

UNIVERSITY OF CALIFORNIA, SANTA BARBARA

CS291A Scalable Internet Services

Eventz

Team: 1337PACK

Aarti Jayesh Jivrajani

Subramaniyam Shankar

Yijun Xiao

Yuning Shen

December 7, 2019

CONTENTS

1	Introduction	3
2	Web Application Description	3
2.1	Data Models	4
2.2	App Features	4
3	Load Testing	6
3.1	Workflows	6
3.2	Optimizations	8
3.2.1	Index optimization	8
3.2.2	Pagination	9
3.2.3	Caching	10
4	Scaling and Cost Benefit Analysis	12
4.1	Scaling experiments	12
4.1.1	Scaling the server	12
4.1.2	Scaling the database	13
4.2	Cost benefit analysis	14
5	Future work	15
6	Summary	17

1 INTRODUCTION

Event sharing platforms provide an easy-to-access online venue for event organizers to create, share, and update event information to keep in touch with their guests, and for people to look up and enrolled in the events they are interested. In this project, we create and deploy an event sharing site that supports events browsing, event creation, guest RSVP, comments, invitation sending, etc; also analyze the service bottleneck and scalability by load testing. Our report organize as follows: we describe the data models and basic features of our web application in Section 2; we introduce the load testing settings, optimizations implemented, and test results in Section 3; we explore horizontal and vertical scaling and analyze cost benefit in Section 4.

2 WEB APPLICATION DESCRIPTION

The web application we created aims to provide a platform for end users to create and share events. A user can create new events, make comments, invite guests, and make RSVPs. Specifically, an unauthenticated user is allowed to create events, browse events, and make comments to an event; an authenticated user can additionally make RSVPs, invite registered users to the events, as well as editing profile and details of hosted events.

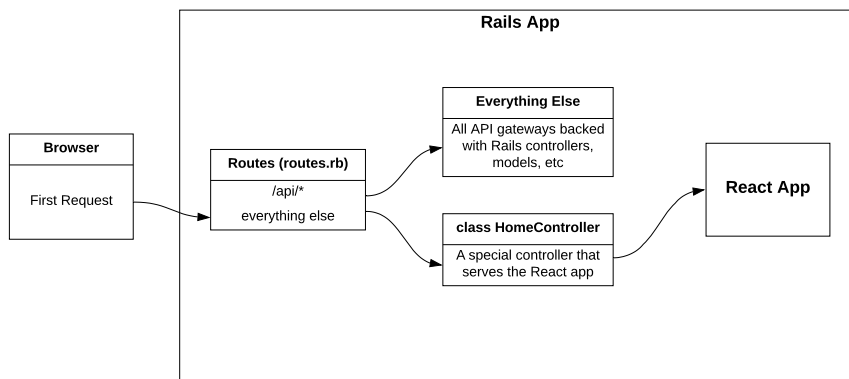


Figure 1: Illustration of the structure of the Rails and React applications.

Our back end is implemented with Ruby on Rails and the front end is implemented with JavaScript React. The server is API only except for the root URI which directs to the landing page. All subsequent queries to the server are responded with data in json format. Figure 1 illustrates the interaction between the server and the client. The application is deployed to Amazon web services (AWS) via Elastic Beanstalk.

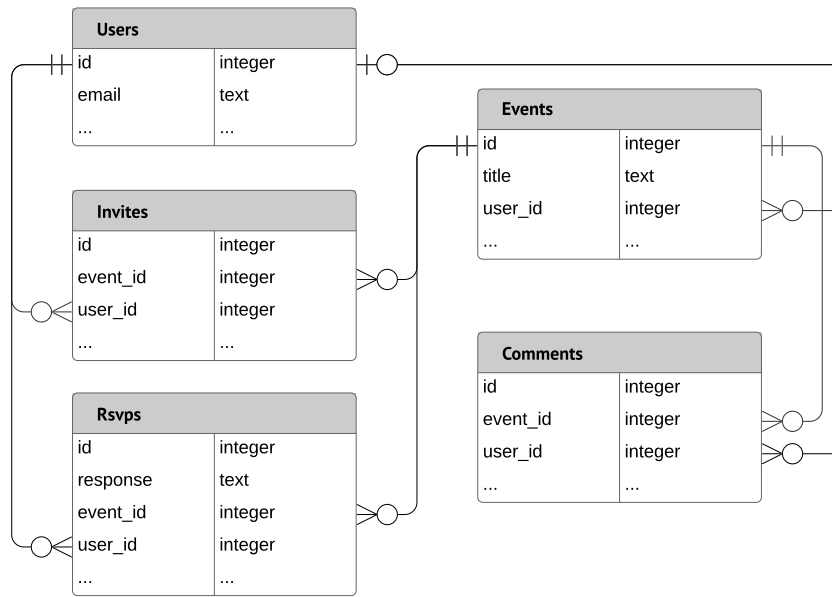


Figure 2: Entity relationship diagram for the event sharing web application.

2.1 DATA MODELS

There are five main models for our event sharing website: Event, RSVP, Comment, User, and Invite. The entity relationship diagram (ERD) is shown in Figure 2 with crow's foot notation. User and event have an optional one-to-many association to allow anonymous users to create events. It is the same case for user-comment association. On the other hand, both invite and RSVP have strict many-to-one association with user and event meaning each RSVP must be made by an authenticated user to a event.


