Reddit Comment Search

CMPE 48A Cloud Computing

Term Project Report

Fall 2022

Leyla Yayladere - 2018400216

Nurlan Dadashov - 2019400300

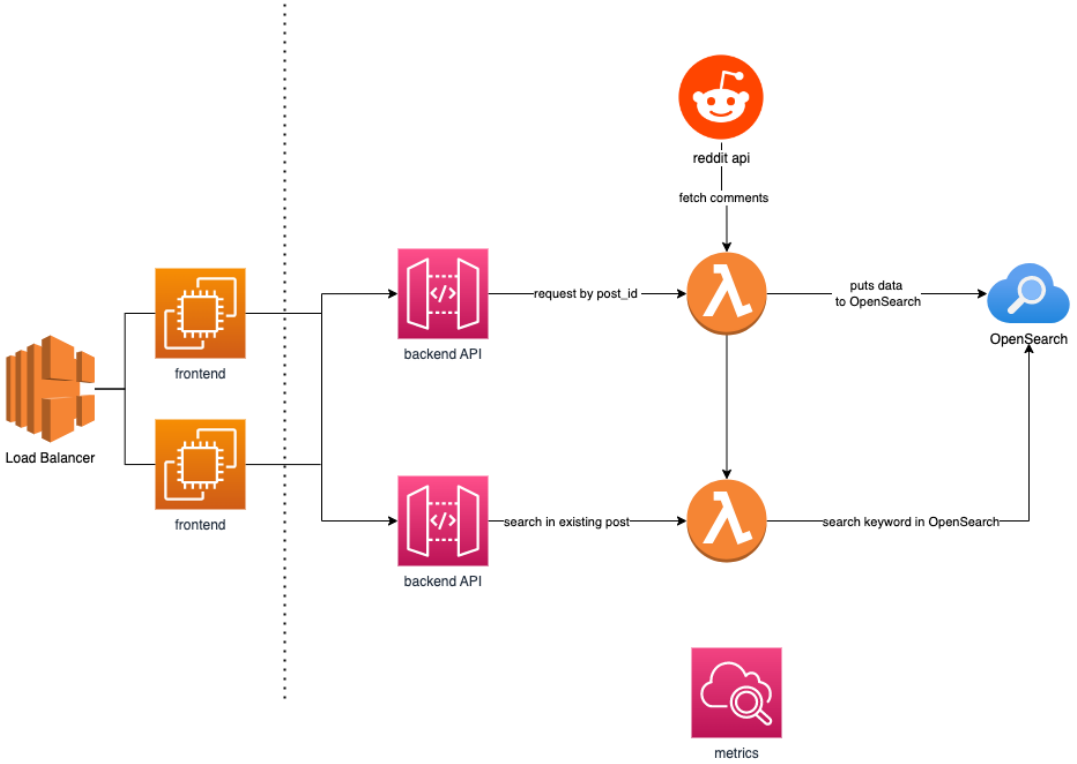
Table of Contents

Application

Our application allows users to search for specific keywords within the comments of a Reddit post. To accomplish this, we use a Reddit bot account and the Python Reddit API Wrapper (PRAW) library to fetch the comments for a given Reddit post ID. These comments are then stored in a search database, with the Reddit post ID serving as the key. When a user makes a request and provides a keyword, our application searches the database for comments containing that keyword and returns the relevant results to the user. This functionality allows users to easily locate and review specific comments within a Reddit post, helping them to quickly find the information they need.

Architecture Design

Our application is designed to utilize the cloud services offered by Amazon Web Services (AWS). We chose to use AWS due to our prior experience and familiarity with the platform.



Backend:

The architecture of our application consists of two separate **Lambda functions**. The first function is responsible for fetching the comments of a Reddit post and storing them in an **AWS OpenSearch** database, using the Reddit post ID as the key. The second function handles searching for a specified keyword within the comments stored in the database.

To enable searching for keywords within previously searched posts, we have implemented an endpoint that triggers the second Lambda function. This allows users to search for keywords within previously searched posts without the need to fetch the comments again, improving the response time of the application. By splitting these tasks into separate Lambda functions, we are able to perform performance testing in different scenarios and optimize the overall performance of the application.

The **AWS API Gateway** acts as a reverse proxy, allowing the frontend of our application to make requests to the backend without the need to directly communicate with the backend servers. This allows us to decouple the frontend and backend of our application, making it easier to maintain and update each component independently. This separation allowed us to work on the frontend and backend concurrently. The frontend of our application was developed in one account, while the backend, including the Lambda functions and OpenSearch database, was developed in another account.

In addition to the functionalities provided by our application, we made use of **AWS CloudWatch** to monitor the logs and key metrics of our service. This tool proved to be invaluable during the debugging process, as it provided us with visibility into the workings of our system.

Frontend:

Our application's frontend is served by two **Amazon Elastic Compute Cloud (EC2)** instances configured as web servers. Our web page features a form that allows users to enter a Reddit post ID and a keyword to search for within the comments of the post. The form includes two submission buttons, one for each of the API Gateway endpoints that trigger the relevant Lambda functions in the backend. When a user submits a request, the search results are returned to the user as a pop-up window, with the keyword highlighted within the text of the comments.

To improve the availability and performance of our frontend, we have implemented a **network load balancer** in front of the EC2 instances. The load balancer distributes incoming traffic across the two instances, helping to ensure that the frontend remains available and responsive to user requests.

Since the concurrency limit of the Lambda functions in our backend represents a bottleneck in the system, we have not configured the load balancer to use autoscaling. This is because autoscaling would result in the creation of additional EC2 instances, which would not necessarily improve the performance of our application. However, we have found the AWS load balancer service to be easy to use and configure for various purposes, such as network or application load balancing. We have also found it easy to bind server instances to a load balancer using target groups, which allow us to monitor the status of each instance and ensure that only healthy instances receive traffic.

Free Tier Limitations

Initially, we had planned to incorporate sentiment analysis for the comments of a given Reddit post into our application. However, after discussing this with our instructor, we determined that the free tier usage limits of the AWS Comprehend service would not be sufficient for our term project. This was due to our desire to conduct performance testing with a large volume of requests to the application.

As a result of this limitation, we had to modify the design of our application to make use of the AWS OpenSearch service for search functionality instead. One advantage of using OpenSearch is that it has free tier usage limits based on hours of usage, rather than on the number of requests made. This made it a more viable option for our project, as it allowed us to conduct performance testing without incurring additional costs.

By carefully considering the limitations of the various AWS services we used and finding solutions to them before beginning implementation, we were able to successfully deploy and test our application while staying within the bounds of the free tier usage limits. However, the free tier limitations did impose a small maximum user capacity on our service.

Below, we have provided a summary of the free tier limitations of the AWS services utilized in our application.

- **AWS Lambda:** 1 million free requests per month and up to 3.2 million seconds of compute time per month. The concurrency limit per AWS Region is 10 invocations at any given time. Lambda function's maximum invocation timeout limit is 30 seconds.
- **AWS OpenSearch:** 750 hours per month of a single-AZ t2.small.search or t3.small.search instance and 10GB per month of optional EBS storage (Magnetic or General Purpose)
- **AWS API Gateway:** 1 Million API Calls Received per month. API Gateway REST API's default maximum integration timeout is 29 seconds.
- **AWS EC2:** 750 hours of Linux or Windows t2.micro instances per month
- **Elastic Load Balancing:** 750 Hours per month shared between Classic and Application load balancers

Performance Testing

During our performance testing, we found it helpful to compare multiple graphs with different test configurations in order to gain a better understanding of the behaviors of our system.

