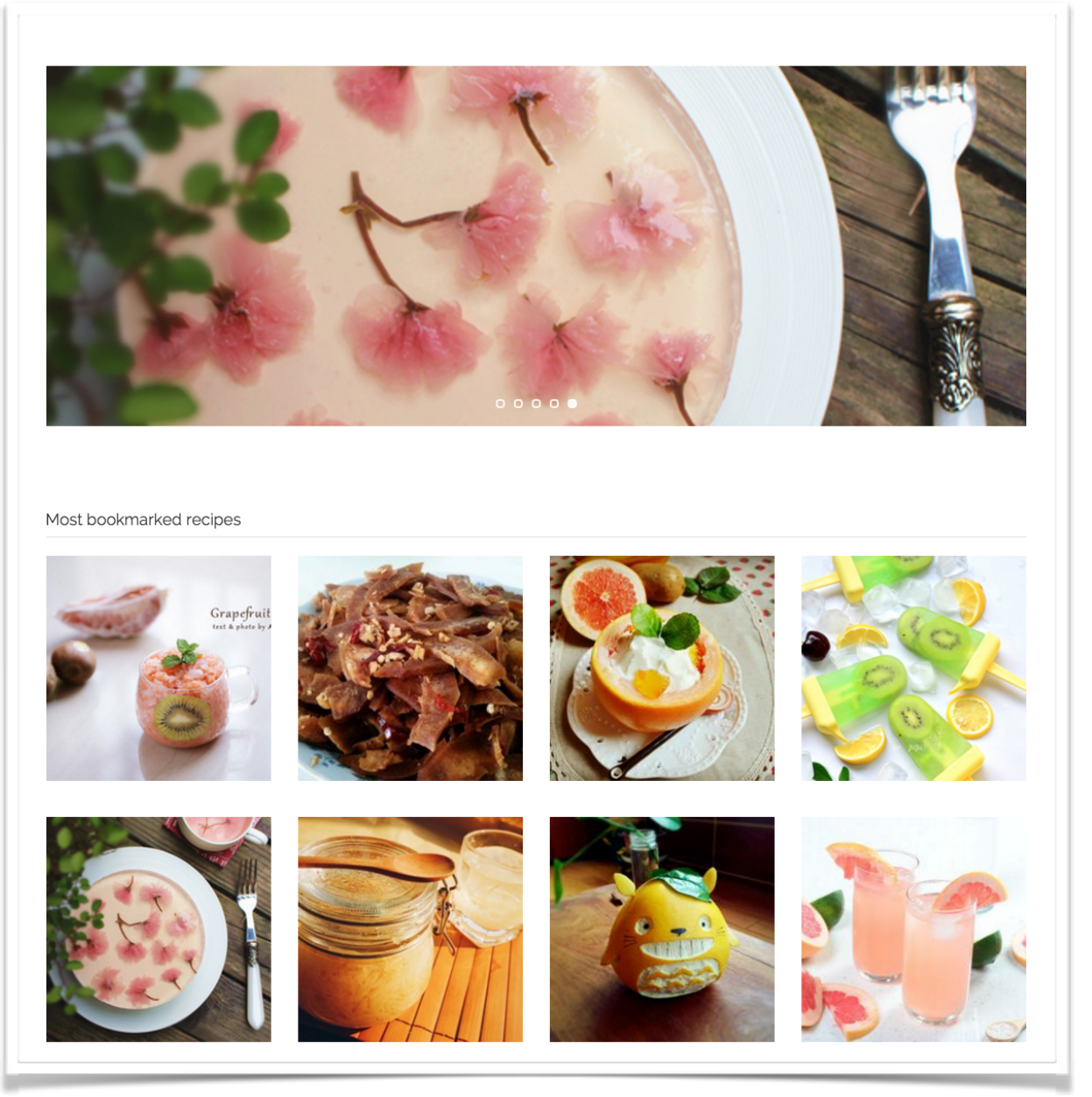
Foodies

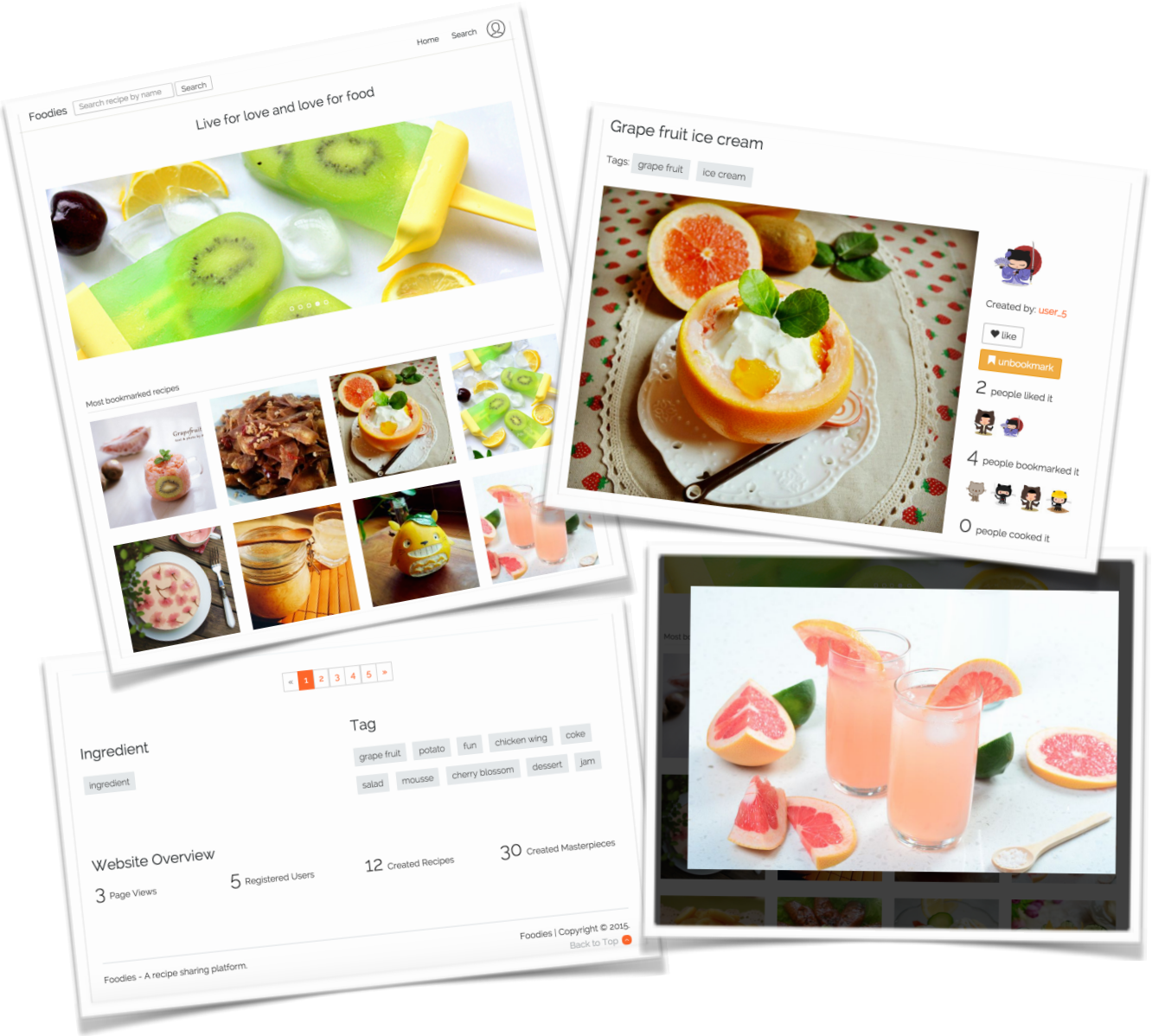
Live for love and love for food



Yuxi Chen
Rigen Mu
Qianyun Zhu
Junchi Zhang
Sha Liu

1. Introduction

If you are worrying about where to find various fantastic recipes, and where to share your own delicious food, the Foodies will be the answer. Foodies is a recipe sharing platform along with social-network features. There are thousands of recipes uploaded by different people each living in different culture. It does not matter whether you are a experienced cook or someone having all meals at restaurant, you will learn our recipes in a incredible speed and quality. Because for every recipe we have provided so much details that lead you to your own masterpiece.



1.1 Recipes

This section is going to show how a recipe and its instructions will be shown to a user. For each recipe show page, its creator, number of like, number of bookmark, and number of people who cook it have been shown on the first part to indicate whether this is a popular one.

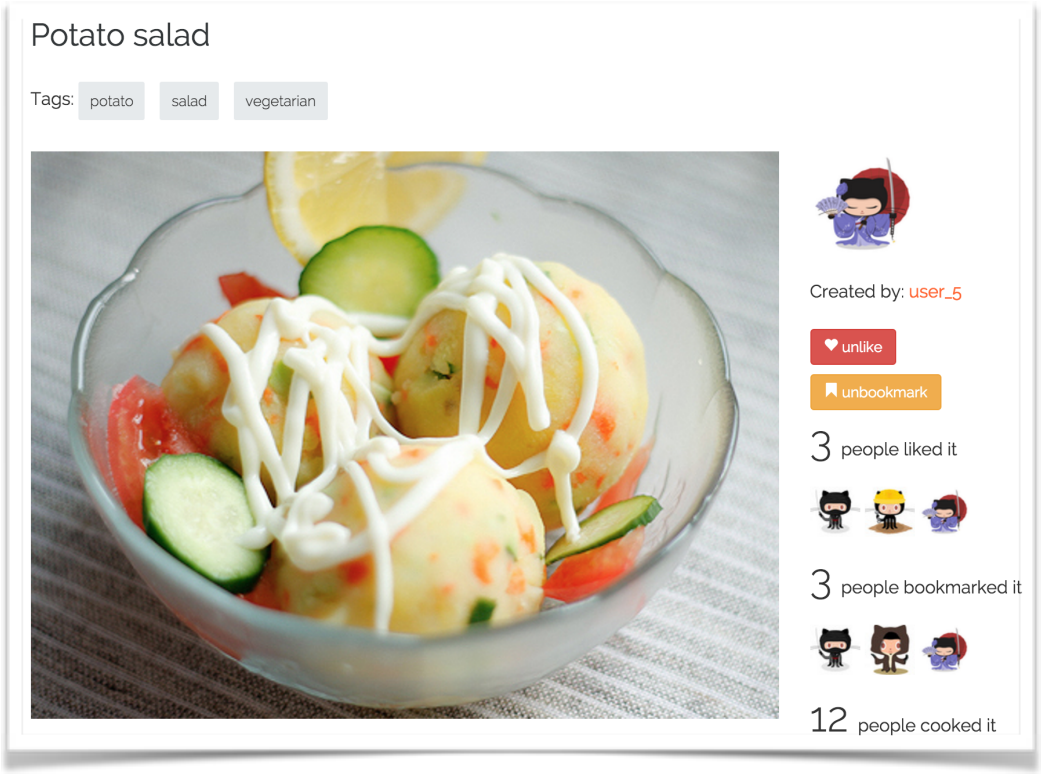


Figure 1.1.1 Recipe Show - 1

By scrolling down the page, instructions for how to cook this dish will be shown including description, ingredients, cook time, step picture and its description.