2.2 APP FEATURES

SIGN UP Any unauthenticated user can choose to sign up for an account. The user email has to be valid and unique. The sign up page is shown in Figure 3(a).

We use bcrypt to encrypt user passwords and only store the password digest in the database. User authorization is done utilizing the pundit package.

LOG IN An unauthenticated can log in with a valid email password pair to be authenticated. The login page is shown in Figure 3(b).


BROWSE EVENTS Any user can browse the events that organized in a reverse chronological order (most recent events first) and check details of a chosen event, as shown in Figure 4.



Sign up

SIGN UP

[Already have an account? Sign in](#)



Log in

LOG IN

[Don't have an account? Sign up](#)

(a)

(b)

Figure 3: (a) Sign up and (b) Log in page.

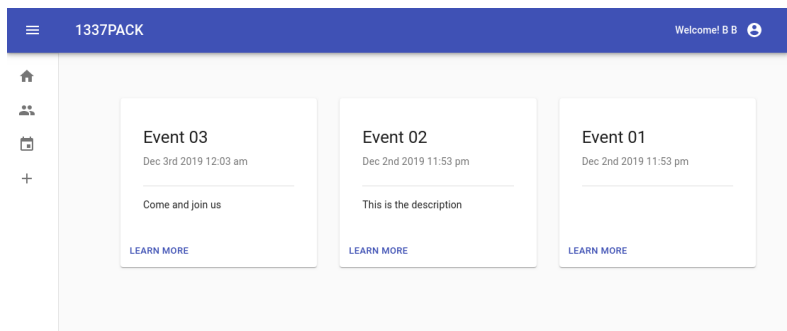


Figure 4: Browse events

CREATE EVENTS Any user including anonymous users can create new events with at least Event Title, Host’s name, and Date & Time, as shown in Figure 5.

COMMENT ON AN EVENT Any user can make comments on an event. For an anonymous user, name needs to be specified with a comment. If the user is logged in, name will be automatically populated with current user information, as shown in Figure 6.

RSVP An authenticated user can RSVP for an event on the event page. Current user’s full name is shown to help the host to recognize guest’s identity, as shown in Figure 7(a).

INVITE An authenticated user can send invites to other users on the event page if the user is authorized. All registered user will be populated in a searchable drop-down menu for the host to choose from. Multiple invites can be sent at once, as shown in Figure 7(b).

The screenshot shows a web form titled "New event" with a close button (X). The form contains the following fields: "Event Title *" (text input), "Hosted By *" (text input), "Event Date & Time *" (text input with the value "December 3rd 12:03 a.m."), "Location" (text input), "Street Address" (text input), and "Event Description" (text area). A blue "POST" button is located at the bottom of the form.

Figure 5: Create new events.

3 LOAD TESTING

In this section, we perform load testing with Tsung to locate the bottleneck of our application. We further introduce three different optimizations that improve our application's performances. Horizontal and vertical scaling are conducted to explore the ideal hardware combination for the application.

3.1 WORKFLOWS

To mimic key user behaviors in different scenarios as well as a combination of user pools in production, we define several simulated workflows and conduct load test with a mixture of workflows with equal probability. The aim of the test is to traverse through our application services to find application bottlenecks and come up with optimization strategies accordingly.

WORKFLOW 1 This workflow simulates a regular guest's behavior (Figure 8) that has following schedule: the user first visit the front page and log in, follow by browsing event list and checking details of an event with comment section loaded, then the user make a RSVP to the event and left. The use will pause (thinktime) accordingly at each step.

WORKFLOW 2 This workflow simulates a typical host's behavior (Figure 9) that has following schedule: the user visit the front page to log in, and then create a new event, follow by brows-

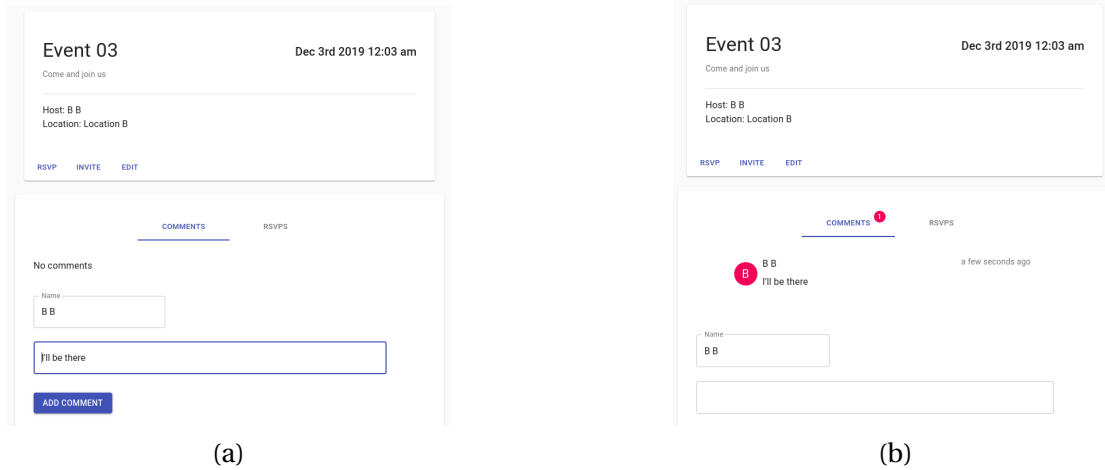


Figure 6: Post comment on an event.



