CS291A
SCALABLE INTERNET SERVICES

Social Network App

Krishna Chaitanya Gudipati

Paul Ren

Benu Madhab Changmai

1. Introduction

The Internet has become an indispensable part of people's lives. With billions of people exploring online all the time, it is vital for any successful web application to be able to handle a large number of requests at any given time. For this project, we created a social network web service - **GauchoBook** application using Ruby on Rails, then we used Tsung load testing tool to see how well our app handles incoming requests as they grow in number. Furthermore, once our app hits the bottleneck when processing requests, we investigate the root cause of such bottleneck and use various remedy techniques to address those issues and see whether our app can overcome the bottlenecks. We will display and features and layout of our application in the next section. In Section 2, we describe each feature that our application supports. In Section 3 we dive into the data models followed by Section 4 which talks about our load testing scenarios as well as the optimization strategies we deployed to tackle the associated bottlenecks. We lay down the future scope of our work in Section 5 and conclude in Section 6.

2. Web App Features

In our GauchoBook application, we have tried to put up a very generic version of any modern day social media application and haven't tried to put forward something novel in terms of functionalities. This is because our primary objective is to delve into the complexities of the features of a social networking app and how to tackle the various bottleneck challenges associated with them. Our web app is designed to help people communicate, interact, and share their life stories online with other users whom they may or may not know. We only ask the user to do a quick sign up before they can start to freely explore the various features we have, including viewing and creating posts, commenting on existing posts, adding a user as a friend, chatting with a friend privately, and editing your own profile page or browsing profiles pages of others. Figure 1. shows the action flow that we would expect from a normal user.

2.1 Login in/Sign up

The first step to use our app would be a signup step for first time user or sign-in step for a user who already has an account. This allows us to provide users with more personalized features.

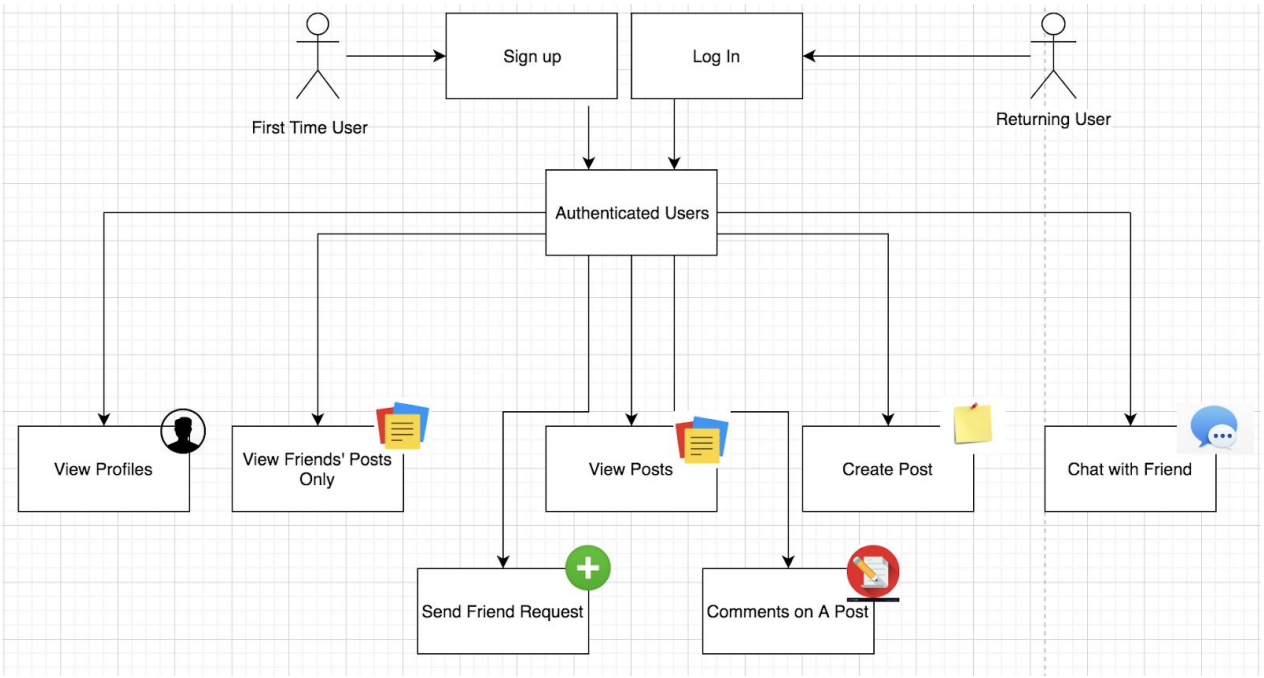


Figure 1. User action flowchart

Figure 2. Signup/Login page

2.2 Homepage

After a user logs in, he or she is going to be redirected to the homepage, where the user can choose different actions at will.

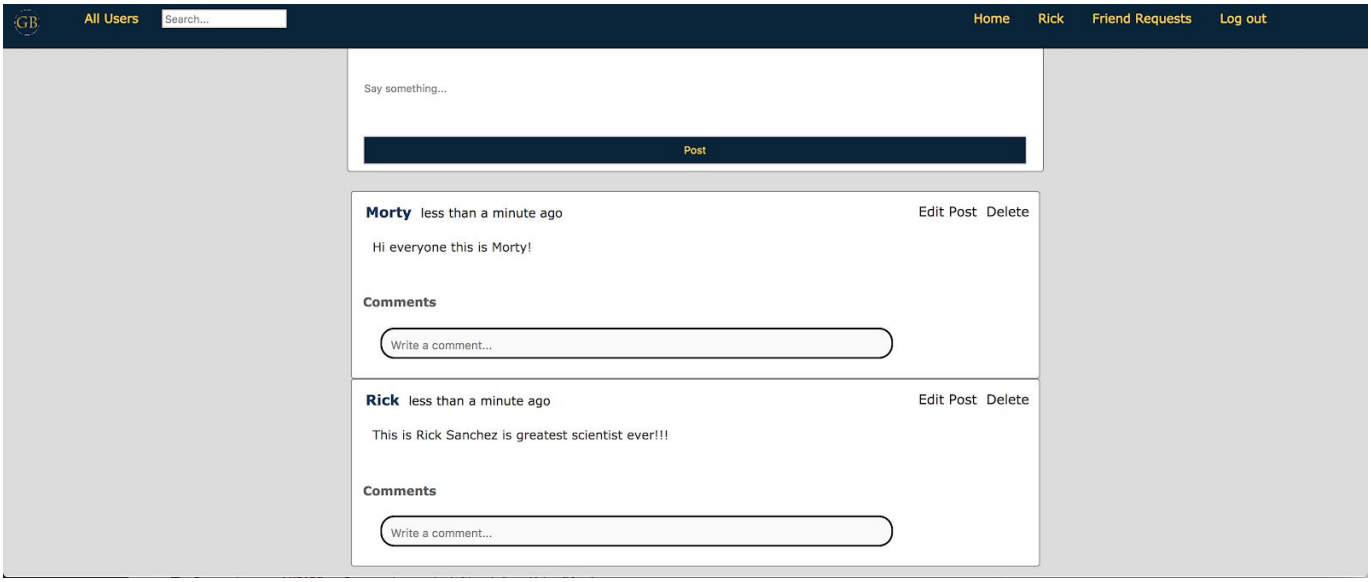


Figure 3. Homepage

2.3 Creating a New Post

After logging in, the user can start to share whatever cool ideas they have.

Figure 4. Creating a new post

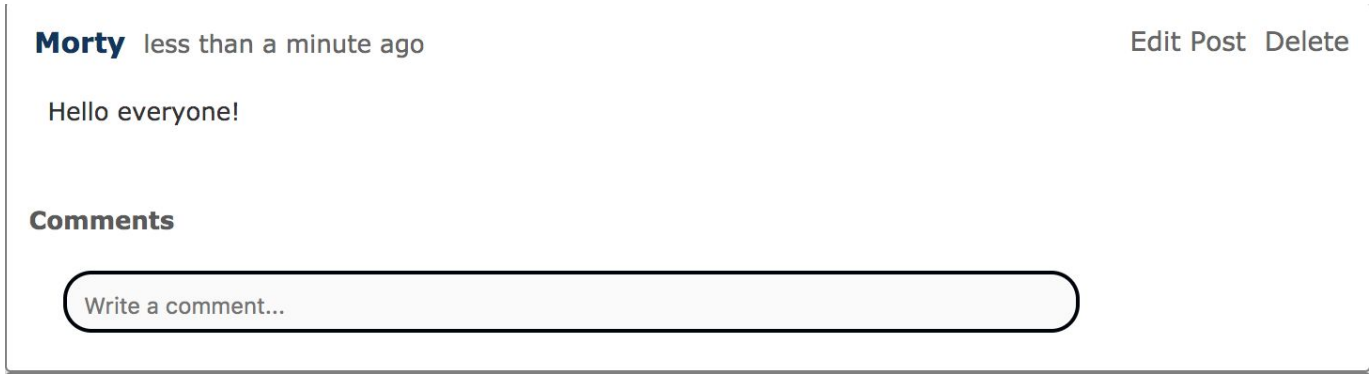


Figure 5. New post created

2.4 Commenting on a post

Users can also make meaningful or meaningless comments on posts of their friends and start to have some active conversation.

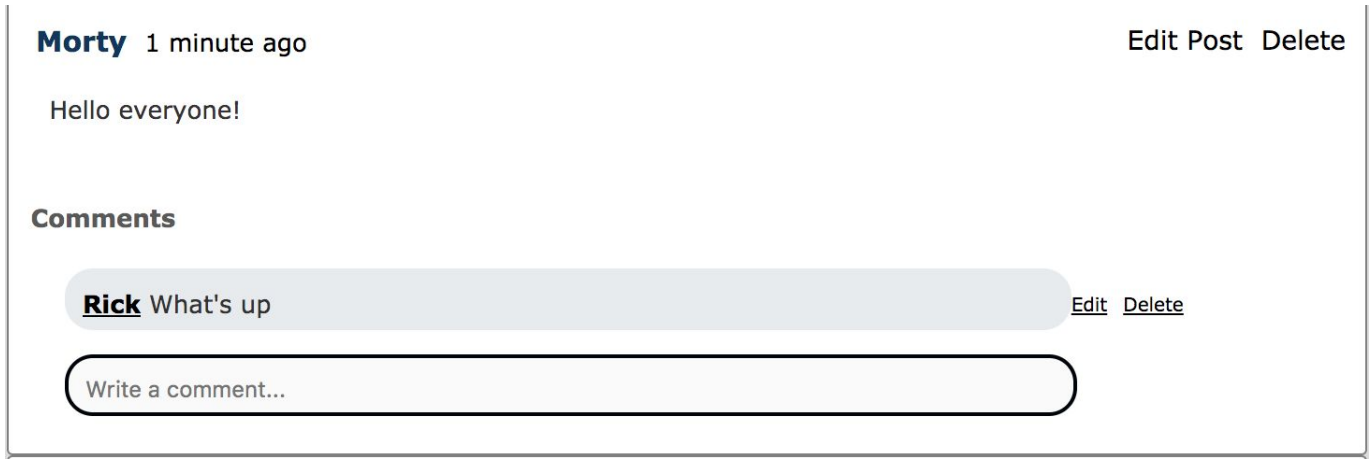


Figure 6. Commenting on a Post

2.5 Viewing All Users

A user can check and see who are all the other users that are currently logged in the app. The user can click into their name to view their profiles if he wants to know more about a certain user.

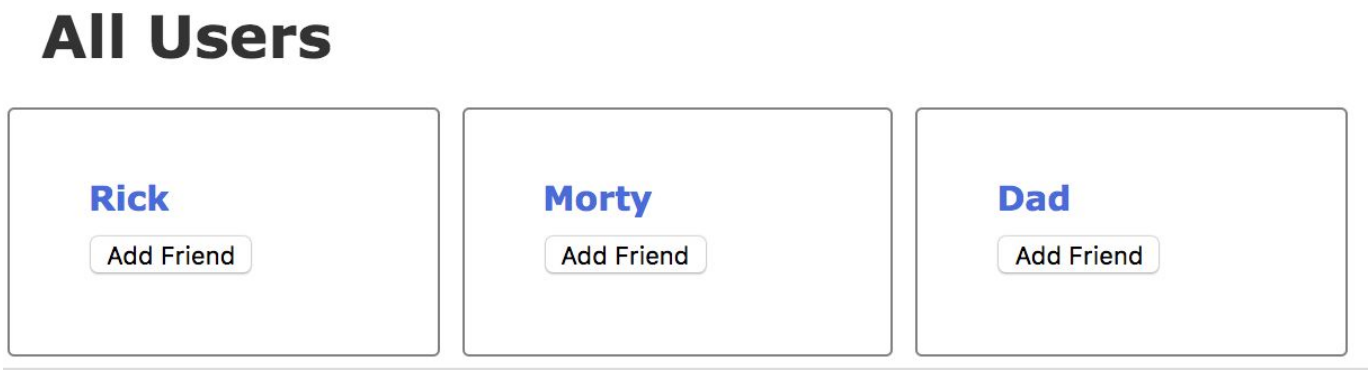


Figure 7. All Users

2.6 Search for users

In the user page, users can search people by providing their username as search query.