For example, we requested a small increase in the concurrency limit of our Lambda function from 10 to 15 in order to observe the impact of this configuration change on the performance of our system. We also compared the performance of our application when calling the API

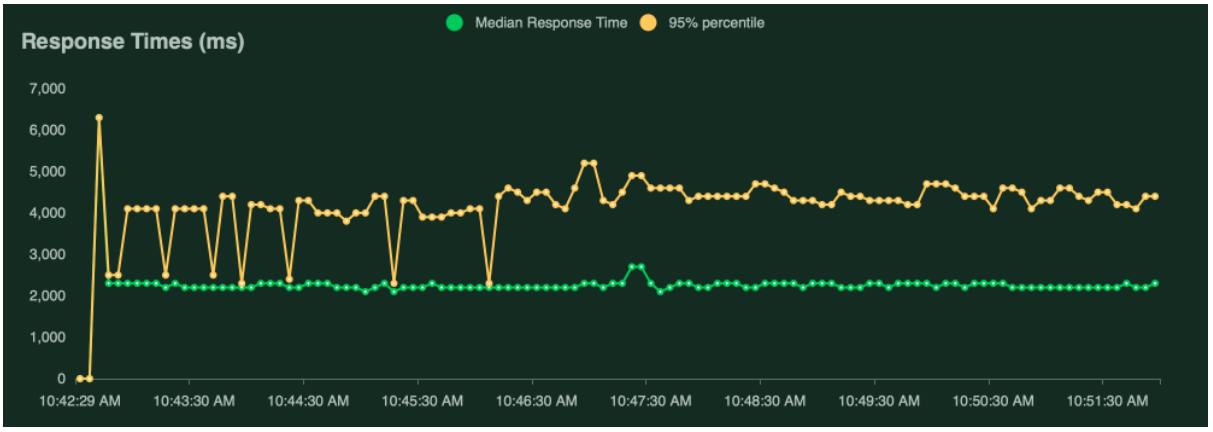
Gateway endpoints with small and large Reddit post IDs, as the size of the post can affect the performance of our service. Additionally, we took into consideration the differences in response time and maximum possible user capacity between the two different endpoints in our application. These differences are due to the design and functionality of the endpoints, and we created test scenarios accordingly to reflect these factors.

By comparing these various configurations and scenarios, we were able to gain insights into the factors that affect the performance of our application and identify opportunities for improvement.

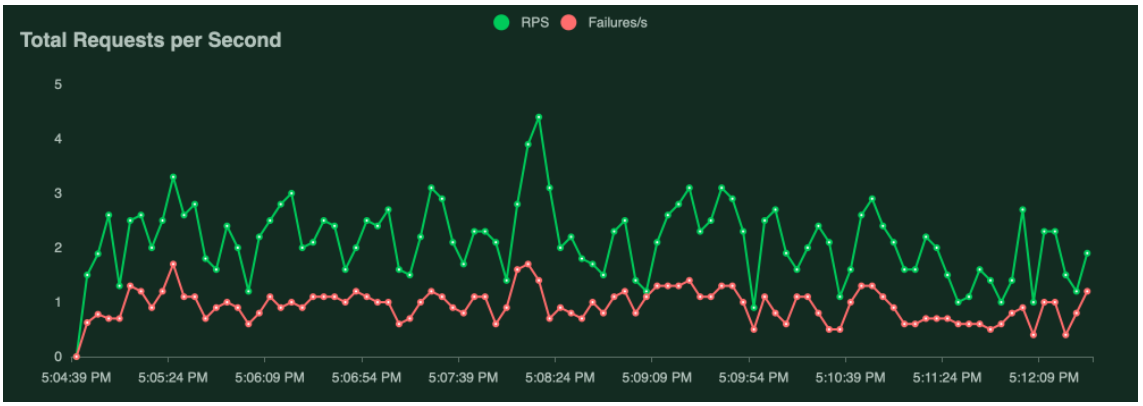
To evaluate the performance of our application, we used **Locust**, a Python-based testing tool that is commonly used for load testing and simulating user behavior. One of the key benefits of using Locust is that it allows us to generate a range of graphs and metrics based on the results of our tests. These graphs can provide valuable insights into the performance of our application under different load conditions.

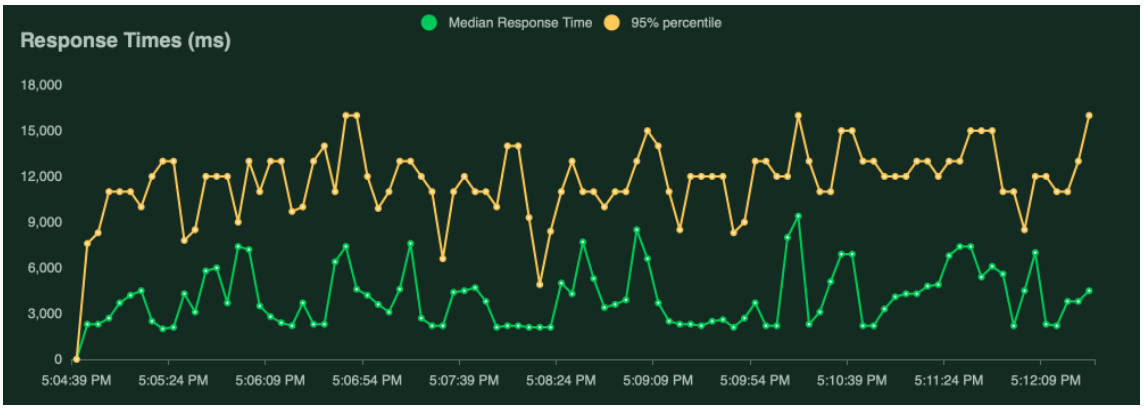
1. comment fetching and keyword searching with small sized post:

1.1. with 5 users and the concurrency limit of 10

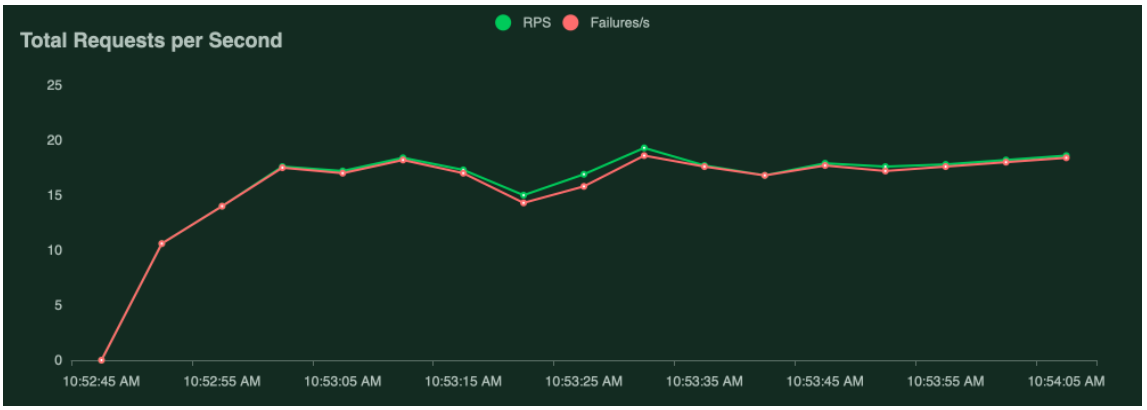


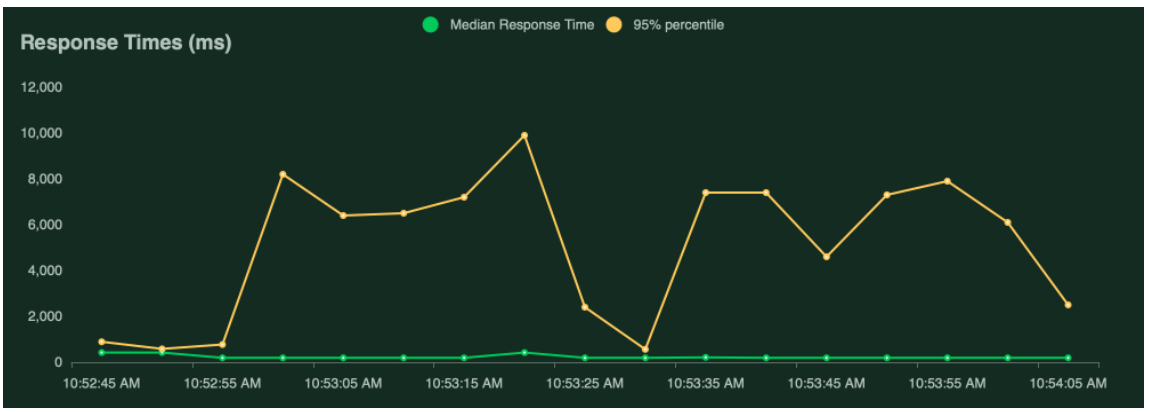
1.2. with 10 users and the concurrency limit of 10



