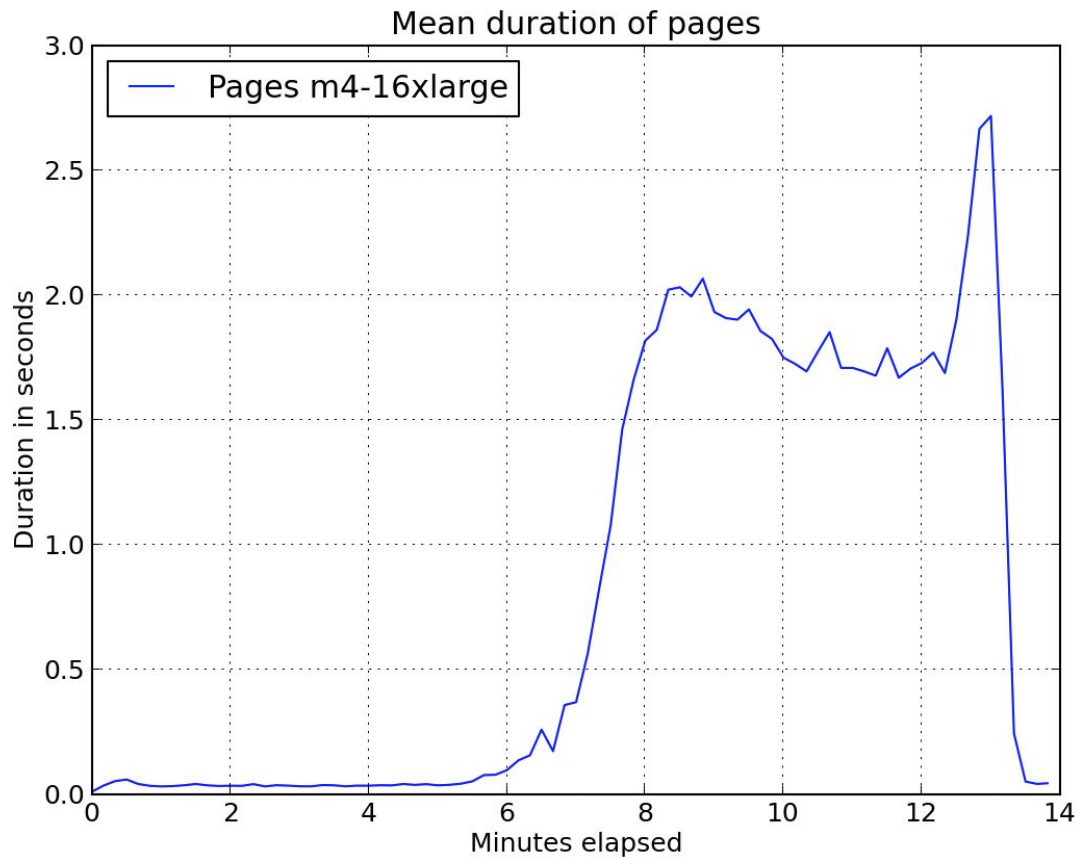


Figure 4.1.12. Response time for m4.16xlarge

The response time around 380 seconds is shown in Figure 4.1.13.

Time (sec)	Response (ms)
360	97.18
370	136.64
380	156.82
390	259.52
400	173.16

Figure 4.1.13. Response time around 380 seconds for m4.16xlarge

We have a slight improvement here. The application successfully survived phase 6 and fails to respond in phase 7.

4.1.7 Summary

Here's the summary of vertical scaling.

Instance	Price	Max Phase Handled
t3.micro	\$7.4/month	3 (2 users/sec)
t3.large	\$60.0/month	3 (2 users/sec)
t3.xlarge	\$119.8/month	5 (6 users/sec)
t3.2xlarge	\$239.6/month	5 (6 users/sec)
m4.4xlarge	\$576/month	5 (6 users/sec)
m4.16xlarge	\$2304/month	6 (10 users/sec)

Figure 4.1.14. Summary of vertical scaling

As we can see in Figure 4.1.14, vertical scaling is very expensive and does not scale very well despite the cost. In the next section, let's see how horizontal scaling performs compared to this.

4.2 Horizontal Scaling

Horizontal scaling refers to the idea of scaling the application by having multiple app servers.



Figure 4.2.0. Diagram of Horizontal Scaling

We will be comparing the performance of using 1, 2, 4, and 16 app servers as shown in Figure 4.2.1.

Instance	vCPU	Memory (GiB)	Price
t3.micro x 1	2 x 1	1 x 1	\$7.49/month
t3.micro x 2	2 x 2	1 x 2	\$14.98/month
t3.micro x 4	2 x 4	1 x 4	\$30.00/month
t3.micro x 8	2 x 8	1 x 8	\$59.90/month
t3.micro x 16	2 x 16	1 x 16	\$119.81/month
t3.micro x 32	2 x 32	1 x 32	\$239.62/month

Figure 4.2.1. Instance comparisons

4.2.1 t3.micro x 1

We used the following command to deploy the application on 1 instance of t3.micro.

```
eb create -db.engine postgres -db.user u -db.pass password --envvars
SECRET_KEY_BASE=linksin --instance_type t3.micro --single
```

Note we are using the same baseline as the vertical scaling. Response time of the test using 1 instance of t3.micro is shown in Figure 4.2.2.

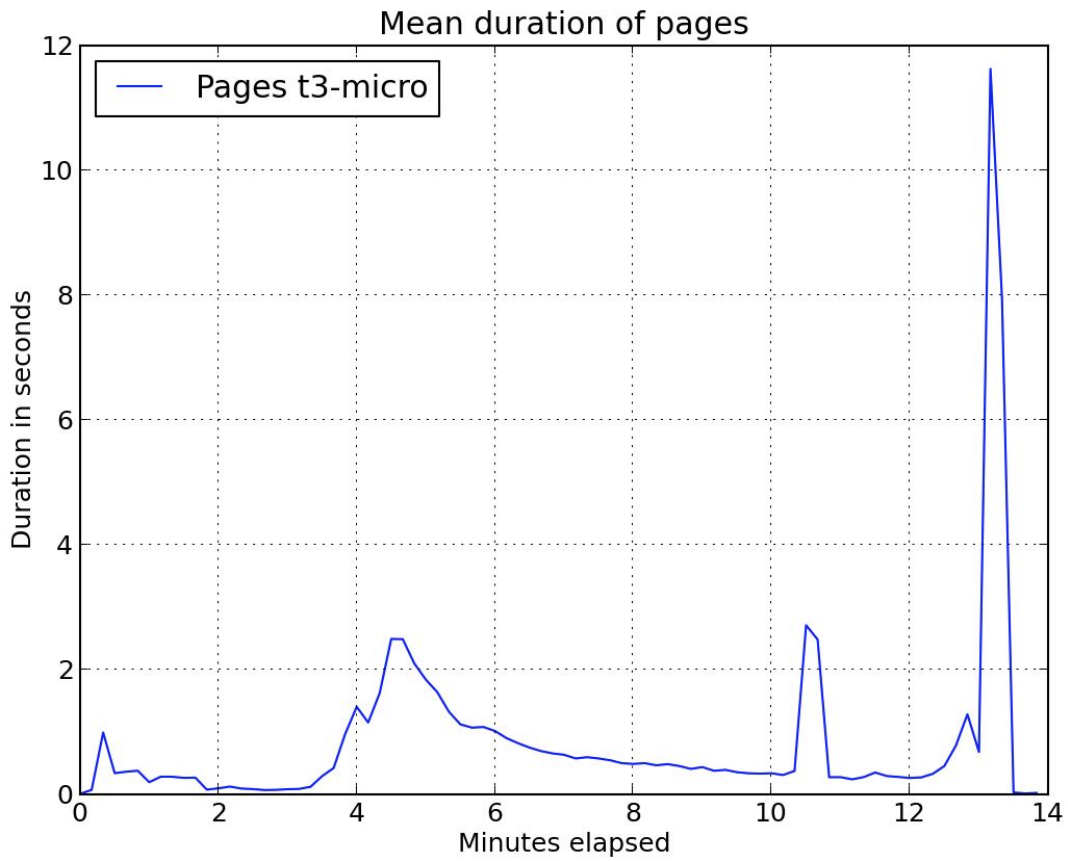


Figure 4.2.2. Response time for t3.micro x 1

The response time around 210 seconds is shown in Figure 4.2.3.

Time (sec)	Response (ms)
190	87.73
200	122.05
210	292.58
220	421.96
230	967.51

Figure 4.2.3. Response time around 210 seconds for t3.micro x 1

As already discussed earlier, the application was able to handle phase 3 but fails at phase 4.

4.2.2 t3.micro x 2

We used the following command to deploy the application on 2 instances of t3.micro.

```
eb create -db.engine postgres -db.user u -db.pass password --envvars  
SECRET_KEY_BASE=linksin --instance_type t3.micro --scale 2
```

Response time of the test using 2 instances of t3.micro is shown in Figure 4.2.4.

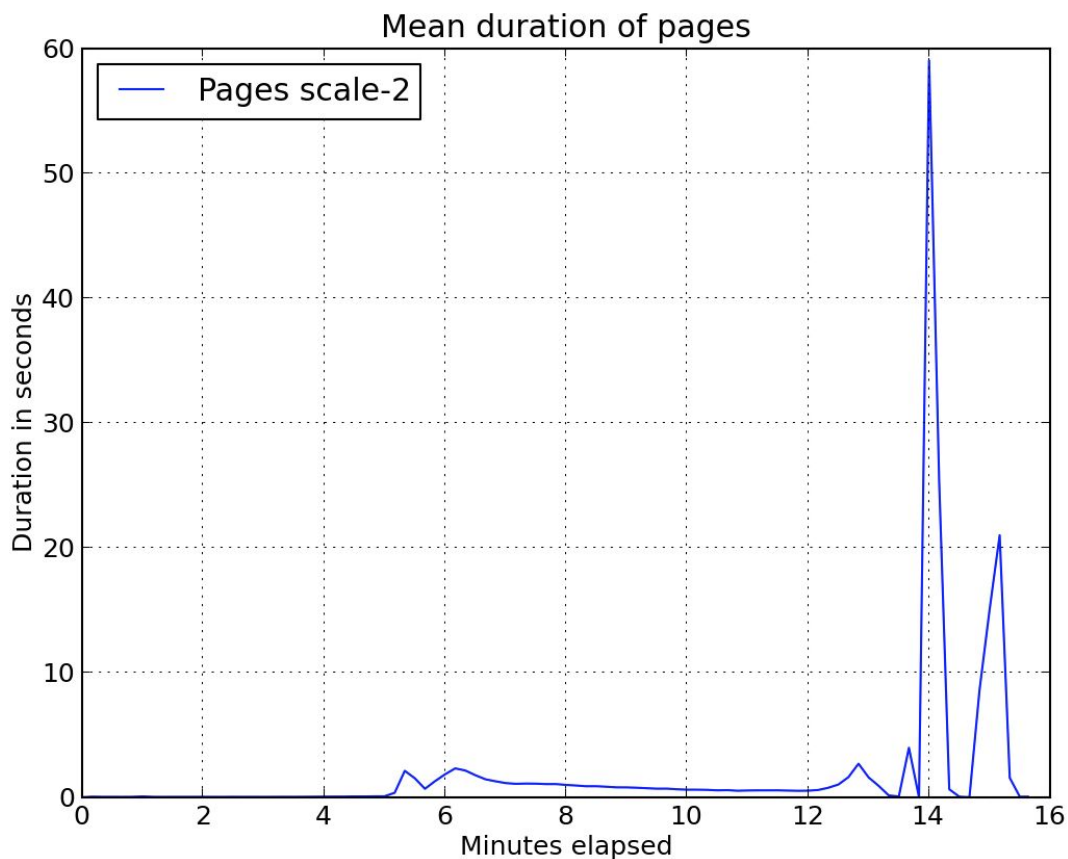


Figure 4.2.4. Response time for t3.micro x 2

The response time around 300 seconds is shown in Figure 4.2.5.