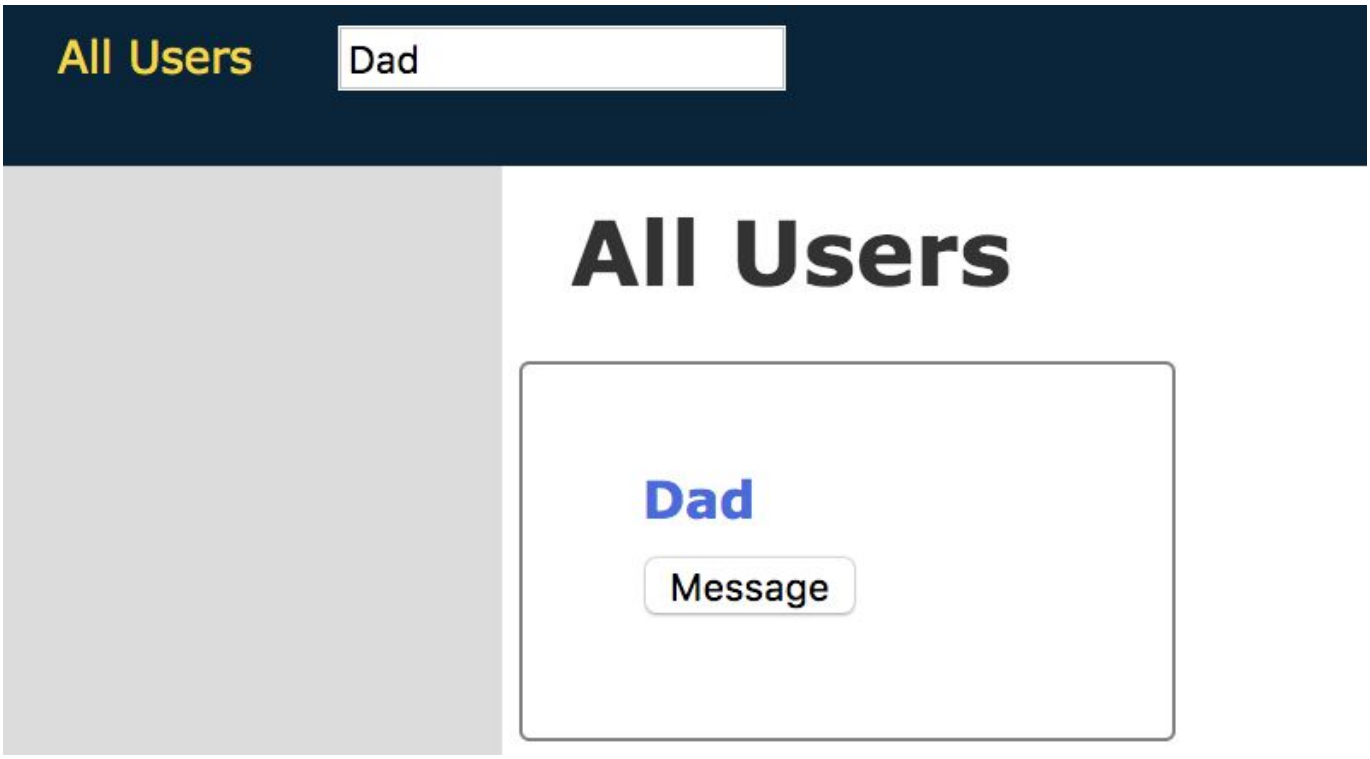


Figure 8. Search for a Specific User

2.7 Add a Friend

Moreover, users can add those they like to be their friends by sending friend requests to other people. After one user issues a friend request, the other user will immediately receive that request and can choose whether to accept it or reject it.

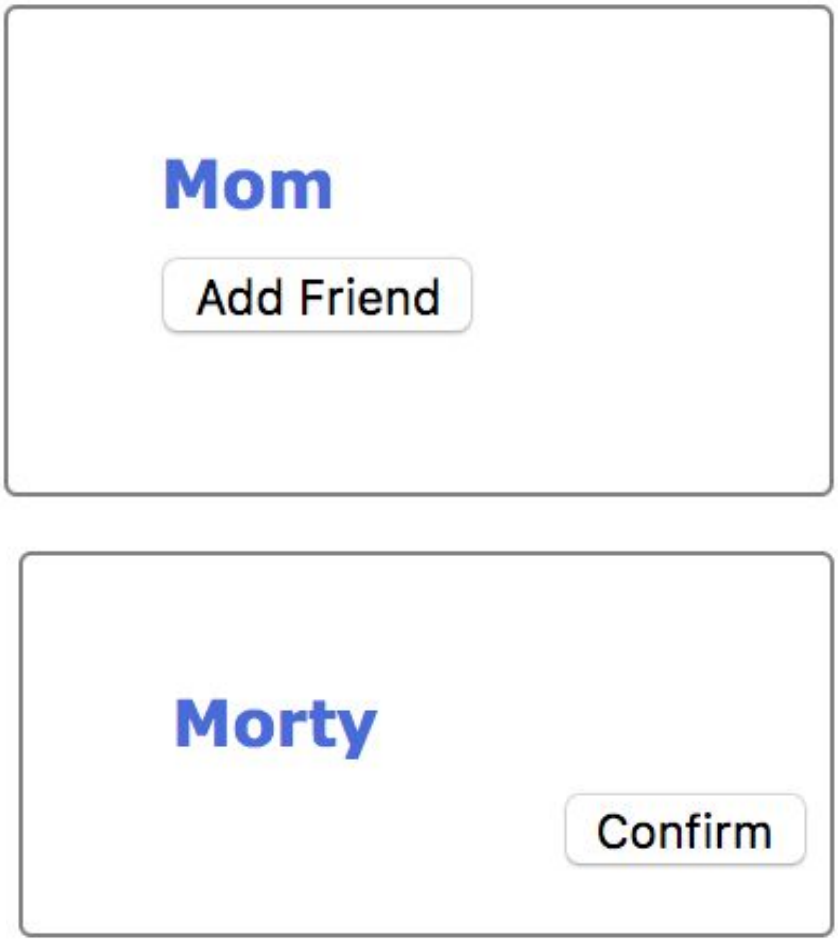


Figure 8. Adding New Friend & Receiving Friend Request

2.8 View a Profile

Each user can maintain his or her unique profile page displaying relevant information that he or she would like other people to see.

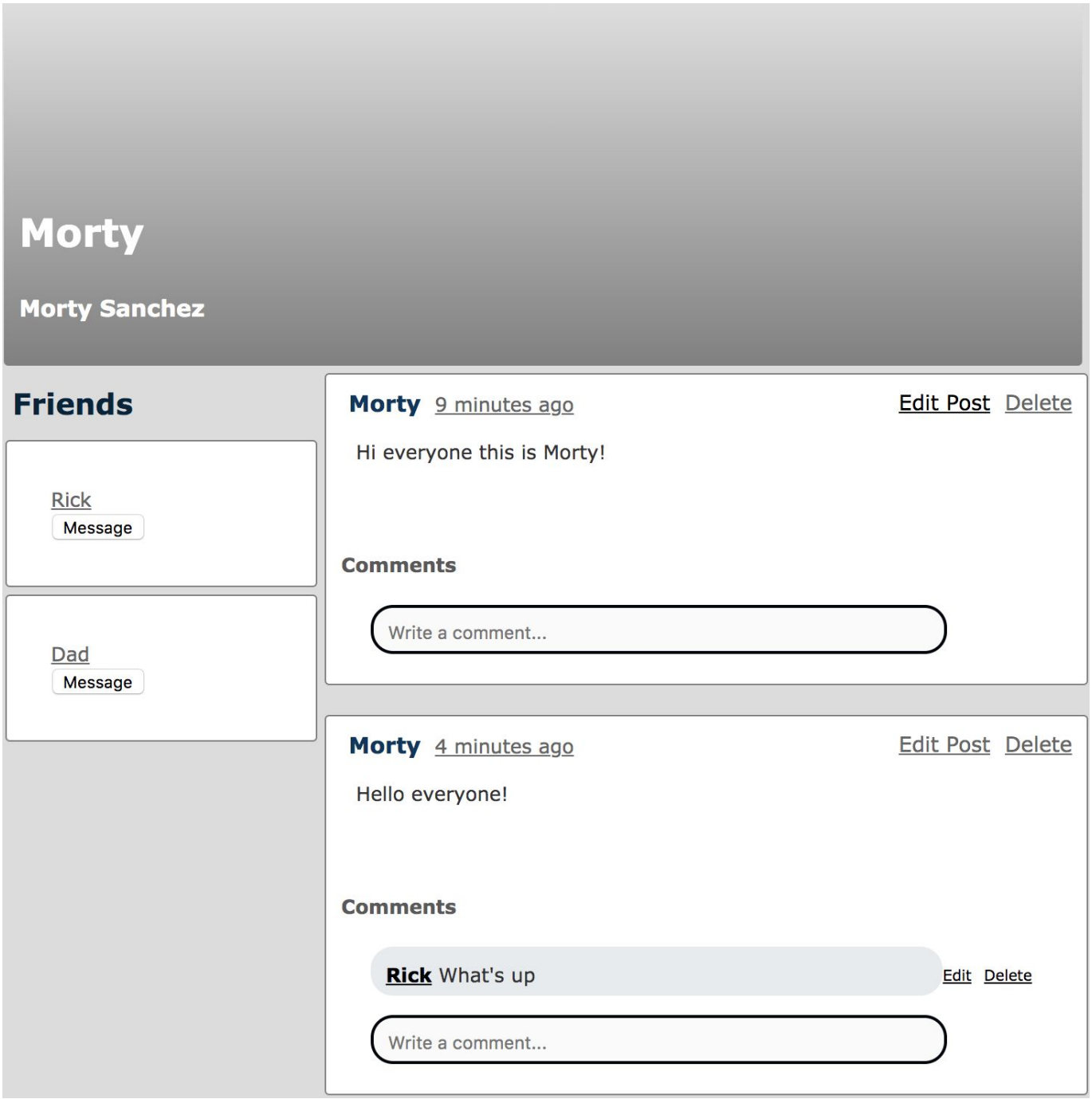


Figure 9. Basic Profile Page

2.9 Message a Friend

Users can directly message their friends and start a conversation with that person.