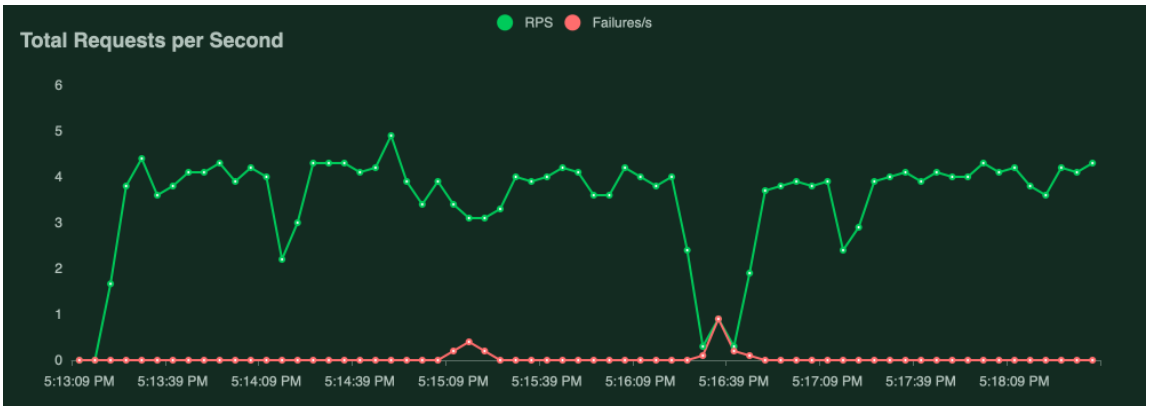


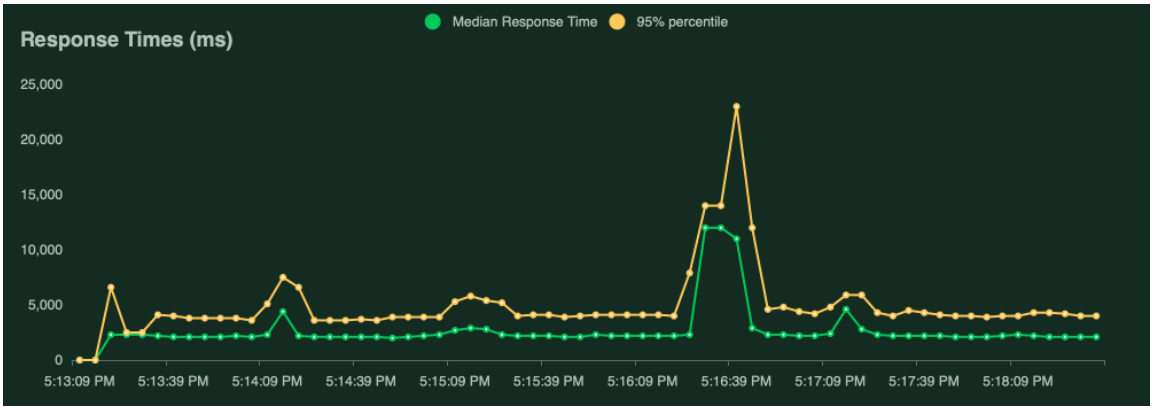
1.3. with 15 users and the concurrency limit of 10



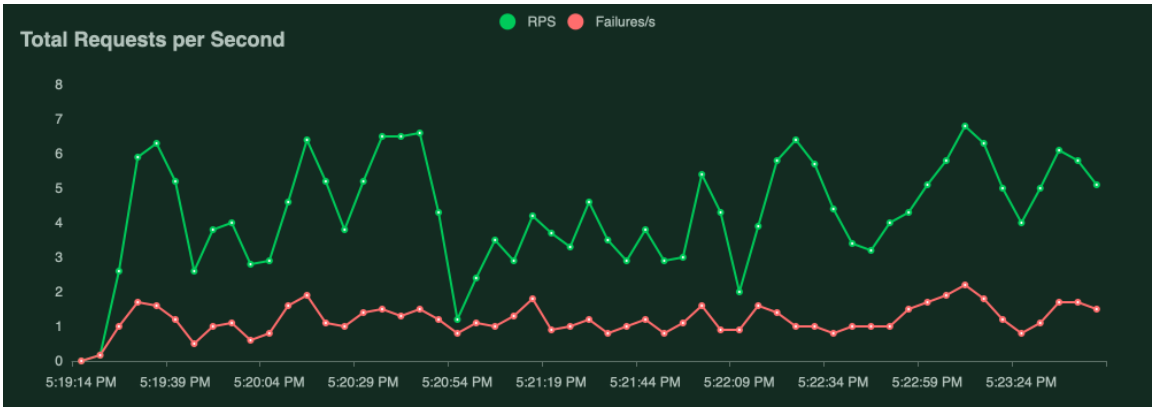


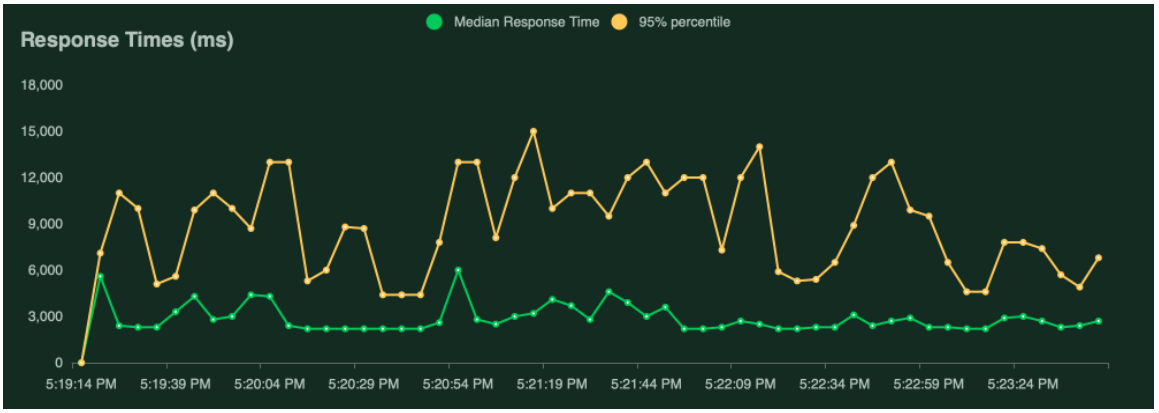
1.4. with 10 users and the concurrency limit of 15



