University of Toronto

Faculty of Applied Science and Engineering

CSC326 Programming Languages

Lab 4 Final Report

Team #	38
Prepared By	Niki Mrvelj: 999058629 Andrew Louis: 998101977 Matthew Marji: 998854912
Instructor	Jianwen Zhu
Due Date	December 8, 2015

1. Final Search Engine Design and Evaluation

1.1 Final Design Summary

Prior to planning and implementing the final design for our search engine - Snake Search, our search engine submitted for Lab 3 still varied from the baseline requirements considerably.

Snake Search's front-end is powered by the Bootstrap front-end framework, which was used to create a responsive front-end on mobile, tablet and desktop platforms (Refer to Appendix A, Section 1). In order to ensure that our front-end stayed maintainable and its components - encapsulated, we used Bottle's default templating system, and created reusable components for page layouts, page tables, and navigational elements.

For our final search engine design we decided to take a holistic approach to enhancing it by implementing features and adding components that create a reliable, scalable web application that focuses on increasing the user's experience and productivity.

The components that describe the front-end features of our final design are: multi-word queries, query autocomplete, and a Google inspired "I'm Feeling Lucky" button.

The improvements that describe the performance enhancements portion of our final design are: deploying a load balancer in front of our AWS instances, and setting up status/uptime monitoring on our AWS instances.

1.2 Detailed Descriptions of Front-End Features of Final Design

1.2.1 Multi-Word Queries

To query the inverted index for more than one word, we followed the following steps:

- Set A to be the set of documents that have word #1
- Set B to be the set of documents that have word #2
- Compute the intersection of A and B

And simply modified the way we queried the persisted inverted index, accordingly.

1.2.2 Autocomplete

We used the Twitter Typeahead.js library (and its Bloodhound suggestions engine) in order to enable users to narrow down their search before viewing their query's search results. In order to facilitate this, search results are also queryable in JSON format via the **/api/?keywords=<keywords>** route which the Bloodhound engine used to generate suggestions.

1.3 Detailed Descriptions of Back-End Features of Final Design

1.3.1 Load Balanced EC2 Instances

A load balancer was placed in front of multiple EC2 instances that are running Snake Search. A load balancer in short, is a service that distributes network traffic across a number of servers (Refer to Appendix B, Section 1). This is used to increase the number of concurrent users and fault-tolerance of the application. Prior to load balancing our EC2 instance, we were not able to have more than 100 concurrent connections. Providing a load balancer alone allowed us to see an increase in users of up to 3000 concurrent connections.

As seen in Appendix B, Section 2, the load balancer (blue circle) is the access point at which all users access Snake Search. The load balancer will determine which instance is of less load and forward the request to that specific instance. Our load balancer has been set up with the same security group, and settings as our EC2 instance and for reliability runs tests every 5 minutes to determine the health of all EC2 instances it is currently connecting to. For our setup, we have **two** EC2 instances running Snake Search on the T2.micro free platform, with the ability to very quickly add additional instances with our deployment scripts.

1.3.2 Status/Uptime Monitoring

We use a third-party service StatusCake (Refer to Appendix B, Section 3 for a visual of our monitoring dashboard) to continuously ensure that our load balancer and instances remain online and intact. The image in the appendix shows the average response times for requests to the load balancer which on average is approximately 300ms. This service will also notify us if there seems to be a large number of concurrent connections slowing down the response times for requests or if the load balancer is not responding to requests. We will then be notified and take manual action.

1.4 Alternate Design and Evaluation

1.4.1 Alternate Design Detailed Description

Our alternate design's features are listed as follows:

Front-End:

- Refactor Using Tornado
- Multi-Word Search
- Search Bar Expression Evaluator (E.g. > 2+2 returns 4)

Back-End:

- Setting Up Load Balancer

The merits of this alternate design over our primary design were that we would be able to get practice pipelining using iterators and generators by doing a refactor with Tornado. The expression evaluator

would increase productivity and user experience by making the user interaction resemble that of Google's, thereby inducing a sense of familiarity.

1.4.2 Final Design Evaluation Process

The key metric used to decide on our final design was the time and effort it would take to implement a particular design to completion.

Instead of tallying up the time estimates for each design to quantify time and effort, we decided to use Agile Story Points using Fibonacci numbers as a normalizing scheme.

As a group, we held a vote for each task in each design, each vote being a Fibonacci number (up to 13 which indicates finding the task superlatively challenging in terms of time and/or effort), and then decided on a vote for each task following a discussion if there were any vote variances.