Figure 10. Adding New Friend & Receiving Friend Request

3.Data Models

In order to achieve the desired effect of the application and to ease maintenance in the future, we've created many models to represent various of entities existing in the user-application interaction process. Figure 9. is an overview represented in a entity-relationship model:

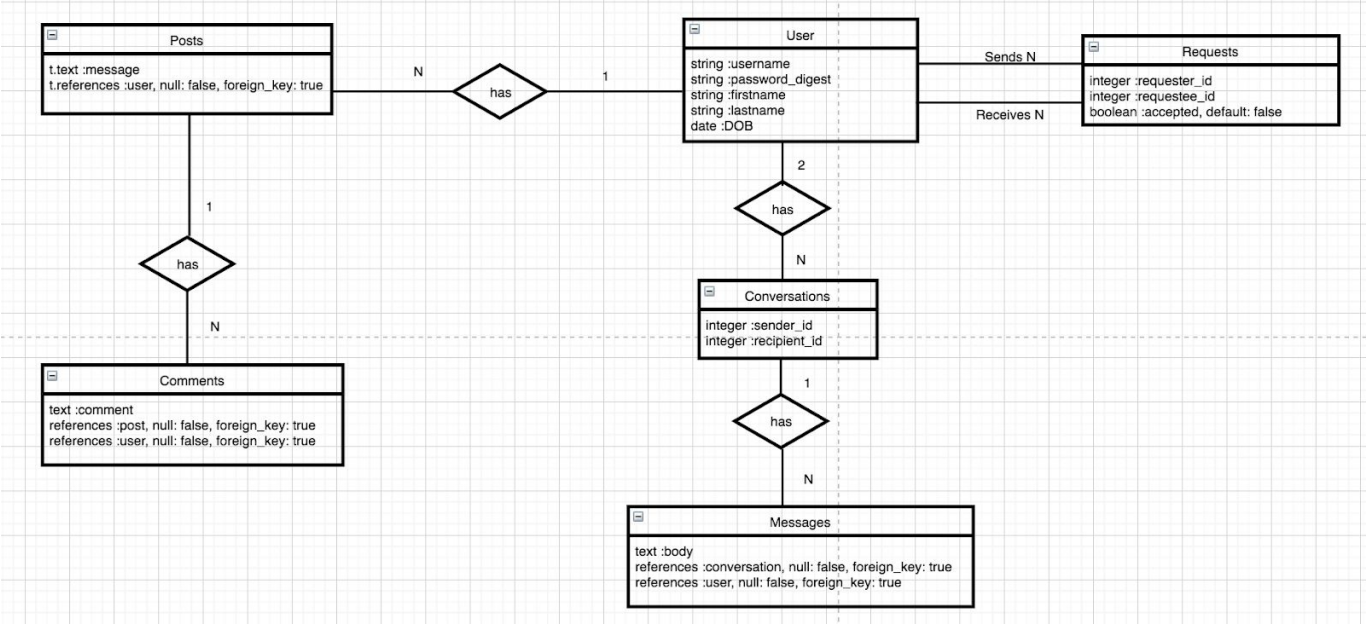


Figure 11. Entity Relationship Model

3.1 Posts Model

The posts model contains the message the post is displaying. The user to which this post belongs, and auto-generated timestamps.

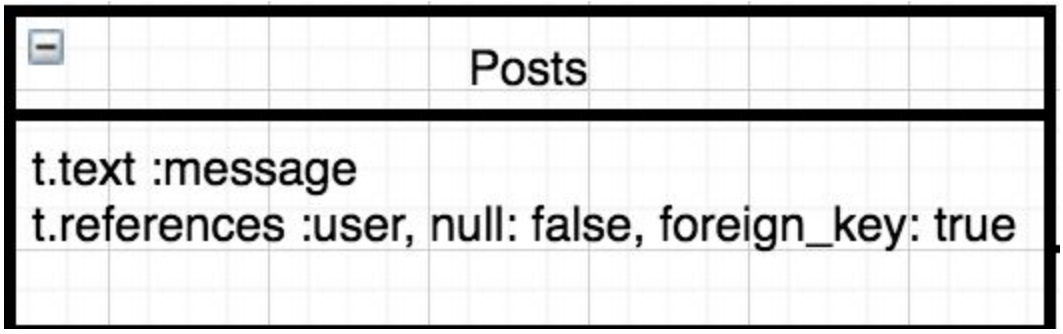


Figure 12. Posts Model UML

3.2 Comments Model

The comments model as well contains the message the comment is displaying. It belongs to a certain user and a certain post, so it has have those as its foreign keys.

Figure 13. Comments Model UML

3.3 Users Model

The users model contains useful information about a particular user. We store the username, password digest, first and last names, and date of birth of a user.

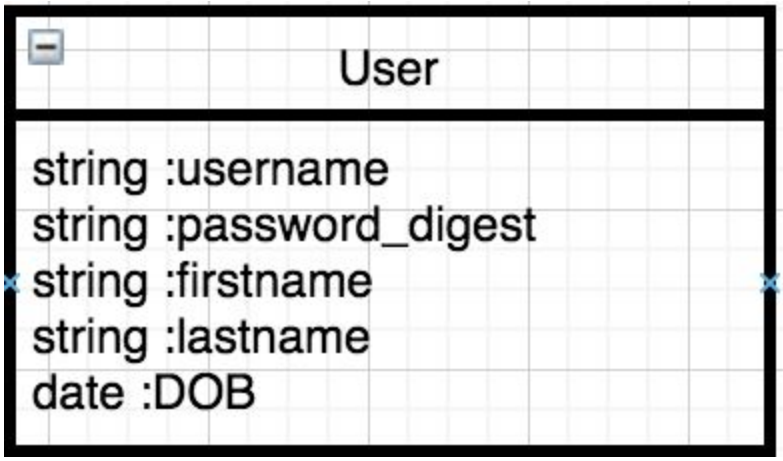


Figure 14. Users Model UML

3.4 Requests Model

The requests model contains the ids of a requester and the corresponding requestee, and a boolean value indicating whether this request has been accepted or not.

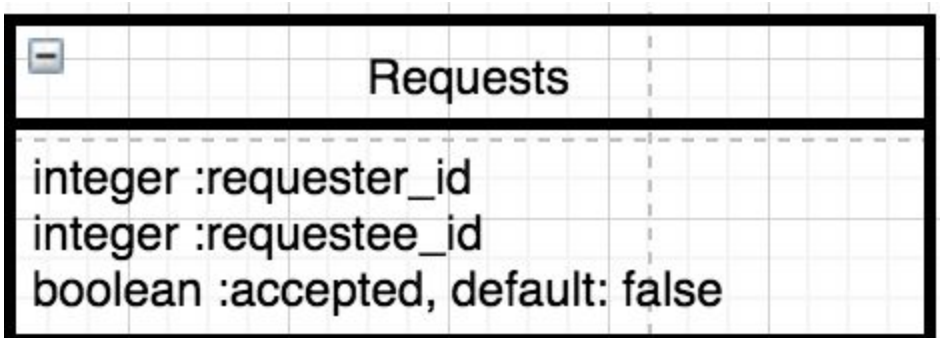


Figure 15. Requests Model UML

3.5 Conversations Model

The conversations model contains the ids of sender and receiver of a certain message. In this way, we can figure out who sent the message to whom and display it to the two users accordingly.

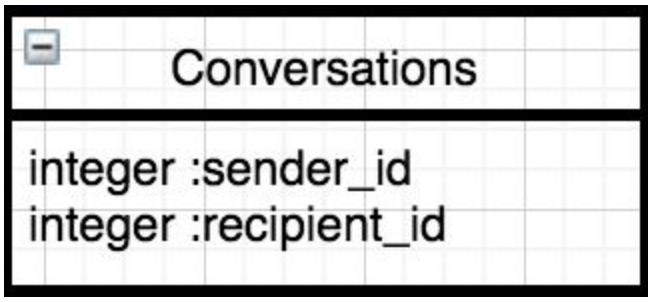


Figure 16. Conversations Model UML

3.6 Messages Model

The messages model contains a text body. It belongs to a certain conversation that happened between two different users, so it has two foreign keys as conversation and users.

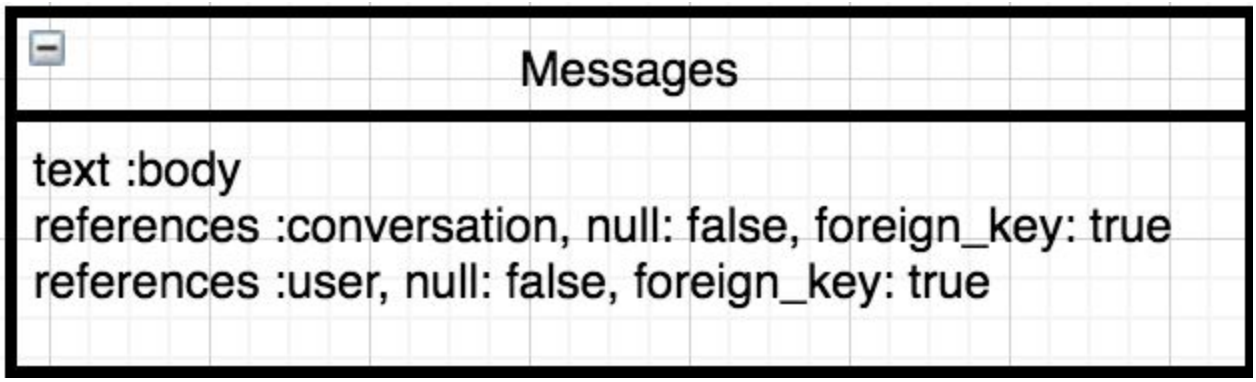


Figure 17. Messages Model UML

4. Load Testing

Tsung is a distributed load testing tool that allows us to pinpoint the bottleneck of our web application in terms of handling user requests. In order to mimic authentic user interaction with our application, we created several flows that represent how a typical user will use our app and navigate through different pages. During the initial phase of our load testing, we constrained the fake user to only have one designated flow of action. In the next phase, we start to mix up two different flows, each with their own probability of occurring. Finally, we mingle up all the flows we can think of and assigned each of them a probability of occurring to simulate the real user interactions with our app. Those actions contain login, reading existing posts, creating a new post, commenting on posts, sending friend requests, and starting a one to one conversation. We believe by generating an abundant number of requests consisting of those different flows of actions we could thoroughly and truthfully test the capability of our web app when it comes to handling a great number of users accessing our web at the same time.

In the next few sections, we are going to present the different flows with which we did load tests on.

4.1 Workflow-1

This workflow simulates a user browsing through the posts at the home page.

1. User visits login page
2. User visits posts index page
3. User waits 1 second (simulates looking around)
4. User logs out

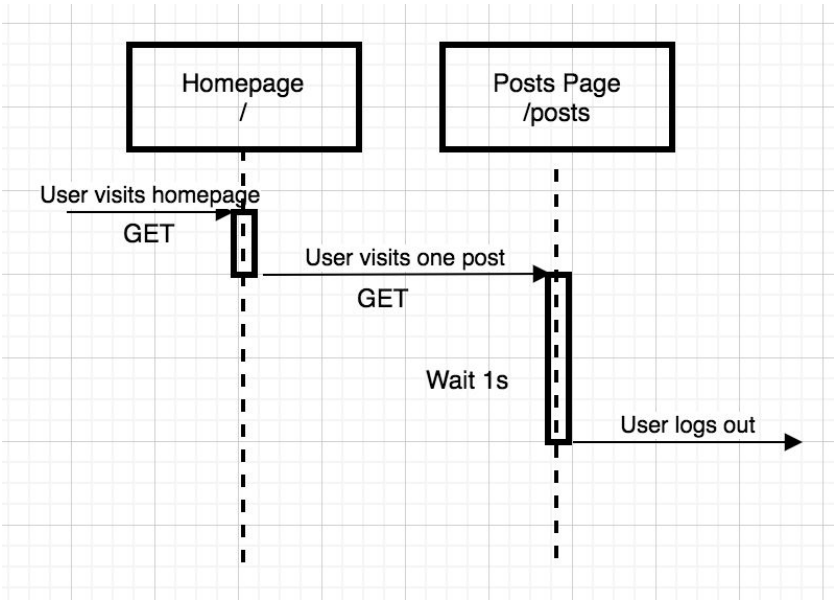


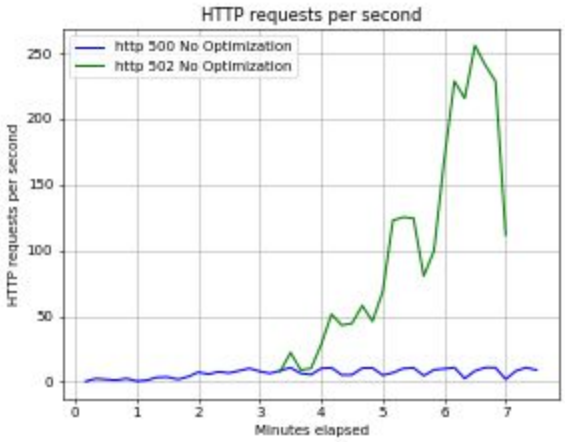
Figure 18. Workflow 1

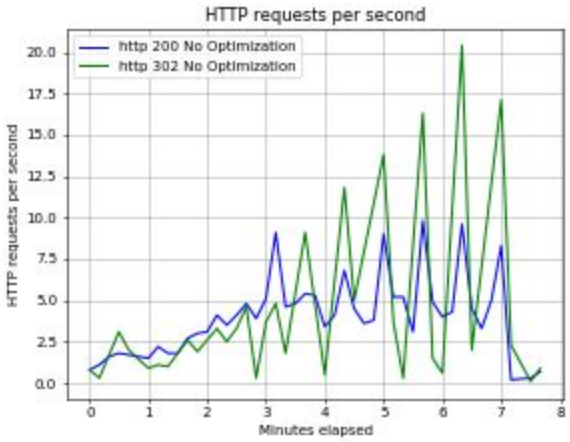
Test Scenario

- **Instance :**
  - Application server**:** c5.large
  - Database: db.m5.large
- **Tsung load:** Total 7 phases. Each phase lasts for 60 seconds.
  - Phase 1: 1 users/sec
  - Phase 2: 2 users/sec
  - Phase 3: 4 users/sec
  - Phase 4: 8 users/sec
  - Phase 5: 16 users/sec
  - Phase 6: 32 users/sec
  - Phase 7: 64 users/sec

Bottleneck Analysis

The initial load test suggests that there are major bottlenecks in the posts index page. From the below graph, we can observe that the 500 responses almost start immediately (i.e after 10 seconds) whereas 502 responses start somewhere in the 4th phase. There are very few 200 and 302 responses. Also, we can observe that the response time for each page is more than 5 seconds which indicates that there are major bottlenecks with the posts index page.