Time (sec)	Response (ms)
280	77.39
290	86.84
300	88.91
310	372.21
320	2129.45

Figure 4.2.5. Response time around 300 seconds for t3.micro x 2

The application was able to handle up to phase 5 and failed at phase 6, which is a huge improvement from only 1 instance.

4.2.3 t3.micro x 4

We used the following command to deploy the application on 4 instances of t3.micro.

```
eb create -db.engine postgres -db.user u -db.pass password --envvars  
SECRET_KEY_BASE=linksin --instance_type t3.micro --scale 4
```

Response time of the test using 4 instances of t3.micro is shown in Figure 4.2.6.

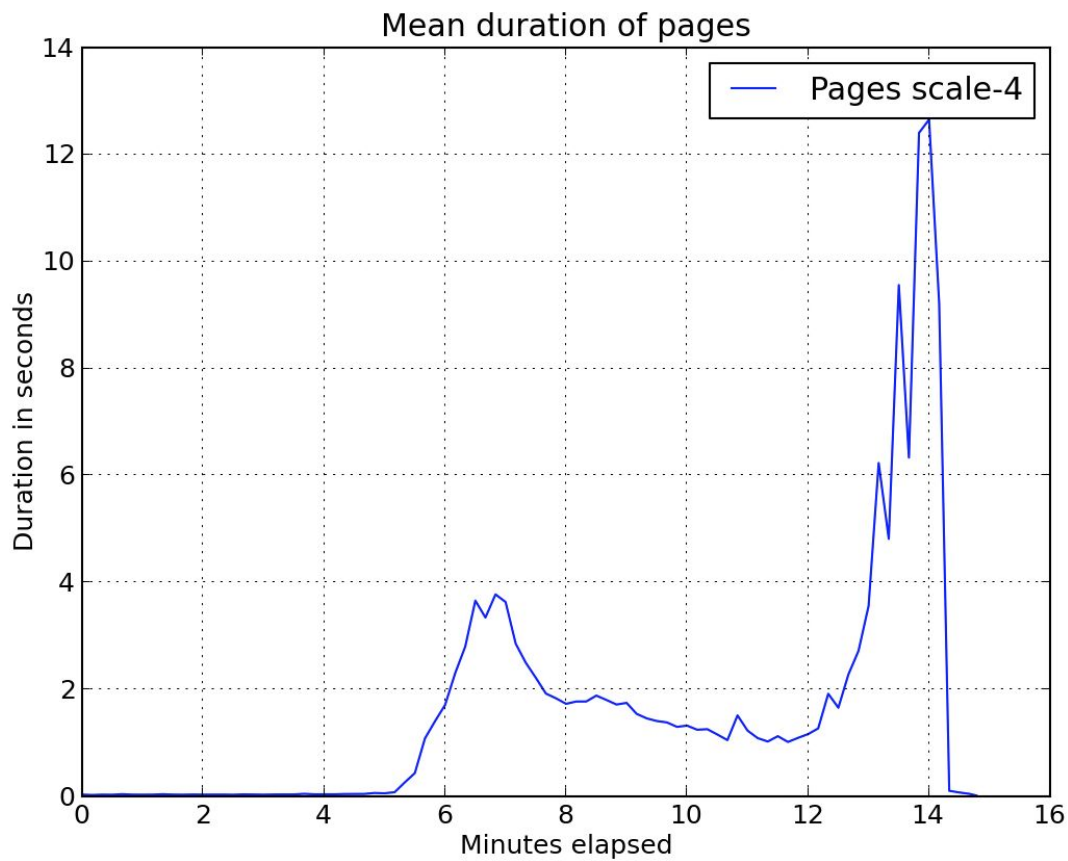


Figure 4.2.6. Response time for t3.micro x 4

The response time around 310 seconds is shown in Figure 4.2.7.

Time (sec)	Response (ms)
290	63.85
300	57.86
310	78.61
320	261.98
330	435.53

Figure 4.2.7. Response time around 310 seconds for t3.micro x 4

There wasn't a significant improvement from last time; the application still fails at phase 6.

4.2.4 t3.micro x 8

We used the following command to deploy the application on 8 instances of t3.micro.

```
eb create -db.engine postgres -db.user u -db.pass password --envvars  
SECRET_KEY_BASE=linksin --instance_type t3.micro --scale 8
```

Response time of the test using 8 instances of t3.micro instances is shown in Figure 4.2.8.

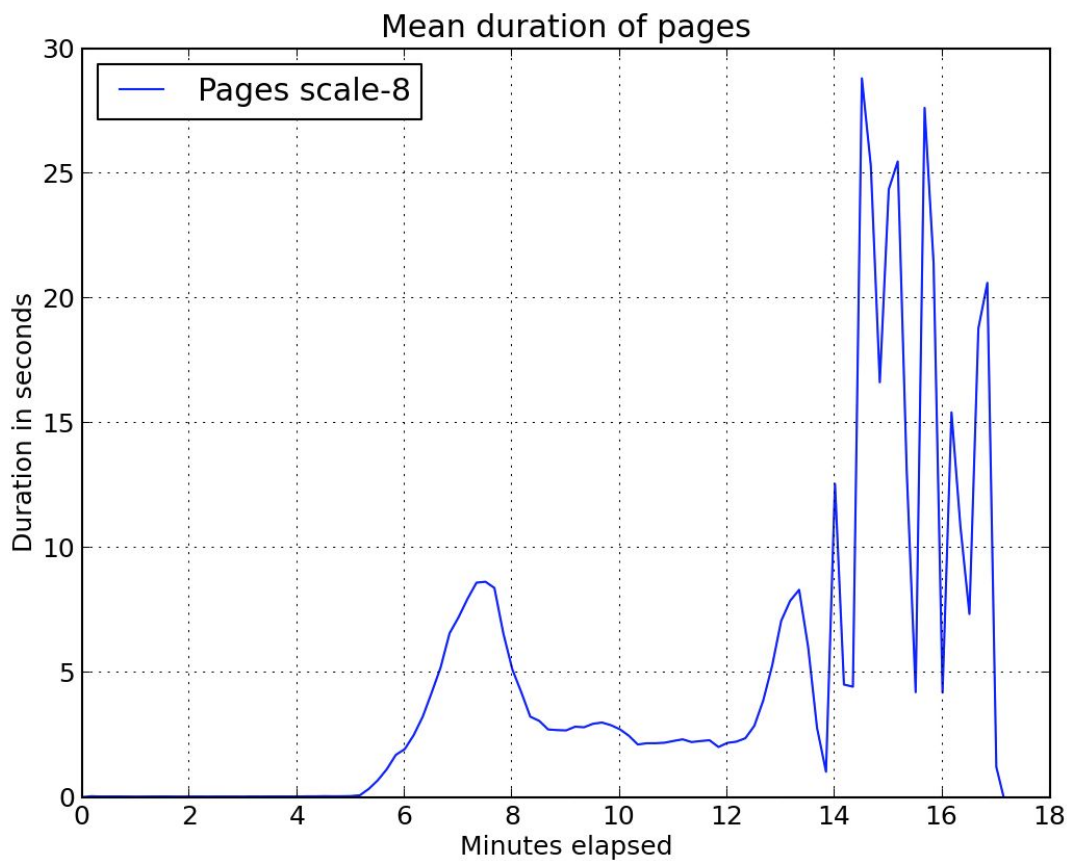


Figure 4.2.8. Response time for t3.micro x 8

The response time around 310 seconds is shown in Figure 4.2.9.

Time (sec)	Response (ms)
290	46.36
300	51.55
310	86.70
320	344.84
330	690.05

Figure 4.2.9. Response time around 310 seconds for t3.micro x 8

There's still not much improvement. The application fails at phase 6.

4.2.5 t3.micro x 16

We used the following command to deploy the application on 16 instances of t3.micro.

```
eb create -db.engine postgres -db.user u -db.pass password --envvars  
SECRET_KEY_BASE=linksin --instance_type t3.micro --scale 16
```

Response time of the test using 16 instances of t3.micro is shown in Figure 4.2.10.

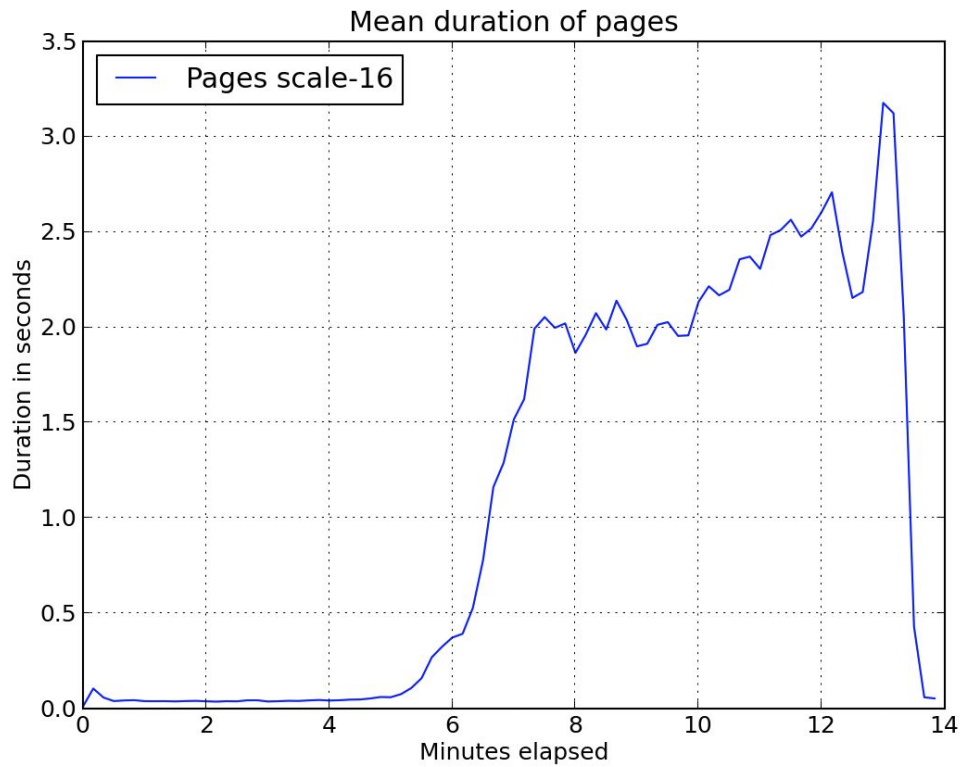


Figure 4.2.10. Response time for t3.micro x 16

The response time around 330 seconds is shown in Figure 4.2.11.

Time (sec)	Response (ms)
310	75.93
320	107.94
330	159.31
340	269.31
350	324.85

Figure 4.2.11. Response time around 330 seconds for t3.micro x 16

The application still fails at phase 6, but it handles the loads longer than before.

4.2.6 t3.micro x 32

We used the following command to deploy the application on 32 instances of t3.micro.

```
eb create -db.engine postgres -db.user u -db.pass password --envvars SECRET_KEY_BASE=linksin --instance_type t3.micro --scale 32
```

Response time of the test using 32 instances of t3.micro is shown in Figure 4.2.12.