The outcome of this evaluation is described in Appendix C, and we decided to implement the feature set we were most confident in fully producing (as quantified by the lower scoring design alternative in Appendix C).

2. Differences Between Final Design and Final Search Engine

Two additional features were implemented that were not part of our final design feature set; namely: multithreaded request handling, and an easter-egg.

The team was able to implement the following two features using remaining time once our assigned tasks were completed.

In addition, our projection for the concurrency capacity of our search engine with a load balancer and multiple instances alone was greatly overestimated. The investigation into increasing concurrency lead to the inclusion of multithreaded request handling which wasn't a part of our original final design.

2.1. Multithreaded Request Handling

The default Bottle server is a simple single-threaded WSGI server. Request handling is severely limited in its ability to serve concurrent requests by the single-threaded nature of the default server. Each request is a blocking operation which blocks other waiting requests waiting to be processed until the current request is handled. Running the application using the **Gevent** server creates a micro-thread out of every connection, and schedules execution time slices among these threads. The number of concurrent connections is then essentially limited by the size of the server's memory.

Tests run using the Apache Benchmarking tool (Refer to Appendix D) indicated that we were able to handle 23 times more concurrent requests than the default WSGI server (saw an increase from being able to execute 1000 requests across 80 concurrent connections with the default server to being able

to execute 3000 requests across 3000 connections). Note, that with the addition of the load balancer, we are now able to handle over **8000 concurrent connections**. See Appendix D, image two. This increase due to the multithreaded request handling as well as the load balancer increased the ability for handling concurrent requests 800x the ability our single instance had after lab two!

We planned on increasing concurrency by using a load balancer alone; however, the combination of non-blocking request handling and the load-balancing approach yields a much more stress tolerant application than applying either of the approaches. The most impressive of this is that we were able to take full advantage of a free platform to be able to create such a well performing, fault tolerant web application.

2.2. Easter Egg

We added an easter egg to the site that maintains the Snake Search theme. To preview the easter egg, click on the animated logo (Alternately, refer to Appendix A, Section 2).

This launches a launches an iframe that serves up the contents of the **/secret** route - yielding a snake themed surprise.

3. Project Testing

There was an emphasis on code testing within the course, assignments, and labs. So for each lab, there was testing done for frontend and backend to ensure correct implementation. Unit tests were written for each new lab. Whenever a new feature or data structure was created, unit tests were created first to ensure that the features when implemented worked correctly. These unit tests also worked in our favour because when introducing new features it ensured that basic functionality from the previous labs worked correctly. The unit test cases written tested both the functionality as well as the database values to ensure that values returned to the front end were accurate. Aside from unit tests, back-end code was always peer reviewed by another developer before it was submitted. Specifically for frontend, it was running “mock searches” and comparing search engine results to expected output and ensuring that result tables were portrayed accurately and queries executed correctly.

4. Lessons Learned

Some of the lessons learned during this project were the benefits of peer code review and time management. Throughout university - project after project, there has always been the lesson of time management. While one would think that at some point you can conquer the lesson of time management, this project proved to be another lesson for time management.

Meeting the challenge of putting out an iteration of Snake Search that went further than simply carrying out the basic requirements set out in the assignments - in the short spans we had to work on the labs always led to a crunch, and reinforced our need to manage our time effectively as a team. This project helped teach us to manage time effectively by breaking down a large project and dividing

it into smaller bite-sized components, and to ensure that we all carried out our team obligations such as team meetings, helping team-members, and of course - finishing our delegated tasks.

A couple of us have had experience in software development at firms of different sizes, what was the same at all places was the importance of peer code reviews, and the use of an online repository. We decide to follow many of these principles which was very beneficial to us and taught us first hand why many successful firms follow these practices. The front-end code and back-end code were both created and stored on Github. Whenever a commit with a new feature or bug fix was created, it was created in a separate branch. At that point, another member of the team reviewed the code, and if the code seemed to work well it was merged into the master branch of the project. If any issues were found (which often times bugs were found), an issue was created on Github and referenced in the pull request. Once the issue was solved, and no other bugs were found, the branch was merged into master. Following this rigid process taught us how important reviewing code is, because it is important that not only bugs are found, but that others see your code and can understand what it does so that if they ever need to access it, they have reviewed it before!