Figure 7: (a) RSVP form and (b) Invite form.

ing event list and check details of an event, and finally leave a comment and exit. The use will pause (thinktime) accordingly at each step.

We do not include invite function in this test as it hit our API endpoints of `get_all_users` and `post_invite`, which share similar logic of endpoint `get_all_events` and `post_comment` that are tested in mentioned workflow. We argue that testing invite behavior may not give us extra insight on service bottleneck than workflow 1/2.

For both workflows, originally design six phases with progressive user arrival rate of 1, 4, 8, 16, 32, 64 users per second. Please note that in some of the tests we scale up the user arrival rates accordingly until the failure of the service is found.

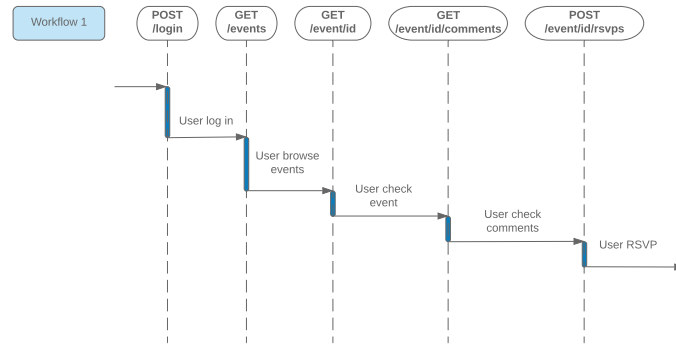


Figure 8: Workflow 1.

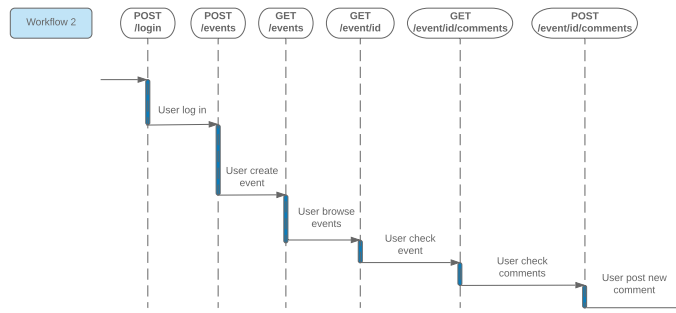


Figure 9: Workflow 2.

3.2 OPTIMIZATIONS

3.2.1 INDEX OPTIMIZATION

In our original implementation, whenever visiting a comment, invite, or RSVP API endpoint, we look up the associated `event_id` in our Event model, which introduces some overhead of database lookup. To improve the performance of comment, invite, and RSVP actions, we instead introduce the constraint of using the `event_id` from Event model as the foreign key for Comments, Invites, and RSVPs models. We expect that by assigning the logic to the database server, the performance of server instances will be improved and unnecessary communication between server and database will be avoided.

We first test on a smaller environment with server and database on `t3.micro` instance for first three phases. We observe decent improvement of index optimization over the no optimization one with delayed 5xx response and decreased response time for phase 2. We further test with six phases on a larger app instance with `c5.xlarge` for server `m5.large` for database. As shown in Figure 10, the improvements are not very significant in larger instances, except for a considerable drop in 4xx responses and a slight increase of 2xx responses. We suspect that in this case, it hit the bottleneck in other part of our application that leads us

to further investigation.