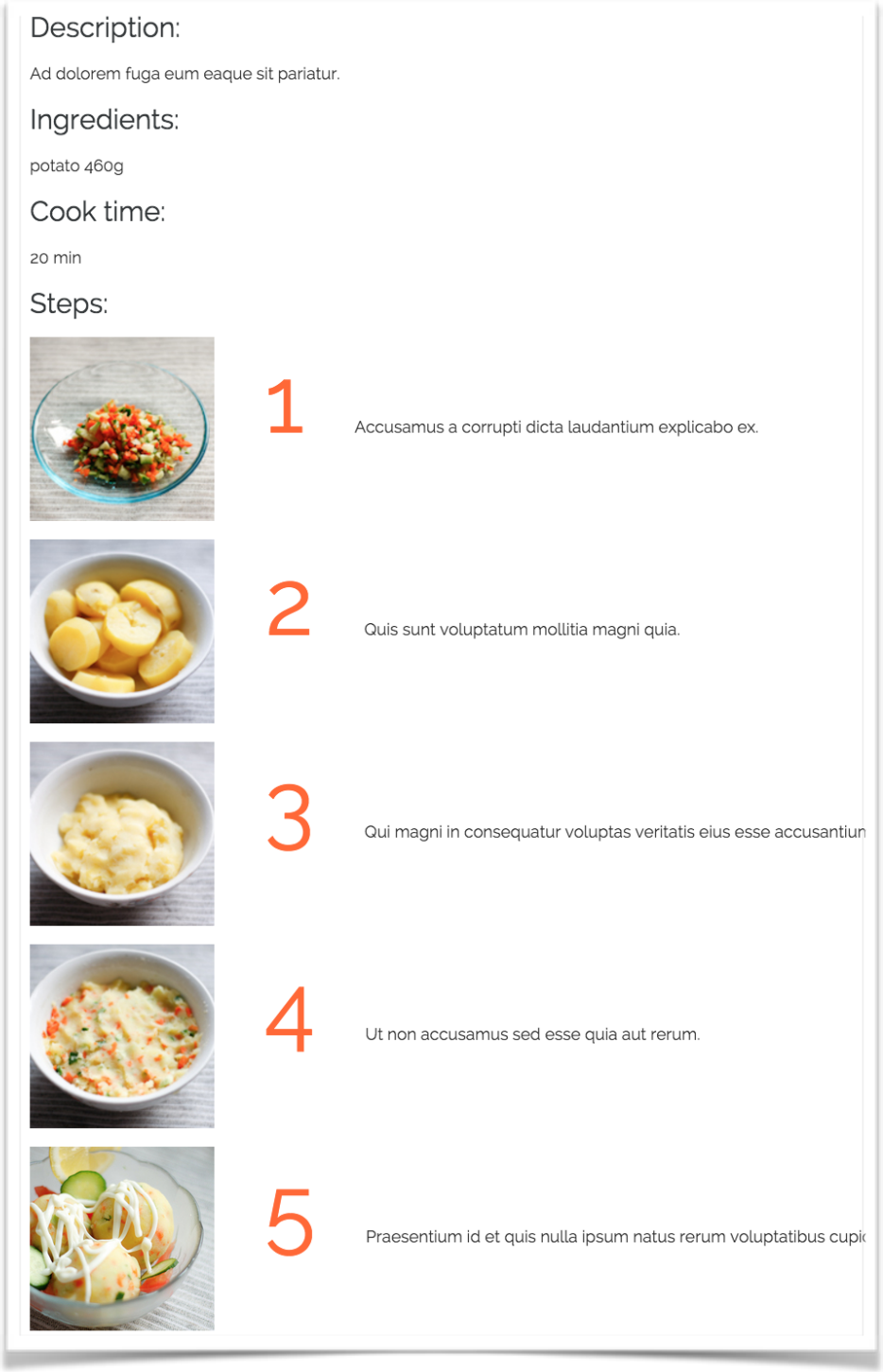


Figure 1.1.2 Recipe Show - 2

1.2 Masterpieces

Sometimes users may be willing to upload their results after learning from a recipe, we called a masterpiece. If you are interested in others' results of cooking this recipe, you can find them easily at the bottom of this page. These picture are uploaded by other learner with or without their own creativity. Of course, you are able to go to these masterpieces detailed page via the link provided by these picture.

Figure 1.2 Masterpieces

1.3 Like & Bookmark

The like and bookmark buttons are shown in each recipe page as Figure 1.1.1 shown. You are welcome to like or bookmark recipes after you sign-in to our application. In addition, you are always allowed to unlike or un-bookmark a recipe if you want to. Remember that the more you contribute, the better the application becomes.

1.4 Tags

Tag section is shown at the bottom of recipe name as Figure 1.1.1 shown. When users creating or editing a recipe, they are allowed to add multiple tags to their dishes. Tags could be taste of the food, category of main ingredient, or which culture does the food originate from. By clicking each tag, users are redirected to search-tag page which illustrated by Figure 1.4 with the result of that tag.

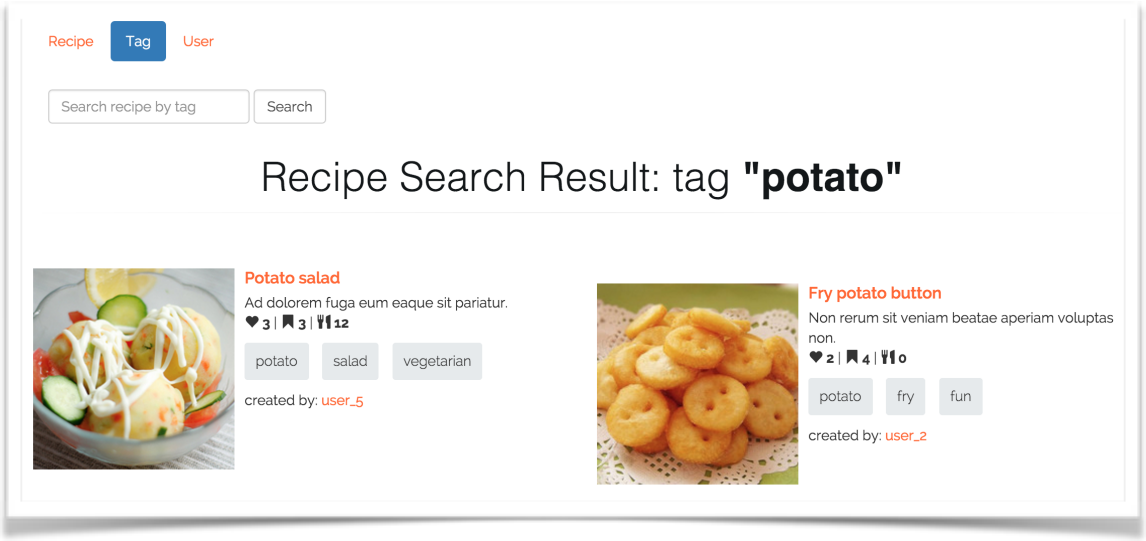


Figure 1.4 Search-tag page

1.5 Search

If user want to search for certain recipe, tag, or other users, our search function is so ready and fast to do that.

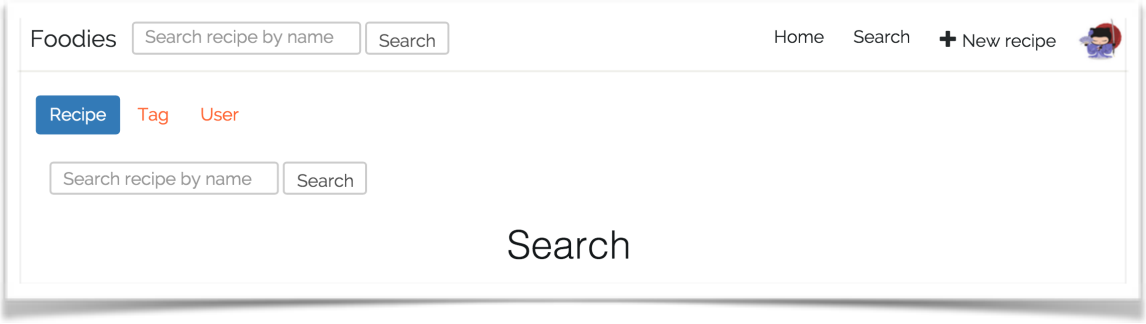


Figure 1.5 Search page

1.6 Create a new recipe

After sign-in, users are able to upload their own recipe by clicking "+ New recipe" button at top-right of any page. By default, our application provide users three ingredients input box and one cooking step input box. User can always add more ingredients and steps by clicking the "+ Add" buttons.

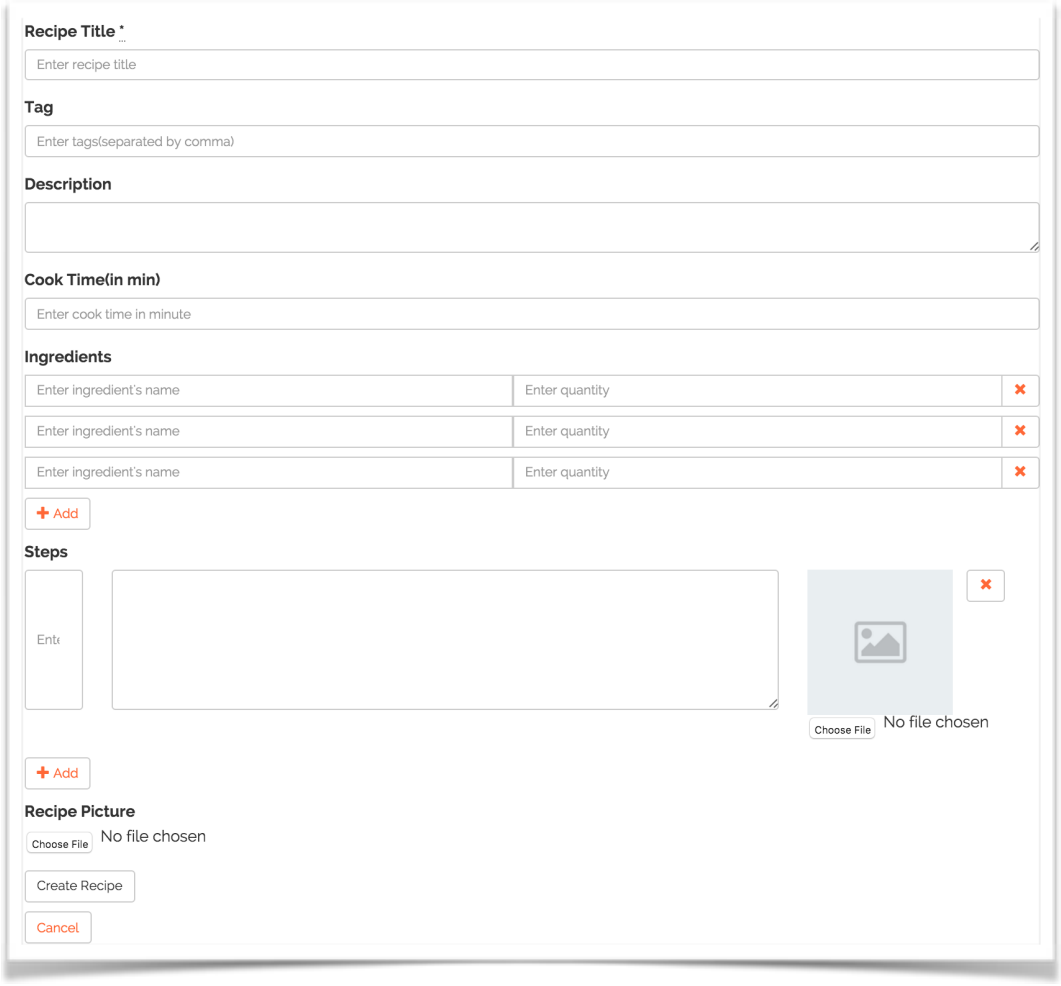


Figure 1.6 Create a new recipe

1.7 User profile

Users are able to edit their profile from user-profile page. In addition, all recipes that have been liked and booked are shown in this page as well masterpieces.

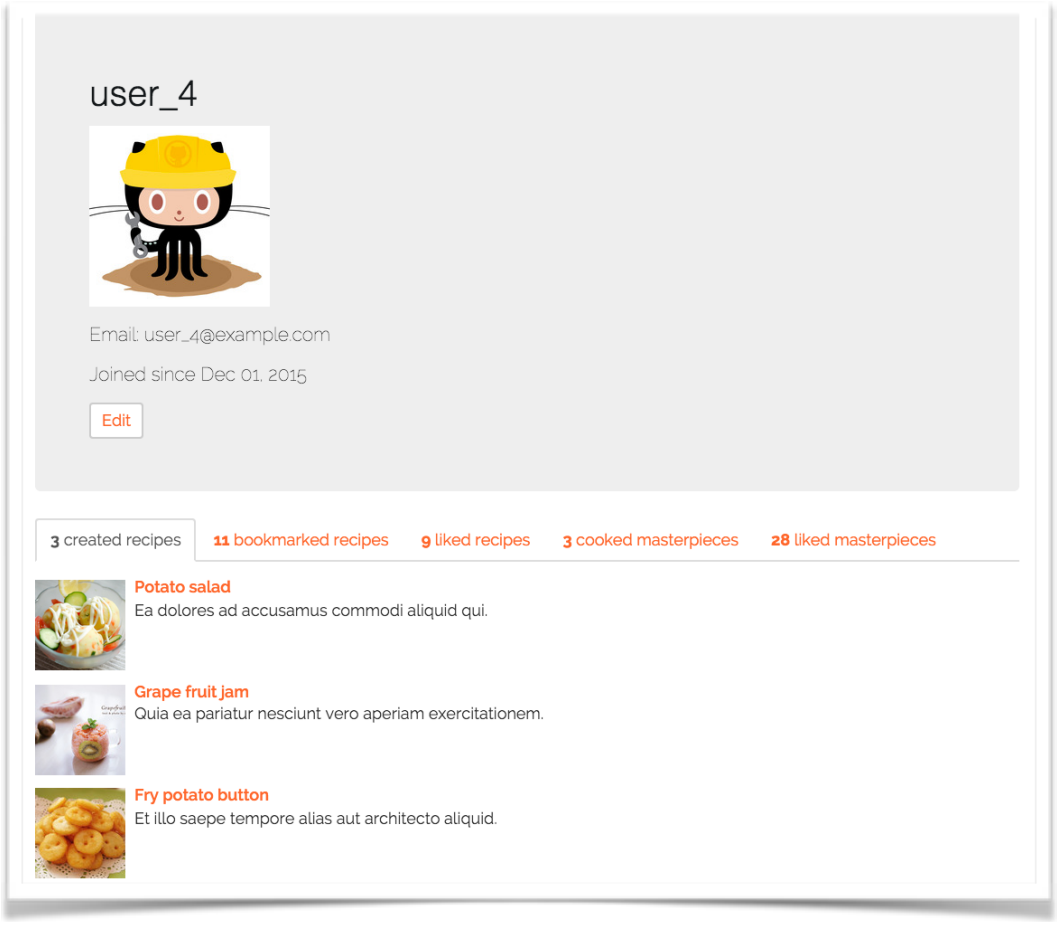


Figure 1.7 User profile

2. Development

Our web application is developed by using agile development methodology, git for version control, pivotal tracker for project management, and travis CI for testing. It starts with listing general features, for example a user can like a recipe, and then data models are defined based on the features.