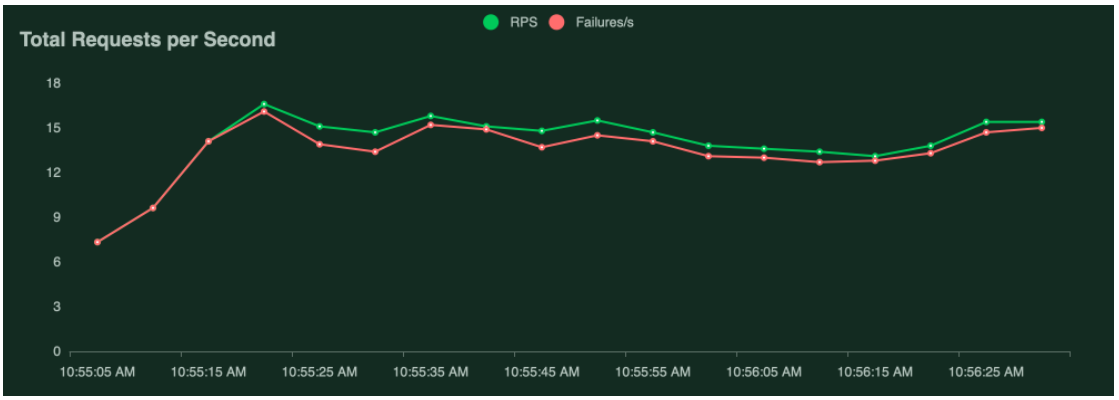


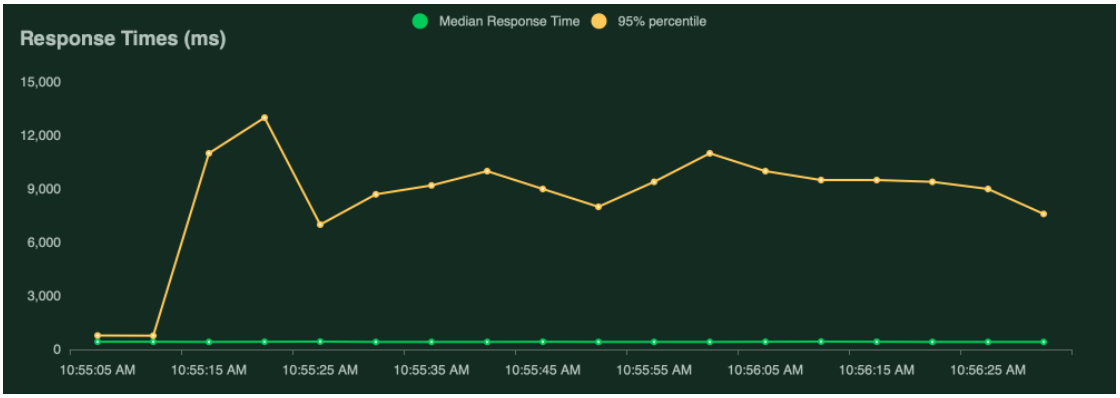
1.5. with 15 users and the concurrency limit of 15





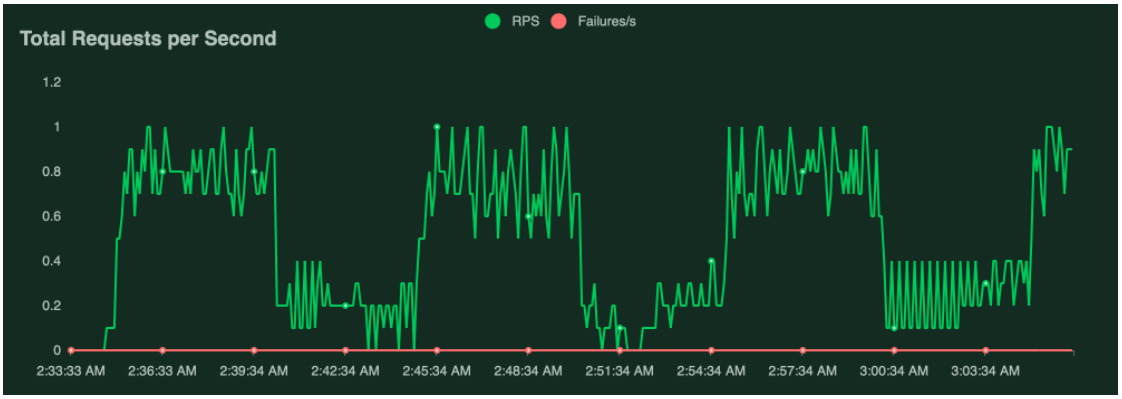
1.6. with 20 users and the concurrency limit of 15

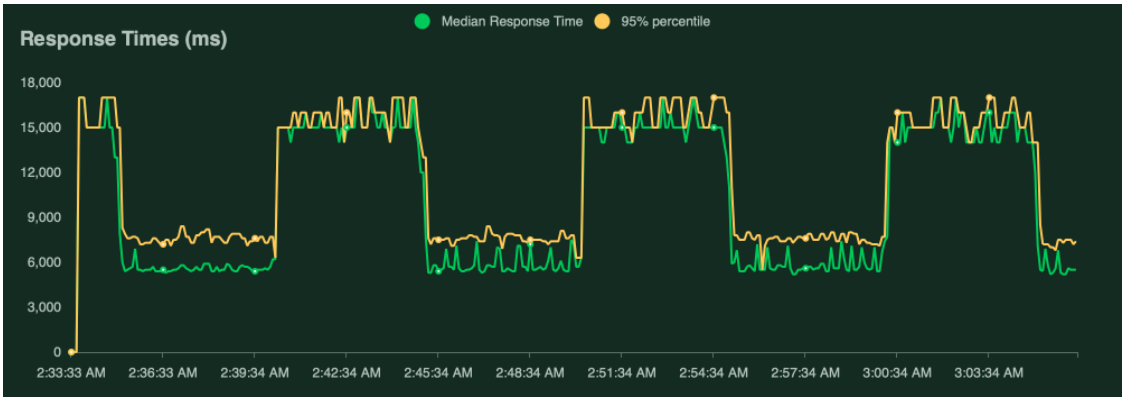




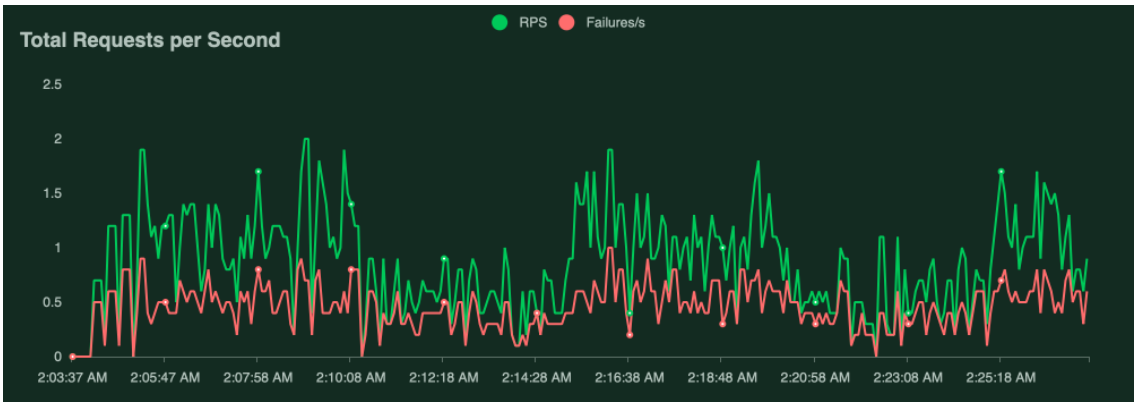
2. comment fetching and keyword searching with big sized post:

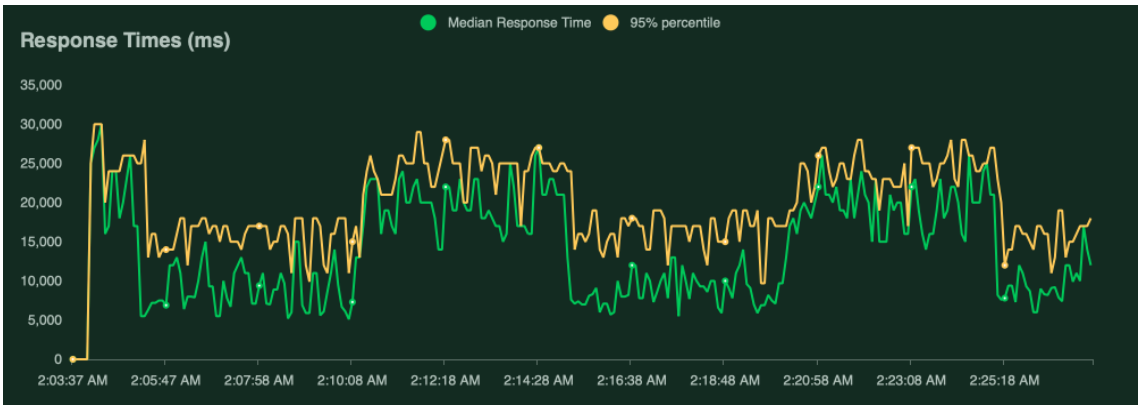
2.1. with 5 users and the concurrency limit of 10



