Extending a Certain Dynamic Priority Queue Worker for OpenFaaS Platform

Shengyu Chen
School of computing and information,
University of Pittsburgh,
Pittsburgh, PA 15213,
Email: SHC160@pitt.edu

Bangyan Li
School of computing and information,
University of Pittsburgh,
Pittsburgh, PA 15213,
Email: bal116@pitt.edu

***Abstract*—In recent years, Serverless Cloud Platforms become more and more popular. Although Openfaas still has some limitations that need to be improved, such as the slow running of the platform, OpenFaaS Platform is still one of the most popular public and easy-to-used platform for developers to implement/extend extra functionality to satisfy their special requirement.**

In this project, the goal is to extend the priority queue worker for OpenFaaS. The purpose of this design is to make certain jobs have higher priority in order to perform their work on the OpenFaaS platform. In our search, there are few works related to this external function. In order to achieve this goal, we use the open source code of OpenFaaS as the platform foundation, combined with operating system knowledge such as priority scheduling algorithms, and designed a new priority queue worker according to our specific needs. In addition, refer to the dynamic queuing algorithm. We optimize priority queue workers and make queue workers dynamic. Then, we deployed the design of the middleware before the gateway of OpenFaaS and the function of processing message in the gateway of OpenFaaS. Finally, we perform performance evaluation and analysis on the design.

Keywords Schedule Queuing Priority, Dynamic Queuing Algorithm, OpenFaaS, Open-source Serverless Platform, Demand Analysis,Platform Analysis(Profit and Structure), Operating System.

I. Introduction

In the daily work, the performance of computers is always an important concern. Computers need to deal with a lot of things, just as there are various deadlines in our lives. We cannot complete it in the order of the date of receiving the work, because everything has priorities. We need to prioritize the most important and urgent matters, and put the less urgent matters behind.

Introducing the priority queue mechanism on the OpenFaaS platform is like adding an authoritative order to a busy workshop. Only when the work is deployed reasonably and steadily, the workers in the workshop will not collide with each other. Largely, significant work and trivial clutter can also be successfully completed within the specified time.

This also has the same application significance in business management. In addition to the magnitude of things, we must also consider the degree of urgency. What we want to see is a reasonable and peaceful allocation of existing resources to make computers work properly.

Not only that, sometime we also need to consider the balance between cost and reward. It means that in specific situation, the first priority of platform is getting benefit. The worker will give priority to tasks with great rewards. This is another factor we need to consider.

In summary, the main purpose of our project is to propose a dynamic priority queue to satisfy the most of demands from users and workers. It can not only make platform work properly but also make platform earn more benefits.

II. Motivation and Demand Analysis

A. Platform' Fairness and Effectiveness

OpenFaaS, as a highly popular open source FaaS framework, will achieve good results when combined with time slices and priority scheduling.

The following dining scenario can help you understand the automatic switching of threads: In a restaurant, you order a hamburger that has not been prepared. In order not to hinder other customers, you stand aside civilly and let the next customer order. At this time, your hamburger is being prepared. Before the hamburger is ready, you stand at the end of its priority ready queue. This seems a bit unfair, because you order the hamburger first, so when the hamburger is ready, the waiter will usually bring it to you first.

Also in the operating system, in order to take into account fairness, most threads waiting for the object will generally use a temporary enhanced priority, so that the thread can execute immediately. When a running thread has used up its amount of time, the computer must decide whether to lower the priority of the thread and whether to let another thread use the processor. For example, if the priority of the thread is lowered, we will find a more suitable thread for scheduling. A more suitable thread is a thread in the ready queue that has a higher priority than the new priority of the currently running thread.

B. Platform's Profit Demand

In the second condition for the same example, if a VIP guest comes, no matter when he comes, he does not need to line up and has the highest priority to enjoy the service whatever order or pick up. It seems to be unfair to other customer, but the VIP member pay much more money than others to buy these services. Meanwhile the restaurant also gain more

benefits from VIP members. This is why VIP member have highest priority.