5. What the Team Would Do Differently

If the team had to redo the project over again, an interesting and different route to take would be to incorporate more of the other members in both sides (backend and frontend) of the project. For the first three labs, work was split up between frontend and backend to ensure work completion. Unfortunately, this split between workloads minimized the exposure each team member had on the opposite workload (frontend isn't familiar with backend work etc). If there was more time for each lab, it would have been nice to perfect and optimize all processes at each step instead of rolling it into lab 4. It would have good to use each prior lab as a stepping stone for the following one with a strong foundation. Due to time constraints, the first two labs were done just to course specifications with much of the work being redone in order to implement enhancements in the final lab. Some parts that took longer than initially thought were backend testing and frontend bottle implementation. There were many exceptions and unique cases that Matthew wanted to test the backend for to ensure there would be no fail cases. For the frontend there was the problem of learning botte.py for session management and implementing this. The team did not initially think this was going to be as new of a topic as it turned out to be.

6. Course Material and Relevance

While doing the lab, the team had to refer to class notes often for rough guidance and outlines. The majority of the backend structure was covered well in the course. Setting up the database as done in class was very helpful. There were no issues with the initial setup. What lacked in class, was the understanding of the SQL language and querying the database for the correct information. Although we have experience in SQL, for those who did not, this may have been a difficult task. The class material proved to be very useful in setting up the databases/result display for the front end but what seemed to be lacking was the direction for navigating Bottle and HTML. Although, HTML is fairly easy to learn; perhaps for some other students who have much less exposure it proved to be an additional difficult task. Setting up Bottle was fairly difficult as there was no direction/knowledge of routes and

wrappers at this point in the course- so you were expected to learn and use material that would be presented in the class much later on.

Another part that was very useful was the lab sections about deploying to cloud services.

7. Time Taken to Complete Labs

Our group consisted of Matthew Marji and Niki Mrvelj for the first two labs and Andrew Louis joined for Lab 3 and Lab 4. The front-end tasks took approximately 3-4 hours for each lab outside of lab sections, while the back-end warranted 5-6 hours of work outside of lab hours. Lab 4, with little instruction was open to many hours of work as we wanted to really move from the basic requirements laid out in Lab 3, to an intuitive search engine with great performance and optimized code.

8. Nonessential Material and Emphasized Material

A part that was useful and the labs should spend more time was deploying to the cloud such as Amazon's EC2, Heroku etc which was extremely important. Understanding performance issues such as load and load balancers was extremely useful as in present day, code is always being run online to some degree. This is useful for students to understand how the real life coding works. For the frontend, it was really interesting and useful to use APIs. The team has integrated APIs before in various projects and agree that it is extremely useful for hosting data structures and web services. This is something that should be emphasized more. The team believes that nothing in the project was useless as everything played a role towards the end product.

9. Other Feedback and Course Recommendations

A key piece of feedback for the course is releasing the lab and assignments in a timely manner. Many students felt cramped for time as labs and assignments were released only days before the due date. This raised a lot of stress and anxiety as many scrambled to finish the work due on top of coordinating their other coursework, capstone work, and external schedules. A big recommendation would be to release the labs well in advance of due dates (same with assignments) to ensure that students have time to complete it with satisfaction and confidence in their submitted work.

10. Responsibility Breakdown

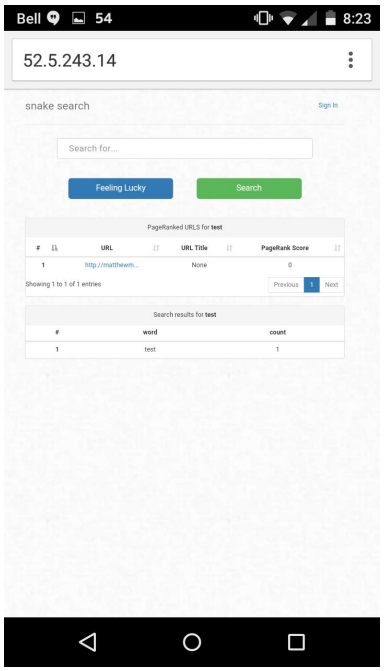
The responsibilities of each member are listed below:

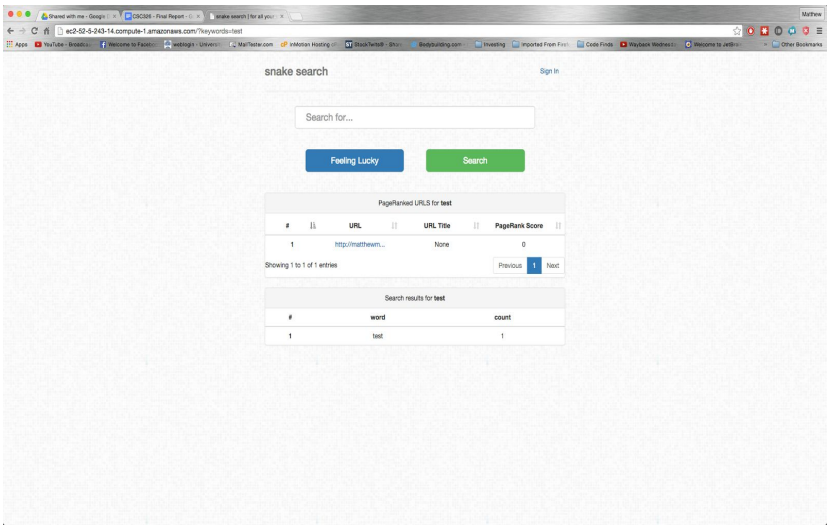
1. Matthew Marji
    a. Backend Implementation (Lab 1)
    b. Backend Implementation (Lab 2)
    c. Backend Implementation (Lab 3)
    d. Backend Implementation and Enhancements (Lab 4)
        i. EC2 Load Balancer setup

        ii. Easter Egg Implementation
    e. Lab 4 Terminator/Creator Scripts
    f. Portions of lab report
2. Niki Mrvelj
    a. Frontend Implementation (Lab 1)
    b. Frontend Implementation (Lab 2)
    c. Frontend Implementation (Lab 3) - Error Page handling, Pagination
    d. Lab Report
3. Andrew Louis
    a. Frontend Refactor to Include Bootstrap and Templating (Lab 3)
    b. Frontend Implementation (Lab 3) - Querying Backend
    c. Frontend Implementation and enhancements (Lab 4)
        i. Multi-word queries
        ii. Query Autocomplete
        iii. I'm Feeling Lucky
        iv. Multi-threaded Request Handling
    d. Frontend Testing
    e. Lab Report

Appendix A: Front-End Features

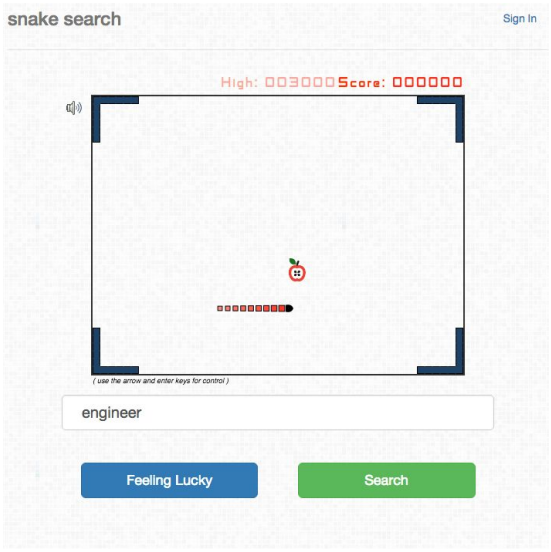
Section 1: Responsive Templated Layouts





The figure above shows the snake search engine front-end display through different platforms (tablet, mobile, and desktop) with a mobile display on the left and desktop display on the right.

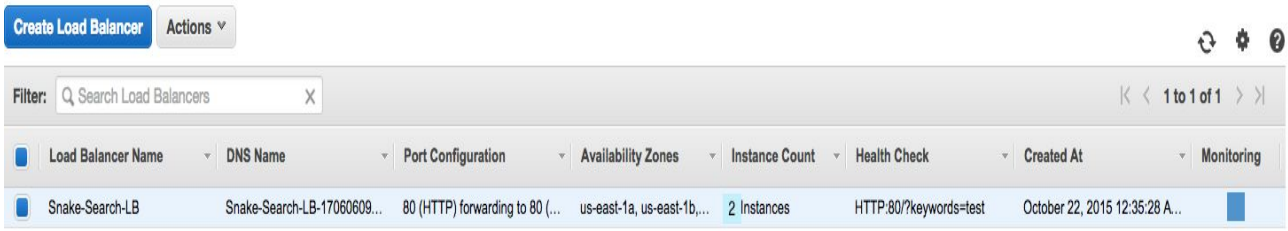
Section 2: Easter Egg Preview



The figure above shows the easter egg within the home page. This launches an iframe yielding the snake themed surprise.