Especially, git plays an important role in our project development. Each team member develops on a specific feature branch and sends a pull request for others to review and comment. Once the pull request approved by all members, it merges with the master branch. Such process allows us to connect commits with features and track git-log more easily. Besides, through story-id in git commit message, pivotal tracker automatically change its status of the story from start to finished.

Apart from the above techniques, six git-wiki pages are created and used to record development tutorials for the whole group as Figure 2 shown. The project also follows test-driven development with the help of travis CI. Before any push to remote git branch, "rake test" is run to ensure the correctness of functions, which helps us fix some logic bugs.

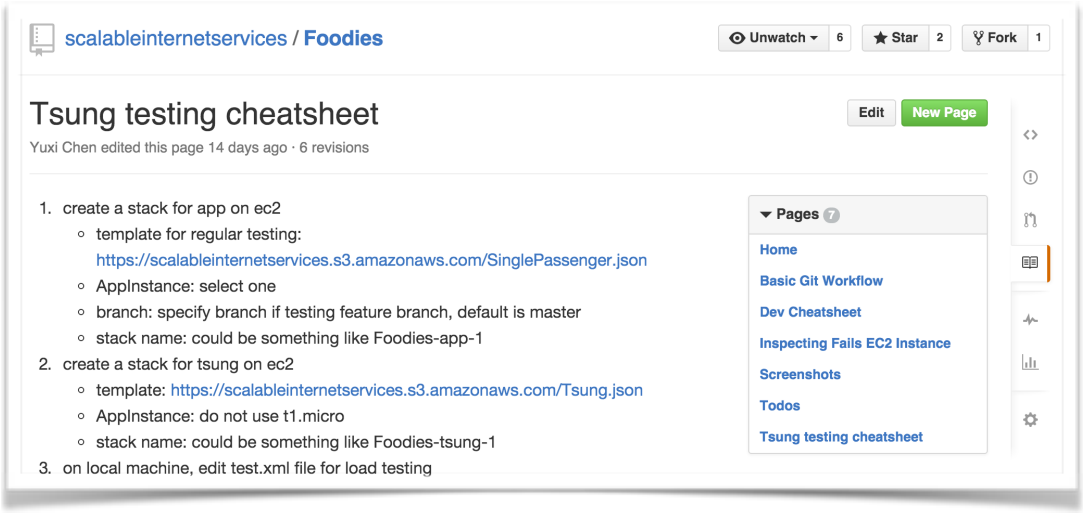


Figure 2 Sample Github Wiki page

3. Application Architecture

We used the model-view-controller (MVC) architecture to build our application. When the user visits a specific url, the app will call related controller and the controller will process the request and generate corresponding view. We also defined several models to maintain the relationship between objects and the database. Our models can be represented as Figure 3 shown.

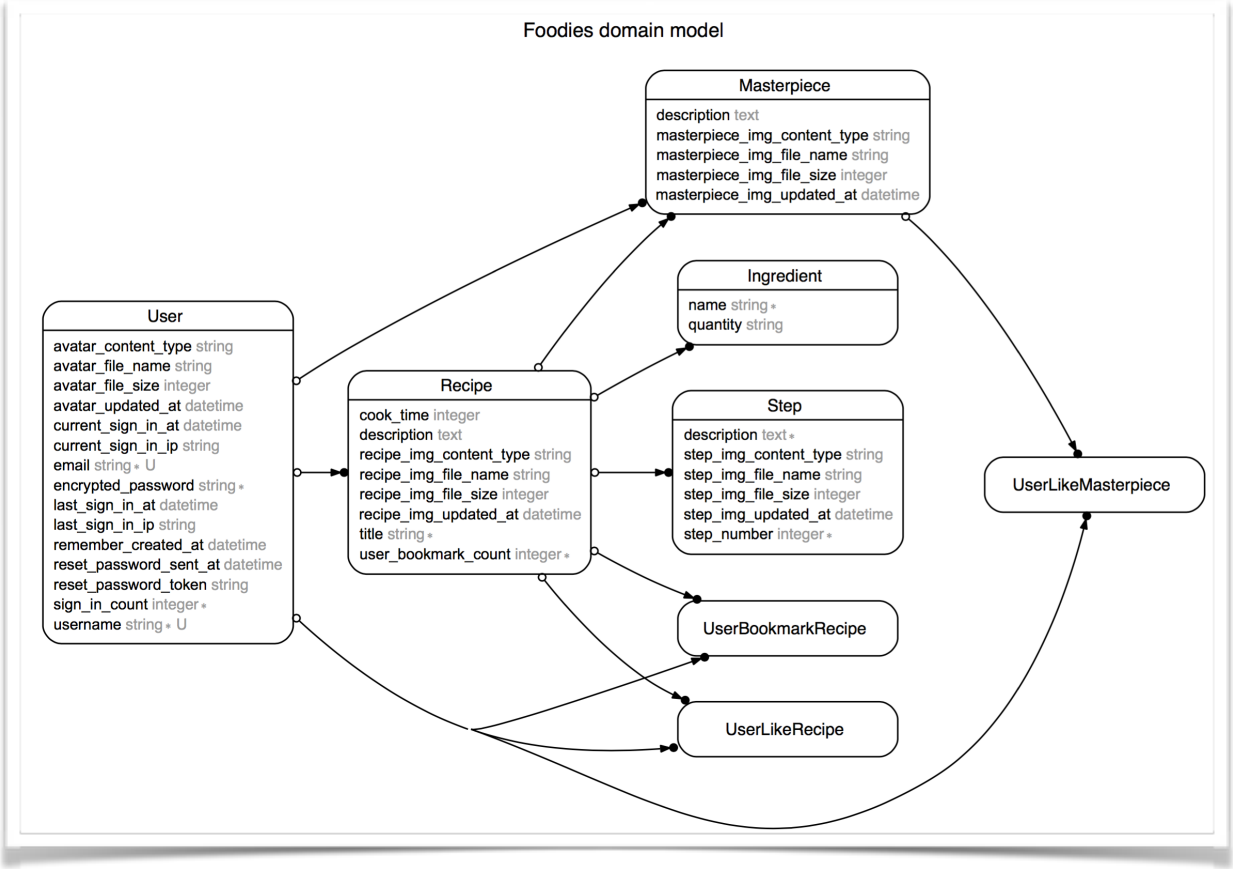


Figure 3 Foodies Domain Model

We have three main models with both views and controllers: User, Recipe, Masterpiece. Each user can create multiple recipes and multiple masterpieces. Each masterpiece belongs to one recipe. Each recipe can have multiple ingredients and multiple steps. Users could also like/bookmark a recipe. To represent such relationship we build models such as UserLikeRecipe which indicates the many to many relationship between User and Recipe.

4. Performance Analysis

4.1 Critical user path

The test dataset contains 500 users, 100 tags, 50 ingredients. It simulates that each user has an average of 2 recipes and 4 masterpieces. In addition, it assumes that each user likes 20 recipes, bookmarks 10 recipes and likes 20 masterpieces on average. And each recipe has an average of 3 tags, 3 ingredients and 2 steps. In summary, the whole dataset consists of 500 users, 1000+ recipes and 2000+ masterpieces.

Phase (30s/P)	1	2	3	4	5	6	7	8
Users / sec	1	2	4	8	16	32	64	128

The test script has 5 phases at first and the number of phases may vary for different optimizations. The critical user path defines 14 user behaviors and assign different weights to these behaviors as Table 4.1 shown. Each user behavior is relevant to one separate xml file and the critical xml is a combination of all these files. The way we separate these files helps us to simulate specific user behavior on targeted pages and check the impact of optimizations more efficiently.

User behavior	Weight
login-logout	1
visit-homepage	10
visit-recipe	20
visit-masterpiece	5
visit-user-profile	5
visit-search-index	1
search-by-name	5
search-by-tag	4
search-by-user	1
create-delete-recipe	1
create-delete-masterpiece	1
like-recipe	2
bookmark-recipe	2
like-masterpiece	1

Table 4.1 User behaviors' weight

4.2 Tsung as a test tool

The scalability of our deployed application on Amazon EC2 is evaluated by Tsung, in which response time, error rates and some other valuable parameters are observed for further performance analysis. Various load tests have been performed including but not limited to server and client caching, different types of SQL optimization, pagination, as well as vertical and horizontal scaling. In each test, a variety of actions of a large number of simultaneous virtual users are simulated to test one optimization on a separate branch on Github.

4.3 New relic

In our optimization, especially for SQL optimization, we used a tool, called New Relic, which provides a performance management solution allowing developers to diagnose and fix problems in application performance in real time. For each user action on the application, New Relic will show how the database is visited as follows. Furthermore, detailed datasets of the time interval for each query is generated, which is quite powerful when the developer is trying to reduce the time for some queries, deduct some unnecessary queries and optimize the overall performance of the application.

Metric	Count	Exclusive		Total	
HomepageController#show	1	91 ms	35%	173 ms	67%
homepage/show.html.erb Template	1	43 ms	17%	47 ms	18%
Rack/ActiveRecord::Migration::CheckPending/call	1	33 ms	13%	246 ms	96%
layouts/application Template	1	30 ms	12%	32 ms	12%
Rack/ActionDispatch::Routing::RouteSet/call	1	30 ms	12%	203 ms	79%
operation/SQLite/select	2	5 ms	2%	5 ms	2%
Remainder	1	24 ms	9%	24 ms	9%

Figure 4.3 New Relic result example

4.4 Initial performance testing

A single t1.micro EC2 instance is used for our initial performance evaluation. The purpose of the initial test is to give an overview of the system performance and hence, to plan on further optimization. We used Tsung for the load test to simulate users' actions on the application and 5 phases is used for this initial test, and the detailed dataset was described in Section 4.1.

The initial Tsung test shows that the average response time for this configuration is 9.21 seconds for a total number of 921 users, which is not acceptable for a practical usage; moreover, the code of 503 is shown almost from the beginning, which is a sign of overloaded request for the app server. There can be multiple reasons for the sluggish response time, such as the configuration of server machine, inefficient database structure for the queries, etc. Therefore, we will optimize the overall application performance in the following sections from various aspects.

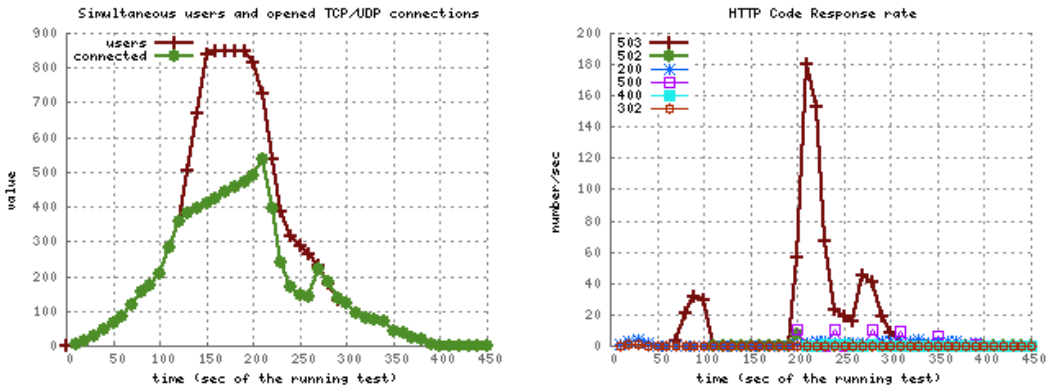


Figure 4.4 Simultaneous Users & HTTP Response

5. Server-side Caching

5.1 Fragment caching

Fragment caching allows the application to cache a fragment of the view and retrieve the view fragment from the cache. And ready for rendering based on a cache key that could uniquely identify the content of the fragment.

5.1.1 Fragment caching on homepage

In the application homepage, top bookmarked recipes are shown on the page as snippets with basic recipe info. Therefore, we use fragment caching to cache the view fragment of each recipe snippet based on recipe's content.

We defined the cache key in homepage_helper.rb like the following:

```
def cache_key_for_recipe_snippet(recipe)
  "homepage-recipe-#{recipe.id}-#{recipe.updated_at}"
end
```

We specify the cache block in homepage show page like the following:

```
<% @top_bookmarked_recipes.each do |recipe| %>
  <% cache(cache_key_for_recipe_snippet(recipe)) do %>
    <div>
      <!-- recipe content containing -->
      <!-- recipe image -->
      <!-- recipe title -->
      <!-- recipe description -->
    </div>
  <% end %>
<% end %>
```

In order to test the performance improvement, we run tsung load test specifically on the session where users simply visit the homepage a lot of times. From the Figure 5.1.1 we can see that the peak response time reduced from 4s to 2s after implementing fragment caching. The server can handle up to 8 users/sec instead of 5 users/sec based on the time that 503 error code started to occur. Also, the 200 OK response rate increased too which indicate higher server performance.