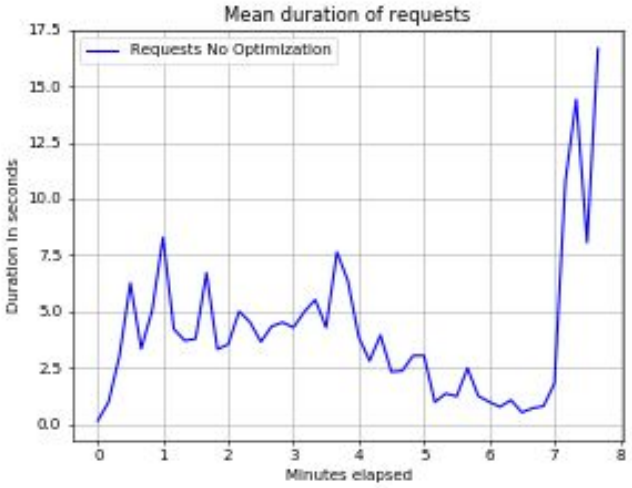
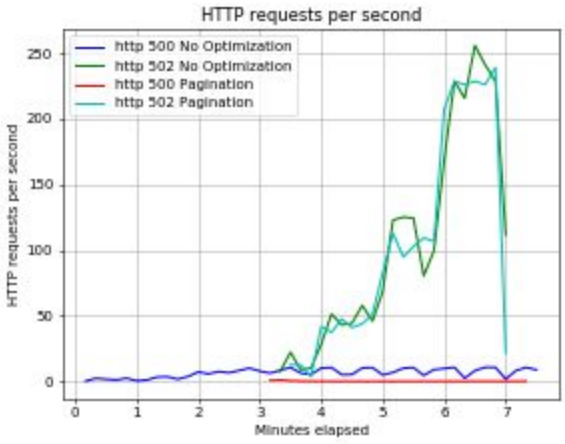


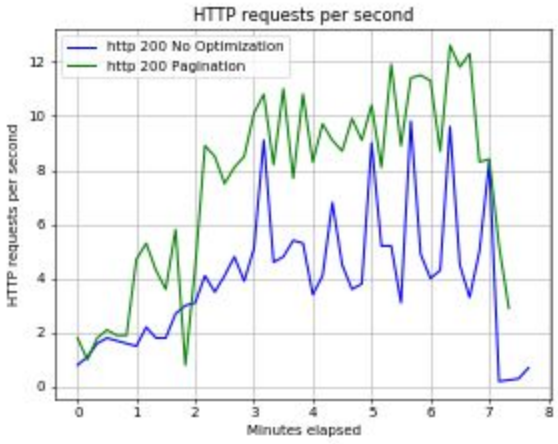
Figure 19.

Optimizations

Pagination

The posts index page displays all the posts in the database. Each post contains its associated user, post content, 0-4 random number of comments and their associated users. The database is seeded with 1500 posts and the average page response time is around 5 seconds. With pagination, we cut the number of posts per page to 50 and from the below graph we can see that the page load time is reduced to less than 100ms. Also, the 5xx responses start appearing somewhere in the phase 4.





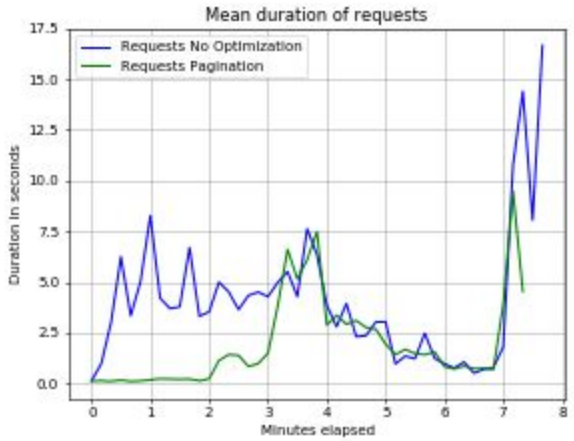


Figure 20.

N+1 Query Optimization

To load the posts index page, 1 query is executed to fetch all the posts, and then a query is executed for each of these posts to fetch its comments and for each comment a query to fetch its associated user. So, we have the N+1 query problem in loading the posts index page. To avoid this problem, we loaded the posts using the "includes" clause for comments and also the users associated with comments. From the below graphs, we can observe that there are no 500 responses after the query optimization and the 502 responses start from phase 5.

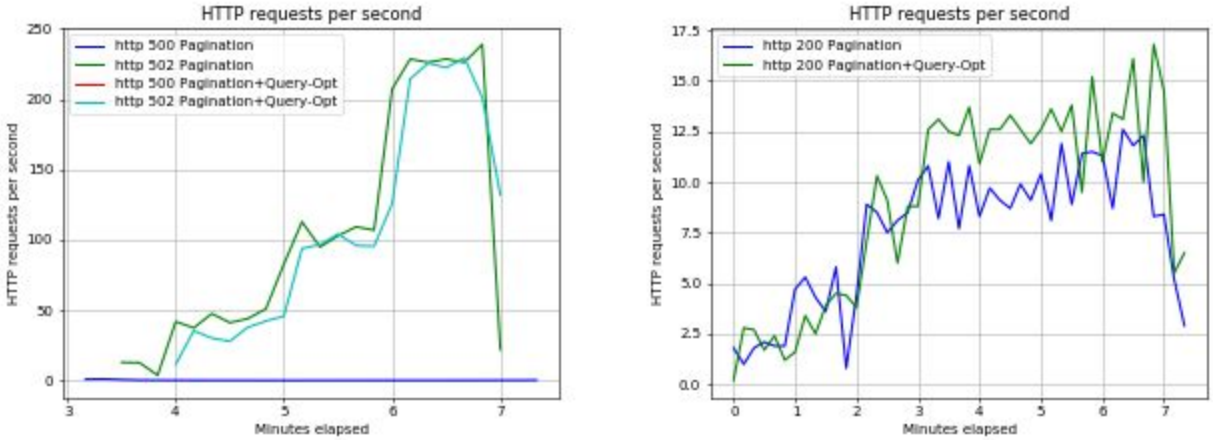


Figure 21.

Caching

After pagination and query optimization, we explored the caching for performance improvement. We implemented the Russian Doll caching and tried to cache the posts and their comments. Although the caching didn't help much in reducing the number of 5xx responses, there is an improvement in the page response time especially in phases 4 and 5.

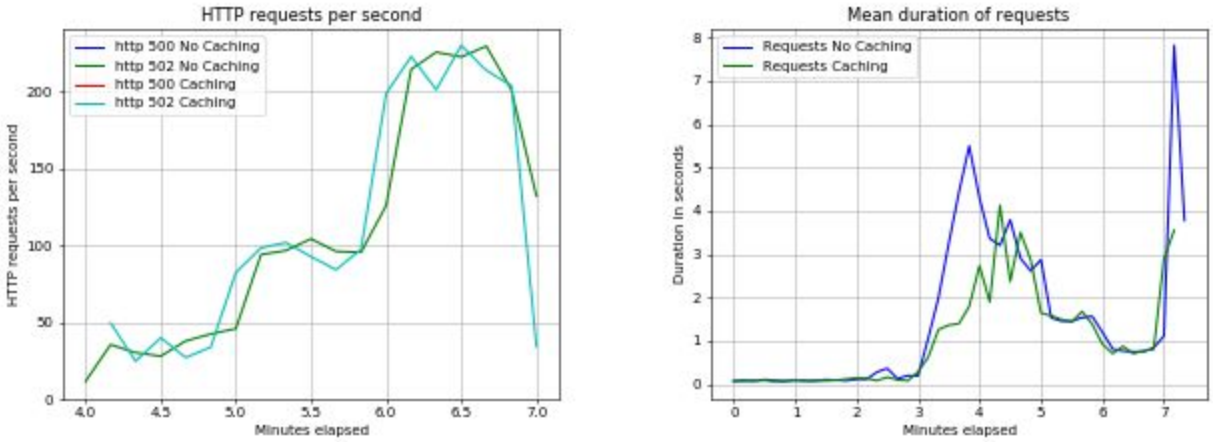


Figure 22.

Vertical Scaling

We performed vertical scaling to identify the bottlenecks in the application server instances. We run the tests using the optimized app on single instances of c5.large, c5.xlarge and m5.2xlarge. Note that the mean response time for c5.large and c5.xlarge are lower than m5.2xlarg in phase 6 due to the 5xx response codes.f

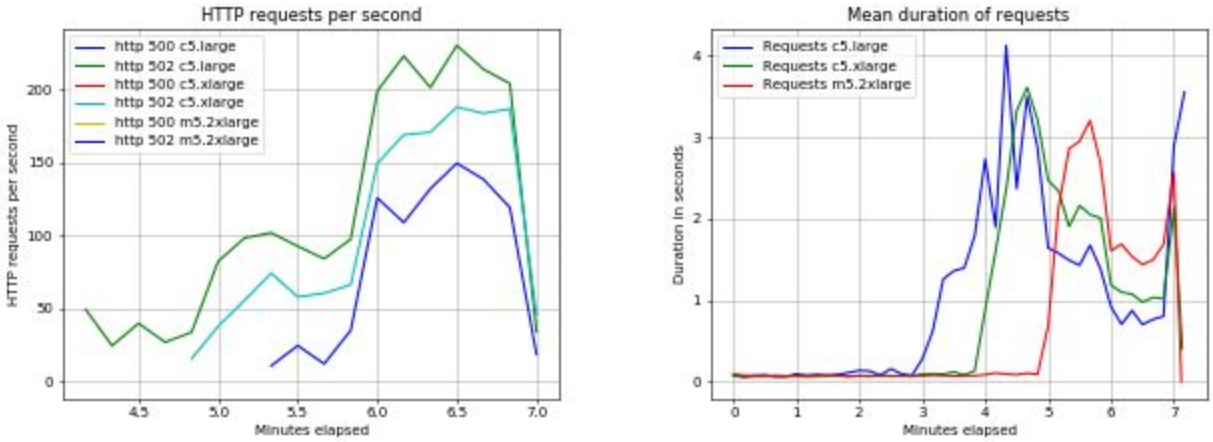


Figure 23.

4.2 Workflow-2

This workflow simulates users creating and editing posts and comments.