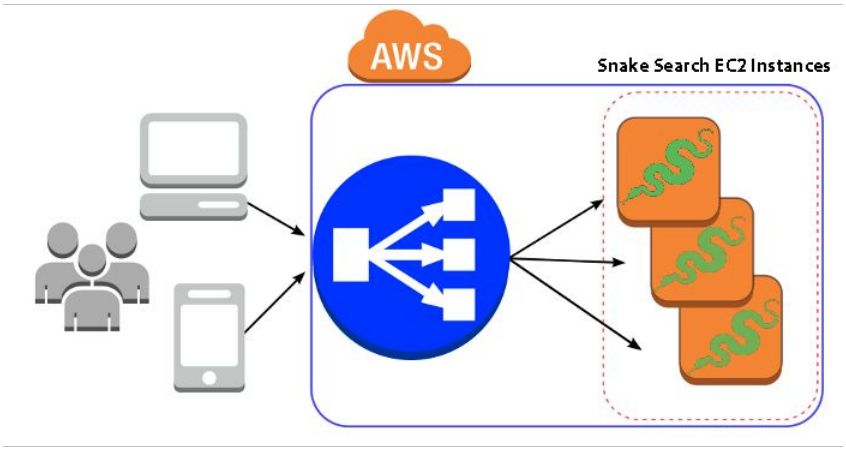
Appendix B: Back-End Features

Section 1: Load Balancer Dashboard



The figure above shows a screenshot of the load balancer used to distribute network traffic across our server. It shows the load balancer name, DHS Name, Port configuration, Availability Zones, Instance counts, and Health check.

Section 2: Load Balancer UML



The figure above is a representation as to how the Snake Search engine operates. The load balancer (blue circle) is the access point of all users at which all users access Snake Search. Load balancer determines the instance with least amount of load and forwards requests to that EC2 instance.

Section 3: Status Monitoring

The figure above is a dashboard snapshot of the third-party service StatusCake used for visual monitoring ensuring that the load balancer and instances remain online and intact.

Appendix C: Agile Story Points Outcome

Alternative Design	Score	**Selected Design**	Score
Refactor using Tornado	13	Multi-Word Search	5
Multi-Word Search	5	Query Autocomplete	5
Expression Evaluator	5	I'm Feeling Lucky	3
Setting Up Load Balancer	5	Setting Up Load Balancer	5
		Uptime Monitoring	5
Total	**28**		**23**

Shown in the point table above is the agile story points outcome between the selected search engine design and the alternative design. These points were fibonacci numbers (up to 13) representing design difficulty. The selected design was chosen based on the lower total outcome.