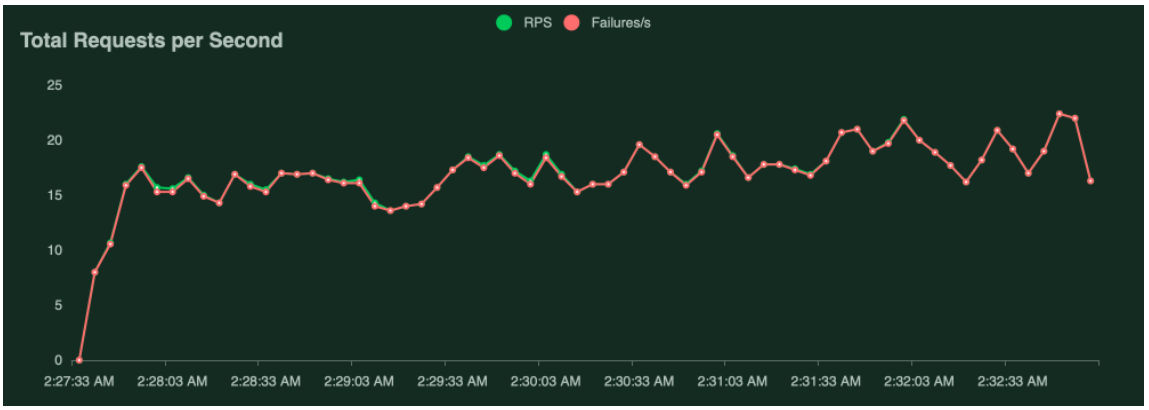


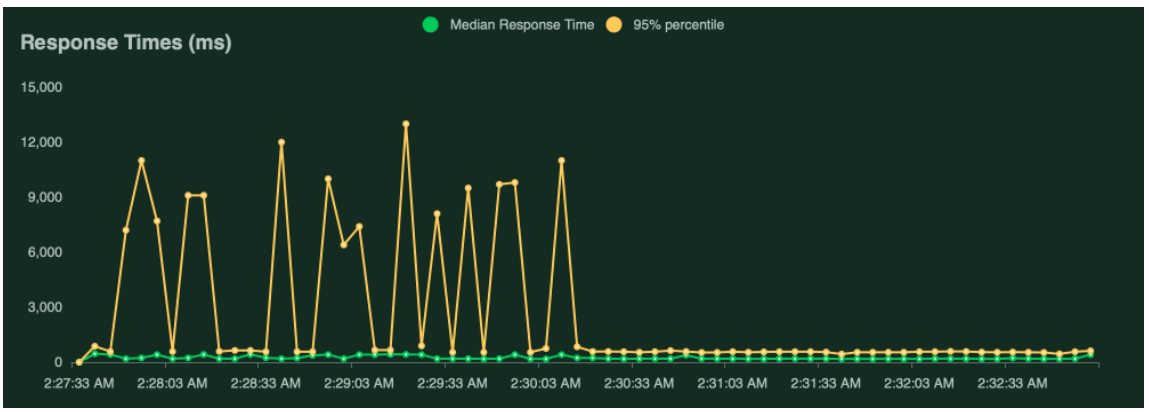
2.2. with 10 users and the concurrency limit of 10