1. User visits login page
2. User logs in and visits his profile page
3. User creates a post
4. User creates a comment
5. User edits the previously created post
6. User deletes the post

7. User logs out

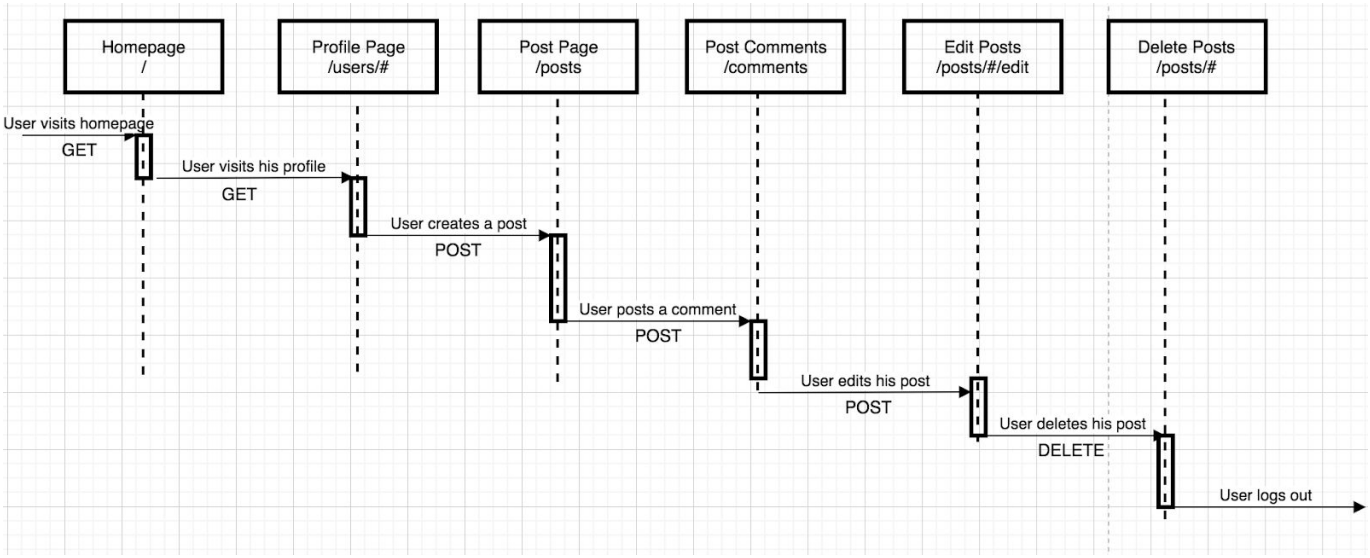


Figure 24. Workflow 2

Test Scenario

- **Instance :**
  - Application server**:** c5.large
  - Database: db.m5.large
- **Tsung load:** Total 7 phases. Each phase lasts for 60 seconds.
  - Phase 1: 2 users/sec
  - Phase 2: 4 users/sec
  - Phase 3: 8 users/sec
  - Phase 4: 16 users/sec
  - Phase 5: 32 users/sec
  - Phase 6: 64 users/sec
  - Phase 7: 128 users/sec

Bottleneck Analysis

The initial load test results show that the 5xx responses appear from phase 3. As there are no significant improvements that can be made to application logic for creating and editing posts and comments. We looked for bottlenecks in the profile page. Since the profile page displays all the posts made by the user, we were hoping there would be scope for improvement.

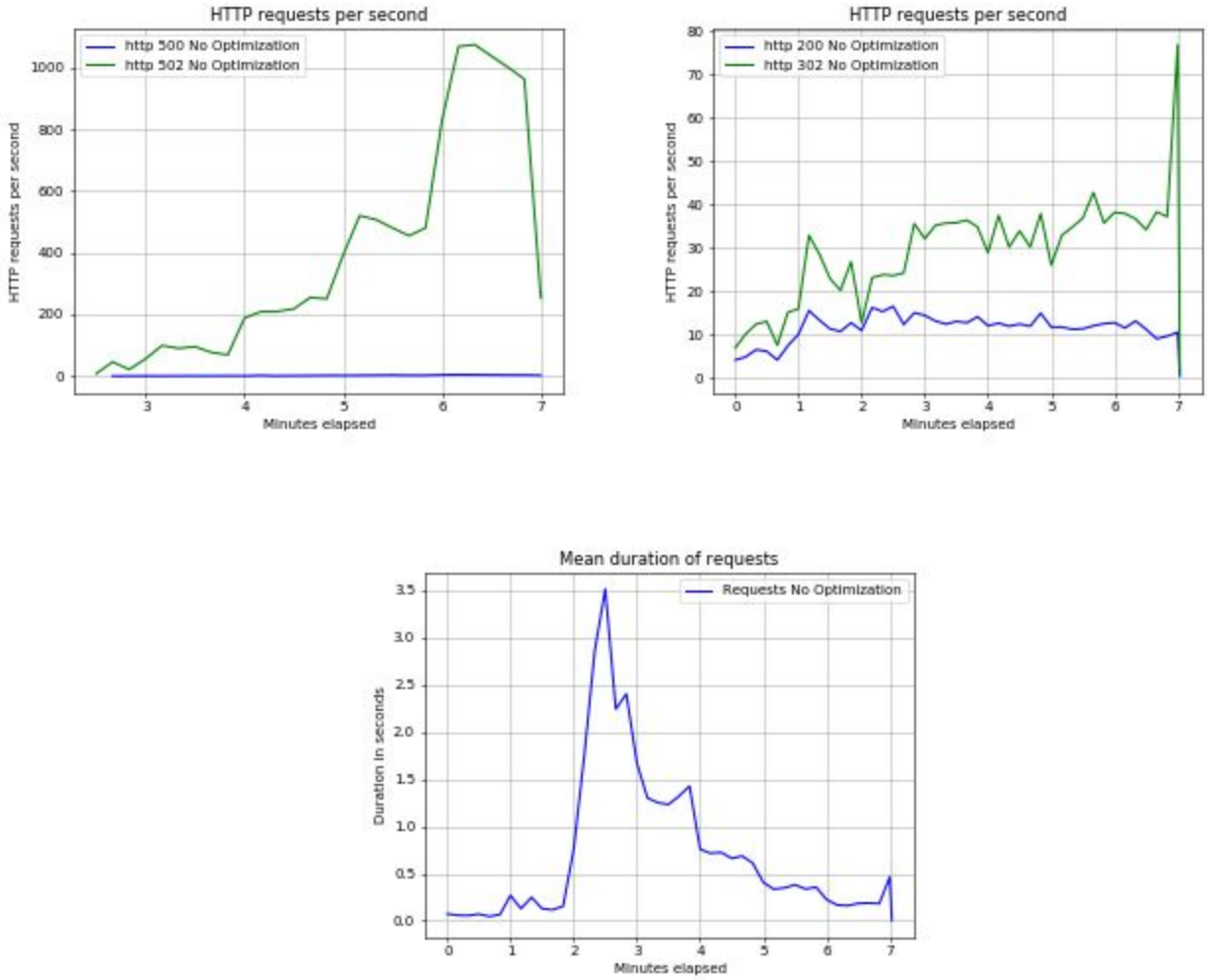


Figure 25.

Optimization

The profile page loads all the posts made by the user from the database. So, we applied the same pagination, N+1 query optimization and caching optimizations that we considered for posts index page. From the below graphs, we can observe that there is a slight improvement in the rate of 200 responses and the rate remains almost same from phase 3. This indicates that the application server is the bottleneck.

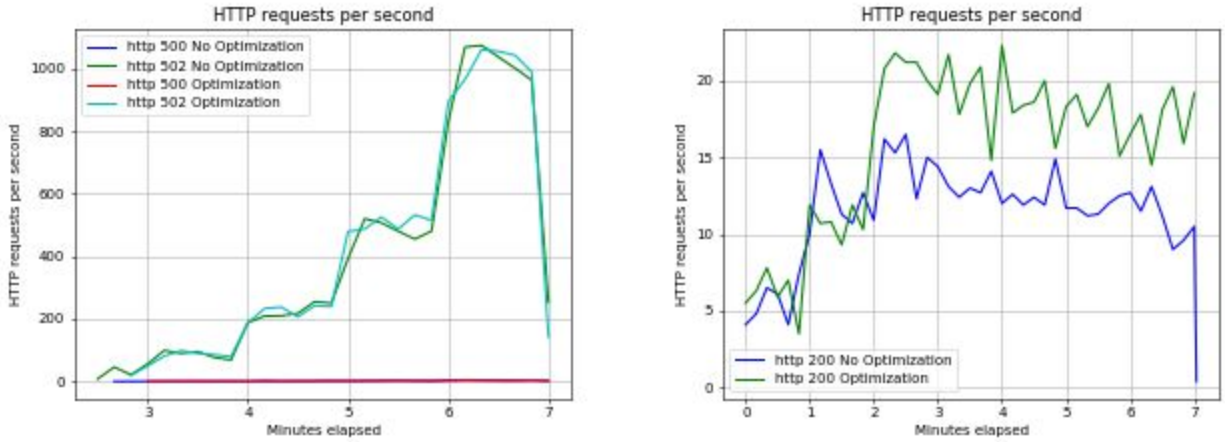


Figure 26.

Vertical Scaling

We performed vertical scaling to identify the bottlenecks in the application server. We run the tests using the optimized app on single instances of c5.large and m5.xlarge.

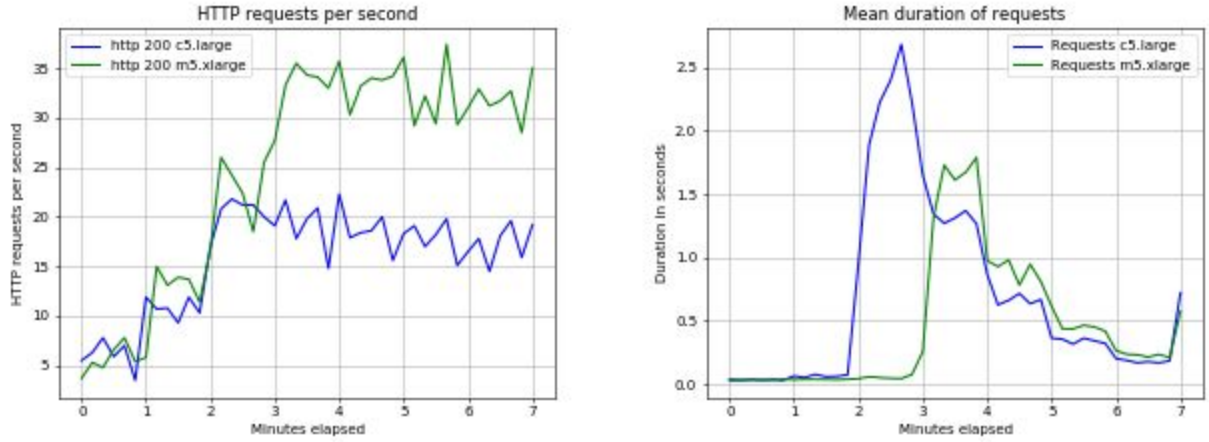


Figure 27.

Horizontal Scaling

We run the tests on 4 instances of m5.xlarge and see a significant improvement in the rate of 200 responses and average response time for a request.

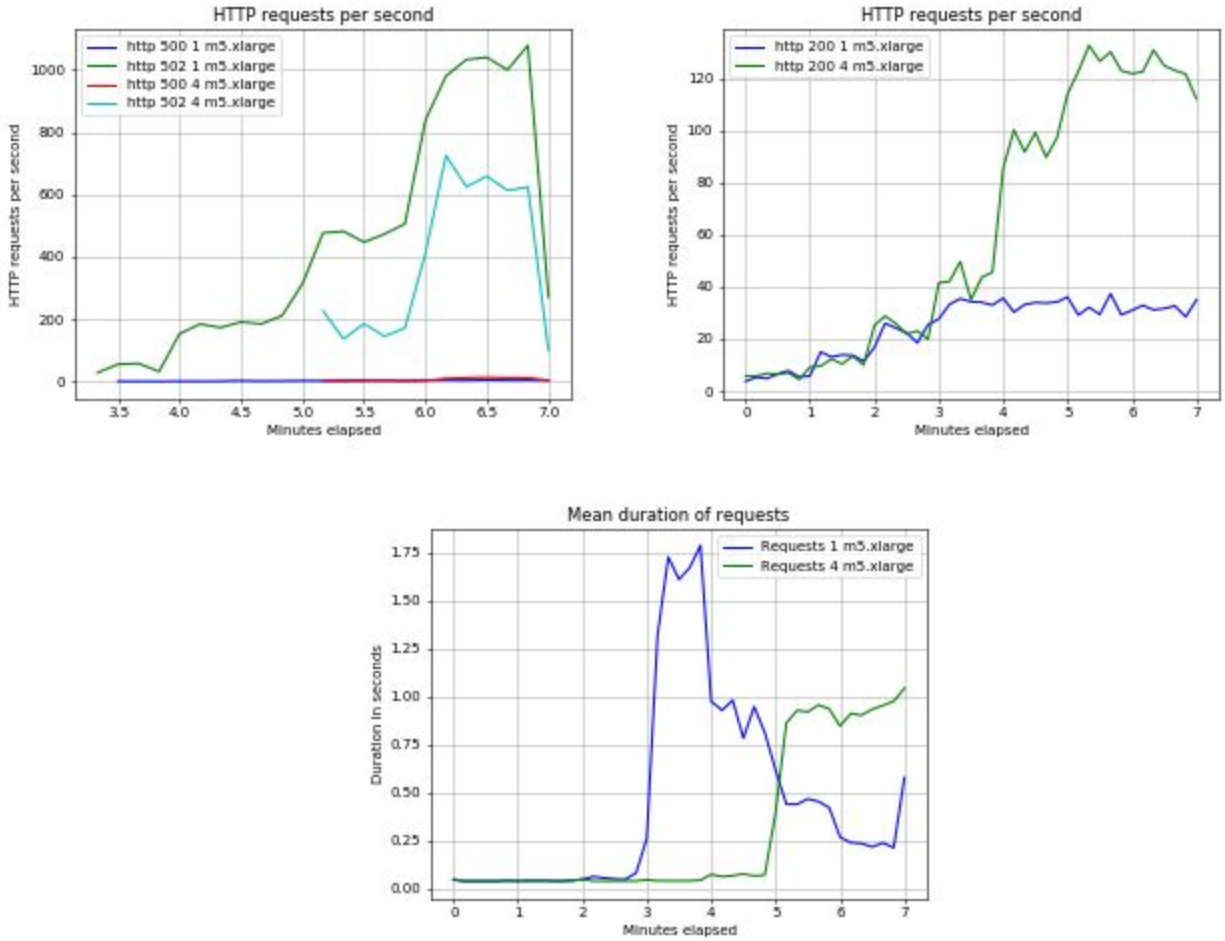


Figure 28.