Analogy to our platform againExpect for taking into account fairness, the platform also need to consider how they earn the rewards. Therefore, from the perspective of gaining revenue, the users who pay more tokens should have higher priority.

C. Summary and Project Goal

According to these three condition, we should consider that how the platform can balance the fairness for all users and gaining revenue. Motivated by this thought, the purpose of these adjustments is not only to improve the overall throughput and responsiveness of the system, as well as to solve potential unfair scheduling phenomena, but also to create the greatest revenue for the platform. As with any scheduling and selecting algorithm, these adjustments are not perfect and can not satisfy all our requirement. Moreover, not all applications will benefit from them. This is another motivation for us to try to make the problem be solved.

In all, we conclude that the initial goal of this is to define a certain priority queue worker to satisfy certain demand. Then, on the basis of the first step, we optimized the design to implement a dynamic priority queue worker to dynamically meet the needs of different customers.

III. Process, Analysis and Approach

Referring to demand mentioned in section 2, we come up our project process showing in the Figure.1. We divide our project into five steps.

A. Process one and two

In the first two steps are searching related materials and analyzing our demands. In this two steps, we come up a question list showing in the following.

- How does this OpenFaaS serverless platform works?
- What is functionality of Gateway in OpenFaaS Platform?
- How does the platform put user requests into queuing buffer?
- Does Platform exist initial priority worker in Gateway?
- Is there any published related work we can refer or replicate?
- How does other serverless platform make money? Can we employ this mechanism into OpenFaaS Platform?
- Is there any special demand we need to consider?
- What algorithm we need to use for defining our priority queue worker?

According to this question list, we do many searches and analysis to clarify our plan, understand the OpenFaaS Structure, and know how to design and implement our priority worker. In this process, we found a challenging issue is that we can not find any public related or similarity work about priority queue worker. Only few people post some comments on blogs and github. So that everything must be solved by ourselves.

Secondly, after careful searches and analysis, and compared with the work of other platforms, we have concluded that we need to pay attention to three main factors in the priority queue work design: time priority (based on the priority of the request to the platform), benefit priority (the priority is determined according to the number of tokens spent by the user), and the function frequency priority (the priority is determined according to the frequency of the function used). Based on these main factors, we combine priority queue scheduling algorithms (such as LRU [11] and LFU [12]) to complete the algorithm design for priority queue workers.

Lastly, we also search, test and understand how the open-FaaS platform works. If you deploy the platform on Docker in your local system, we find that the openFaaS platform has an execution delay problem. Therefore, one of our thoughts is, after the deployment of priority staff, will the platform have more delays? In this situation, where can we deploy priority queue worker so that the platform can still work efficient? In this case, we come up two methods to extend our worker: the first method is to extend the priority queue worker in the middleware before the request enters the OpenFaaS gateway; the second method is to implement the priority queue worker by modifying the message processing function after subscription. In the implementation part, I will explain this issue in more detail.

B. Process Three, Four and Five

After clarifying this issue, we continue to design priority queue workers for two different methods through Python and Go languages respectively, testing in docker on local computer system. Finally, we conduct a performance evaluation to analyze our design.

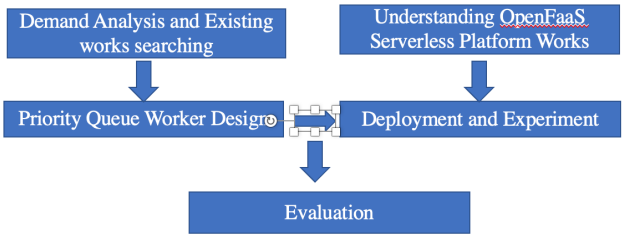

Fig. 1. Project Process

IV. Related Work

A. Serverless Computing

Serverless computing (also known as function-as-a-service or FaaS for short) is the latest paradigm in cloud computing in which the developer deploys functions to the cloud and delegates the operational and management tasks of servers to the provider [3]. The developer can focus on designing and implementing the application instead of spending time on the

management, operation, and maintenance of the infrastructure. [8]

Another feature of serverless is autoscaling. When the developer deploys a function to serverless computing, they do not need to worry about how their code should scale. No matter how many times concurrent events to the function are triggered, the serverless provider will serve them by running new instances.[4]