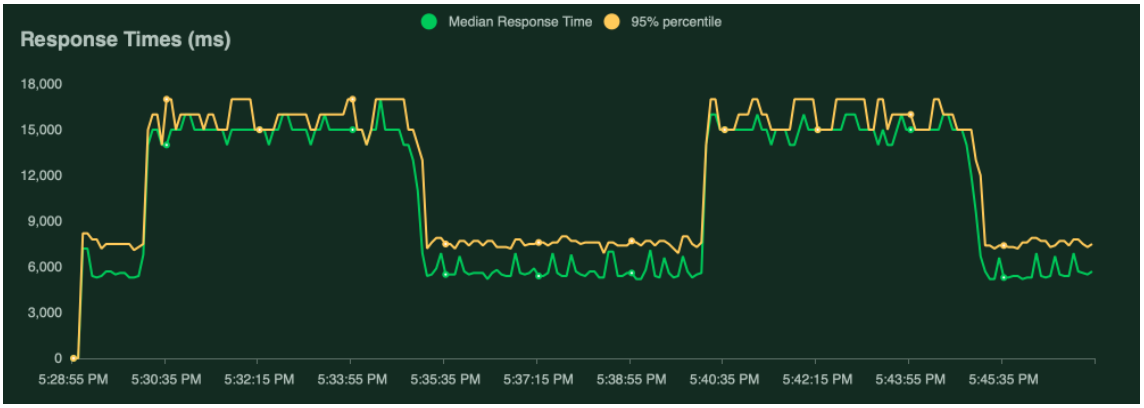


2.3. with 15 users and the concurrency limit of 10

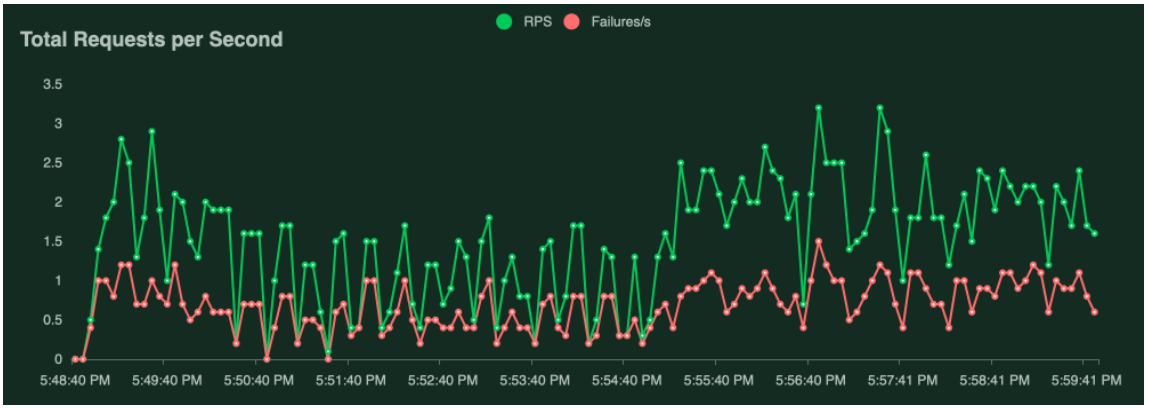


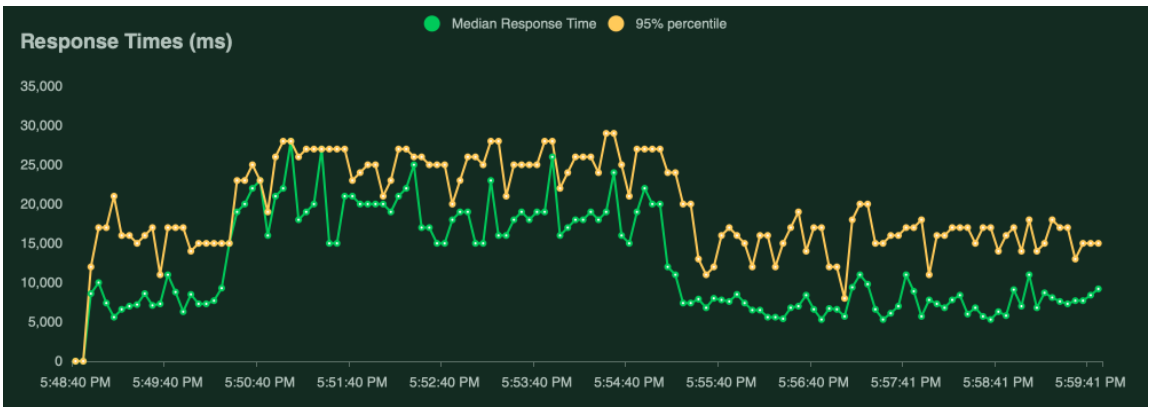


2.4. with 10 users and the concurrency limit of 15



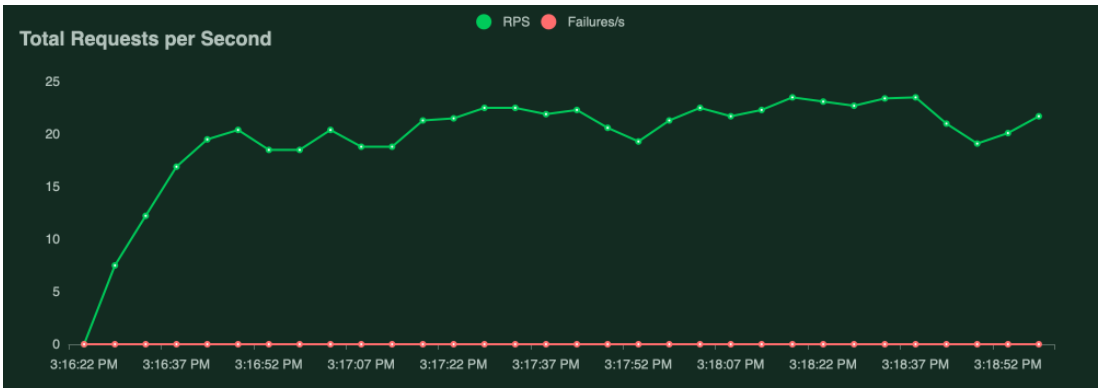
2.5. with 15 users and the concurrency limit of 15

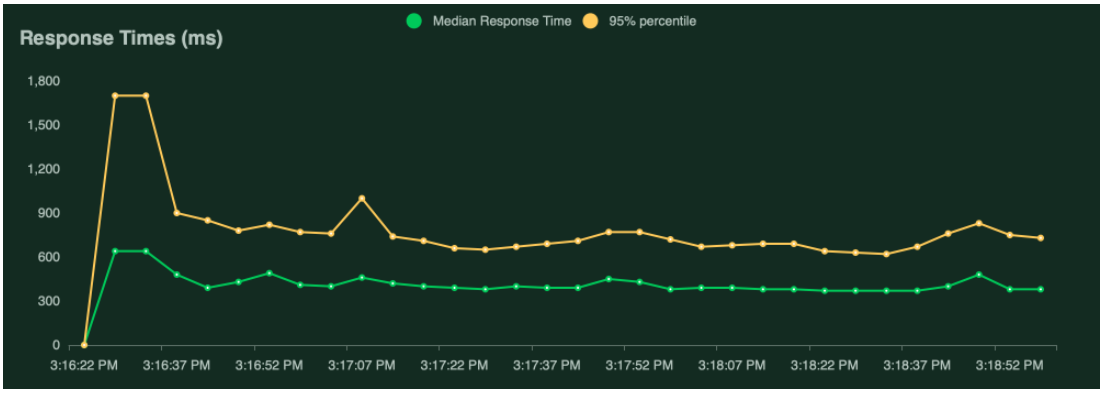




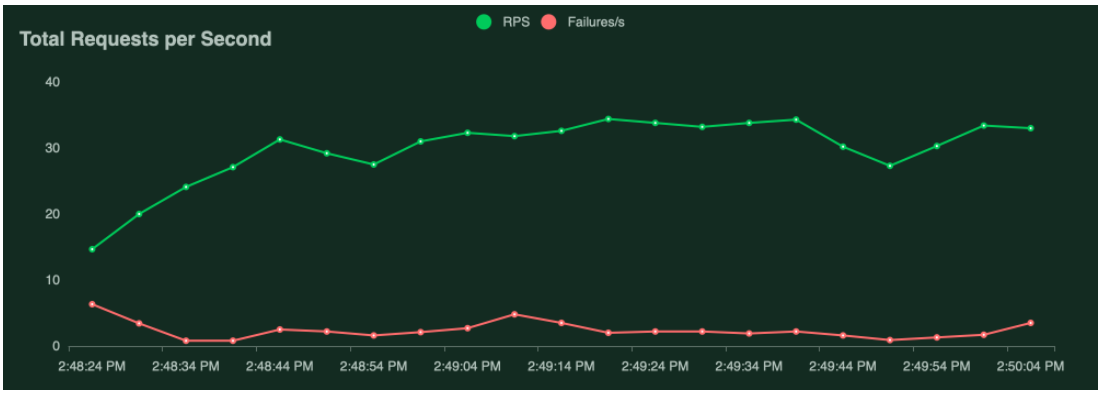
3. keyword searching:

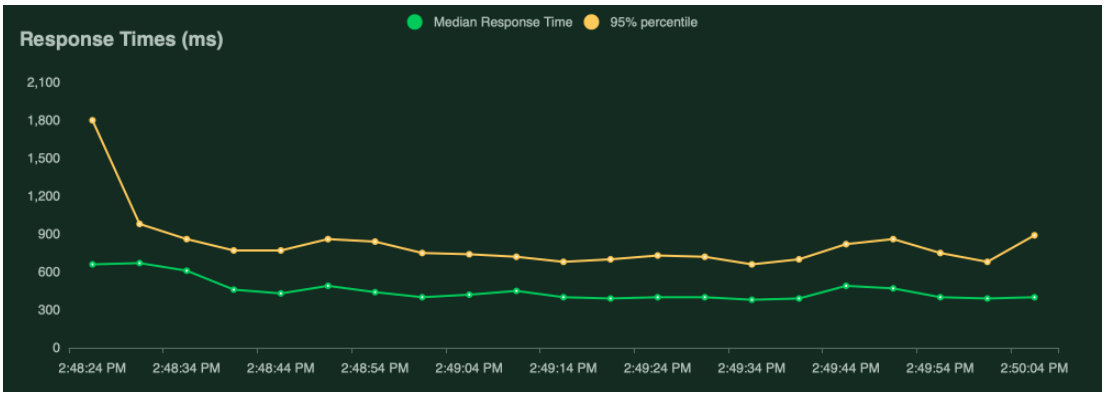
3.1. with 10 users and the concurrency limit of 10