4.3 Workflow-3

This workflow simulates users browsing through the users page and sending friend requests.

1. User visits login page
2. User logs in and visits the users page
3. User sends a friend request to a user
4. User views all the received friend requests
5. User logs out

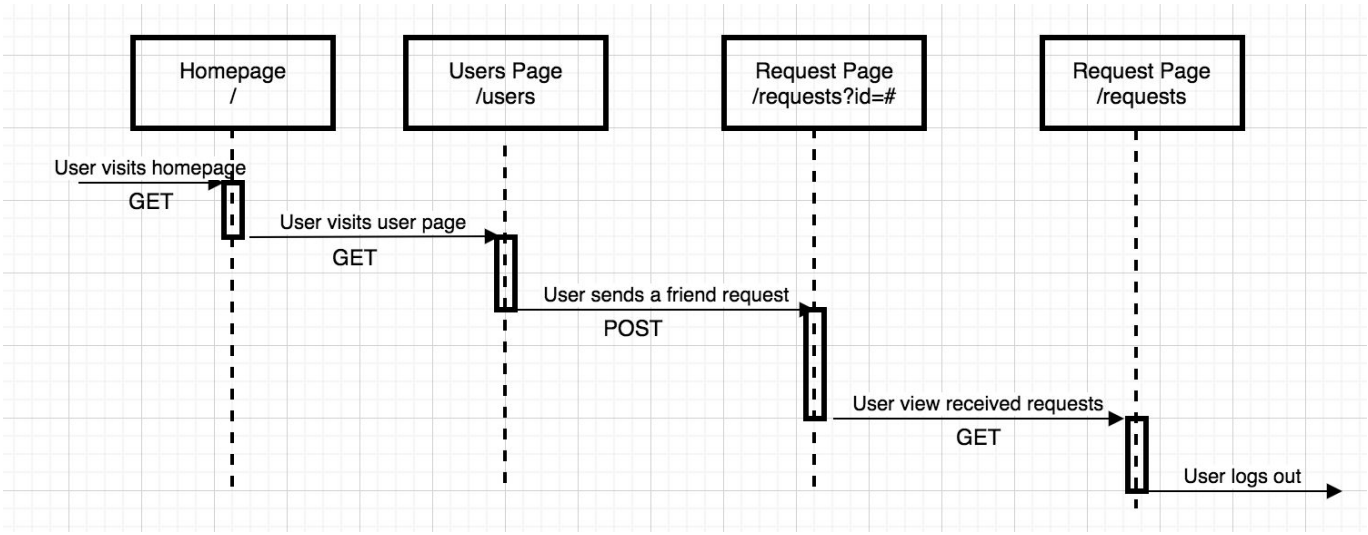


Figure 29. Workflow 3

Test Scenario

- **Instance :**
  - Application server**:** c5.large
  - Database: db.m5.large
- **Tsung load:** Total 7 phases. Each phase lasts for 60 seconds.
  - Phase 1: 2 users/sec
  - Phase 2: 4 users/sec
  - Phase 3: 8 users/sec
  - Phase 4: 16 users/sec
  - Phase 5: 32 users/sec
  - Phase 6: 64 users/sec
  - Phase 7: 128 users/sec

Bottleneck Analysis

The initial load test results suggests that there are major bottlenecks in loading the users index page. From the below graph, we can observe that the 500 responses almost start immediately (i.e after 10 seconds) whereas 502 responses start somewhere in the 3rd phase. Also, we can observe that the response time for each page is more than 4 seconds . Since there is not much load associated with the friend requests and also not much improvements can be made to the application logic for that part we suspect that the users index page for bottlenecks and look for possible optimizations.

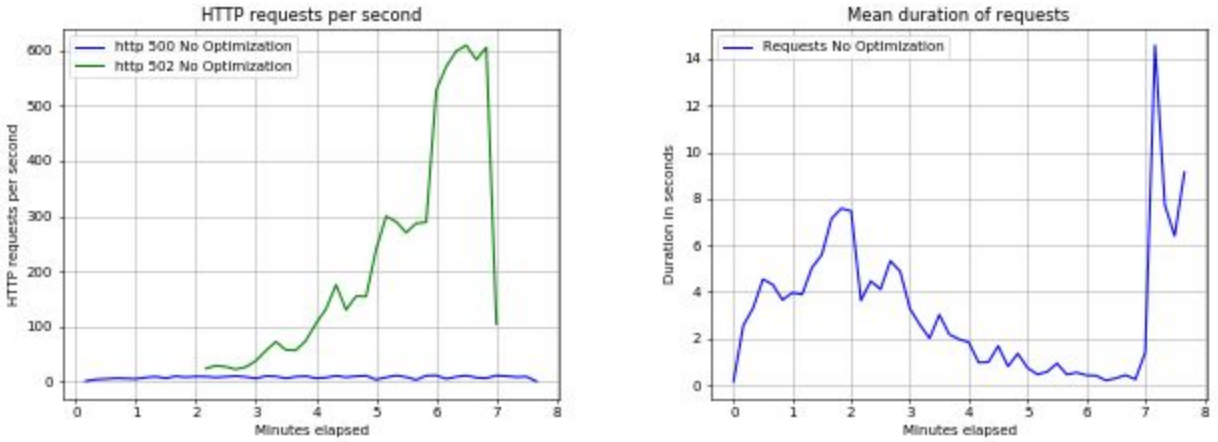


Figure 30.

Optimizations

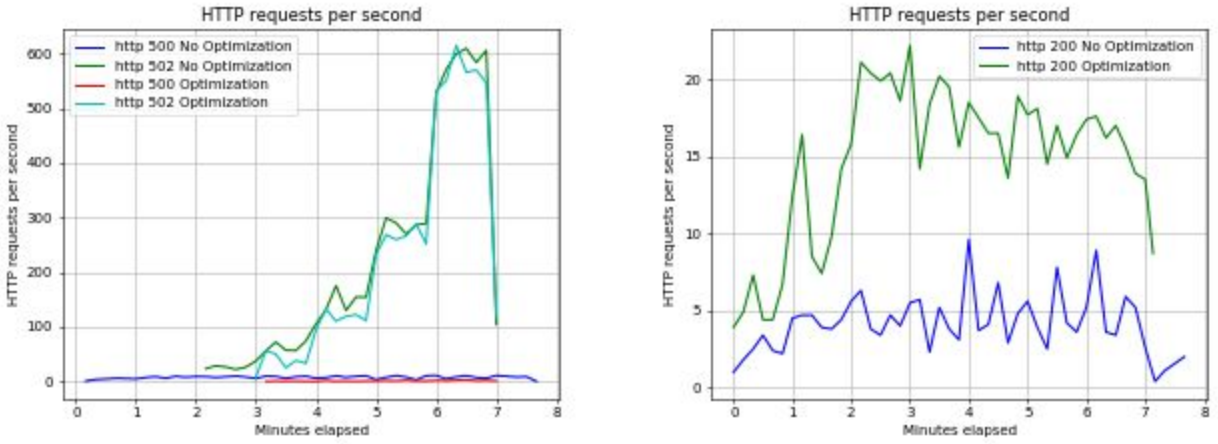
Pagination:

The users index page displays all the users both friends and non-friends in the database. The database is seeded with 1000 users and the average page response time is around 4-5 seconds. With pagination, we cut the number of users per page to 50 and from the below graph we can see that the page load time is reduced to less than 150ms. Also, the 5xx responses start appearing somewhere in the phase 4.

Caching

After pagination, we explored the caching for performance improvement. The users page is updated only when a new user signs up. We implemented the simple fragment caching and tried to cache each of the users individually.

With pagination and caching the average response time reduced from 5 seconds to less than 150ms and also the 5xx responses start appearing from phase 4.



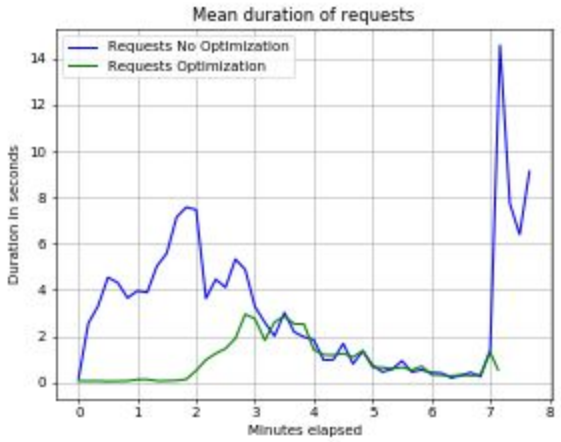


Figure 31.

Vertical Scaling

We performed vertical scaling to identify the bottlenecks in the application server instances. We run the tests using the optimized app on single instances of c5.large, m5.xlarge.

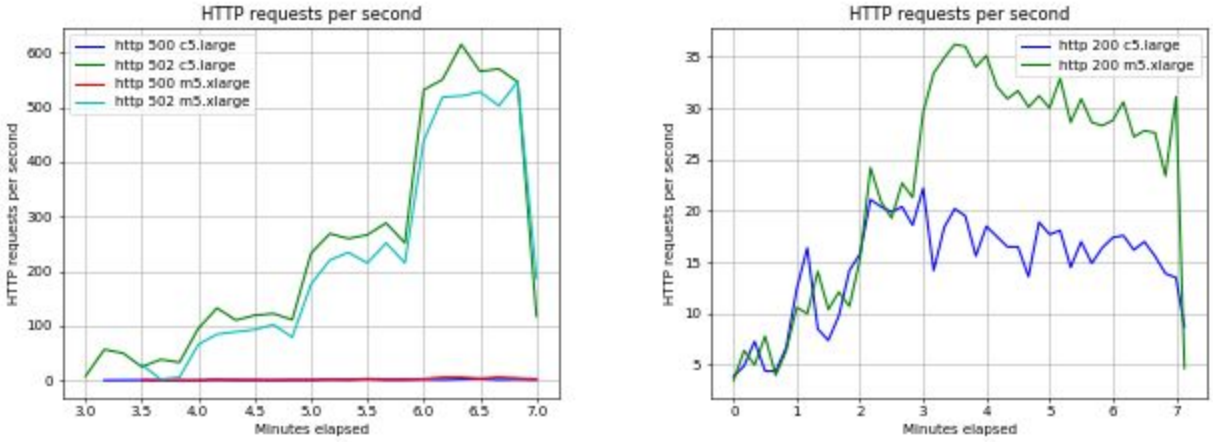


Figure 32.

Horizontal Scaling

We run the tests on 4 instances of m5.xlarge.