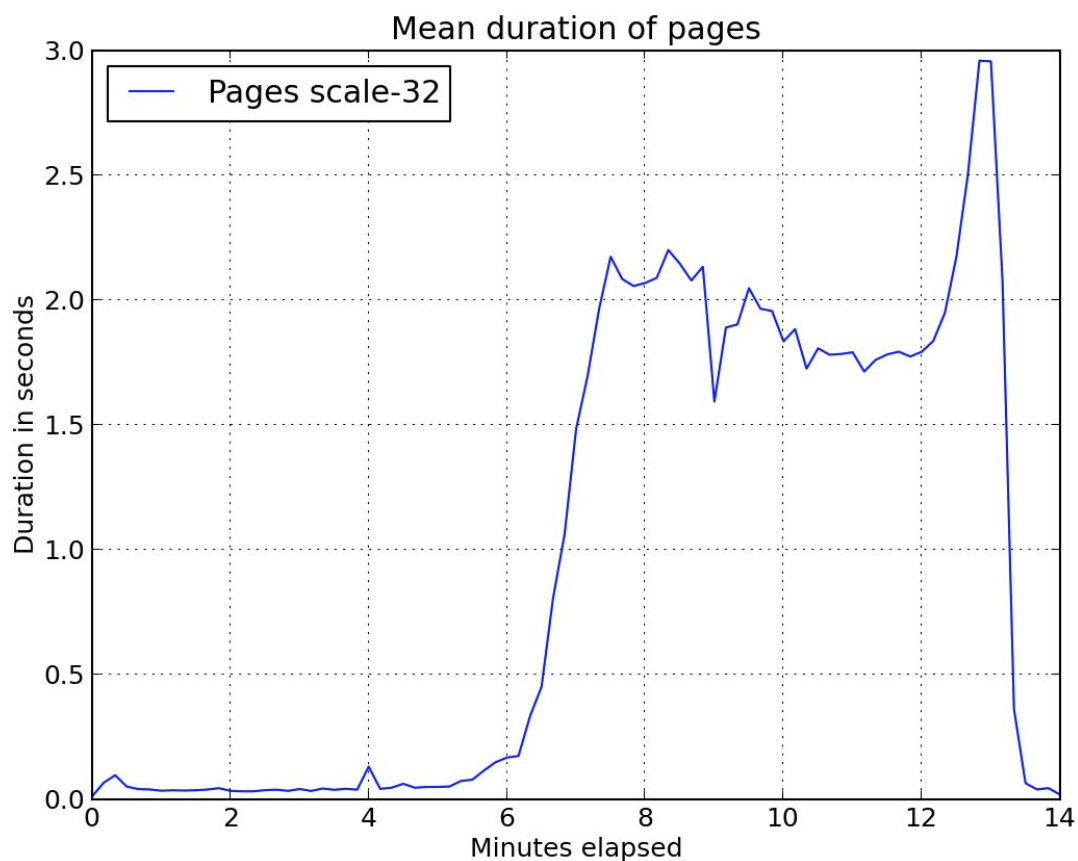


Figure 4.2.12. Response time for t3.micro x 32

The response time around 380 seconds is shown in Figure 4.2.13.

Time (sec)	Response (ms)
------------	---------------

360	167.68
370	174.12
380	332.76
390	450.94
400	805.90

Figure 4.2.13. Response time around 380 seconds for t3.micro x 32

Although the application fails at phase 7, it can finally handle phase 6.

4.2.7 Summary

Here's the summary of horizontal scaling.

Instance	Price	Max Phase Handled
t3.micro x 1	\$7.49/month	3
t3.micro x 2	\$14.98/month	5
t3.micro x 4	\$30.00/month	5
t3.micro x 8	\$59.90/month	5
t3.micro x 16	\$119.81/month	5
t3.micro x 32	\$239.62/month	6

Figure 4.2.14. Summary of horizontal scaling

Recall that, with vertical scaling, it cost \$2304/month to handle phase 6. On the other hand, with horizontal scaling, we only needed to \$239.62/month to handle phase 6. Given this, we effectively save \$2064.38/month when choosing horizontal scaling over vertical.

4.3 Pagination

Our baseline app utilized pagination, but it initially used the “will_paginate” gem, which is bulky and slow. Aside from adding unnecessary bulk to our rails application, it also used two queries for pagination: one to get a count of all records in a table, and another to fetch the number of records per page. This count query is unneeded for our application since we do not care how many pages of records we have. Thus, a switch to the “pagy” gem was made due to it being faster, lighter, and overall more efficient. Additionally, the pagy

gem has numerous options that we utilized to get rid of the unnecessary query, consequently speeding up our application.

Our method for generating the users to match with was also modified to be based off of pagination to reduce page load time. Initially, our user matching method involved grabbing 20 random users from the database and checking each of them individually to make sure they are not already matches of the current user or the user itself. After the user has swiped through these 20 users, the page is refreshed to load more users. Additionally, each time the swiping page is loaded a count of the whole user table is done, which is rather costly. With the switch to the pagy gem, the count query is omitted, and five users are drawn at a time. Instead of refreshing the page when these five cards are swiped through, we are using an AJAX request to load the next set of users.

```
def show
  @user = User.find(params[:id])
  @userlist = randomShow(@user).select do |user|
    conversation = Conversation.between(user.id, @user.id)
    conversation.empty?
  end
end

def randomShow(user)
  User.where.not(id: user.id).order("RANDOM()").limit(20)
end
```

Figure 4.3.0. User matching before pagination optimization

```
def show
  @user = User.find(params[:id])
  @pagy, @users = pagy_countless(User, items: 5)
  @userlist = @users.shuffle.select do |user|
    conversation = Conversation.between(user.id, @user.id)
    conversation.empty? && (user.id != @user.id)
  end
end
```

Figure 4.3.1. User matching after pagination optimization

```

if (numCards <= 2 && nextPage > 0) {
  $.ajax({
    url: '?page=' + nextPage
  }).done(function (result) {
    cards = $(result).find("#card_container").children()
    $('#card_container').children()[0].remove()
    $('#card_container').prepend(cards);
    numCards += cards.length - 1;
    nextPage++;
  }).fail(function () {
    nextPage = -1;
  });
}

```

Figure 4.3.2. Ajax query for adding more cards

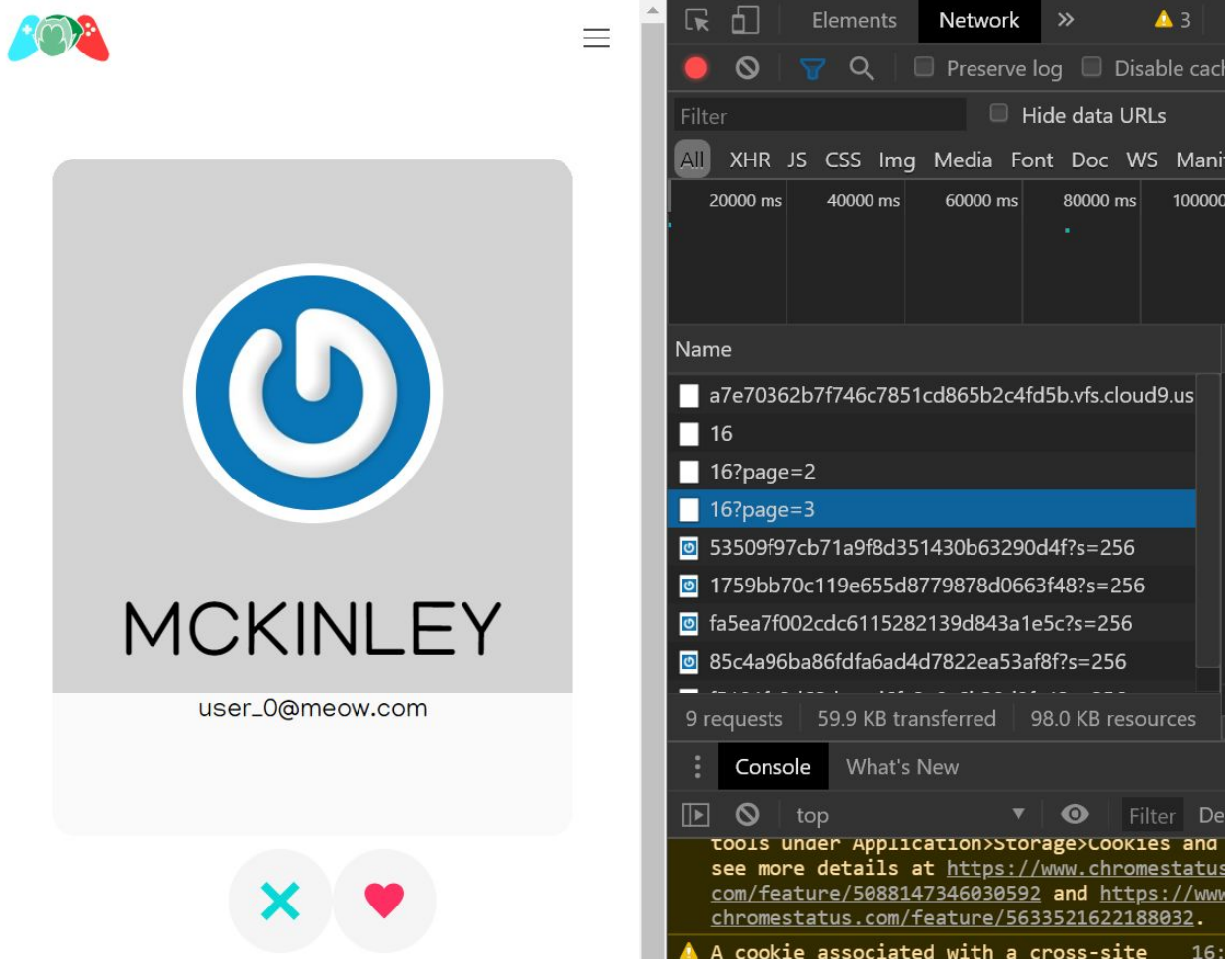


Figure 4.3.3. User matching pagination in action

We deployed our app using a t2.micro instance. The resulting response time is shown in Figure 4.3.4.

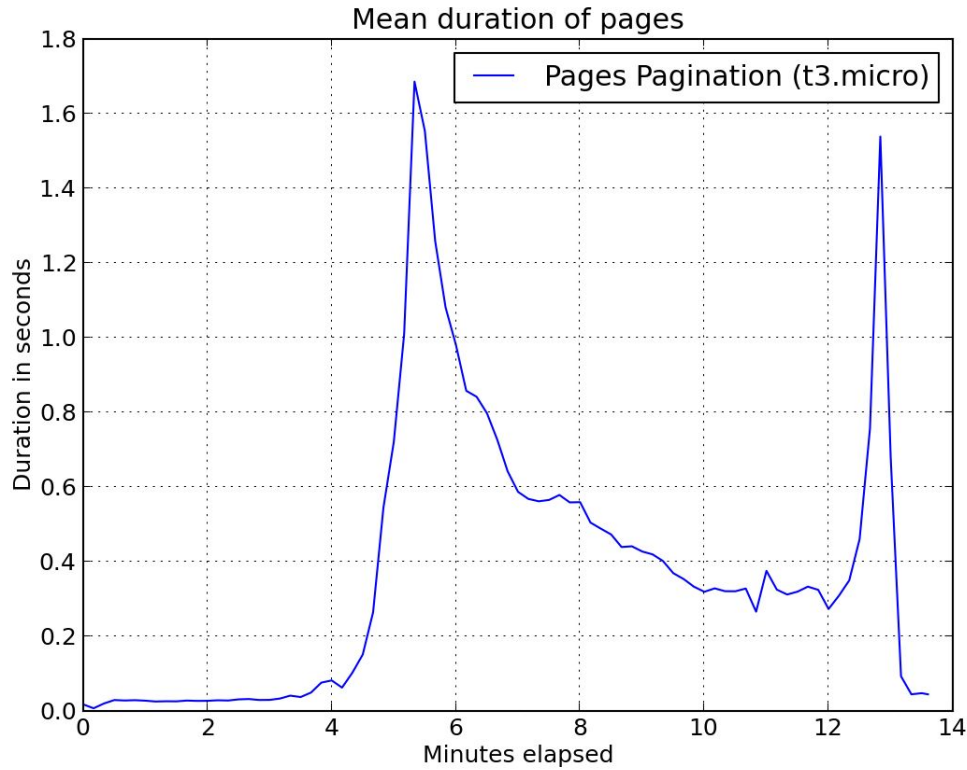


Figure 4.3.4. Response time for pagination optimizations

The response time around 270 seconds is shown in Figure 4.3.5.