Master	Master with fragment caching

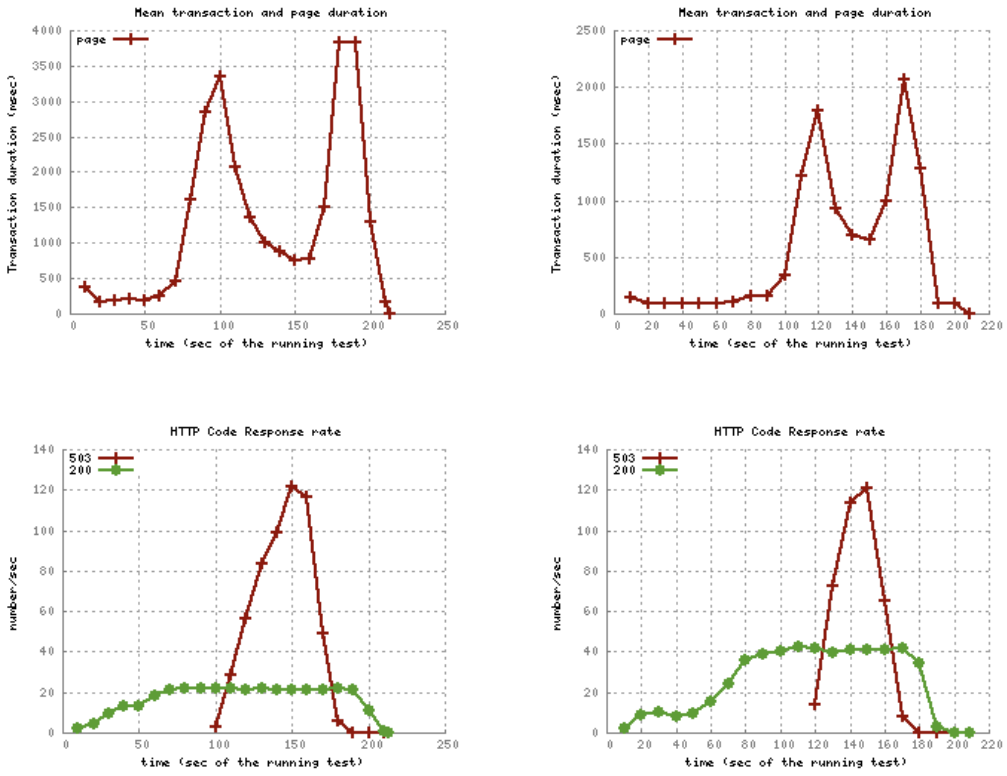


Figure 5.1.1 Fragment Caching Result

5.1.2 Fragment Caching on homepage

Besides caching each recipe snippets on homepage, we can also cache the whole recipe table if everything in the table doesn't change. This is the so-called Russian Doll Caching technique.

We defined the cache key in homepage_helper.rb like the following:

```
def cache_key_for_recipe_snippet(recipe)
  "homepage-recipe-#{recipe.id}-#{recipe.updated_at}"
end

def cache_key_for_recipe_table
  max_updated_at =
   @top_bookmarked_recipes.max_by(&:updated_at).updated_at
   "homepage-recipe-table-#{max_updated_at}"
end
```

We specify the cache block in homepage show page like the following:

```
<% cache(cache_key_for_recipe_table) do %>
  <% @top_bookmarked_recipes.each do |recipe|%>
    <% cache(cache_key_for_recipe_snippet(recipe)) do %>
      <div>
        <!-- recipe content containing -->
        <!-- recipe image -->
        <!-- recipe title -->
        <!-- recipe description -->
      </div>
    <% end %>
  <% end %>
<% end %>
```

To test the performance improvement, we still ran tsung load test specifically on the session where users simply visit the homepage a lot of times. From the Figure 5.1.2 we can see that the peak response time reduced from 4s to 1s after implementing Russian Doll Caching, which is even better than simple fragment caching. The server can handle up to about 15 users/sec instead of 5 users/sec based on the time that 503 error code started to occur. Also, the 200 OK response rate increased even more, almost trippled, and this indicated that the server could hanlde more requests. The number of 503 code decreased a lot which means the server had higher availability.

Master	Master with Russian Doll

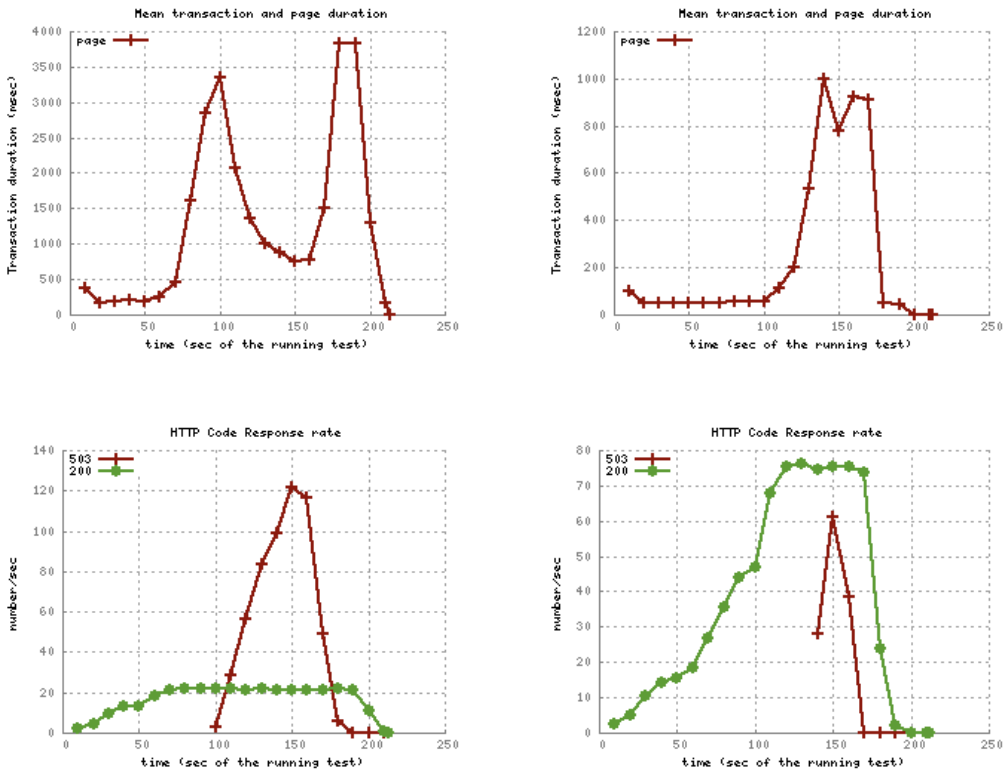


Figure 5.1.2 Russian Doll Cashing Result

5.1.3 Fragment Caching on other pages

We also did fragment caching on other pages and it boost some performance of the server. The different approaches we tried and tested are listed in Table 5.1.3.

Fragment Caching Approach	Detail Implementation
Application Layout	cache the HTML head, header bar and footer
Recipe New Page	cache the whole form for creating a new recipe
Masterpiece New Page	cache the whole form for creating a new masterpiece
Recipe Show Page	cache the recipe view fragment cache the masterpiece snippet fragment fort a recipe
Masterpiece Show Page	cache the masterpiece view fragment
Search Result Page (Russian Doll Caching)	cache each recipe snippet fragment cache the whole recipe search result table
Login / Signup Page	cache the whole form for login / signup
Profile Page	cache the user info view fragment

Table 5.1.3 Approaches for Fragment Caching

For the approaches that caching static content such as recipe_new page and login page, we didn't see a great performance improvement. The rails server probably need the same amount of time either generating static html or read cached html from cache store.

For the approaches that cache dynamic content such as recipe_show page, we observed slight performance improvement. However, if we cache too much content for example in search_result page, we observed slight performance degradation. In the search_result page, caching the whole table might take up a lot of space so that when the memory was full, older fragments would be cleared out.

Due to frequently switching cache fragments in and out, we tend to miss the cache we want. Writing the cache also takes extra time that's not worth it because we quickly throws the cache out due to space limit. Therefore, we can reflect from the

test result that we should only cache fragment that are commonly used and rarely change. Furthermore, it could be the case that the smaller the cache fragment, the better.

5.2 Low Level Caching

Low level caching is often used to cache stable but computationally expensive query results. However, our project does not have many suitable cases as most content in our web pages keeps changing. One fit is the display of user count, recipe count and masterpiece count on our homepage. Rather than constant count(*) queries, it stores count information in memory as following query.

```
@user_count = Rails.cache.fetch( "user_count", expires_in: 1.minutes )
```

In this case, count can be fetched directly from cache instead of database. However, most content in homepage keeps changing and thus low level caching count does not help a lot.

5.3 Memcaching

The rails default caching uses memery_cache. Memcache is another kind of cache store and we used the dalli gem as the memcache store. We used memcache with the same cache mechanism as the previous Russian Doll Caching approach in 5.1.2. The load testing result also show performance improvement. From the figures we can see that the peak response time reduced from 4s to 1.2s.

The server can handle up to about 10 users/sec instead of 5 users/sec based on the time that 503 error code started to occur. Also, the 200 OK response rate increased a lot, and this indicated that the server could hanlde more requests. The number of 503 code decreased a lot which means the server had higher availability.

Before memcaching	After memcaching

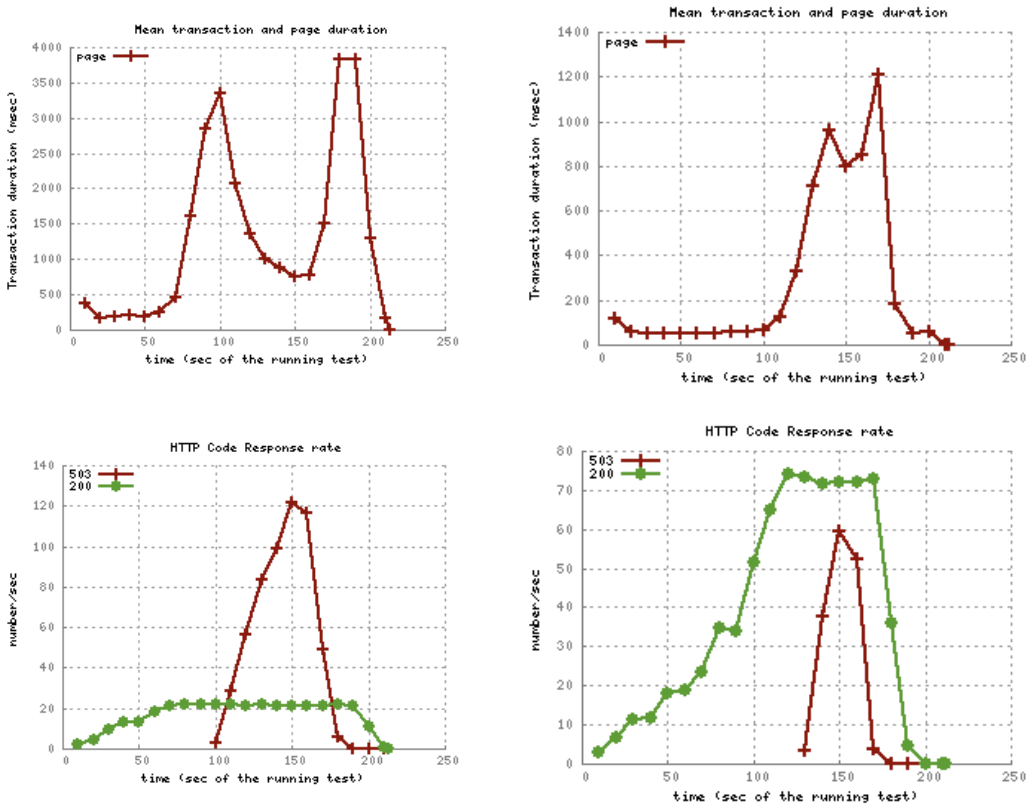


Figure 5.3 Memcahing Result

6. Client-side Caching

Client side caching is used in homepage controller with "stale?" method shown as follow.

```
if stale?([Recipe.all, ActsAsTaggableOn::Tag.all, User.all, Masterpiece.all,
current_user == nil ? 0: current_user.id ])
```

The "stale?" method sets etag and last modified headers based on updated_at columns in recipe, tag, user, and masterpiece tables plus current_user.id, which allows server to determines if the request is stale by comparing the computed etag with the etag in the request.

With this configuration, the first request to Homepage#show invokes the full request while the subsequent requests skip view rendering and get a 304 not modified. Client side cache helps avoid view loading from client perspective as well as bypass request process for the server.

Figure 6 compares the response time and number of SQL between the first and second request. We can see a decrease of 170ms in response time and 12 SQL statements reduced.

Timestamp	Resp. Time	URL	
22:46:00	26 ms	/	[Detail] [SQL (4)]
22:45:44	195 ms	/	[Detail] [SQL (16)]

Figure 6 Response Time Comparison

7. SQL Optimization

7.1 Add index

Expanding on the search algorithm efficiencies, a key area in database performance is how fast the data can be accessed. In general, adding index to columns is one way to improve accessing performance.