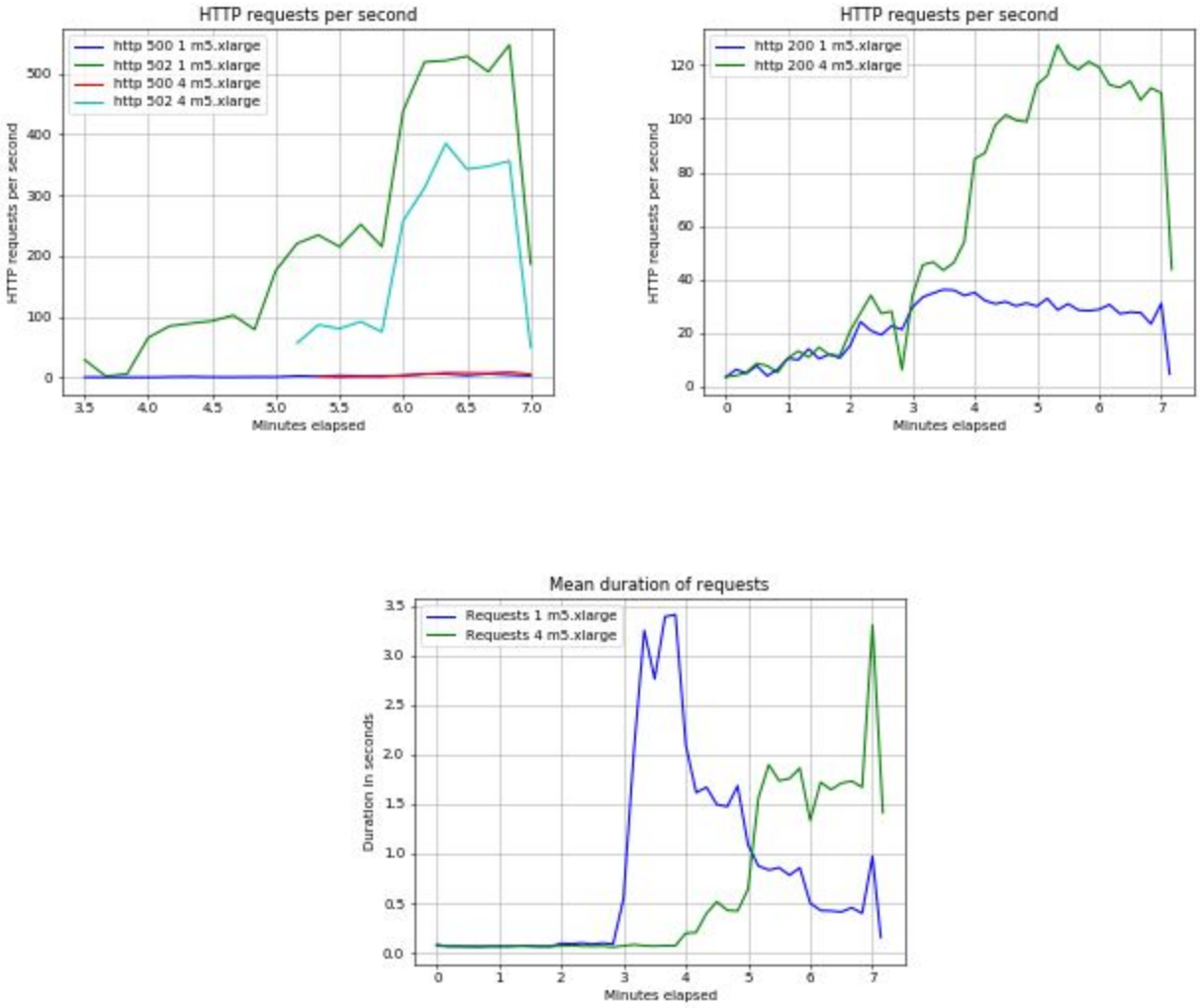


Figure 33.

4.4 Workflow-4

This workflow simulates users browsing through the profile page and sending messages to friends.

1. User visits login page
2. User logs in and visits the profile page
3. User selects a friend and opens a conversation which loads all the previous messages
4. User sends a message
5. User logs out

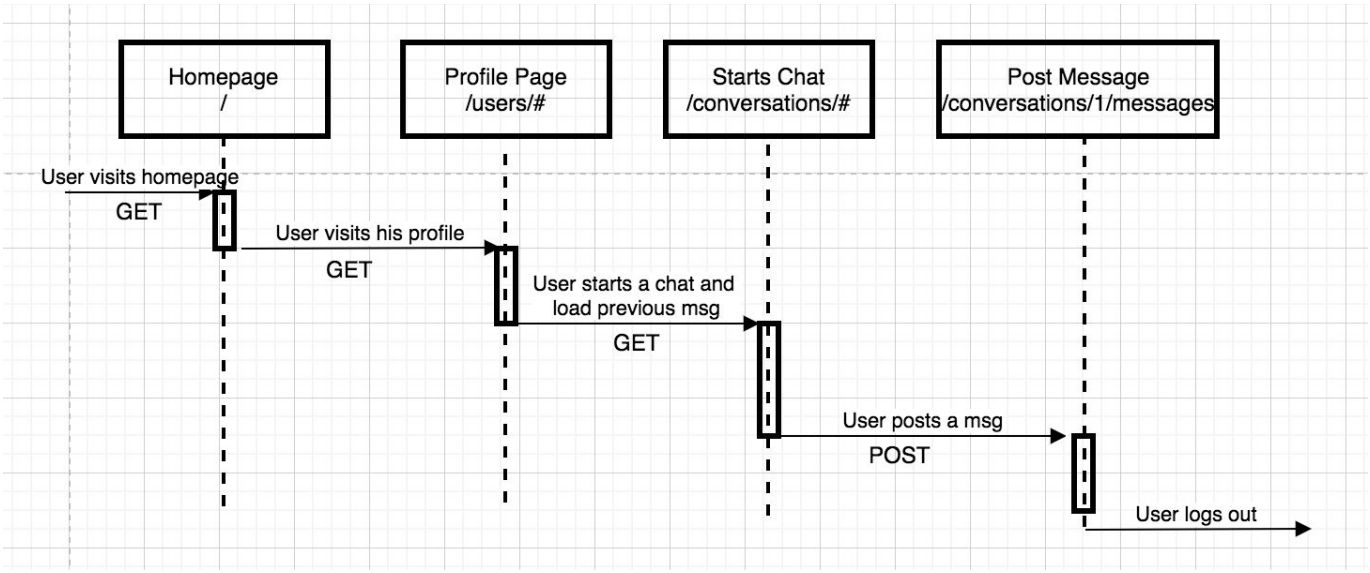


Figure 34. Workflow 4

Test Scenario

- **Instance :**
  - Application server**:** c5.large
  - Database: db.m5.large
- **Tsung load:** Total 7 phases. Each phase lasts for 60 seconds.
  - Phase 1: 2 users/sec
  - Phase 2: 4 users/sec
  - Phase 3: 8 users/sec
  - Phase 4: 16 users/sec
  - Phase 5: 32 users/sec
  - Phase 6: 64 users/sec
  - Phase 7: 128 users/sec

Bottleneck Analysis

Since we have already optimized for the bottlenecks in the profile page as part of workflow-2, we ran the tests to look for bottlenecks in the conversation page. From the below graphs for initial tests, we observe that the 5xx responses start appearing from phase 3 and the average response time spikes to 6 seconds. The conversation page loads all the messages between two users. The database is seeded with 1000 users and 200 messages between two users who are friends. So we decided to perform the pagination and caching on conversations page and see if they provide any improvement.

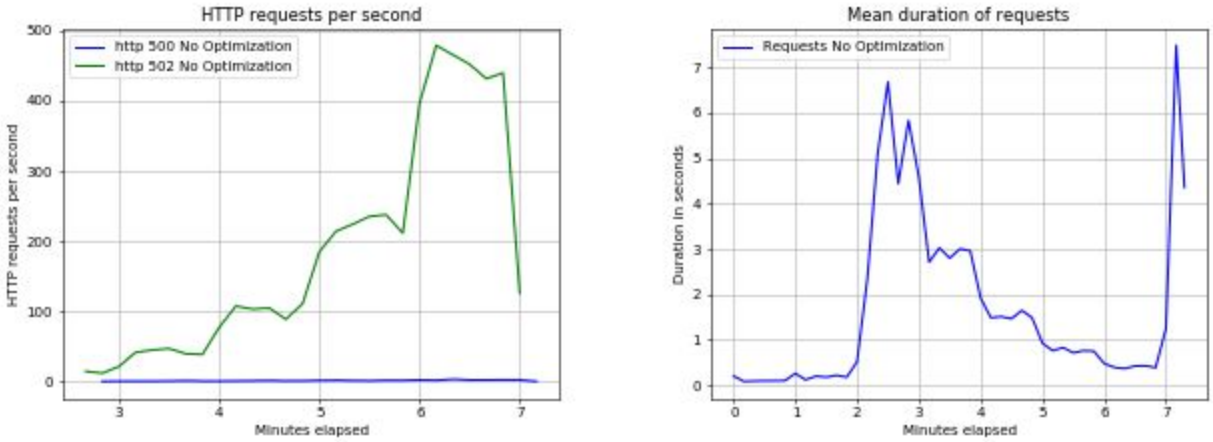


Figure 35.

Optimizations

With pagination, we cut the number of messages per page to 30 and also cached the messages. We can see from the below graphs that this optimization provided a little improvement in terms of 5xx responses and the average page response time went down to less than 2 seconds during the phase 3.

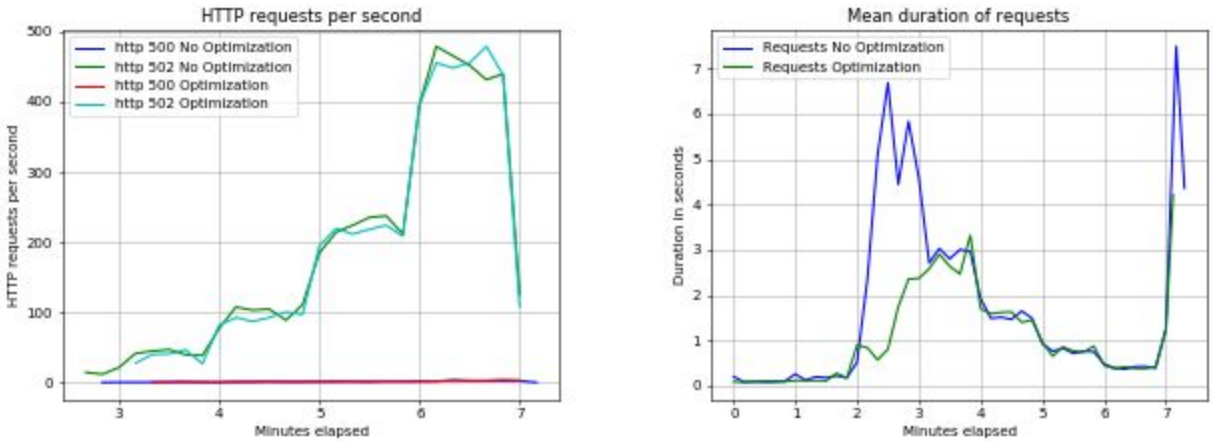


Figure 36.

Horizontal Scaling

We run the tests on 4 instances of c5.large to identify the bottleneck in application server. We observe that the 5xx responses start appearing in phase 6 i.e at 64 users/sec and also there is a 4x times improvement in the 200 responses and the average response time reduced to less than 80ms for phases 3 and 4.

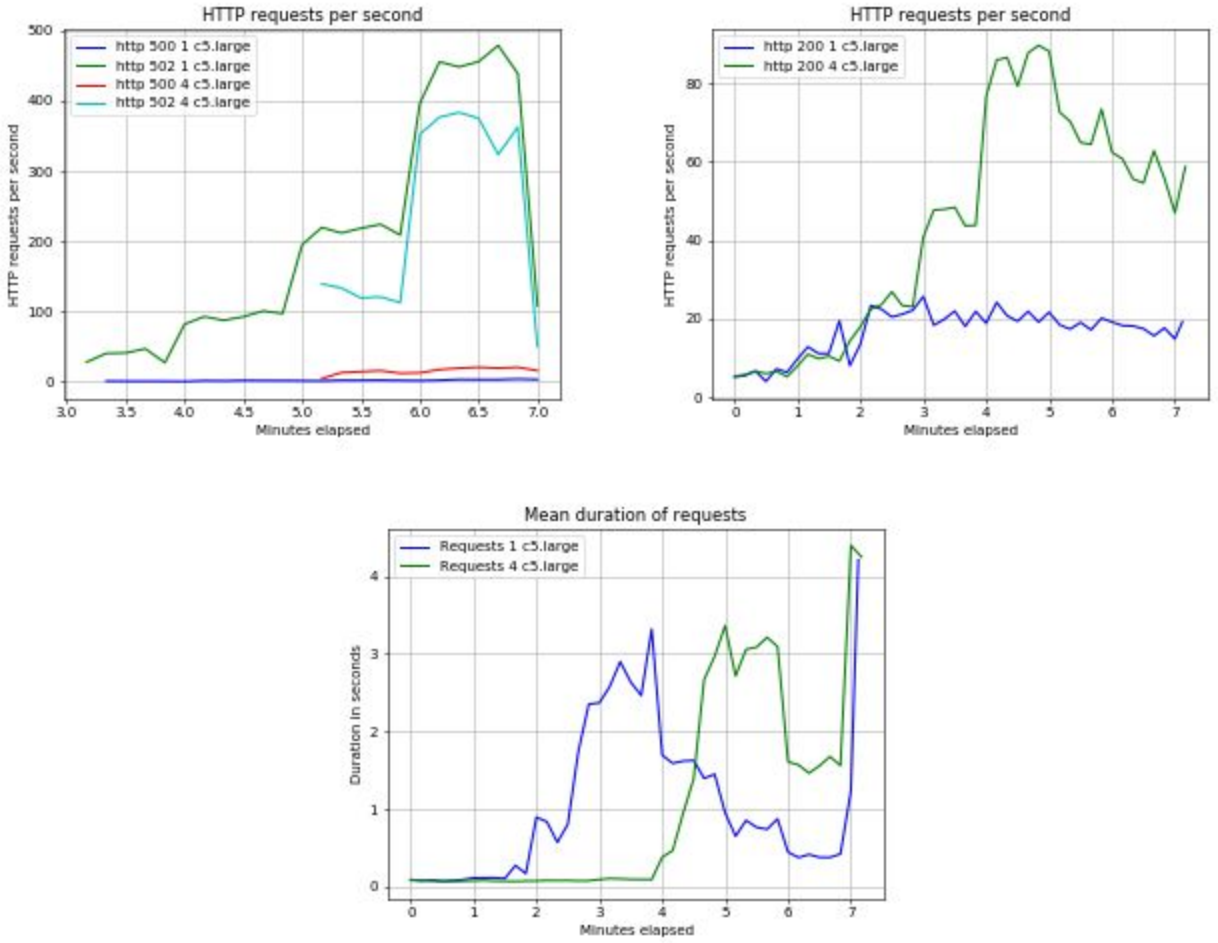


Figure 37.