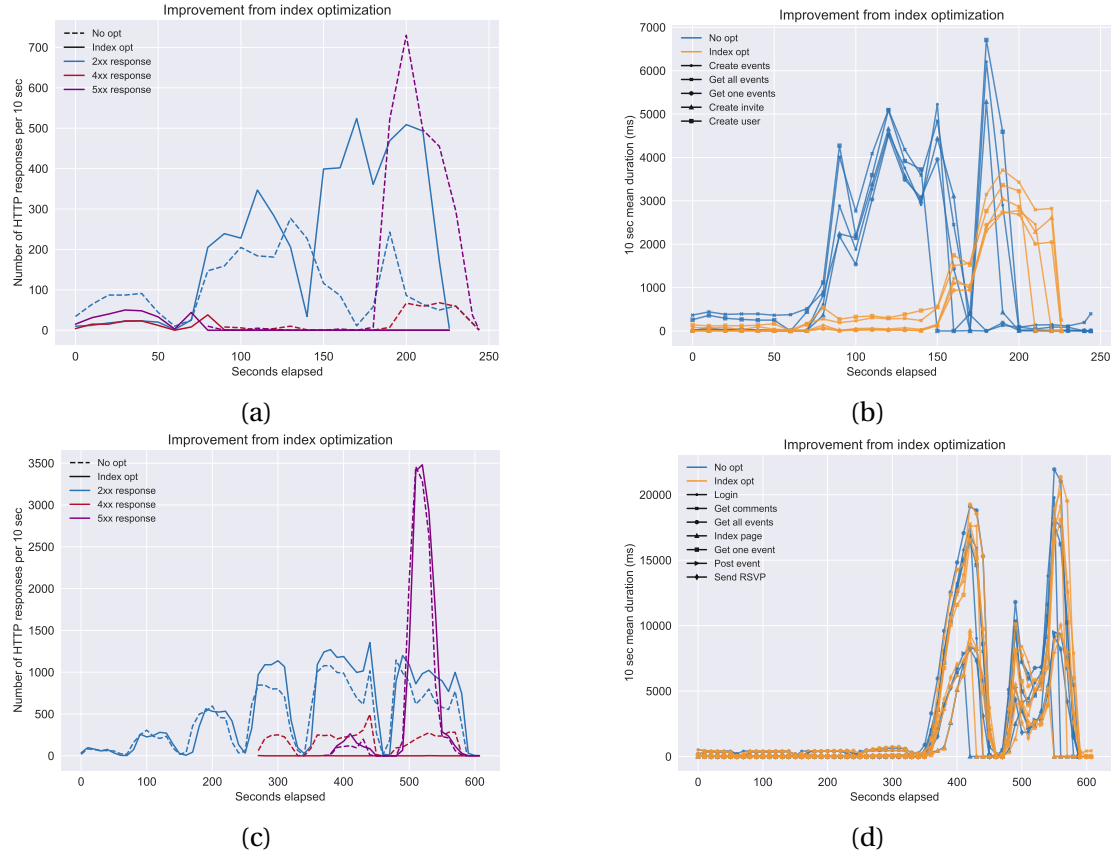


Figure 10: Tsung test results of HTTP responses (a) and transaction duration (b) for our application with or without index optimization on `t3.micro`. Additional test results on `m5.large` and `c5.large` are shown in (c) and (d). All statistics were counted or averaged within 10 seconds.

A further evidence is that, as the above comparison is done with only 1k seeded records in the database, we can not conclude that index optimization helps in all possible settings. As an example, we increase the number of seed record to 10k and conduct the same tests. The results are shown in Figure 11. We can observe that there is a significant deterioration in performance of the app with index optimization.

3.2.2 PAGINATION

In previous analyses, we observed a long transaction duration for Get all events, especially in 10k case (Figure 11(b)), as in our workflows, whenever a user needs to access a particular event, the user has to go to the all events page. When the database is seeded with a very high number of records, each time a huge number of records will be processed and returned. For

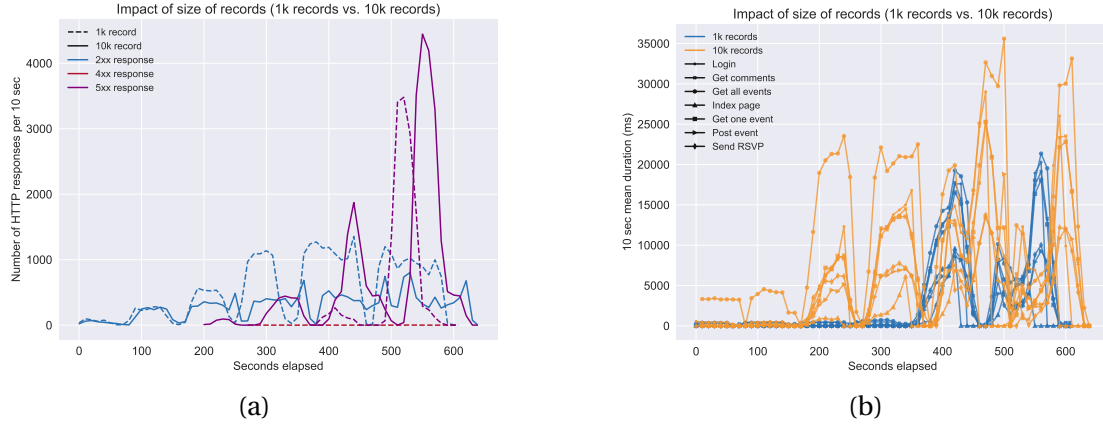


Figure 11: Tsung test results of HTTP response (a) and transaction duration (b) for our application with index optimization with 10k seeded users plus events (orange) compared to 1k seeded users and events (blue). All statistics were counted or averaged within 10 seconds.

majority of the users, they will not need more than 10-20 event records at one time, so processing and fetching all the other records (events) from the database introduced an additional overhead. By introducing pagination, only a set of 10 records is processed and sent back to the user, which we expect to greatly improve the overall performance of our application.

With only index optimization present in our logic, it is clearly evident that as the number of record increases, the performance reduces drastically. In contrast, the test results of application with pagination clearly shows that even with a high number of records seeded in the database (10k), the number of 5xx responses drastically decrease to a very low level and the transactions are much faster up to phase 5 (Figure 12).

It is still inconclusive, however, that without pagination, the bottleneck lies in fetching a huge number of records from the database or processing all those records in the app server.

3.2.3 CACHING

When database is the bottleneck, adding caching can be an improvement to reduce DB load. We cache two different types of models: active users and active events with an expiration time of an hour. We compare the effects of caching in addition to previous two optimizations. The results in Figure 13 indicate a marginal (if not none) improvement after adding caching to index and pagination optimized models. We argue that, given the Tsung test with 50% probability of cache miss, the benefits of using cache will decrease as the number of incoming users increase. From another aspect, this observation leads to a hypothesize that the database might be less likely to be the bottleneck of our current application, compared the app server.