In our application, table updating and inserting are the actions which performed less than selecting certain information from database. Therefore, adding index can result more work load that can be handled by server. As Figure 7.1 show, server with indexes starts to return errors later than the one without index. In addition, the first peak period perform 60% better than original application without indexes. Server with indexes are good to respond to client lower than 2s

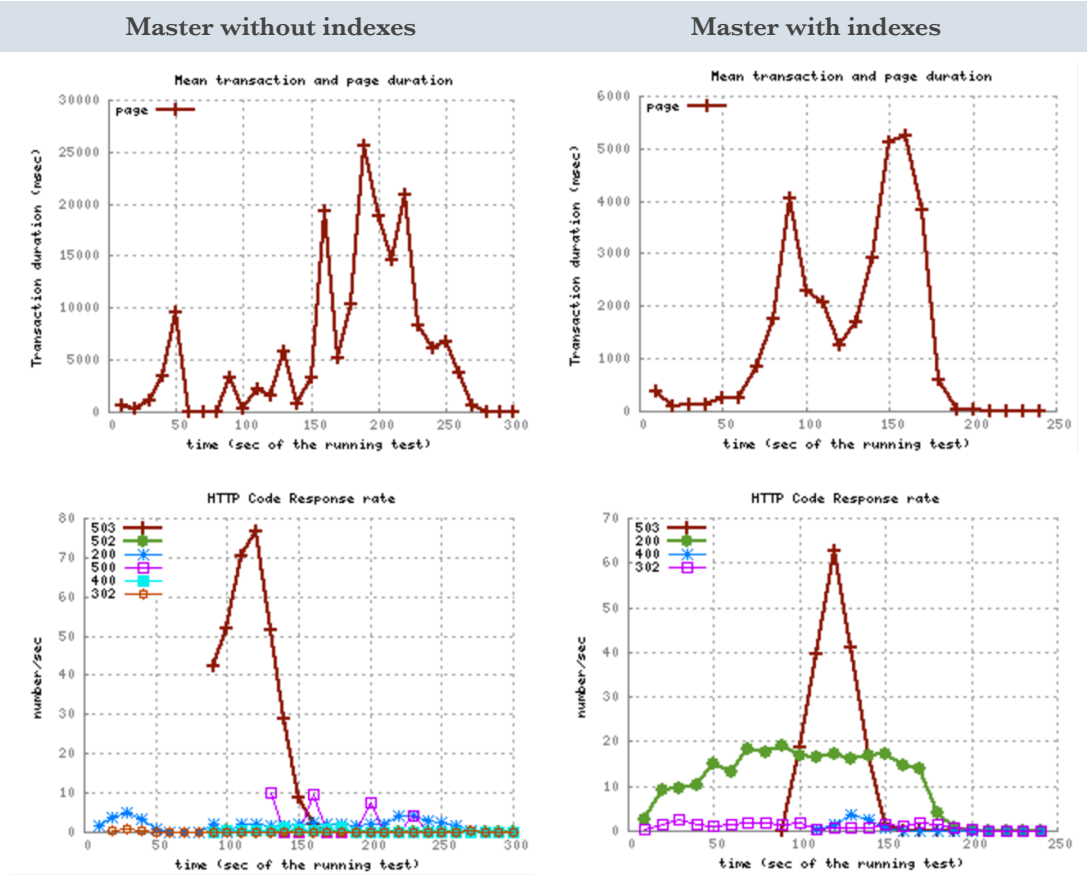


Figure 7.1 With/Without Indexes Comparison

7.2 Optimizing count(*)

We observed a lot of count(*) queries and found out that we are calling count method on already loaded entries. For example, in recipe show page, we display the number of users liking the recipe by calling count method on the object array @users_liking_it. Given that we already loaded all the users liking the recipe, we could avoid extra count(*) calls by using the length method on the object array instead.

We used the critical.xml to test the optimization performance and saw slight improvement. The average transaction time reduced from 1.45s to 1.38s.

7.3 Counter cache

Counter cache is being used to find the total number of belonging objects more efficiently. One of the cases in recipe_show view can well illustrate the use of counter cache. Before using counter cache, it needs to query user_like_recipe table and count number of rows related to recipe "r1" to get user_like_recipe count. After adding counter_cache (as Table 7.2 shown), recipe table has one more column named user_like_count. Now user_like_count can directly return the count of the users who liked recipe "r1" without querying user_like_recipe table. Similar cases exist in masterpiece_show and profile_show views.

```
class UserLikeRecipe < ActiveRecord::Base
 belongs_to :user, counter_cache: :liked_recipe_count
 belongs_to :recipe, counter_cache: :user_like_count
end
```

It turns out that counter cache does not improve the performance probably because constant update on count column can cost time.

7.4 Preload

We found out that we have the N+1 problem in our application because we are making extra calls to database when we want specific associated data over and over again. We can use includes to preload and reduce a lot of query calls.

For example, in our search result page, we are displaying recipes that satisfy the search constraint with recipe information, along with the user info of the recipe creator and the tags info of the recipe. N+1 problem occurs because for each recipe, the application will ask the database for the user info by querying the users table and tag info by querying the tagging table based on the recipe id. Below is the query log for related calls when displaying 4 recipes on the search result page:

```
ActsAsTaggableOn::Tag Load (0.2ms)  SELECT "tags".* FROM "tags" INNER JOIN "taggings" ON "tags"."id" =
"taggings"."tag_id" WHERE "taggings"."taggable_id" = ? AND "taggings"."taggable_type" = ? AND
"taggings"."context" = ?  [["taggable_id", 10], ["taggable_type", "Recipe"], ["context", "tags"]]
User Load (0.2ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT 1  [["id", 5]]
ActsAsTaggableOn::Tag Load (0.4ms)  SELECT "tags".* FROM "tags" INNER JOIN "taggings" ON "tags"."id" =
"taggings"."tag_id" WHERE "taggings"."taggable_id" = ? AND "taggings"."taggable_type" = ? AND
"taggings"."context" = ?  [["taggable_id", 9], ["taggable_type", "Recipe"], ["context", "tags"]]
User Load (0.2ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT 1  [["id", 1]]
ActsAsTaggableOn::Tag Load (0.2ms)  SELECT "tags".* FROM "tags" INNER JOIN "taggings" ON "tags"."id" =
"taggings"."tag_id" WHERE "taggings"."taggable_id" = ? AND "taggings"."taggable_type" = ? AND
"taggings"."context" = ?  [["taggable_id", 8], ["taggable_type", "Recipe"], ["context", "tags"]]
User Load (0.2ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT 1  [["id", 4]]
ActsAsTaggableOn::Tag Load (0.2ms)  SELECT "tags".* FROM "tags" INNER JOIN "taggings" ON "tags"."id" =
"taggings"."tag_id" WHERE "taggings"."taggable_id" = ? AND "taggings"."taggable_type" = ? AND
"taggings"."context" = ?  [["taggable_id", 7], ["taggable_type", "Recipe"], ["context", "tags"]]
CACHE (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT 1  [["id", 1]]
```

To solve the N+1 problem and extra reduce query calls, we used includes when getting the recipes and then rails will be smart enough to make only two calls to find all the user info and tags info for all the 4 recipes together.

```
@recipes = Recipe.search(params[:search]).order("created_at
DESC").includes(:creator, :tags)
```

The query log is shown below:

```
User Load (0.4ms)  SELECT "users".* FROM "users" WHERE "users"."id" IN (5, 1, 4)
ActsAsTaggableOn::Tagging Load (0.4ms)  SELECT "taggings".* FROM "taggings" INNER JOIN "tags" ON
"tags"."id" = "taggings"."tag_id" WHERE "taggings"."context" = ? AND "taggings"."taggable_type" = 'Recipe' AND
"taggings"."taggable_id" IN (10, 9, 8, 7)  [["context", "tags"]]
```

As can be seen, the overall query time can be reduced using the preloading optimization technique. The average transaction time is reduced from 1.45s to 1.29s.

7.5 Raw SQL

When users search recipe, we show the count of users liking, bookmarking, cooking a recipe as shown in Figure 1.4. Originally, for each recipe snippet, recipe table has to be joined with uses_liking_it, users_bookmarking_it and masterpiece table and calculate the total count. For example, given N recipes as search results, 3N count(*) calls are made to get count information. In order to avoid repeated queries, raw sql(find_by_sql) is used to get the recipe information along with all count numbers in a single query.

From the load testing results shown in Figure 7.5, the performance improvement has been reflected by both response time decreasing and error codes percentage lowering. With the raw sql technique, the server can handle up to eight users a second instead of two. In addition, the average response time is reduced and the peak period is deferred.

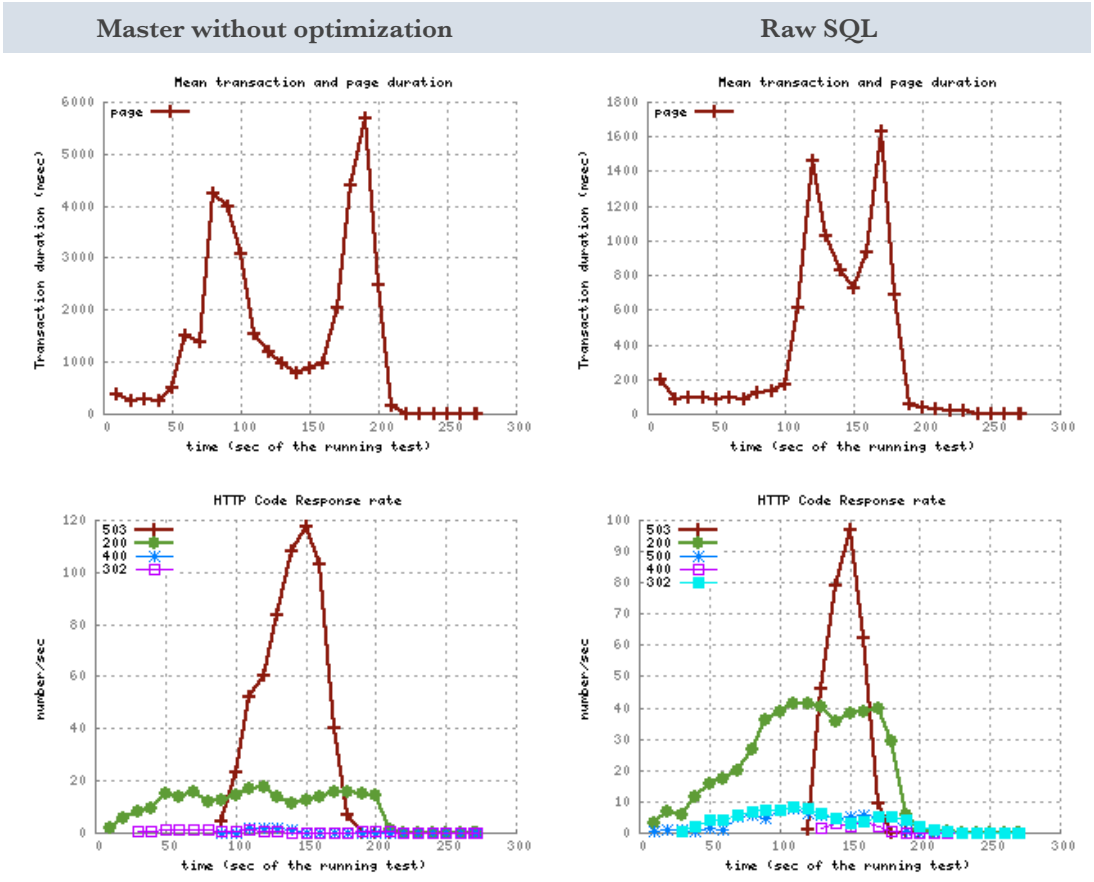


Figure 7.5 Raw SQL Performance

7.6 Select columns

When controllers pass values to views for display, it may be more efficient to pass columns selected by views instead of all the columns.

It can be seen from Figure 7.6, there is slight decrease in peak response time but the server still serve up to 2 users per second after optimization. In other words, the optimization does not have a significant impact. In terms of http response rate, it even gets worse than original one. One guess is that selecting specific columns may be more complicated than selecting all columns and thus sometimes consumes more time.