4.5 Workflow-5

This workflow simulates users searching for particular users. We have basically designed this workflow to check for database bottlenecks with the search query.

1. User visits login page
2. User logs in and visits the users page
3. User searches for users with some query name
4. User logs out

Test Scenario

- **Instance :**
    - Application server**:** 8 m5.xlarge
    - Database: db.m5.large and db.m5.xlarge
- **Tsung load:** Total 7 phases. Each phase lasts for 60 seconds.
    - Phase 1: 2 users/sec
    - Phase 2: 4 users/sec

- Phase 3: 8 users/sec
- Phase 4: 16 users/sec
- Phase 5: 32 users/sec
- Phase 6: 64 users/sec
- Phase 7: 128 users/sec

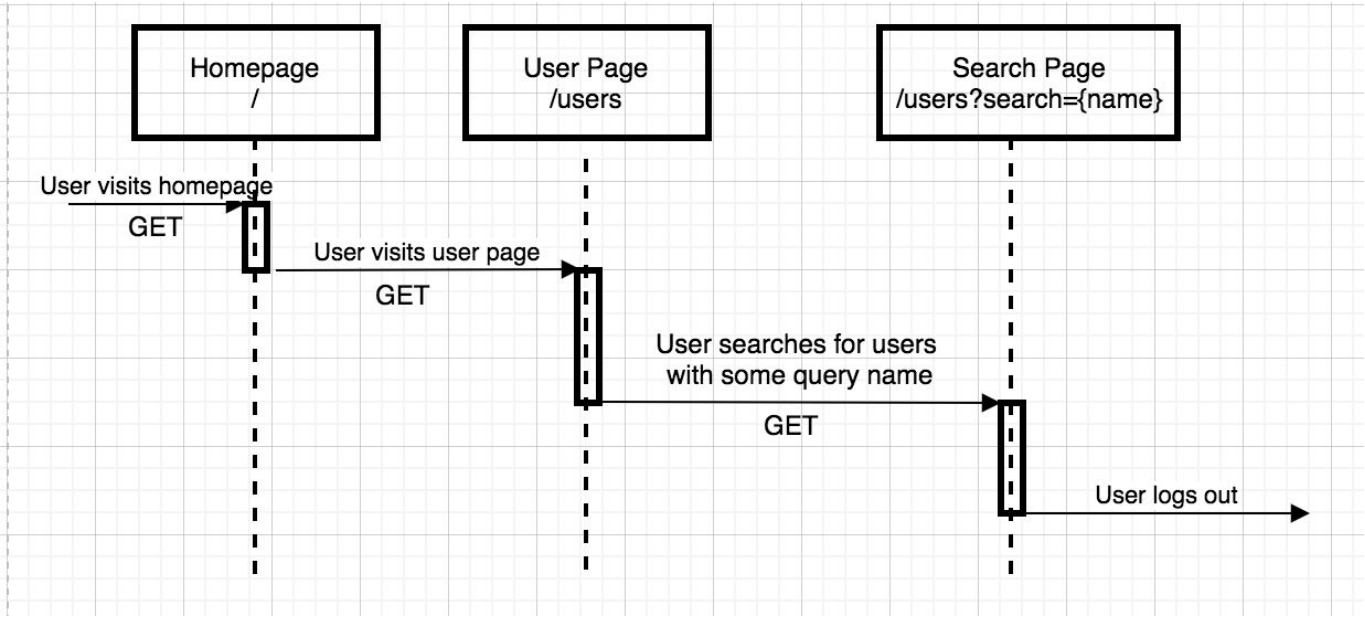


Figure 38. Workflow 5

For our tests, we took 8 instances of m5.xlarge so that to avoid the application server bottleneck. We were not getting any performance improvements by vertically scaling the database from m5.large to m5.xlarge so we are not putting any graphs for the tests and move on to cost analysis on all flows.

4.6 Maximum Load and Cost Analysis

In this section, we perform the cost analysis on all flows using different hardware configurations.

Test Scenario

We used the following tsung load for each of the hardware configurations.

- **Flow:** We have used the following percentages for each of the flows during Tsung tests.
  - Flow-1 : 30%
  - Flow-2 : 20%
  - Flow-3 : 15%
  - Flow-4 : 20%
  - Flow-5 : 15%
- **Tsung load:** Total 7 phases. Each phase lasts for 60 seconds.
  - Phase 1: 2 users/sec
  - Phase 2: 4 users/sec
  - Phase 3: 8 users/sec
  - Phase 4: 16 users/sec
  - Phase 5: 32 users/sec
  - Phase 6: 64 users/sec
  - Phase 7: 128 users/sec

We have done the cost analysis by finding the best hardware configuration for a user arrival rate. The below graph compares the cost of hardware configuration with the user arrival rate it supports (user arrival rate for the phase before the one where 5xx responses start occuring). So depending on the users arrival rate, we can switch to those hardware configurations which minimize the cost for supporting that user rate. Using this information, we can be better prepared to handle various levels of users' arrivals during different times of the day. For example, if the data show that at peak times there are 60 users/sec accessing the site then we can switch to (4 c5.2xlarge, db:m5.xlarge) which costs 1.7$/hour to support that user rate. Notice that, in our case we are not getting any improvements by scaling the database indicating that it is not a bottleneck.

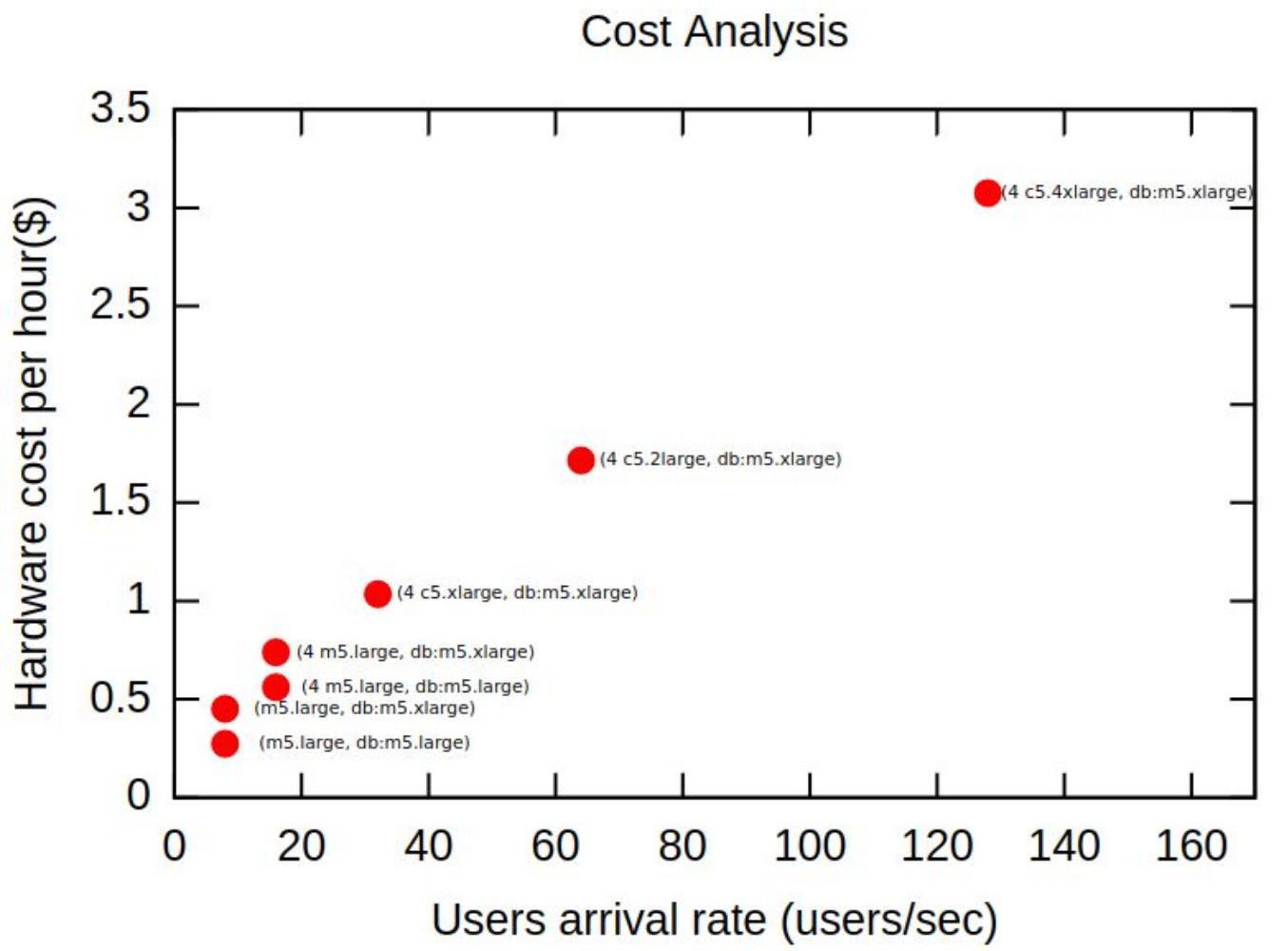


Figure 39. Cost analysis graph

We also wanted to find the maximum load that our application could handle. So we ran the tests using 16 instances of m5.4xlarge application servers and m5.xlarge database instance costing around 12.6$/hour. This configuration could handle user arrival rate of 1024 users/sec.

5. Future Work

Our future work can be summarized mainly in terms of feature additions and optimizations.
Firstly, there are several areas in our application where dynamic AJAX loading of components in a page can lead to significant improvements in page-loading times. For example, we can view comments in a post using a ‘load more comments’ button, which will load comments only when asked for.
Secondly, our conversation page currently needs to be refreshed to load new messages. In the future we would like to add streaming of messages which would enable automatic loading.
Thirdly, we would also like to enable users to search for content besides just users from our search bar. This would involve indexing posts by keywords so that users can search for content using keywords.
Finally, adding images as part of post content will likely create bottlenecks on both application server and database side. So we would also like to explore the various optimizations required to address such bottlenecks.

6. Conclusion

In this report, we gave an overview of our social network application (GauchoBook) implemented using Rails framework, discussed the results from several load tests performed using Tsung and explored some optimization techniques to address the scalability issue.

We have designed several workflows to simulate all possible user scenarios and performed load tests on each of these workflows individually to identify the bottlenecks associated with each user action/request.
Initially, most of the workflows couldn’t handle more than 1 user/sec. Then we explored some of the common optimization techniques like pagination, N+1 query problem and Caching. We also ran tests by horizontally and vertically scaling the instances. We also performed the cost analysis to identify the right hardware configuration to handle a specific user arrival rate. After performing all the optimizations, our application was able to handle 1024 users/sec with 16 m5.4xlarge application servers and m5.xlarge database instance.

Finally as part of future work, we describe some of the improvements and optimizations that we could have made to our applications.