Time (sec)	Response (ms)
250	62.6
260	102.66
270	151.21
280	264.48
290	545.03

Figure 4.3.5. Response time around 270 seconds for pagination optimizations

With pagination optimizations, our app was able to respond in a timely manner until the middle of phase 5. This shows improvement over the baseline, which fails in phase 4.

Instance	Max Phase Handled
w/o pagination	3
w/ pagination	4

Figure 4.3.6. Pagination summary

4.4 Processes/Threads

We can maximize the usage of the application server by using multiprocessing and multithreading. When we have multiple processes running the application, each process will run in different cores, and thus can handle more requests. When we have multiple threads, while we are waiting for a response from the database, we can switch the thread and run other stuff, and thus increase the performance of the application.

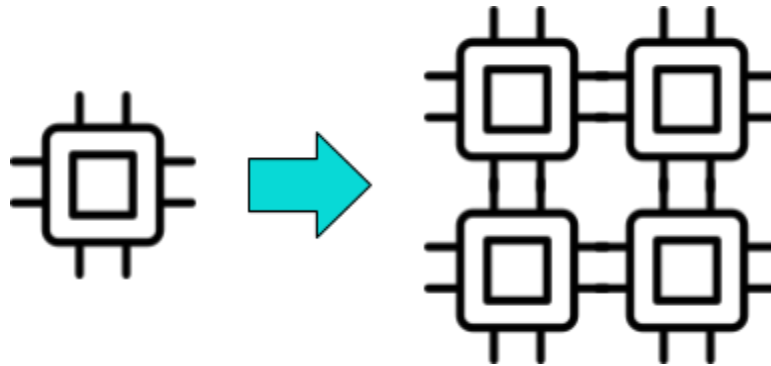


Figure 4.4.0. Diagram of multiprocessing and multithreading

Because EC2 provides default puma configuration, we need to override it using ebextensions as follows:

```

# t is number of threads
# w is number of workers

commands:
  01backup_config:
    command: "cp -n
/opt/elasticbeanstalk/support/conf/pumaconf.rb
/opt/elasticbeanstalk/support/conf/pumaconf.rb.original"

  02_edit_comment_default_threads_config:
    command: "sed -i 's/threads 8, 32/threads t, t/'
/opt/elasticbeanstalk/support/conf/pumaconf.rb"

  03_delete_default_worker_config:
    command: "sed -i '/workers/d'
/opt/elasticbeanstalk/support/conf/pumaconf.rb"

  04_insert_new_worker_after_threads_config:
    command: "sed -i '/threads/a workers w'
/opt/elasticbeanstalk/support/conf/pumaconf.rb"

```

We will be testing on m4.4xlarge EC2 instance and m5.24xlarge database instance. We chose such a large database to avoid the database from being the bottleneck of the performance. m4.4xlarge EC2 instance has 16 vCPU, which means there are 8 cores and each core has 2 threads.

4.4.1 Process x 1 & Thread x 1

Response time with 1 process and 1 thread is shown in Figure 4.4.1.

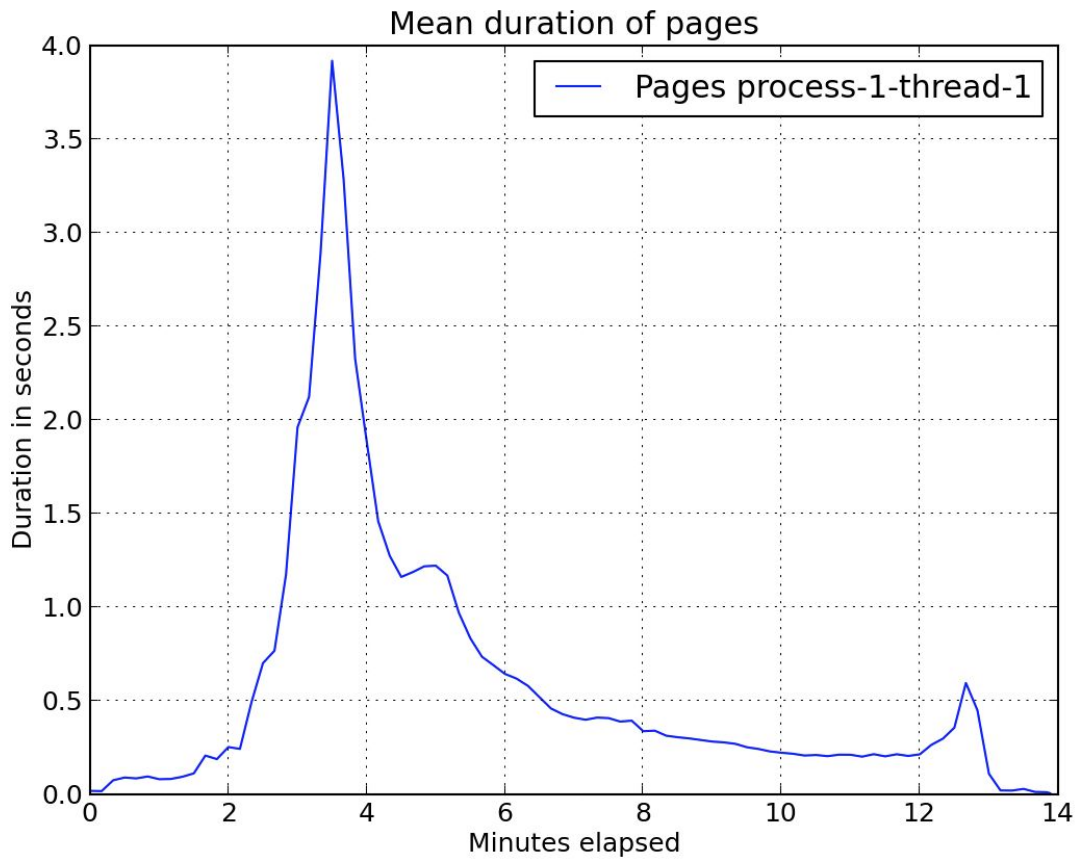


Figure 4.4.1. Response time for process x 1 & thread x 1

The response time around 100 seconds is shown in Figure 4.4.2.

Time (sec)	Response (ms)
80	94.13
90	112.24
100	207.4
110	188.53
120	252.85

Figure 4.4.2. Response time around 100 seconds for process x 1 & thread x 1

The application was able to handle phase 1, but fails at phase 2. We will use this result as the baseline for multi-process and multi-thread testing.

4.4.2 Process x 8 & Thread x 1

Response time for 8 processes and 1 thread is shown in Figure 4.4.3.

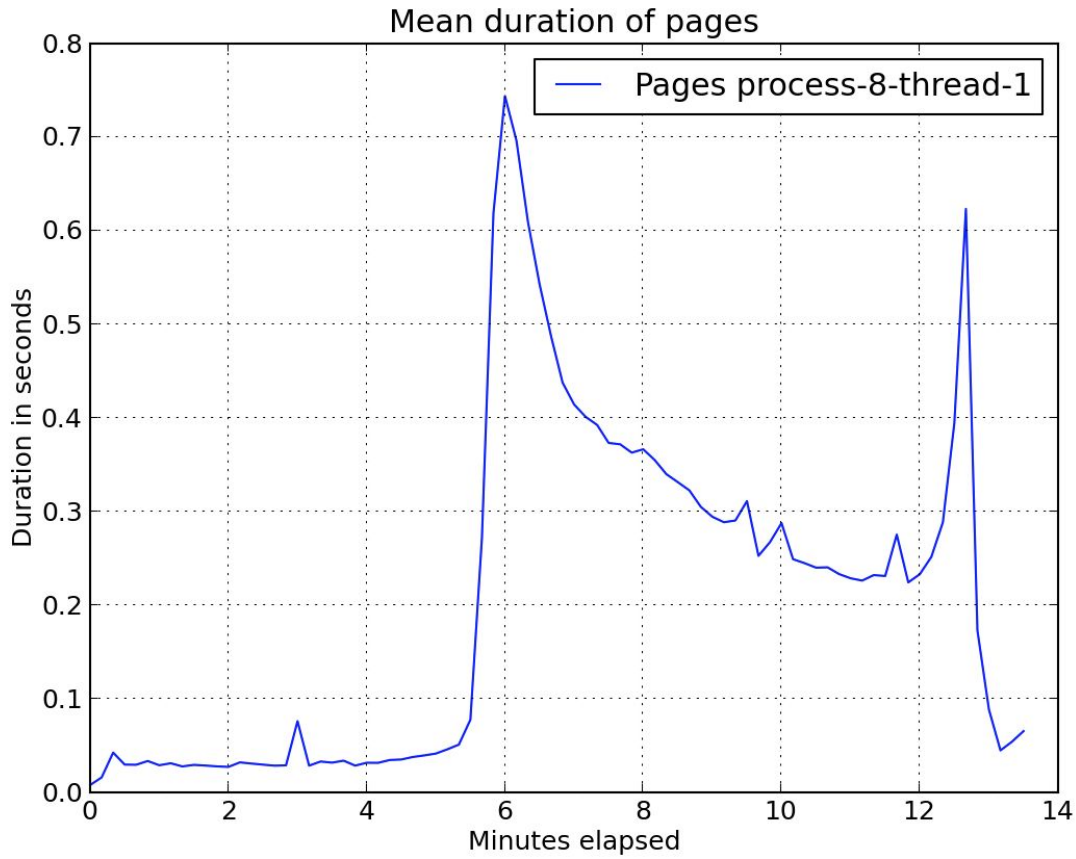


Figure 4.4.3. Response time for process x 8 & thread x 1

The response time around 330 seconds is shown in Figure 4.4.4.

Time (sec)	Response (ms)
310	46.04
320	51.05
330	77.66
340	271.57
350	618.86

Figure 4.4.4. Response time around 330 seconds for process x 8 & thread x 1

The application now can handle phase 5 and fails at phase 6, which is a huge improvement from the previous result

4.4.3 Process x 1 & Thread x 8

Response time for 1 process and 8 threads is shown in Figure 4.4.5.