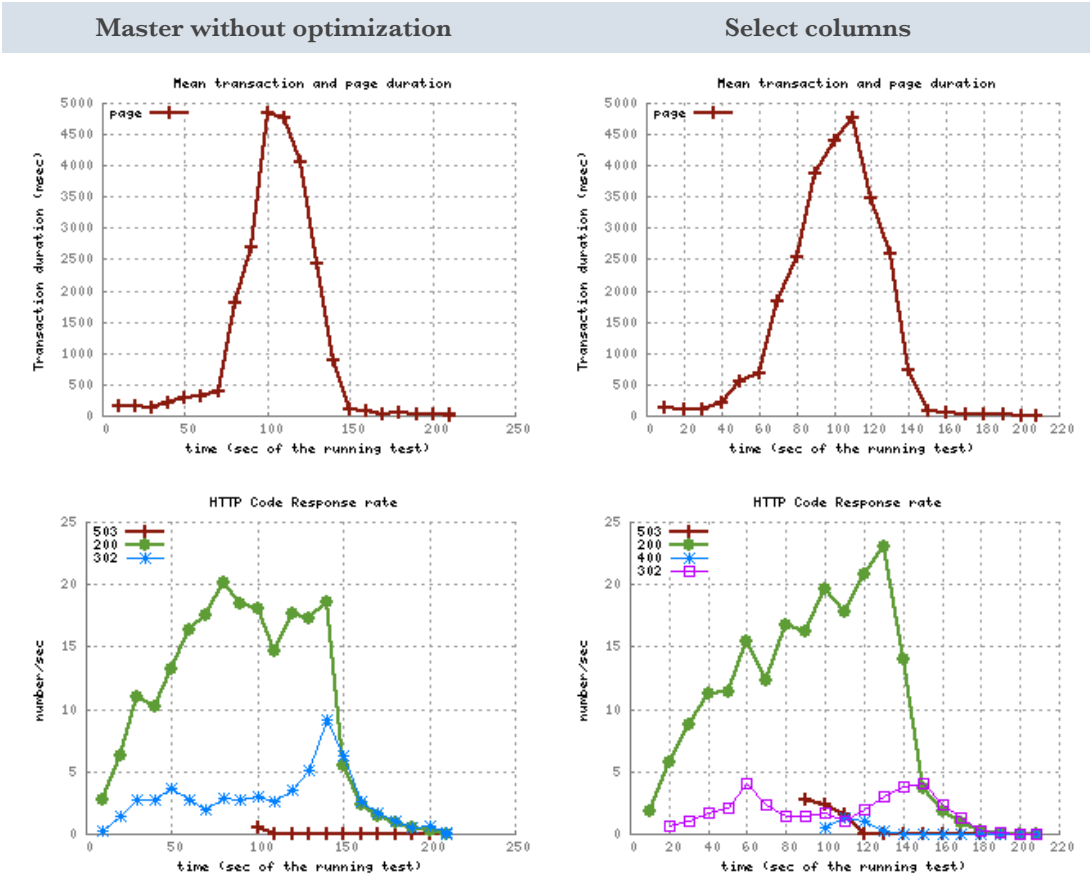


Figure 7.6 Select Columns Performance

8. Pagination

Each time users visit homepage or search something, the app renders all the results at a time. Large amounts information displayed on the screen at the same time will make the response time quite slow. In order to solve this problem, we utilized the kaminari pagination method, so that the results are shown on different pages, each page has limited results (such as 20), Then the rendering time will be highly reduced.

The Table 8 compares the performance before and after pagination. Since the seed data is very large, the improvement is significant. When searching 'a', the time decreased by 94%, when visiting homepage, the time is decreased by 63% after pagination. The performance with and without pagination after deploying on AWS is shown in Figure 8.1 and Figure 8.2.

	Before pagination	After pagination
Search "a"	Completed 200 OK in 13489ms (Views: 12821.4ms | ActiveRecord: 666.3ms)	Completed 200 OK in 856ms (Views: 829.7ms | ActiveRecord: 25.0ms)
Visit homepage	Completed 200 OK in 158ms (Views: 108.7ms | ActiveRecord: 3.6ms)	Completed 200 OK in 59ms (Views: 51.0ms | ActiveRecord: 1.6ms)

Table 8 Pagination Result

Without pagination	With pagination

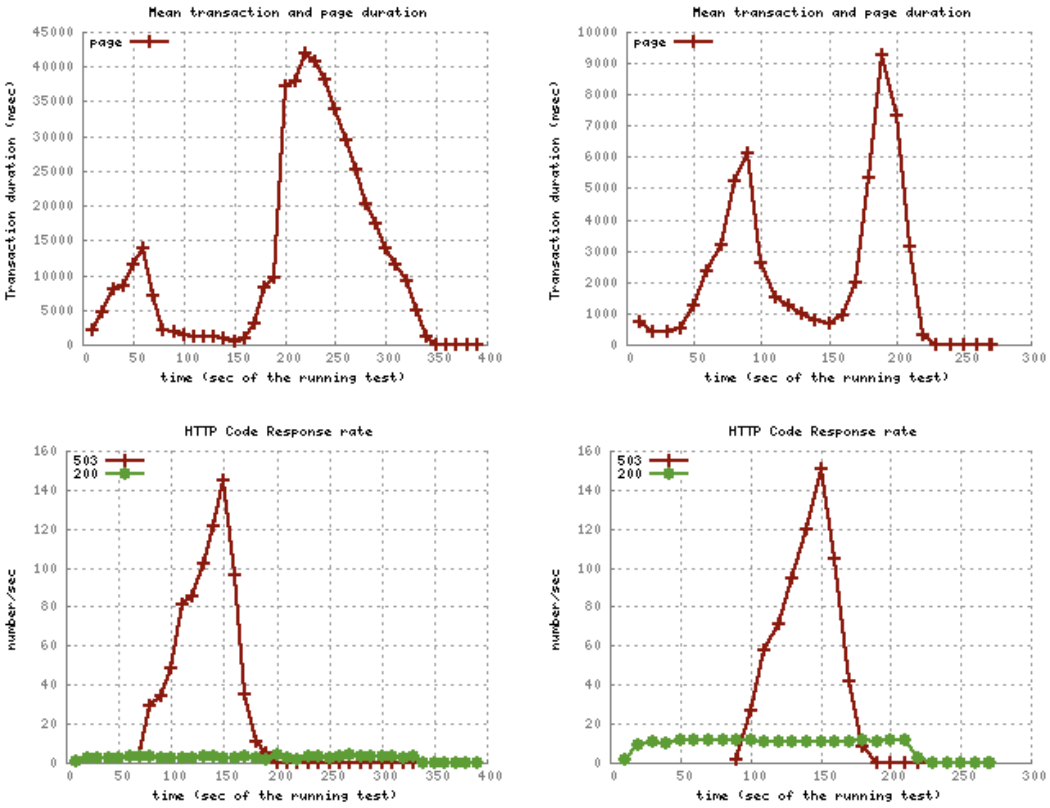


Figure 8.1 Pagination performance comparison (Search "a")

Without pagination	With pagination

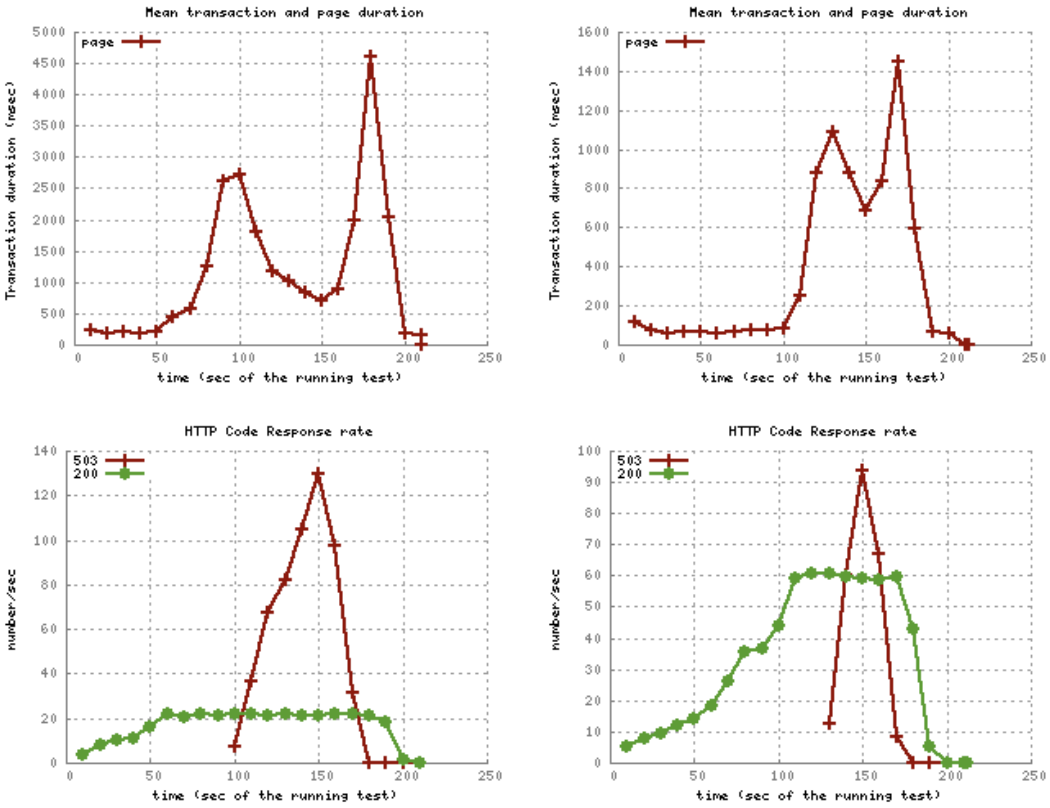


Figure 8.2 Pagination performance comparison (Homepage)

In the figures above, it is obvious that pagination reduced the response time significantly. The performance improvement has also been reflected by higher 200 response rate and lower 503 response rate.

9. Vertical Scaling

Vertical scaling refers to a way of scaling application by adding more power to existing machine, such as CPU, RAM, etc. With faster cpu, more memory, better storage and higher bandwidth, the server performance will be improved significantly.

In order to test the application's vertical scalability, we used six different instance types: M2.xlarge, M2.2xlarge, M2.4xlarge, M3.large, M3.xlarge and M3.2xlarge. For both M2 and M3 instances, they balance the performance of memory and CPU well. However, better and more consistent performance and SSD storage guarantee that M3 outperforms M2 in most cases.The details of different instances are compared below in Table 9.

Instance Type	vCPU	Memory (GiB)	Storage (GB)
M2.xlarge	2	6.5	17.1
M2.2xlarge	4	13	34.2
M2.4xlarge	8	26	68.4
M3.large	2	7.5	32
M3.xlarge	4	15	80
M3.2xlarge	8	30	160

Table 9 Instances Comparison

Each test was performed with a tsung stack running on the same instance. In order to show the improvement of vertical scaling, we compared the response time of request , 200 and 503 response rate of different instances. The results are illustrated from Figure 9.1 to Figure 9.6

We can see that after upgrading from instance M2.xlarge to M2.2xlarge, the mean response time decreased from1.67 sec to 1.06 sec, which corresponding to over 37% lower response time. Going from M3.large to M3.xlarge, the mean response time decreased from 1.83 sec to 1.48 sec, reduced the response time by 20%. Finally, when choosing M3.2xlarge, the mean response time is 0.86sec, reduced 53% corresponding to M3.large, which is also a huge improvement. So as the

features of instances improve, the response time decrease, which meets the expectations. Similarly, the 200 response rate improves and 503 response rate decreases as the features of instances improve.