It is worth mentioning that we implemented the low level caching using the caching mechanism provided by Rails, which uses file store for caching. This mechanism could be in-

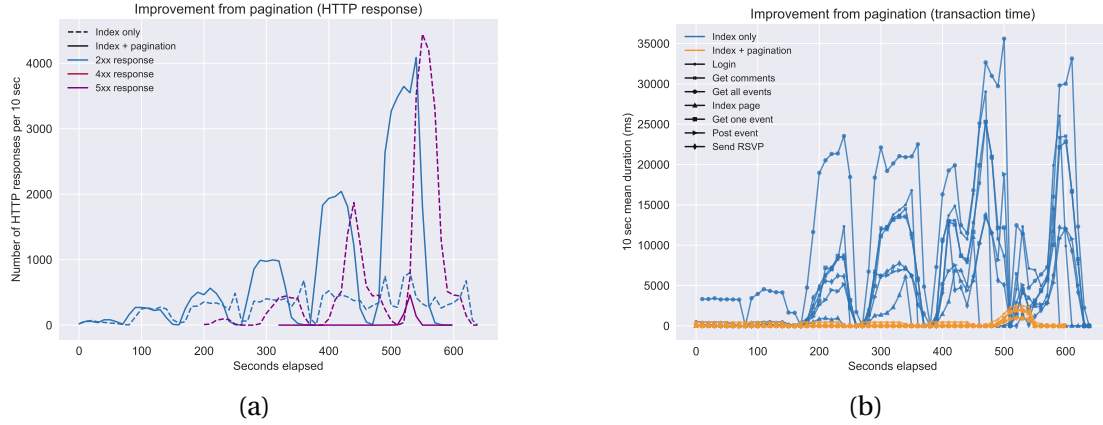


Figure 12: Tsung test results of HTTP responses (a) and key transaction duration (b) for our application with pagination + index optimization (orange) compared to index optimization only application (blue). All statistics were counted or averaged within 10 seconds.

efficient as it introduced the overhead of deserializing the cache store tree in case of cache retrievals. Since this mechanism uses local cache (instead of a decentralized one), horizontal scaling up our app servers would invariably lead to cache miss.

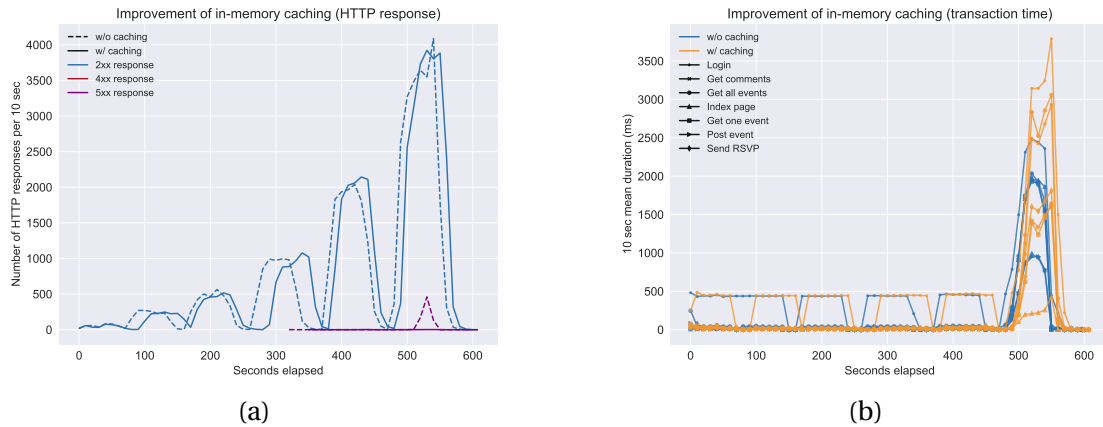


Figure 13: Tsung test results of HTTP response (a) and transaction duration (b) for our application with pagination + index + caching optimization (orange) compared to the one without caching (blue). All statistics were counted or averaged within 10 seconds.

4 SCALING AND COST BENEFIT ANALYSIS

4.1 SCALING EXPERIMENTS

We further scale testing our application with all three optimizations we implemented. There are three variables involved in the tests - instance size (both DB and EC2 instance), number of replicas of the app server instances and max number of incoming users per second. We are cost-aware while scaling out and scaling up. For the baseline, we use `c5.1large` and `m5.1large` instances for EC2 and DB respectively.

We first investigate whether the database or the server instance is the performance bottleneck to test our previous hypothesis. Figure 14 shows the Tsung test results with different server and database instance combinations. We can observe that upgrading the server instance from `c5.1large` to `c5.xlarge` improves the performance significantly, while upgrading the database instance from `m5.1large` to `m5.2xlarge` almost has no effect on the performance. This observation indicates that the server instance is our current bottleneck.