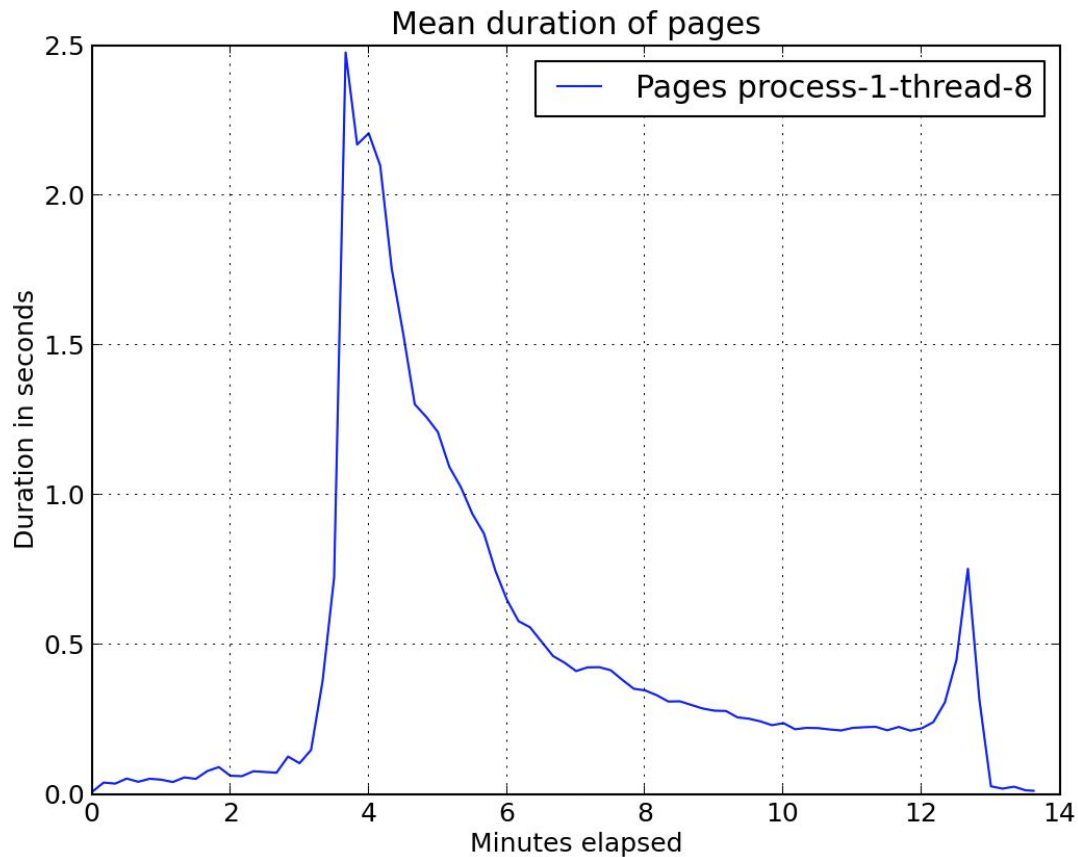


Figure 4.4.5 Response time for process x 1 & thread x 8

The response time around 190 seconds is shown in Figure 4.4.5.

Time (sec)	Response (ms)
170	126.35
180	104.05
190	148.40
200	378.11

210	723.94
-----	--------

Figure 4.4.5. Response time around 190 seconds for process x 1 & thread x 8

The application can handle up to phase 3 and fails at phase 4, which is a decent improvement from the baseline.

4.4.5 Summary

Here's the summary of optimization using multi-process and multi-thread.

Instance	Max Phase Handled
process x 1 & thread x 1	1
process x 8 & thread x 1	5
process x 8 & thread x 1	3

Figure 4.4.6. Summary of optimization with multi-process & multi-thread

From the table above, we can conclude that the more processes/threads we have, the more loads it can handle.

4.5 SQL Optimization

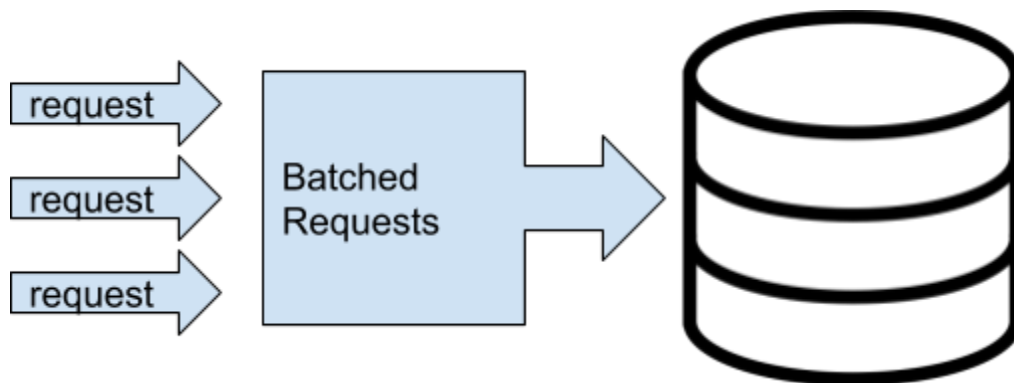


Figure 4.5.0. Diagram of SQL Optimization

Upon inspection, our app suffered in multiple places from the N+1 query problem and with redundant calls to the database, like fetching the current user multiple times in one page load. Using eager loading, the redundant

queries were batched together into one query call. Selection of extraneous columns was also filtered out. For instance, when batch loading the recipient and sender in conversations we only select the id, name, and email columns. Batch loading and removing redundant calls greatly reduced the number of hits we make to our database, as seen in Figure 4.5.1 and Figure 4.5.2.

```

Rendering conversations/matches.html.erb within layouts/application
Conversation Load (0.2ms) SELECT "conversations".* FROM "conversations" WHERE ((conversations.send_id = 1 OR conversations.recv_id = 1))
l app/views/conversations/matches.html.erb:3
User Load (0.1ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 2}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:5
Profile Load (0.1ms) SELECT "profiles".* FROM "profiles" WHERE "profiles"."user_id" = ? LIMIT ? [{"user_id", 2}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:20
CACHE User Load (0.0ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 1}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:5
User Load (0.1ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 4}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:6
Profile Load (0.1ms) SELECT "profiles".* FROM "profiles" WHERE "profiles"."user_id" = ? LIMIT ? [{"user_id", 4}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:20
CACHE User Load (0.0ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 1}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:5
User Load (0.1ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 11}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:6
Profile Load (0.1ms) SELECT "profiles".* FROM "profiles" WHERE "profiles"."user_id" = ? LIMIT ? [{"user_id", 11}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:20
CACHE User Load (0.0ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 1}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:5
User Load (0.1ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 6}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:6
Profile Load (0.2ms) SELECT "profiles".* FROM "profiles" WHERE "profiles"."user_id" = ? LIMIT ? [{"user_id", 6}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:20
CACHE User Load (0.0ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 1}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:5
User Load (0.1ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 8}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:6
Profile Load (0.1ms) SELECT "profiles".* FROM "profiles" WHERE "profiles"."user_id" = ? LIMIT ? [{"user_id", 8}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:20
CACHE User Load (0.0ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 1}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:5
User Load (0.1ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 12}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:6
Profile Load (0.1ms) SELECT "profiles".* FROM "profiles" WHERE "profiles"."user_id" = ? LIMIT ? [{"user_id", 12}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:20
CACHE User Load (0.0ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 1}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:5
User Load (0.1ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 15}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:6
Profile Load (0.1ms) SELECT "profiles".* FROM "profiles" WHERE "profiles"."user_id" = ? LIMIT ? [{"user_id", 15}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:20
CACHE User Load (0.0ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 1}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:5
User Load (0.1ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 5}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:6
Profile Load (0.1ms) SELECT "profiles".* FROM "profiles" WHERE "profiles"."user_id" = ? LIMIT ? [{"user_id", 5}, ["LIMIT", 1]]
l app/views/conversations/matches.html.erb:20

```

Figure 4.5.1 Queries loading matches.html before batching

```

Processing by ConversationsController#matches as HTML
User Load (0.6ms) SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ? [{"id", 1}, ["LIMIT", 1]]
l app/helpers/sessions_helper.rb:18:in 'current user'
Rendering conversations/matches.html.erb within layouts/application
Conversation Load (0.7ms) SELECT "conversations".* FROM "conversations" WHERE ((conversations.send_id = 1 OR conversations.recv_id = 1))
l app/views/conversations/matches.html.erb:3
User Load (0.2ms) SELECT "users".* FROM "users" WHERE "users"."id" IN (?, ?) [{"id", 2}, ["id", 1]]
l app/views/conversations/matches.html.erb:3
Profile Load (1.1ms) SELECT "profiles".* FROM "profiles" WHERE "profiles"."user_id" IN (?, ?) [{"user_id", 1}, ["user_id", 2]]
l app/views/conversations/matches.html.erb:3
User Load (0.2ms) SELECT "users".* FROM "users" WHERE "users"."id" IN (?, ?, ?, ?, ?, ?, ?) [{"id", 1}, ["id", 4}, ["id", 11}, ["id", 6}, ["id", 8}, ["id", 12}, ["id", 15}, ["id", 5]]
l app/views/conversations/matches.html.erb:3
Profile Load (0.2ms) SELECT "profiles".* FROM "profiles" WHERE "profiles"."user_id" IN (?, ?, ?, ?, ?, ?, ?) [{"user_id", 1}, ["user_id", 4}, ["user_id", 5}, ["user_id", 6}, ["user_id", 8}, ["user_id", 11}, ["user_id", 12}, ["user_id", 15]]
l app/views/conversations/matches.html.erb:3

```

Figure 4.5.2 Queries loading matches.html after batching

Additionally, the user matching function was rewritten as a custom SQL query. Originally, we first pulled 20 random users from the databases, then individually queried conversation for each of these random users. The custom query removes the needs for these individual checks. The custom SQL query is shown below in Figure 4.5.3. Based off database times alone, these changes seemed like optimizations. However, in actuality these changes made our app perform worse.

```
def randomShow(user)
  User.find_by_sql("SELECT * FROM users u
                  WHERE ((u.id != #{user.id}) AND
                        NOT EXISTS
                        ( SELECT id
                          FROM conversations c
                          WHERE (c.send_id = u.id) OR
                                (c.recv_id = u.id)
                        ))
                  ORDER BY RANDOM()
                  LIMIT 20;")
  #User.where.not(id: user.id).order("RANDOM()").limit(20)
end
```

Figure 4.5.3. Direct SQL query

Response time for SQL optimizations is shown in Figure 4.5.3.

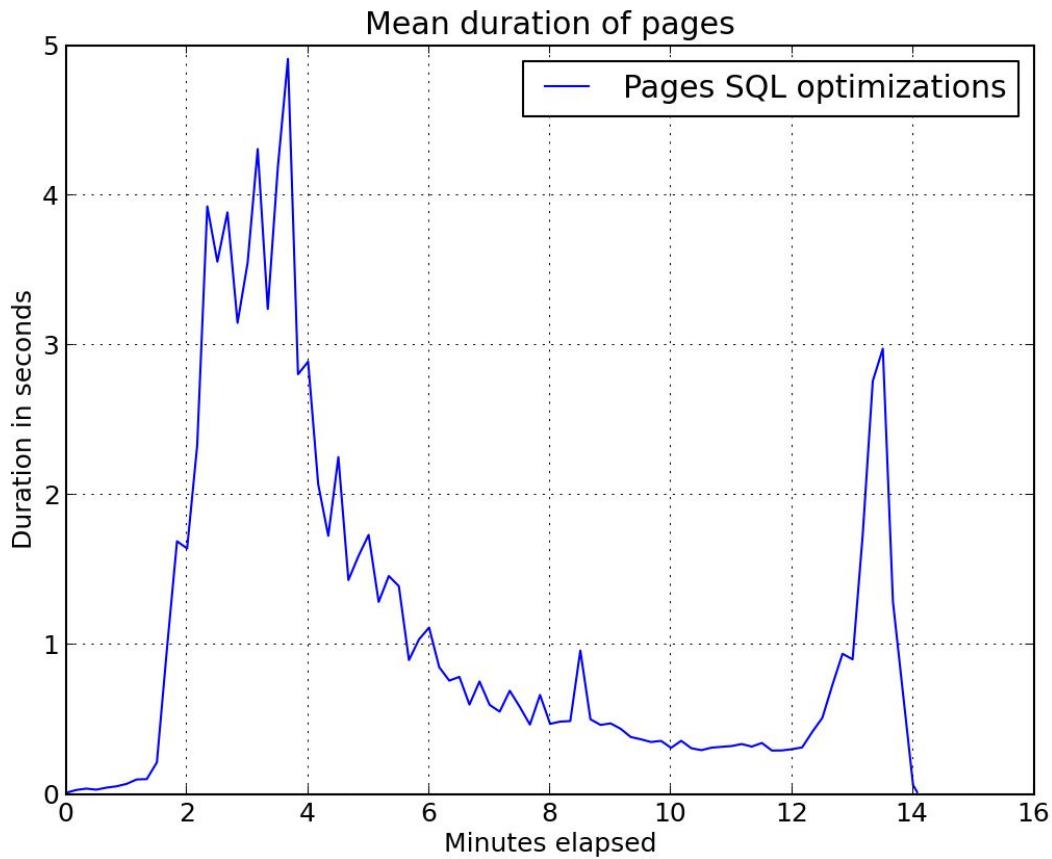


Figure 4.5.3. Response time for SQL optimization attempt

The response time around 90 seconds is shown in Figure 4.5.4.