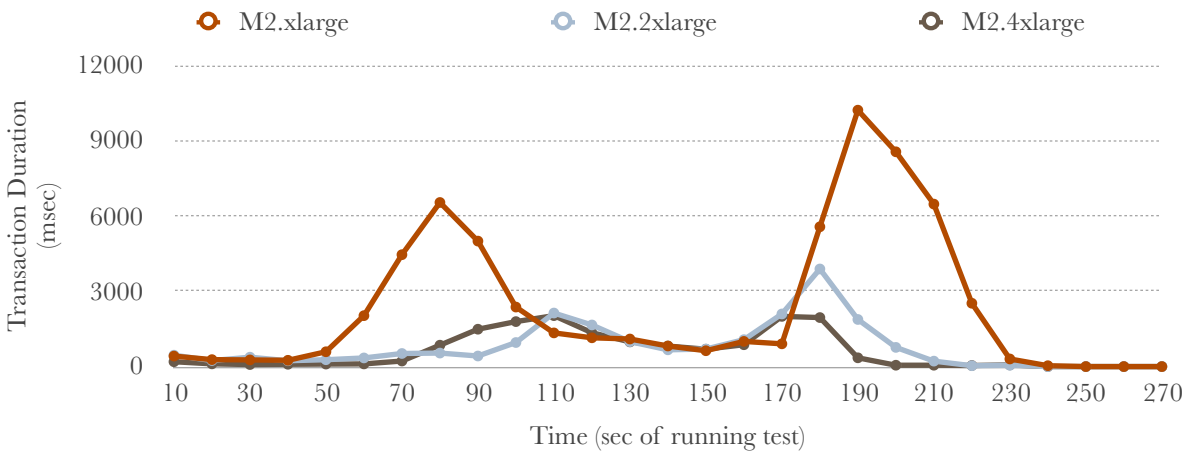


Figure 9.1 Mean Transaction and Page Duration

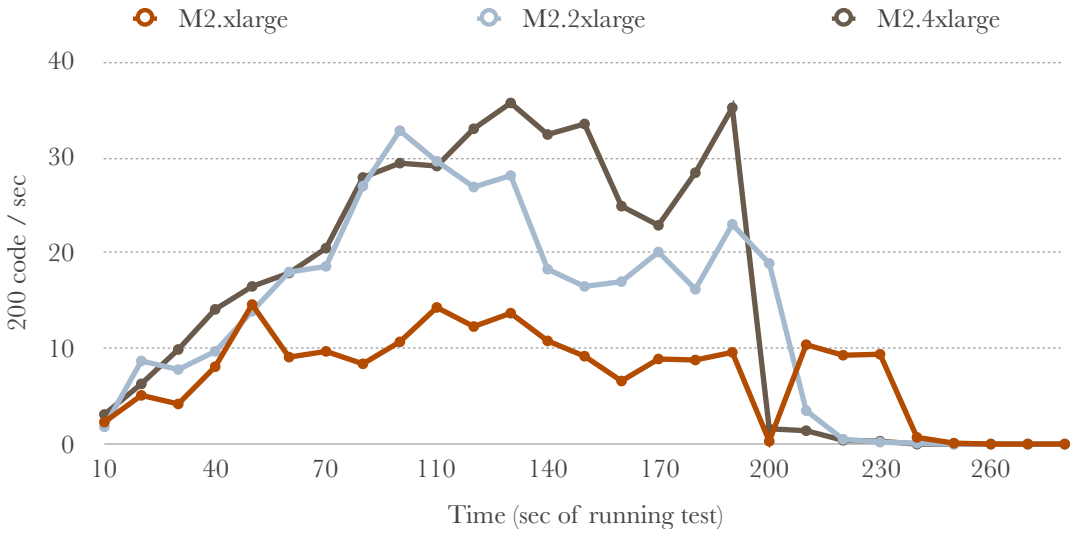


Figure 9.2 Code 200 Response Rate

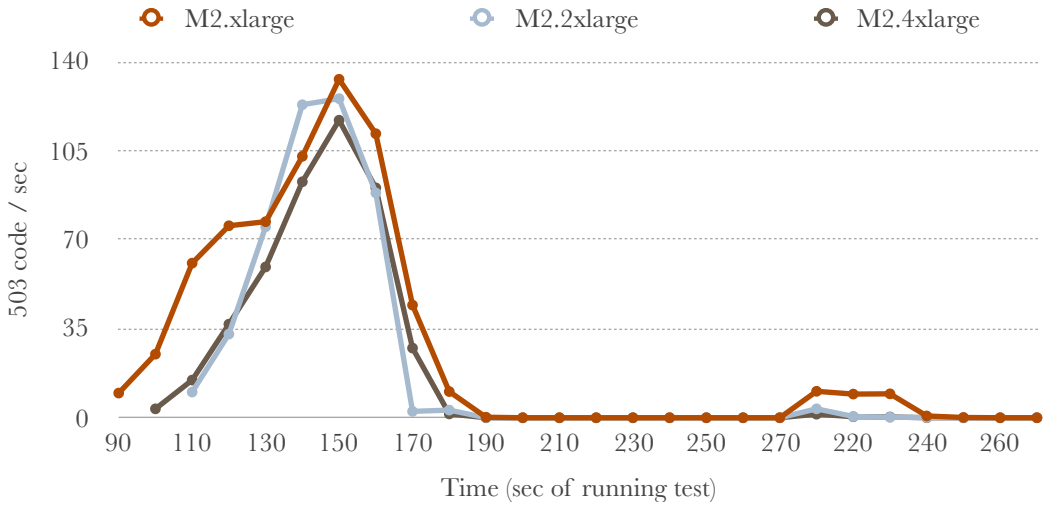


Figure 9.3 Code 503 Response Rate

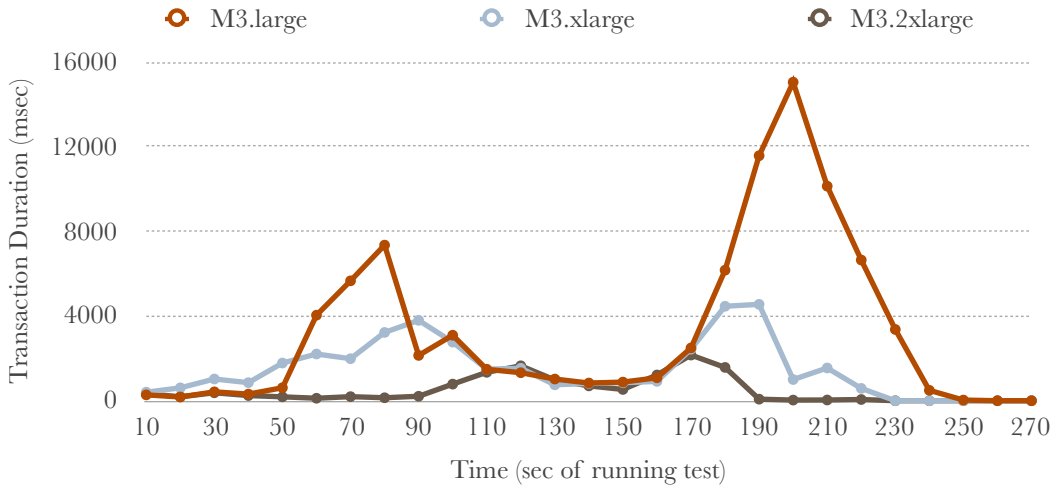


Figure 9.4 Mean Transaction and Page Duration

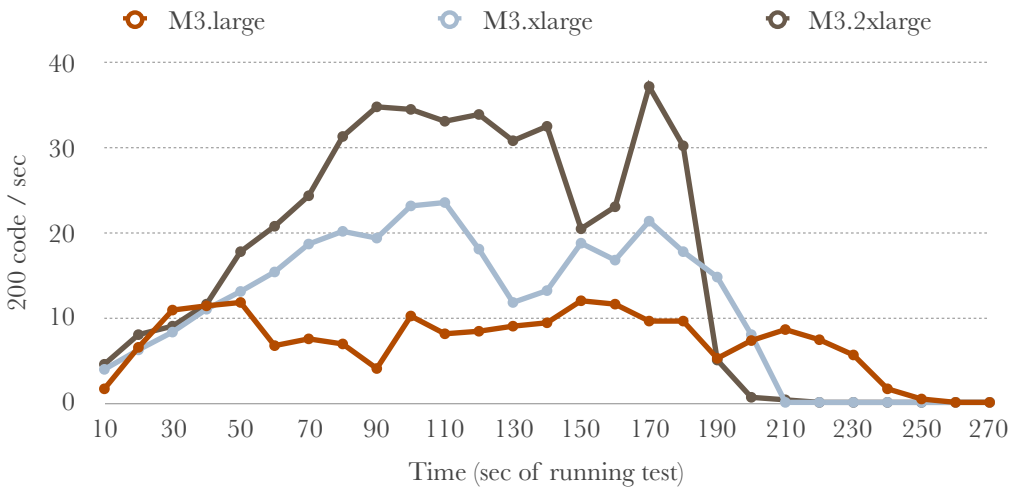


Figure 9.5 Code 200 Response Rate

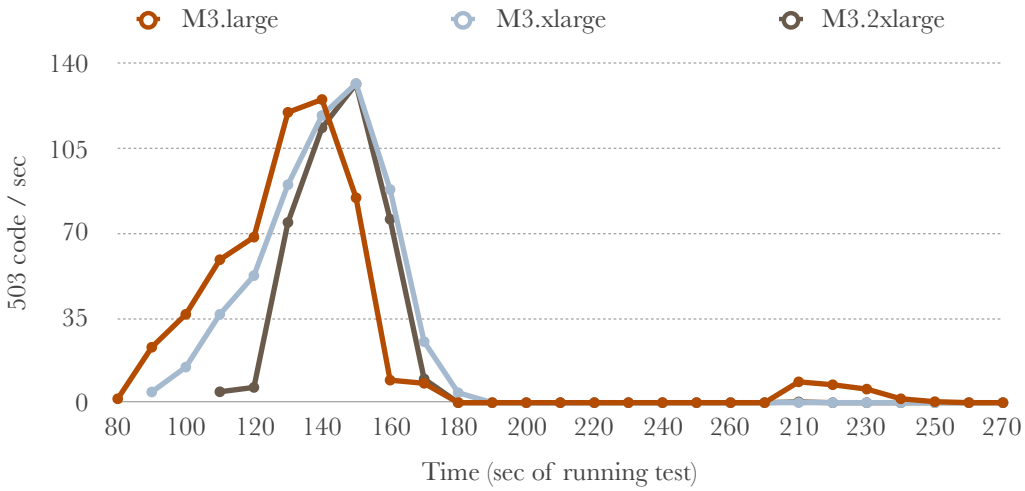


Figure 9.6 Code 503 Response Rate

10. Horizontal Scaling

To handle more users, the system can also be horizontally scaled by adding more server machines to the pool of resources that the application uses, where the load will be divided with a elastic load balancer across these server machines. Moreover, all the servers are further connected to a remote database server, thus all of the actions related to the database occur remotely (shown in Figure 10.1).

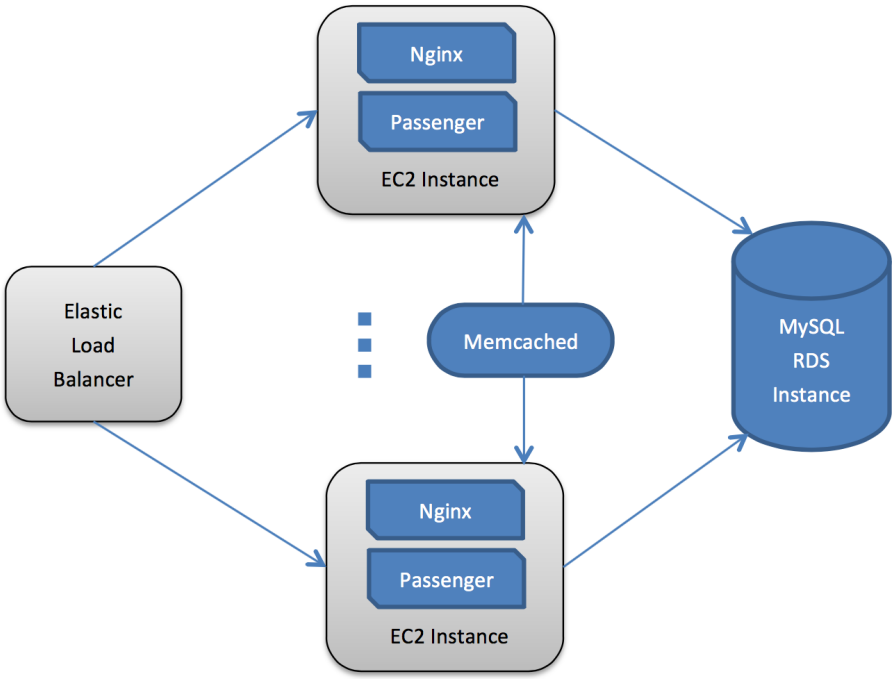


Figure 10 Amazon EC2 Setup

Different sets of configuration are used for the horizontal scaling. First is to change the number of app-server instances to launch; second is to try using different type of machines (i.e. m1.xlarge and m2.4xlarge); and third is to use different number of phases (i.e. different number of simultaneous users) in order to compare the performance.

10.1 Two Instances of m1.xlarge machine (5 phases)

From the generated Tsung report, the mean page and request time for this configuration are both 1.76 seconds. The code of 503 appears at around 130 second after the test, which is a sign of temporary overload of the server due to exceeding requests. Therefore, more instance will be added for a better performance.

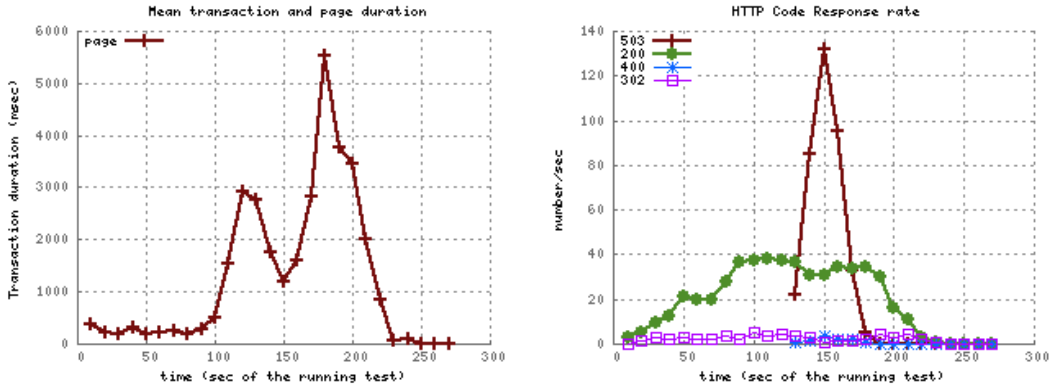


Figure 10.1 Two Instances m1.xlarge result

10.2 Four Instances of m1.xlarge machine (5 phases)

From the generated Tsung report, the mean page and request time for this configuration are both 1.62 seconds. Compared with case shown as Figure 10.1, there is a decrease in the mean time. Since more instances are added, the load of HTTP request is divided up and thus reduced the corresponding mean page and request time.