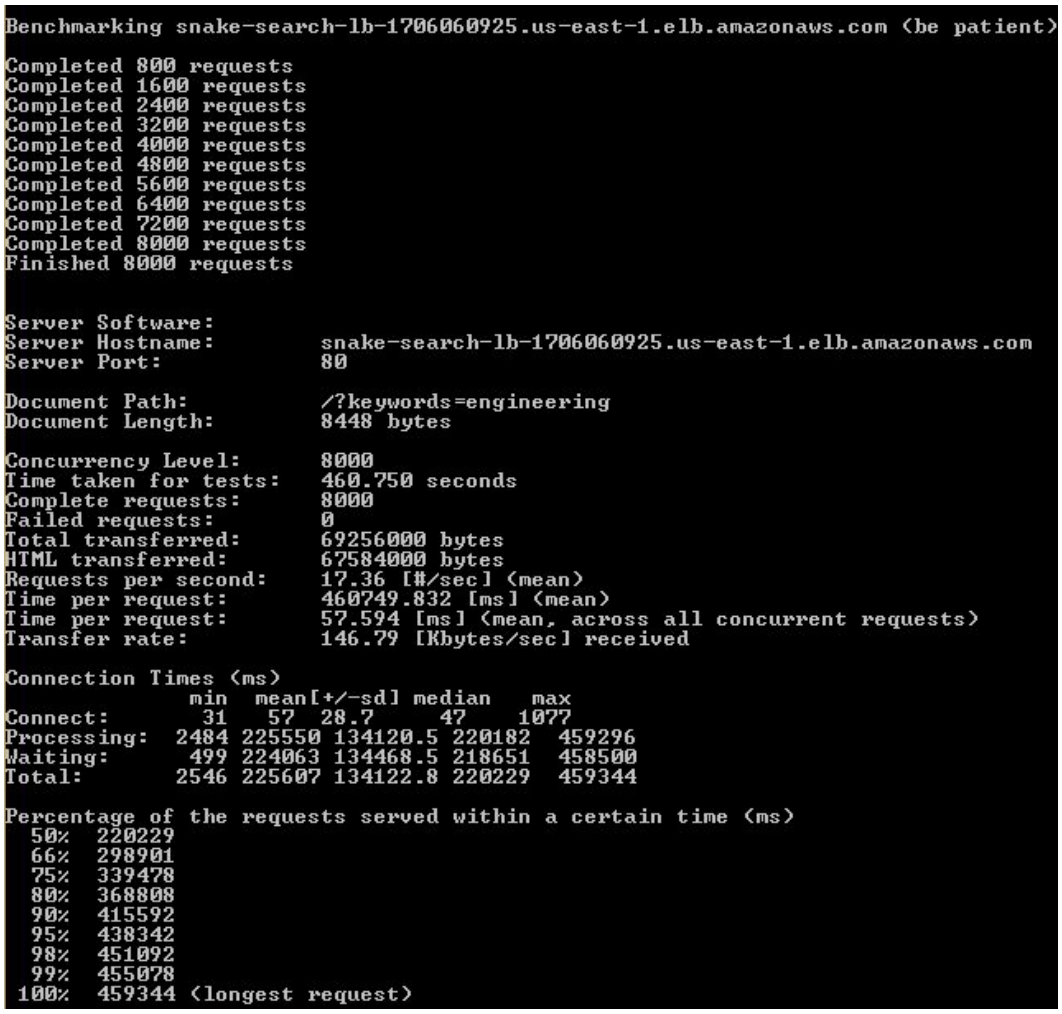
Appendix D: Apache Benchmarking Tool Results

```
Server Port:            80

Document Path:          /?keywords=test
Document Length:        4855 bytes

Concurrency Level:      3000
Time taken for tests:   137.363 seconds
Complete requests:      3000
Failed requests:        0
Total transferred:      15195000 bytes
HTML transferred:       14565000 bytes
Requests per second:    21.84 [#/sec] (mean)
Time per request:       137363.298 [ms] (mean)
Time per request:       45.788 [ms] (mean, across all concurrent requests)
Transfer rate:          108.03 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       28   45   8.4     47     157
Processing:   765 68866 39282.0  68673  136709
Waiting:      172 68473 39434.3  68268  136597
Total:        813 68911 39281.9  68721  136758
```

The above snapshot shows the tests run using the Apache Benchmarking tool indicating that the search engine was able to handle 23 times more concurrent requests than the default WSGI server.

The screenshot below shows the tests run using the Apache Benchmarking tool on the load balancer that has been setup. We were able to handle **8000 concurrent requests** with absolutely no failed connections or timeouts!

We likely could have handled more requests but our test machine used for benchmarking seemed to crash when testing 9000 or more concurrent requests so we could not fully test the abilities of our machine. It is likely that our setup could handle close to 10,000 concurrent requests.



```
Benchmarking snake-search-lb-1706060925.us-east-1.elb.amazonaws.com (be patient)

Completed 800 requests
Completed 1600 requests
Completed 2400 requests
Completed 3200 requests
Completed 4000 requests
Completed 4800 requests
Completed 5600 requests
Completed 6400 requests
Completed 7200 requests
Completed 8000 requests
Finished 8000 requests


Server Software:
Server Hostname:        snake-search-lb-1706060925.us-east-1.elb.amazonaws.com
Server Port:            80

Document Path:          /?keywords=engineering
Document Length:        8448 bytes

Concurrency Level:      8000
Time taken for tests:   460.750 seconds
Complete requests:      8000
Failed requests:        0
Total transferred:      69256000 bytes
HTML transferred:       67584000 bytes
Requests per second:    17.36 [#/sec] (mean)
Time per request:       460749.832 [ms] (mean)
Time per request:       57.594 [ms] (mean, across all concurrent requests)
Transfer rate:          146.79 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       31   57  28.7     47    1077
Processing:  2484 225550 134120.5 220182  459296
Waiting:      499 224063 134468.5 218651  458500
Total:       2546 225607 134122.8 220229  459344

Percentage of the requests served within a certain time (ms)
  50%  220229
  66%  298901
  75%  339478
  80%  368808
  90%  415592
  95%  438342
  98%  451092
  99%  455078
 100%  459344 (longest request)
```