Time (sec)	Response (ms)
70	99.79
80	102.25
90	213.71
100	970.89
110	1690.38

Figure 4.5.4. Response time around 90 seconds for SQL optimization attempt

The application can handle up to phase 1 and fails at phase 2, which is actually worse performance than the baseline. This is likely due to multiple factors. One reason the performance worsened is the time it takes to convert a batch of queries to active records is somewhat significant. Thus, even though the number of hits to the database have been lessened and

query times reduced, there was still an overall increase in time needed to fetch records from the database and load them into the application. Another factor that possibly led to this performance decrease is batching requests in the conversation controller. We only need the information for the sender or the recipient depending on which one the user is, but in batching the requests for this information we end up getting both. This results in getting one unneeded record from the database with every conversation, which is probably what ruined the performance.

4.6 Client-side Caching

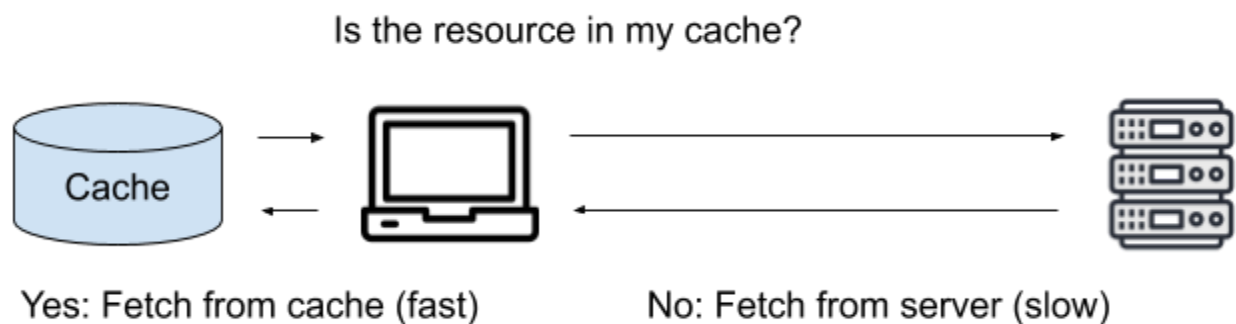


Figure 4.6.0. Diagram of client-side caching

Using Rails and its built-in methods for caching, we added client-side caching to our controllers for our app. More specifically, we cached all methods that would fetch a model instance (or class object). This was done using the 'if stale?' command, which allowed us to cache the objects based on model-specific parameters. We did client-side caching for the following controllers:

1. Events_controller.rb
2. Users_controller.rb

In our users controller, we focused on caching the `@user` object, which would fetch from the User model every time it was called or updated. Through client-side caching, we would reduce this by nesting such queries in the following conditional statement:

users_controller.rb

```
if stale?([@user, @user.updated_at, @user.events])
```

```
@user = User.find(params[:id])
end
```

This conditional caches the `@user` object based on the object itself, its update time, and the events it has stored. This was used when showing the user, updating the user, and editing the user information.

Similarly, with events, because we are fetching both users and events, we would use the previous conditional and the following conditional:

events_controller.rb:

```
if stale?([@event, @event.updated_at])
  @event = Event.find(params[:id])
end
```

This was used when showing events and updating events per user. We did not cache messages, because in our load testing, we typically assumed that users would not modify their messages after they have started them. Thus, it would be unnecessary to modify them. Similarly, with conversations, we would not have to cache them frequently, because the controller does not frequently fetch a conversation object from the database.

Using these modifications, we ran our tests with the following instance:

```
eb create -db.engine postgres -db.user u -db.pass password --envvars
SECRET_KEY_BASE=linksin --instance_type t3.micro --single
```

Running tests, we compared the results of client-side caching with the original baseline instance with no prior optimizations. Only client-side caching was used in this instance. The application was also deployed on a machine with 1 process and 8 threads. As shown from Figure 4.6.1 and Figure 4.6.2, the application tends to fail near 223 seconds. This, in comparison to Figure 4.4.5, suggests that there is little to no improvement, as both instances fail roughly near the middle and the end of phase 3, without reaching phase 4.

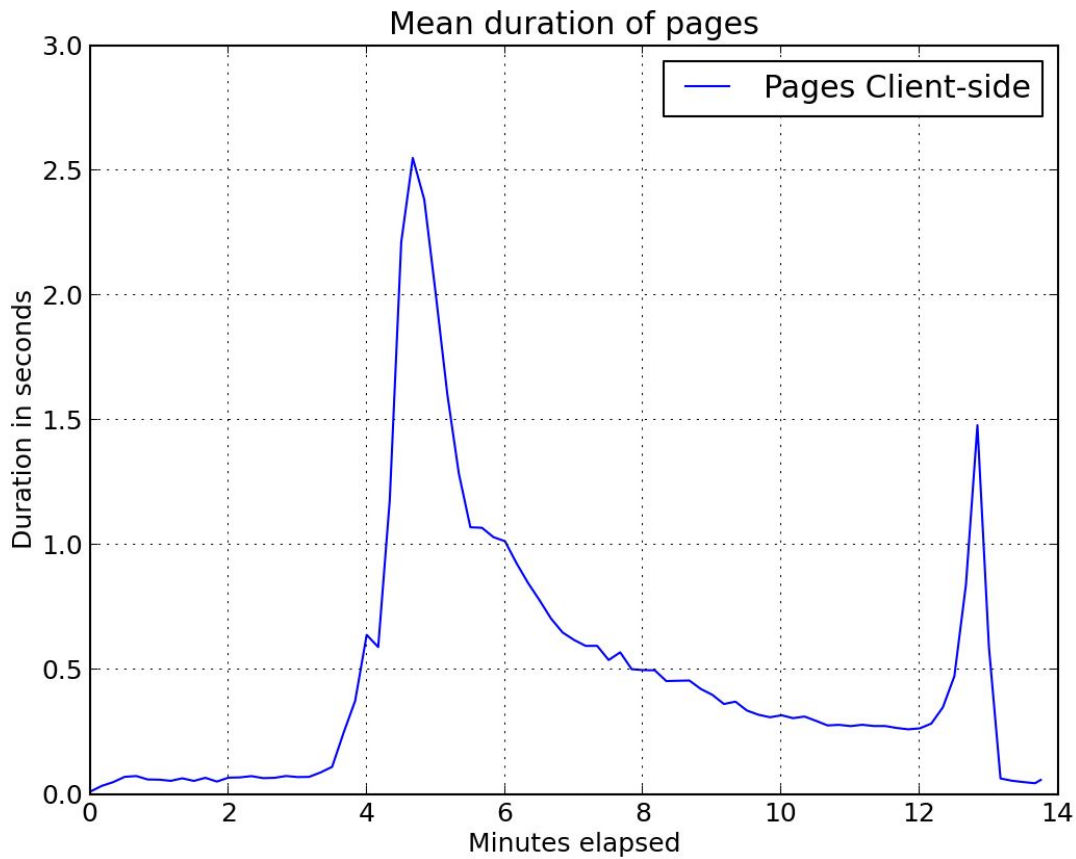


Figure 4.6.1. The change in response time over load testing script. Use of client-side caching for deployment.

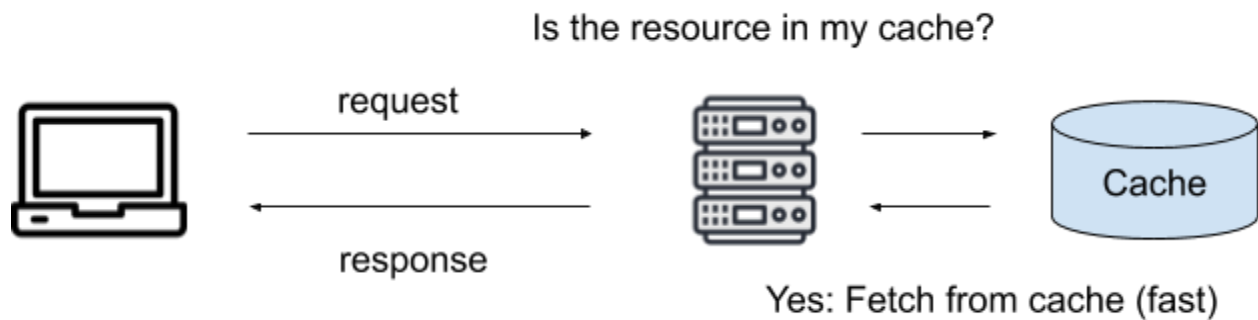
Time (sec)	Response (ms)
190	61.12
200	145.88
210	337.30
220	624.67
230	1205.47

Figure 4.6.2. The change in response time over load testing script near end of Phase 3 (180s - 240s).

According to lecturers Andrew Mutz and John Rothfels, and after further investigation on Tsung, we realized this result occurred due to the nature of our load-testing script. As Tsung times how well the server responds to

requests from the Tsung script, it does not use browser caching, as it specifies the requests made without bypassing requests through caching. Thus, in a commercial setting, client-side caching would improve our application performance, but as is, it does not do so effectively.

4.7 Server-side Caching



4.7.0. Diagram of server-side caching

Our server-side caching approach can be separated into 2 stages: we first performed fragment caching on critical views in our application, and then we deployed a memory cache using ElastiCache on AWS. The results from both iterations were then evaluated together to observe the benefits of combining server-side caching with client-side caching.

Fragment caching was implemented on 3 primary pages:

1. Our conversations page (Found in 'views/messages/index.html.erb')
2. Our card-swiping page (Found in 'views/users/_gamercard.html.erb')
3. Our event-creation page (Found in 'views/layouts/_eventcard.html.erb')

Implementing fragment caching on these 3 pages helped cache parts of the view that would be fetched continuously, thus reducing the number of loads when fragments of the view contained the same information. This was particularly important in these 3 pages because each page fetched and queried from our database, thus producing a performance bottleneck.

For our events page, we originally had the following code:

```
<% if @user.events.empty? %>
```

```

...
<% @user.events.each do |event| %>
...
<% end %>
<% end %>

```

This would mean that we are fetching all the events from a particular user every time we load the events page. However, unless the user modifies an event, or unless a user adds or removes an event, the list of events do not change. Hence, using Russian doll caching, we can cache the events fragment as follows:

```

<% cache(Event.cache_key_for_eventList) do %>
...
<% @user.events.each do |event| %>
  <% cache(Event.cache_key_for_event(event)) do %>
    ...
    <% end %>
  <% end %>
<% end %>

```

This method of caching calls two functions (found in our model 'event.rb'):

```

def Event.cache_key_for_event(e)
  "event-#{e.id}-#{e.updated_at}-#{e.members.count}"
end

def Event.cache_key_for_eventList
  "eventList-#{Event.maximum(:updated_at)}-#{Member.maximum(:updated_at)}"
end

```

The inner cache fragment caches each individual event based on its id, update time, and the members it contains. Doing so will generate a unique

key for the event every time it is modified. Additionally, the outer cache fragment caches the nested events list. This is because the events for a particular user is typically unchanged unless they have been invited to a new event. Hence, we only update the latest cached events if the entire database of Events, and the corresponding database of Members in those events have been updated.