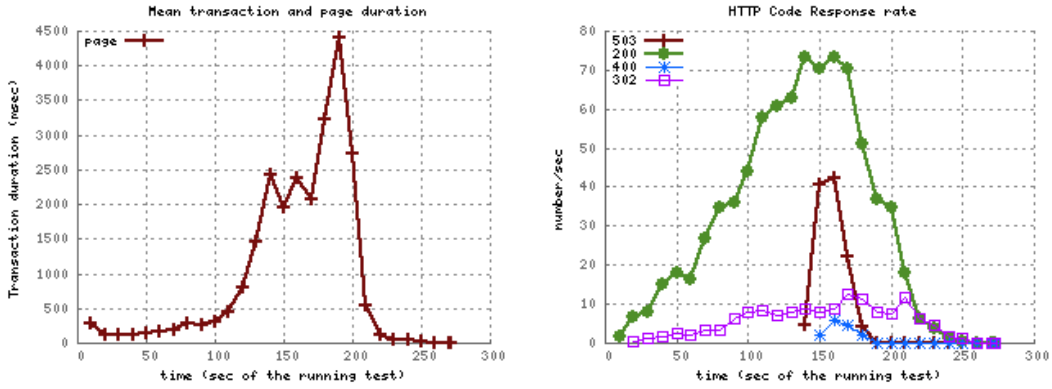


Figure 10.2 Four Instances m1.xlarge result

10.3 Eight Instances of m1.xlarge machine (5 & 7 phases)

From the generated Tsung report, the mean page and request time for this configuration are both 0.52 seconds. Still for the same reason, the load is divided and relocated to more instances, and result in further decreasing in mean page and request time. The sever can handle requests so well that not any code of 503 is shown during the whole test.

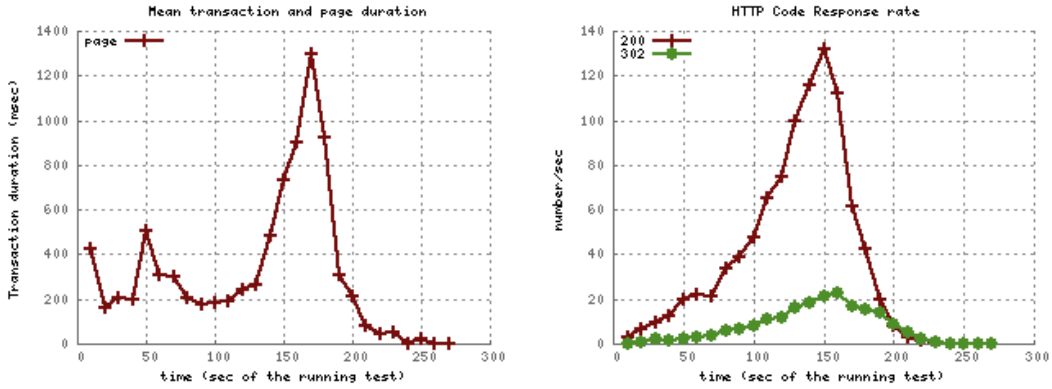


Figure 10.3.1 Eight Instances m1.xlarge result of 5 phases

Since there is no 503 code shown for the case above, now we increase the load by adding 2 more phases in this test. From the generated Tsung report, the mean page and request time for this configuration are both 1.90 seconds, which is significantly increased compared to the last case, since the number of users is about four times as before (i.e. 901 users before and 3712 users for the current case) and also their corresponding number of requests.

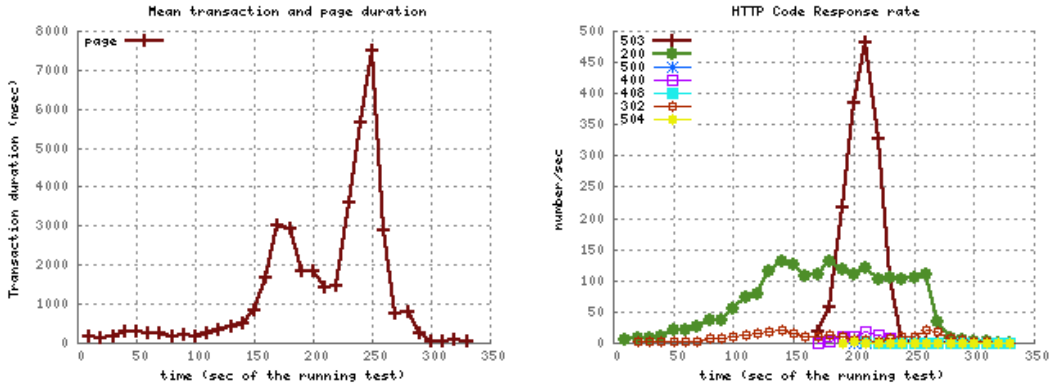


Figure 10.3.2 Eight Instances m1.xlarge result of 7 phases

10.4 Two Instances of m2.4xlarge machine (5 & 7 phases)

Now for a comparison to the case shown as Figure 10.1, we will still use 5 phases. From the generated Tsung report, the mean page and request time for this configuration are both 1.14 seconds, which is shorter than in previous case shown in Figure 10.1 (i.e. 1.76 seconds). This is probably because the initial configuration of the server machine is better than in the set of m1.xlarge instance.

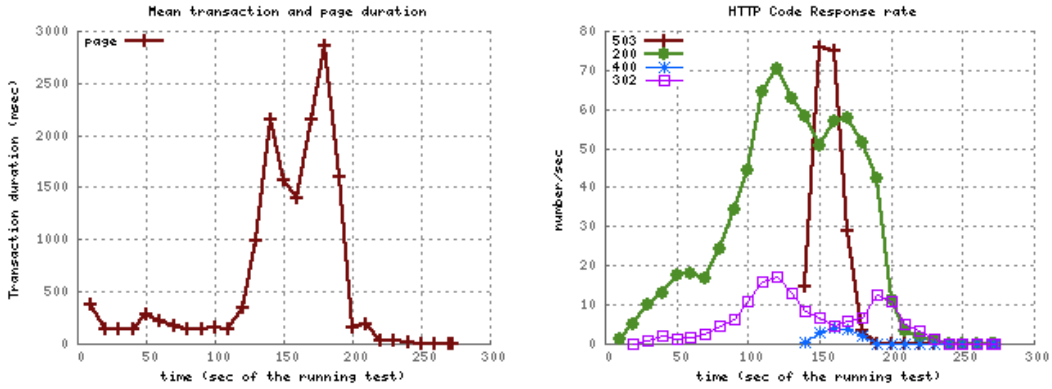


Figure 10.4.1 Two Instances m2.4xlarge result of 5 phases

Since the initial configuration of the instance for m2.xlarge is better, we increase the load by adding two more phases (i.e.7 phases) in the following test. From the generated Tsung report, the mean page and request time for this configuration are both 0.73 seconds. There is a 36% decrease in the mean time for page and request, even with four times users than in above case (i.e. 905 users before and now 3802).

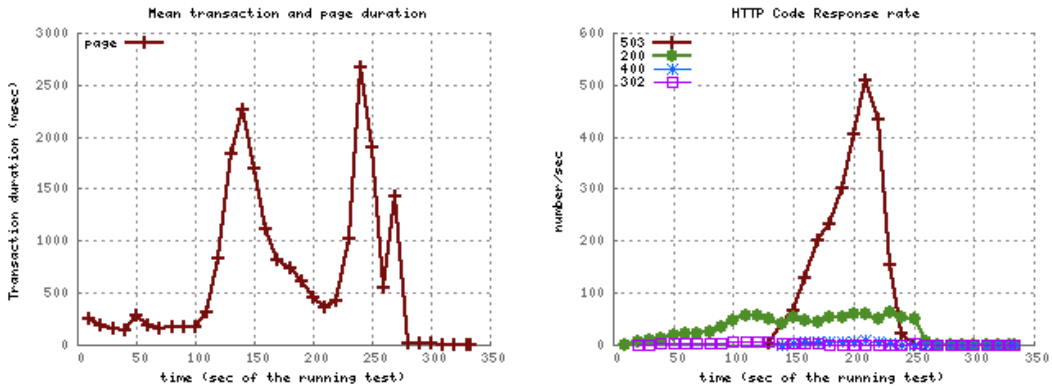


Figure 10.4.2 Two Instances m2.4xlarge result of 7 phases

This is probably because there is more redirection by the elastic load balancer. From Figure 10.4.2, we can see that the 503 code appears around 10 seconds earlier in this case than previous one. Moreover, the number of code 200 decreases significantly, which tells the decrease of average time is at the expense of reducing correct connection (represented by code 200). Thus, in fact, the performance of the system is decreased due to heavy load.

10.5 Four Instances of m2.4xlarge machine (7 phases)

From the generated Tsung report, the mean page and request time for this configuration are both 0.93 seconds, which is longer than the case before due to more redirection by the load balancer. However, the system performance is better since there is more 200 code shown than before.

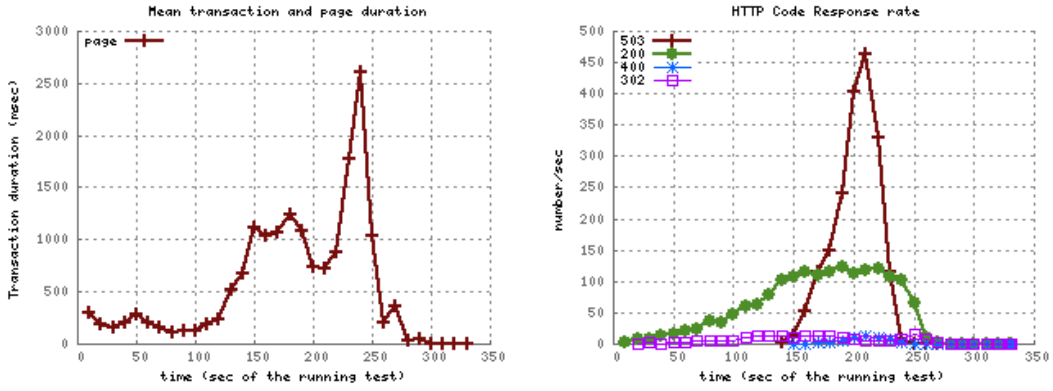


Figure 10.5 Four Instances m2.4xlarge result of 7 phases

10.6 Eight Instances of m2.4xlarge machine (7 phases)

From the generated Tsung report, the mean page and request time for this configuration are both 1.24 seconds. This situation is similar as in case B2, more redirection by the load balancer increases the the mean time of page and request, but better actual performance of the system since there is more code 200 shown.

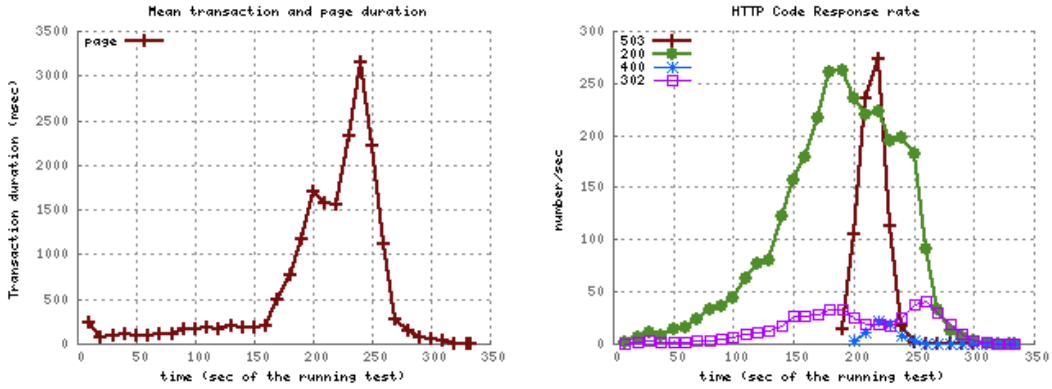


Figure 10.6 Eight Instances m2.4xlarge result of 7 phases

11. Further development

Various optimizations are done during the whole process of development. However, even if the methods that have been used lead to significant improvement, further optimizations are still needed for better user experiences. Configuration optimizations are worth trying. For example, Nginx and Passenger are HTTP server and app server that can handle multiple requests concurrently. They can help the application to schedule HTTP requests and rails processes more efficiently. In addition, sharding is another approach that can further improve the overall performance of the application. It breaks up a single database into multiple databases, thus solved the problem of the single database performance bottleneck.

On the other hand, the development of new features should be continued for the convinence of current users and furthermore, to attract more new users. There are many interesting features we'd like to have included but not limited to "collection", which can be created by a user who wants to make a series of recipes of interest as a personal list, and what's even better is this collection can also be liked, bookmarked and shared; the function of following other users who creates great recipes, which increase the interactions between users; and so on.

For better user experience as a social network point of view, users should be able to comment recipes, masterpieces, and collections. The creators of recipes, masterpieces, and collections can view all comments and reply them or try to improve their own works.

12. Conclusion

Through the course, our team built a scalable recipe sharing platform where users can share recipes, upload masterpieces, and explore recipes from others. Agile development, with the use of git, pivotal tracker and Travis CI facilitates our development a lot. Also, Pair programming and team collaboration inspire us to solve more technical problems .

The path to optimization is not easy for us. Load tests with Tsung and Newrelic help better monitor performance of our application, but it is painful to simulate submitting a form to the server though as many details are involved in a post request. Various optimization methods including server fragment cache, client side cache, sql optimization are implemented in our project. But in practice, it is hard to observe significant improvement and tradeoffs between error rate and response time exist often. From our experiments, horizontal scaling, vertical scaling and fragment cache significantly improve the performance. It is observed that cache works well when the cached results rarely change and expensive to compute, which is not common in our project as most content in our page keeps changing. Client side cache also contributes a lot as it avoid view rendering time. In addition, SQL optimizations, especially adding index , preloading and raw sql can save many sql queries and optimize active record time.