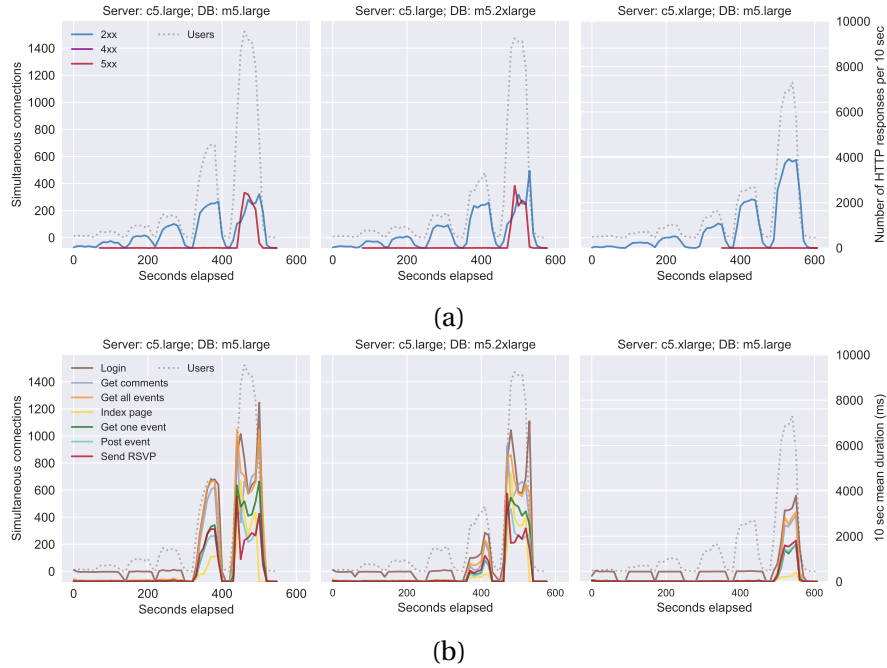


Figure 14: Tsung test results of HTTP response types (a) and request/transaction duration (b) for our application in different server / database instance combination settings. All statistics were counted or averaged within 10 seconds.

4.1.1 SCALING THE SERVER

Based on the above observation, we continue to design our scaling experiments with a fixed database instance (`m5.1large`). As price-wise two `c5.1large` are at the same rate as one `c5.xlarge`,

we want to compare performances of different scaling settings (vertical and horizontal scaling) at the same price rate.

instance type	4 instances	8 instances	16 instances
c5.large	max rate 64/s	max rate 128/s	max rate 256/s
c5.xlarge	max rate 128/s	max rate 256/s	-
c5.2xlarge	max rate 256/s	-	-

Table 1: Server instance scaling experiment settings.

For the base setting with four c5.large instances, we use the original Tsung test file with 6 phases in total with incoming users at the rate of 1, 4, 8, 16, 32, 64 users per second. As we scale up horizontally and vertically, we increase the max rate of incoming users. We perform this test up to 256 users/second. The scaling experiment settings are listed in Table 1.

HTTP responses from the tests are shown in Figure 15, with similar scale of horizontal or vertical scaling, (e.g. doubling the number of servers vs doubling the server size), the improvements of serving numbers of incoming users are similar that using 2x resource can support 64 users/sec and using 4x resources can support 128 users/sec. However, when we further look into response during for critical transactions, scaling server vertically has greater benefits on decreasing response time compared to scaling server horizontally. Another observation is that during the test, the CPU usage of database hits 100 % for 4x resource experiments, indicating we are hitting the bottleneck of database.

4.1.2 SCALING THE DATABASE

Continuing from previous section, it is evident that we hit the bottleneck of database. To verify the observation, we horizontally scaled our application server to 128 instances of c5.xlarge to ensure that all the 5xx that we might receive is due to the bottleneck in the database. We stress test our app with one phase of user arrival rate of 256 users/sec on three different database instances as Figure 17. We see that the bottleneck from database is eliminated when we use m5.4xlarge with very few 5xx responses and low CPU consumption as showed in Figure17(c).

As mentioned in Section 3.2.3, implementing caching would help overcome a database bottleneck. These tests are performed with low level caching from Rails and the bottleneck is still hit. As previously argued, the local caching does not help much when 128 application server instances are used to perform these tests, due to the high number of cache misses. Therefore, we hypothesized that it could benefit from a decentralized caching. To test this theory we implement a caching using ElastiCache for Redis. The Redis instance was chosen to be a very big instance r5.24xlarge, to remove any cache based bottlenecks. Using the same tests condition, it was observed that with a db instance of m5.2xlarge, the CPU utilisation decreased to around 73 % as Figure 17(d)