With both the conversations and card-swiping page, we originally employed a similar method of Russian doll caching, but realized there were some issues:

1. With the conversations page, the database of messages and conversations would only be updated when a user sends or edits a message. Hence, if a user had multiple conversations open, the fragment would not swap between conversations, but instead, displayed the cached version (unless a new message pops up).
2. With the card-swiping page, the database of users would only be updated only if a new user signs up to our application. Hence, if we used Russian doll caching, every user would be swiping the same set of cards until a new user signs up (because they are all retrieving the most updated cached version of the list)

To avoid reducing the user experience of our application, we chose not to use fragment sharding for these particular pages, because it would interfere with the way the application functioned. Instead, our conversations and cards page only cached each individual user, conversation and message, as shown below:

`__gamercard.html.erb`:

```
<div class="LI_GamerCardContainer">
...
  <% @userlist.each do |user| %>
    <% cache(User.cache_key_for_user(x)) do %>
      ...
    <% end %>
  <% end %>
</div>
```

User.rb:

```
def User.cache_key_for_user(x)
  "user-#{x.id}-#{x.updated_at}-#{x.events.count}"
end
```

Index.html.erb (for conversations and message):

```
<div class="columns" >
  <div class="column is-3" style="overflow: auto;">
    <% @conversations.each do |conversation| %>
      <% cache(Conversation.cache_key_for_conversation(conversation)) do %>
        ...
      <% end %>
    <% end %>
  ...
  <div style="height: 90%; overflow-y: scroll" id="message-window">
    <% @messages.reverse.each do |message| %>
      <% cache(Message.cache_key_for_message(message)) do %>
        ...
      <% end %>
    <% end %>
  </div>
  ...
</div>
```

Conversation.rb:

```
def Conversation.cache_key_for_conversation(x)
  "conversation-#{x.send_id}-#{x.recv_id}-#{x.updated_at}"
end
```

Message.rb:

```
def Message.cache_key_for_message(x)
  "message-#{x.user_id}-#{x.conversation_id}-#{x.created_at}"
end
```

Finally, we also combined this with ElastiCache, with help from documentation provided by Teaching Assistant Rishab Ketan Doshi. We first began by creating a memcache cluster on AWS ElastiCache. We then deployed our existing Rails app with the following changes to our Gemfile and production.rb file:

Gemfile:

```
group :production do
  gem 'pg', '1.1.4'
  gem 'dalli-elasticache'
end
```

Production.rb:

```
endpoint =
  "linksin-memcache-2.5sqcdv.cfg.usw2.cache.amazonaws.com:11211"
elasticache = Dalli::ElastiCache.new(endpoint)
config.cache_store = :dalli_store, elasticache.servers,
{:expires_in => 1.day, :compress => true}
```

From there, we deployed our instance using the following commands:

```
eb create -db.engine postgres -db.user u -db.pass password --envvars
SECRET_KEY_BASE=linksin --instance_type t3.micro --single linksin-memcache
```

According to Rishab, this would fail, as our current cache, deployed on the LinksIn security group, did not allow our instance, which is on a separate security group, to connect through a TCP connection.

Hence, we first added a new inbound custom TCP rule to our memcache cluster, so that it would enable TCP connections to our instance. From there, we re-deployed our instance, and it deployed as intended.

Combining this with the optimizations we made in Section 4.6, we deployed our instance with the following configurations:

```
eb create -db.engine postgres -db.user u -db.pass password --envvars
SECRET_KEY_BASE=linksin --instance_type t3.micro --single
```

Running tests, we compared the results of both client-side caching and server-side caching with the original baseline instance. Only client-side and server-side caching was used in this instance. The application was also deployed on a machine with 1 process and 8 threads. As shown from Figure 4.7.1 and 4.7.2, the application tends to fail near the middle of Phase 4. This, in comparison to Figure 4.4.5, suggests that server-side caching improves response time with increased loads.

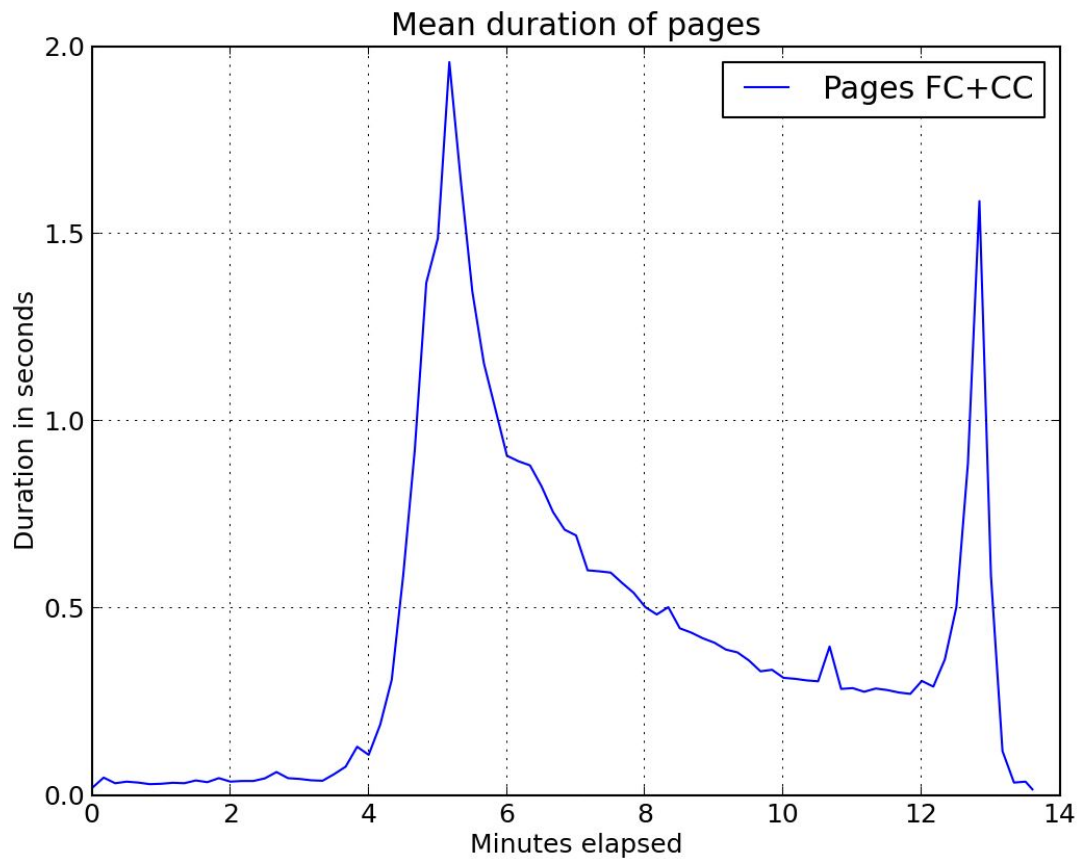


Figure 4.7.1. Mean duration of pages response time with Fragment Caching (FC) and Client Caching (CC). Memcache Cluster was also used in ElastiCache

Time (sec)	Response (ms)
230	195.31
240	174.50
250	452.51
260	931.32

270	1342.46
-----	---------

Figure 4.7.2. Table of response times near phase where application begins to fail. Application was deployed with fragment caching, client caching, and memcaching.

As hypothesized, the impacts of fragment caching and server-side caching using ElastiCache increased the number of 304 HTTP Responses, shown in Figure 4.7.1. These responses indicated that the cached-copy of an object or data on the client side is still up-to-date with that on the server. Hence, by caching frequently used data, we reduced the time taken typically to read and write data.

4.8 Read-Slaves

For most applications including Linksin, reads are much more common than writes. Thus, we can have read-slave databases that handle only the reads and master database that handles only the writes and and apply the changes to the read-slaves. Then we can easily horizontally scale the database by increasing the number of read-slaves. For simplicity, we will be implementing one master database and one read-slave.

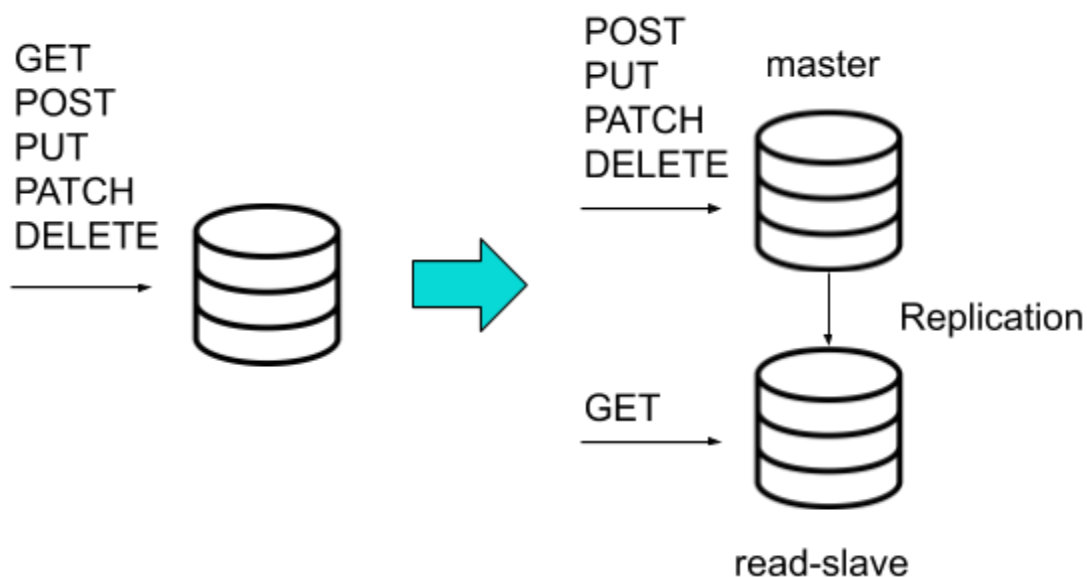


Figure 4.8.0. Diagram of read-slave

First of all, we need to change some code in our application to redirect reads to read-slave and write to the master. We tell the rails application where it can find the two databases `primary` and `primary_replica`.

```
# config/database.yml
...

production:
  primary:
    adapter: postgresql
    database: <%= ENV['RDS_DB_NAME'] %>
    encoding: UTF8
    host: <%= ENV['RDS_HOSTNAME'] %>
    password: <%= ENV['RDS_PASSWORD'] %>
    port: <%= ENV['RDS_PORT'] %>
    username: <%= ENV['RDS_USERNAME'] %>
  primary_replica:
    adapter: postgresql
    database: <%= ENV['RDS_DB_NAME'] %>
    encoding: UTF8
    host: ""
    password: <%= ENV['RDS_PASSWORD'] %>
    port: <%= ENV['RDS_PORT'] %>
    username: <%= ENV['RDS_USERNAME'] %>
    replica: true
```

Notice the hostname of `primary_replica` is empty at the moment. This is okay for now because we will be creating a replicated database after first deployment through AWS RDS console. Then we update the host with the endpoint of the replica, and redeploy the application again.