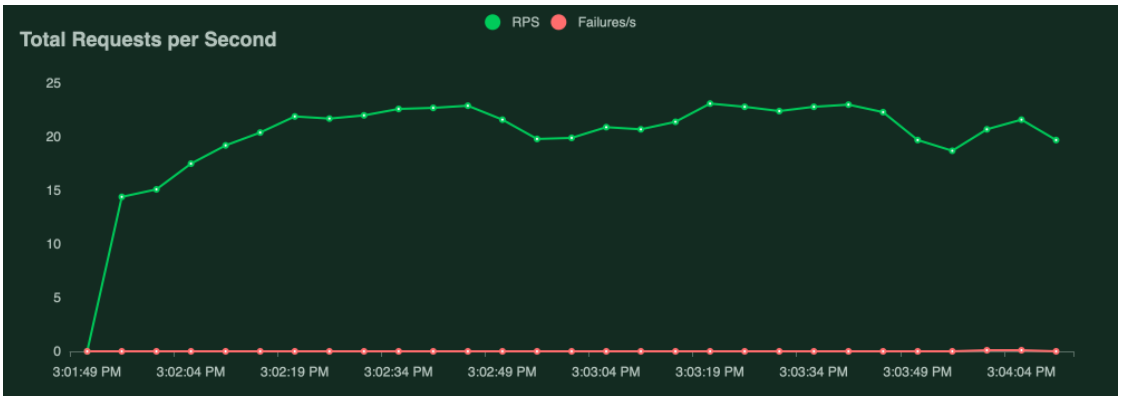


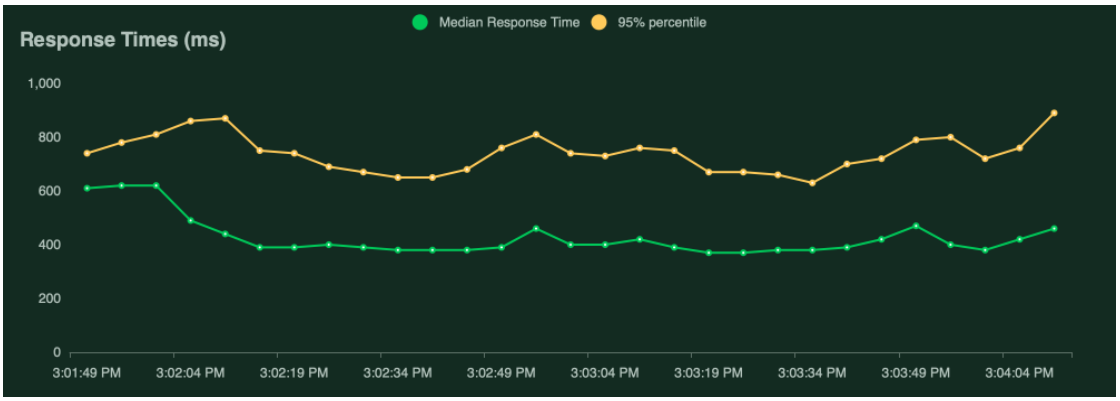
3.2. with 15 users and the concurrency limit of 10



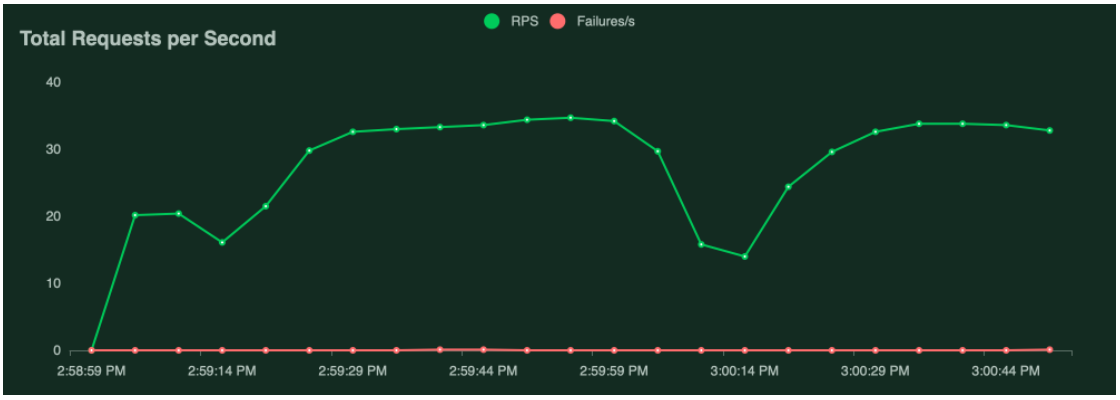


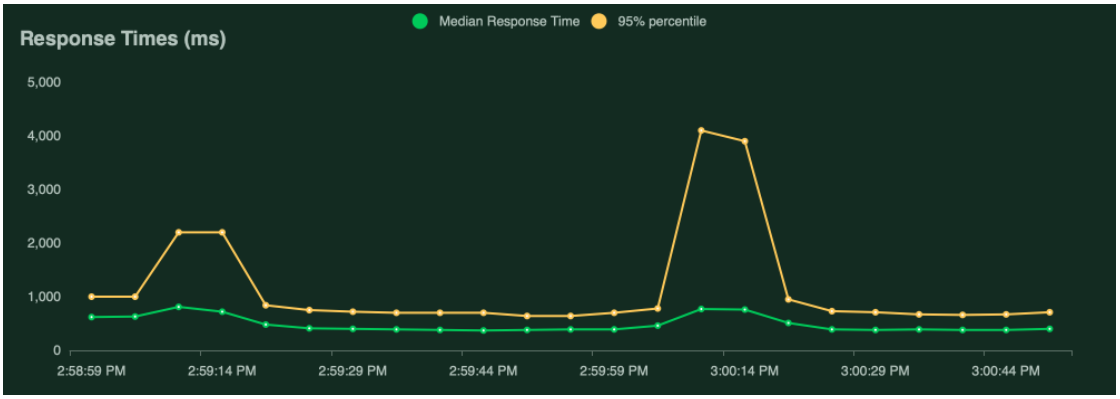
3.3. with 10 users and the concurrency limit of 15



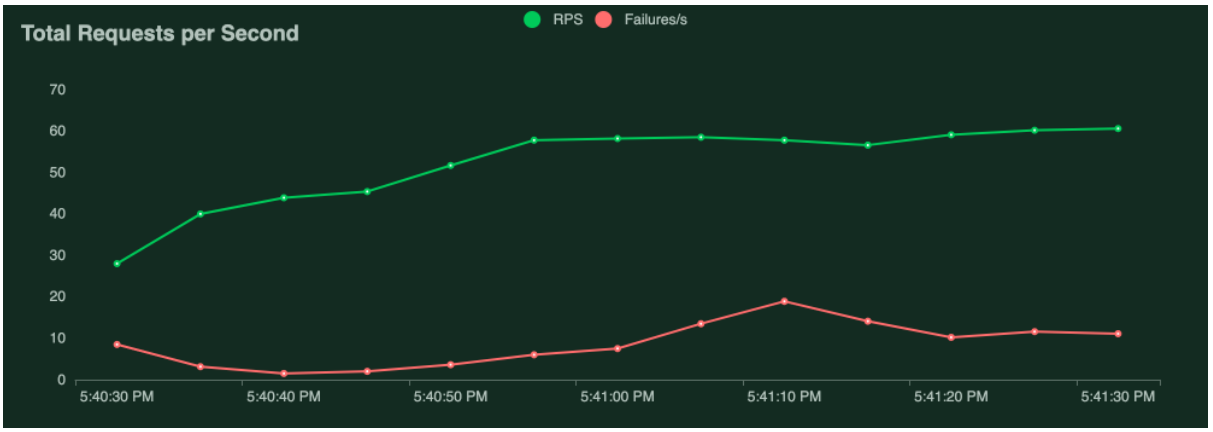


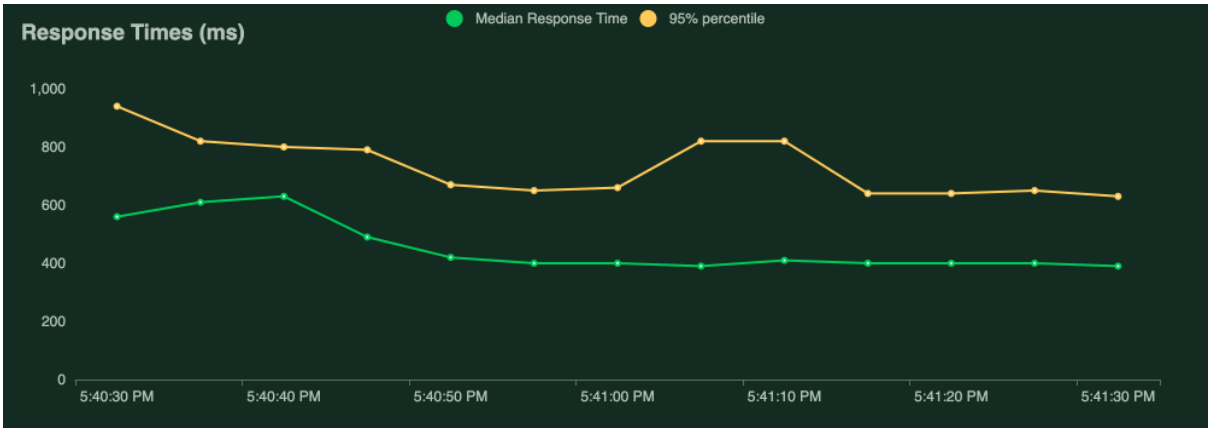
3.4. with 15 users and the concurrency limit of 15