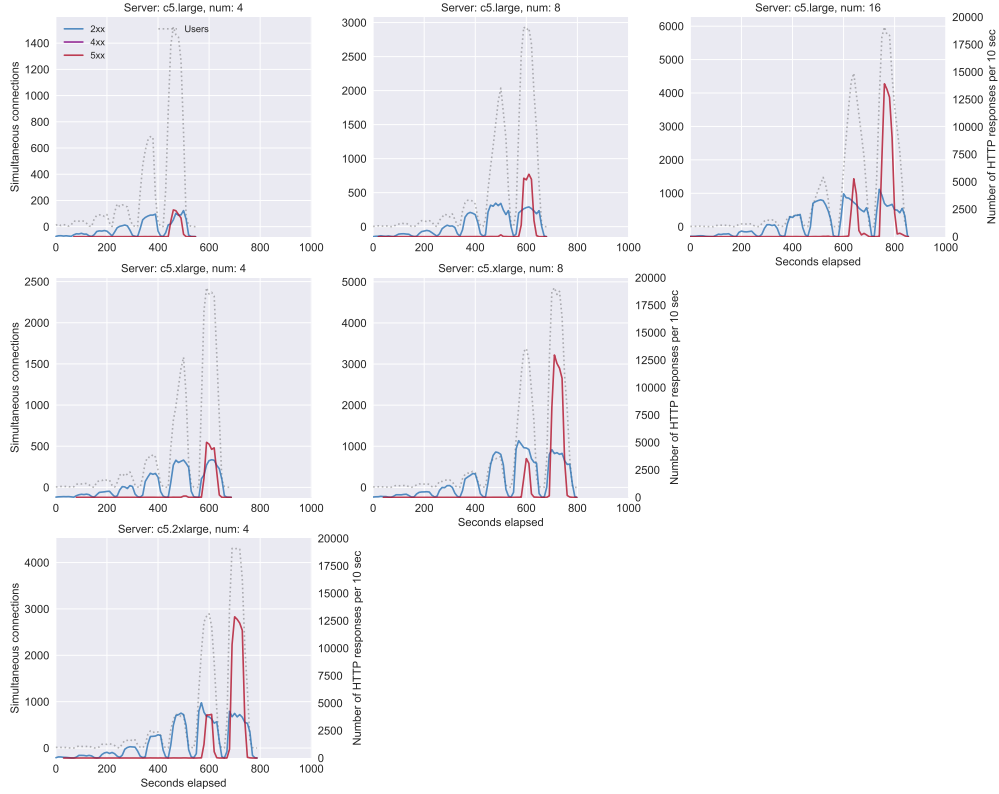


Figure 15: Server vertical and horizontal scaling results (HTTP responses) with database size fixed at m5.large

4.2 COST BENEFIT ANALYSIS

From the aforementioned results it is evident that there are two competing factors in our application, response times and availability. Keeping the price constant it is possible to have a lower number of instances with higher resources as compared to a deployment with higher number of instances with lower resources.

From the results as shown in Figure 15, it is seen that having better resources per instance gives better response times as compared to the other deployment which is able to support more number of users (i.e delay the onset of 5xx errors). If response times are more important, then fewer instances of higher "quality" should be preferred over higher "quantity" of lower quality instances. The latter is preferred if higher availability is the priority.

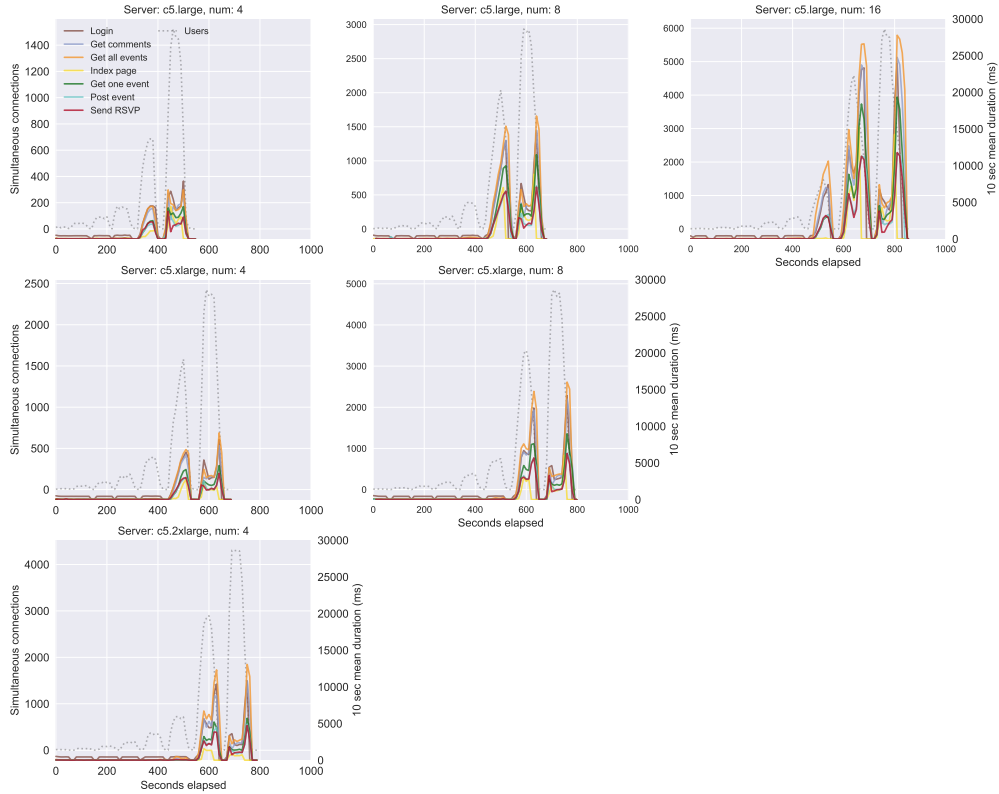


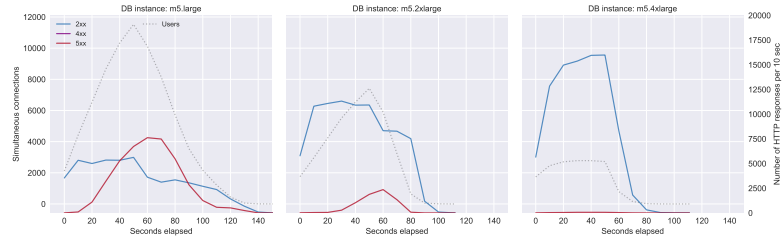
Figure 16: Server vertical and horizontal scaling results (transaction duration) with database size fixed at m5.large