Next, we need to tell the rails application that there are two databases `primary` and `primary_replica` it needs to be aware of.

```
# app/models/application_record.rb
class ApplicationRecord < ActiveRecord::Base
```



```

self.abstract_class = true
connects_to database: { writing: :primary, reading: :primary_replica }
end

```

Then, we need to activate the database connection switching. We want the rails to redirect all the POST, PUT, DELETE and PATCH requests to the primary database, and GET and HEAD requests to primary_replica if it exceeds the delay of 2 seconds. If the delay is within 2 seconds, we will handle these requests on primary database to guarantee “read your own write” principle.

```

# config/application.rb
module Linksin
  class Application < Rails::Application

    ...

    config.active_record.database_selector =
      { delay: 2.seconds }

    config.active_record.database_resolver =
      ActiveRecord::Middleware::DatabaseSelector::Resolver

    config.active_record.database_resolver_context =
      ActiveRecord::Middleware::DatabaseSelector::Resolver::Session

  end
end

```

Finally we deploy the application, replicate the database, then update the hostname of the replica as discussed earlier.

```

# config/database.yml

```

```
...  
  
production:  
  ...  
  primary_replica:  
    adapter: postgresql  
    database: <%= ENV['RDS_DB_NAME'] %>  
    encoding: UTF8  
    host: "linksin.civkgqjsrtyg.us-west-2.rds.amazonaws.com"  
    password: <%= ENV['RDS_PASSWORD'] %>  
    port: <%= ENV['RDS_PORT'] %>  
    username: <%= ENV['RDS_USERNAME'] %>  
    replica: true
```

Redeploy the application, then we are good to go.

We tested read-slave on m4.4xlarge instance to make sure application server does not become the bottleneck of the performance.

4.8.1 Master database x 1

The response time without read-slave is shown in Figure 4.8.1. Note we are reusing the test result from 4.1.5.

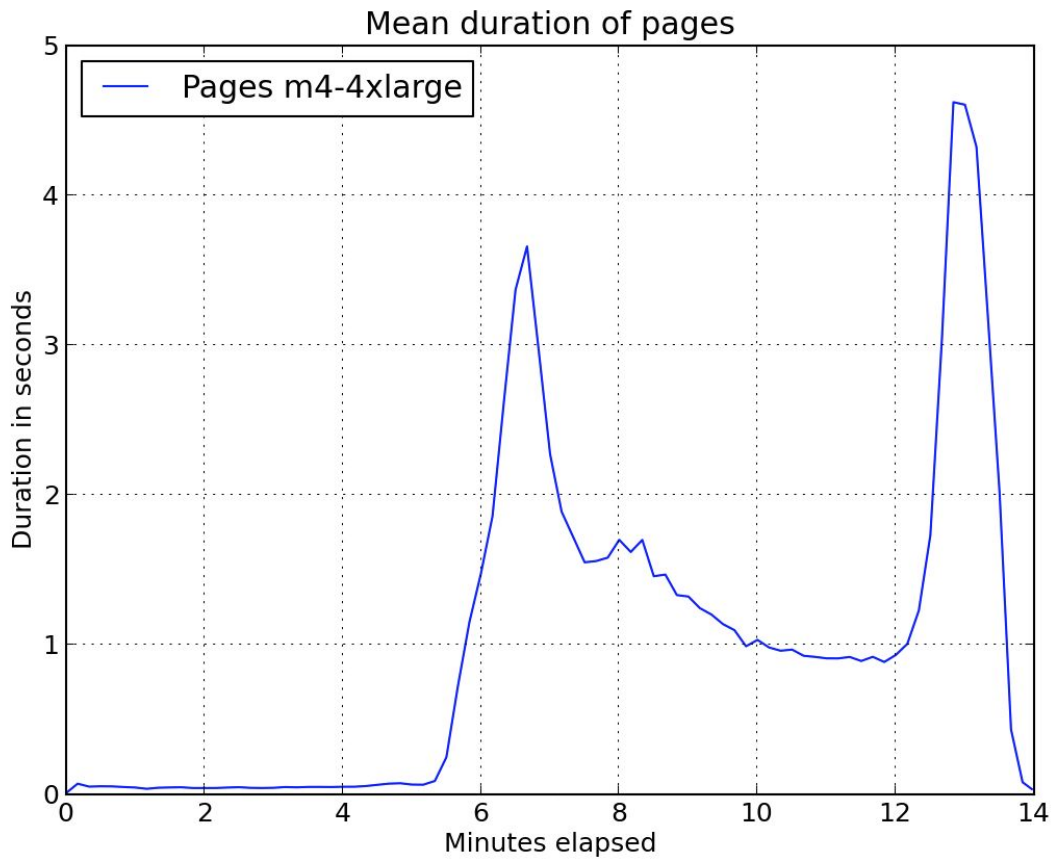


Figure 4.8.1. Response time w/o read-slave

The response time around 320 seconds is shown in Figure 4.8.2.

Time (sec)	Response (ms)
300	65.65
310	64.84
320	89.74
330	248.16
340	723.47

Figure 4.8.2. Response time around 320 seconds w/o read-slave

The application successfully handles phase 5 but fails at phase 6. We will use this as the baseline to check the performance improvement.

4.8.2 master database x 1 & read-slave x 1

The response time with read-slave is shown in Figure 4.8.3.

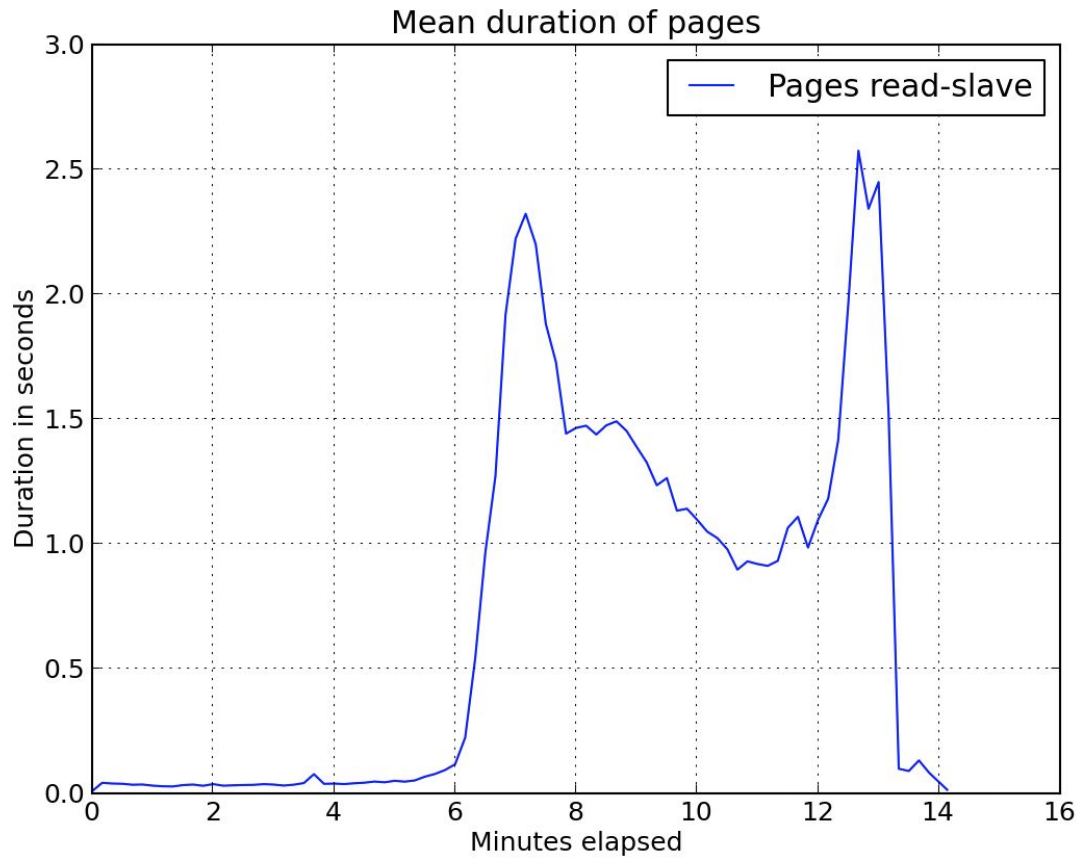


Figure 4.8.3. Response time w/ read-slave

The response time around 360 seconds is shown in Figure 4.8.4.

Time (sec)	Response (ms)
340	77.82
350	93.63
360	116.66
370	223.71

380	542.53
-----	--------

Figure 4.8.4. Response time around 360 seconds w/ read-slave

The application can now handle phase 6 and fails at phase 7, which is still an improvement from the previous section.

4.8.3 Summary

Here's the summary of optimization with read-slave.

Instance	Max Phase Handled
w/o read-slave	5
w/ read-slave	6

Figure 4.8.5. Summary of optimization with read-slave

As you can see in Figure 4.8.5, there is a slight performance improvement with one instance of read-slave. As discussed in the beginning of this section, read-slave databases can be scaled horizontally. If we have multiple read-slaves, we would have observed further performance improvement.

5. Further Development

5.1 Database Sharding

Sharding involves the process of splitting up the application's data into two or more smaller chunks, partitioning these chunks across separate tables. Using Rails 6's multiple database feature we initially attempted to pseudo-shard our databases by splitting our user records among multiple databases via email. However, as a model can only be connected to one database, this would have meant splitting our user model into three separate models, as well as figuring out how to combine them again when generating matches and conversations. This requires complex logic and a good deal of additional time to implement that we did not have. Thus, a hold has been put on attempts at database sharding until Rails's multiple database feature supports it.

5.2 Improving Events

At the time of our project submission, users can create events and invite matched friends to the event. While events will hold information about which users are invited, there is currently no functionality for users to accept invitations or join events. Given more time for development of the application, this is an area we'd definitely like to explore and improve upon further.

5.3 Real-time Messaging

In its current state, our messaging feature only loads new messages when the page is refreshed. Thus, the user will not get new messages until they send out a message, which causes a rerender, or if they reload the page. Obviously this is not ideal, as messages could go unseen for long periods of time. Additionally, real-time messaging capability is expected of any modern-day chat application, making real-time messaging via ActionCable is a primary feature to add on in further development.

5.4 Push Notifications

At the moment, users do not get notified when someone messages them or if they are invited to an event. If the application is running on different tab of the browser, then we can activate the sound notification when there is a new message or invite to keep the user up to date, and thus improve the user experience.

5.5 Linking External Accounts

Currently we only have a text field for users to specify their external gaming accounts instead of a dedicated field to store these information (There is nothing preventing users from inputting something else). Once we have OAuth working, we would be able to import the games they liked, and other users can view them to better know about them.

5.6 Combining All Optimizations

We implemented every optimization technique on different branches. We could merge all these branches to the master and see how much performance improvement we make overall.

6. Conclusion

At the beginning of this quarter we were Rails and Ruby newbies with no experience in building scalable web apps. Over the course of this quarter, we managed to build, deploy, and scale a responsive web application on Rails that allows gamers to connect with each other via a Tinder-esque interface. Using agile development and git, we were able to work on features simultaneously. With Travis CI integration, we were able to make sure that any merges we attempted would not break our application, allowing us to merge our feature branches together without much difficulty. We also learned a good deal about the intricacies of deploying applications to AWS and how to troubleshoot when an Elastic Beanstalk environment is not instantiating or terminating as planned.

Initially, load testing was rather difficult for us to implement due to critical path, user matching, requiring two users to swipe on each other. After creating a load-testing branch that loosened this constraint, we were able to create a script to test our critical path. We found that our app's baseline performance was less than ideal due to factors such as costly loads of large, unused JavaScript files. Thus, different optimizations were attempted individually; these optimizations are vertical scaling, horizontal scaling, paginations, multiple processes and threads, SQL optimizations, client-side caching, serve-side caching, and read slaves. Though we did not get the chance to combine our optimizations due to AWS accessibility issues, we were still able to achieve substantial improvements. Our baseline application was only able to make it through phase 3. Our best optimizations, however, were able to make it through phase 6, handling 5x more users per second. We found that scaling and pagination provide the most significant performance enhancements.