3.5. with 25 users and the concurrency limit of 15



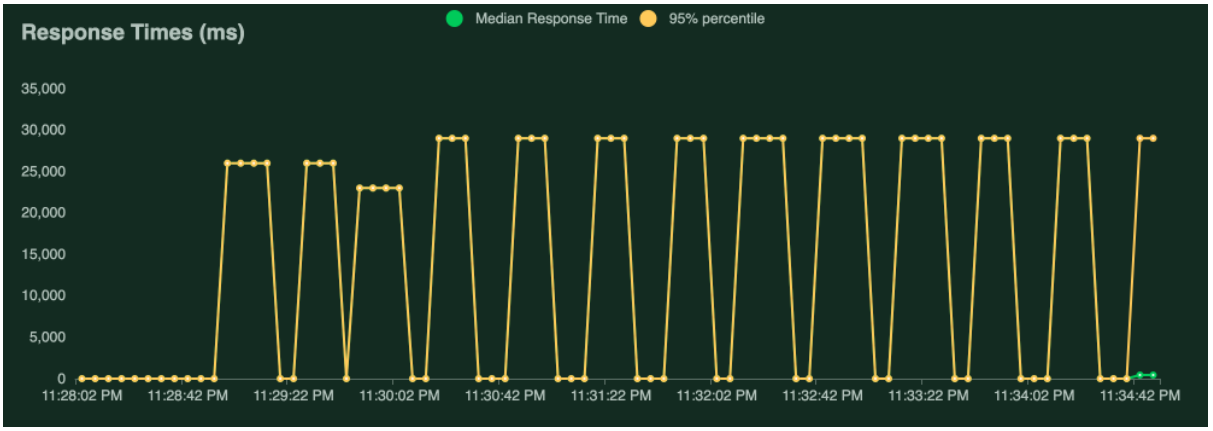


4. comment fetching and keyword searching with only one user but a larger-sized post that typically results in a timeout:

4.1

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/Prod/reddit	13	10	29000	29000	29000	28313	23245	29494	281	0	0
	Aggregated	**13**	**10**	**29000**	**29000**	**29000**	**28313**	**23245**	**29494**	**281**	**0**	**0**

4.2



Observations from Performance Testing

In this section, the above graphs will be analyzed. It can be easily seen at all charts that the request rate per second and the response time of our service tended to fluctuate inversely. This behavior is a result of the way that the testing tool, Locust, is designed.Rather than specifying a fixed request rate per second, Locust allows users to specify a fixed number of virtual users, which can then be adjusted. These virtual users simulate real users interacting with the application and make requests at a rate that depends on the response times of the system. As such, when the response time increases, the virtual users are likely to wait longer before making additional requests, which can lead to a decrease in the overall request rate per second.

The behavior for the second lambda, which is responsible for keyword searching, is straightforward. This lambda can easily handle the number of users less than or equal to its concurrency limit (*3.1., 3.3., 3.4.*). When the number of users is more than the concurrency limit, some failures can be observed (*3.2., 3.5.*). When a concurrency limit is reached and another request is made, that request would be rejected leading to failure. Since the lambda operates quite fast (600-800ms), the error is not that high compared to the similar case of the first lambda.

The behavior for the first lambda, which is responsible for both comment fetching and keyword searching, is complex. This lambda can easily handle the number of users less than its concurrency limit as did the second one (*1.1., 1.4., 2.1., 2.4*). The size of the post does not significantly affect the reliability of service by looking at the failure rate, as long as it is small enough to be fetched under the lambda timeout (30 second for free tier). However, larger post sizes tend to increase the response time, which can negatively impact its overall performance. If the number of users exceeds the concurrency limit, most of the requests result in a failure (*1.3., 1.6., 2.3.*). While testing the API with a “big” post, interesting behavior was observed. In (*2.1., 2.2., 2.4., 2.5.*), a periodic increase/decrease can be observed in response time graphs. This is probably due to Reddit APIs slowing down the response to the lambda every five minutes. We can see how performance of the system can be significantly affected by the external API. The performance of the first lambda is limited due to the concurrency limits of the second lambda as well, since it is invoking it to perform the search. AWS limits lambda concurrency per region as well. So, when we are opening lambdas of both types, the concurrency limit for each of them effectively decreases. That is why, in contrast to the second lambda, the first lambda cannot handle the case where the number of users is equal to the concurrency limit (*2.2., 2.5*).

During the performance testing for the first Lambda function (Part 1 and Part 2), we observed two different error responses for failures: *500 Server Error: Internal Server Error* and *502 Server Error: Bad Gateway*. However, in the testing for the second Lambda function (Part 3), we only encountered the 500 Server Error: Internal Server Error. Upon further investigation, we were able to identify the reasons for these different error types.The 500 Server Error: Internal Server Error is typically given when the first Lambda function is unable to run due to exceeding its concurrency limit. On the other hand, the 502 Server Error: Bad Gateway is

given when the first Lambda function attempts to trigger the second one, but receives an error message and forwards it to the Gateway as the response. Since the Gateway cannot understand this message, it throws a Bad Gateway error. Additionally, the response times for these two errors are different. The 502 Server Error: Bad Gateway takes longer to receive, while the 500 Server Error: Internal Server Error can be obtained almost immediately. This difference in error behavior might contribute to the fluctuations observed in the graphs from Part 1 and Part 2, while the graphs in Part 3 tend to be less fluctuating.

So far, we have only discussed the concurrency limitations of Lambda functions. However, it is worth noting that both API Gateway and Lambda functions have a timeout limit of 29 and 30 seconds, respectively. Thus, the bottleneck is 29 second. In our performance testing, we did not observe any issues related to these timeout limits in the charts from Part 1, because the response time for the large-sized posts was below the timeout limit. Similarly, we did not see any issues in the charts from Part 2, because the response time was much lower due to the functionality of the second Lambda function. However, with even larger-sized posts, in Part 4, we observed that response time was close to the 29-second timeout limit of API Gateway and the failure rate was around 75% (*4.1.*) even with only one user due to *504 Server Error: Gateway T*imeout. It is worth mentioning that no requests had a response rate exceeding the timeout limit in Response time graph (*4.2.*).

The response time of our service can be further improved by applying advanced configurations to AWS OpenSearch such as increasing the number of data nodes or using multiple clusters to AWS OpenSearch. However, these advanced configurations are not available in the free tier of the service, so we were unable to test their impact on the performance of our application.

Challenges

During the performance testing of our application, we encountered challenges in interpreting the results of our tests. In particular, we observed very different behaviors in our API testing graphs.

Upon further investigation, we discovered that these variations were due in part to the limitations on burst calls imposed by the Reddit API. Since our application relies on this API to fetch comments for a given Reddit post, the performance of our service is not solely dependent on the components in the cloud, but also on Reddit's policies for throttling requests to their endpoints.

Overall, this challenge highlighted the importance of considering the dependencies and limitations of external APIs when designing and testing a cloud-based application. By taking these factors into account, we were able to better understand and interpret the results of our performance tests.

Future Work

As a potential future improvement to our application, we could consider addressing the issue of timeout limitations by implementing workarounds such as dividing larger jobs into smaller parts. This approach could help to reduce the time required to complete a given task and potentially improve the overall performance of the application.However, implementing this type of solution was beyond the scope of our current project.

Another potential approach for improving the performance of our application could be implementing caching to reduce the number of requests made to external APIs.

GitHub

https://github.com/nurlandadashov02/reddit_search/tree/main

Demo

https://drive.google.com/file/d/1hKeYNOvakVXW1FWwVcaU7WTgqUipaZPK/view?usp=sharing