5 FUTURE WORK

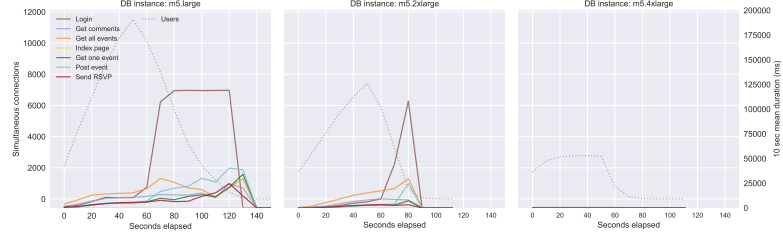
From the results and implementation of the application it is evident that the application can be improved in mainly two ways - Making the application more feature-rich and having more optimizations in place in order to scale the application further.

One good-to-have feature in our application would be a search functionality. Especially in cases when a user creates a lot of events and/or receives a lot of invites. This feature would go a long way in improving the user experience. Another feature we would like to add is a ranking chart where we list events based on popularity (which is decided by number of hits, RSVPs and an overall visitor count).

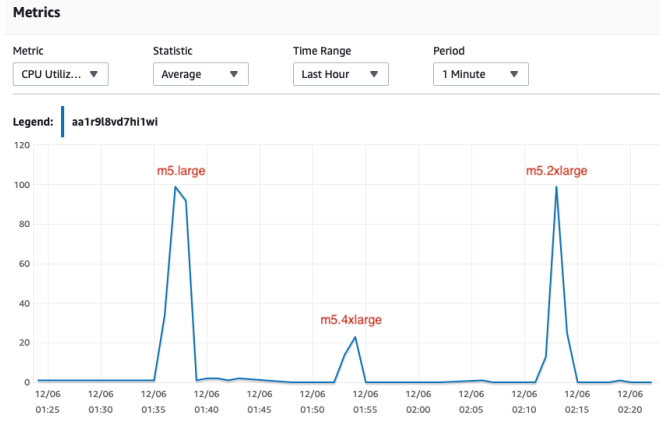
It was evident from our results that while vertically scaling the DB did help scale better, horizontally scaling the DB instances would help us scale well as well as provide a better reliability and fault-tolerance. Caching popular events(based on ranking as described above.



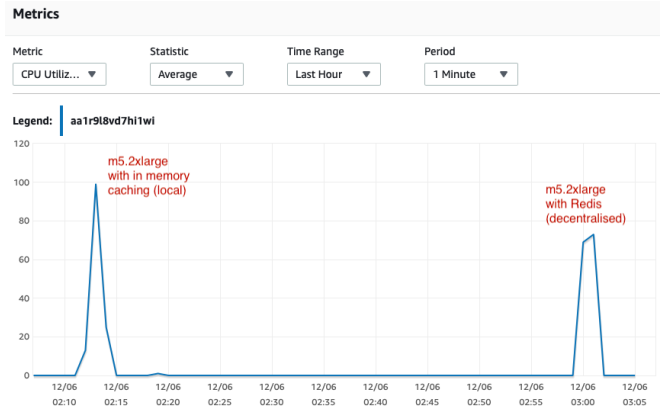
(a)



(b)



(c)



(d)

Figure 17: Results of vertical scale of DB instances on HTTP responses (a), transaction duration (b), and CPU usage (c). (d) shows the benefit of using a decentralised cache (Redis) over a localised cache (in memory cache)

6 SUMMARY

There is an ever increasing emphasis on reliability and fault-tolerance. With engineers promising very high SLAs to the customers, we know the importance of developing a fault-tolerant and scalable system.

At first, we had the basic skeleton of the application in place, which allowed us to design our Tsung workflows. Once we had our base application deployed, we started the load testing. At lower user-arrival rates, we noticed DB latency, which led us to introduce index optimization. On load testing this further, we introduced pagination and caching in order to improve the DB read thorough-put. While these optimizations helped us eliminate the 5xx at the baseline EC2/DB configuration, we were not able to support a very high user arrival rate. Scaling our app server both horizontally and vertically helped us achieve that. After scaling the app server horizontally considerably, we were able to hit the DB bottleneck. In order to eliminate that, we scaled our DB vertically.

By performing these scaling activities, we learned that deploying a scalable application involves hitting a sweet spot between vertical and horizontal scaling. On one hand, scaling horizontally provides reliability and on the other, scaling vertically allows us to support more users. The importance of scale testing an application is one of our major takeaways from this project.