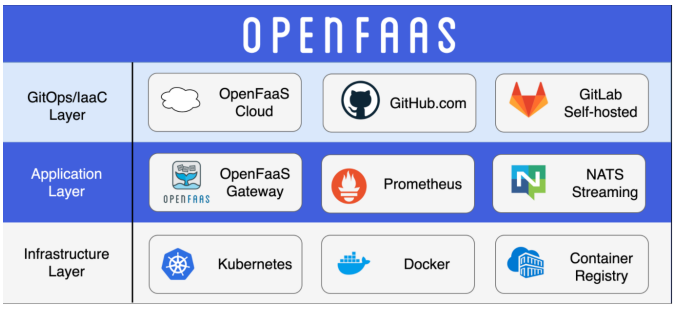

Fig. 2. OpenFaaS Architecture

B. OpenFaaS

What is OpenFaaS? OpenFaaS is a framework and infrastructure preparation system for building serverless applications. It originated from the serverless framework in the Docker Swarm and now supports other kinds of infrastructure backends, such as Kubernetes or Hyper.sh. Functions in OpenFaaS are containers. Any program written in any language can be packed as a function by leveraging the container technologies of Docker. This enables us to fully reuse the existing code to consume a wide range of web service events without rewriting the code. OpenFaaS is a great tool for modernizing old systems to run on a cloud-based infrastructure. The detail architecture shows in the Figure 2.[5]

C. Priority Queue

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue. Generally, the value of the element itself is considered for assigning the priority. For example, The element with the highest value is considered as the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element. In other cases, we can set priorities according to our needs.[8]

In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first. Also, priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.[7]

D. Dynamic Priority scheduling algorithm

According to Wikipedia, dynamic priority scheduling is a type of scheduling algorithm in which the priorities are calculated during the execution of the system. The goal of dynamic priority scheduling is to adapt to dynamically changing progress and form an optimal configuration in self-sustained manner. It can be very hard to produce well-defined policies to achieve the goal depending on the difficulty of a given problem.

Earliest deadline first scheduling and Least slack time scheduling, Least Recently Used algorithm, and Least Frequently Used algorithm are examples of Dynamic priority scheduling algorithms.[10]

1. Initialize/Randomize function priority list
2. Receive the requests within a period and put it in queue(order by time)
3. Refer function priority list, re-order the received requests in queue
4. Check benefit priority(tokens) in each requests and move up/down priority level in queue -- sort it again.
5. Copy the re-order list to new list1 and clear up this re-order list
6. Pop up the requests from new list1 by order to next steps' function execution.
7. Conditional: Update function priority list within a certain period(Make priority queue dynamic):
   - LRU and LFU
8. Go back the the step two

Fig. 3. Algorithm Design

V. Priority Queuing Algorithm Design

In previous section, we mentioned that there are three main factor need to be consider for our priority algorithm design. They are time priority, function frequency priority and benefit priority. So according to these three factors, we come up our algorithm showing in the following Figure 3. And I will also explain more in detail why we will design in this way.

First of all, we know that multiple functions are deployed in the opanFaaS platform. Therefore, we need to initialize the function priority list according to some requirements we need. For example, we can make required functions have a higher priority than other functions. Secondly, after initialization, the priority queue worker will collect requests received within a certain period of time in a certain time and order. Then, the priority queue worker will reorder the request queue referring to the function priority list again. Next, according to each request (benefit priority) owned by the token, the worker will increase its order and sort the queue again. Finally, the worker copies the sorted queue to the new queue, which can pop

requests in order to perform the next step. At the same time, the sorted queue will be cleared and return to the second step. This is initial design we set up. However, as we mentioned in the introduction, our ultimate goal is to make our priority queue dynamic. It mean that we plan to make the function priority list update within a certain time. How can we update the priority list?

Here, we introduce the concept of dynamic priority scheduling algorithm to make our priority queue workers dynamic. In this case, the reference function priority list is not fixed and will be updated. The priority queue worker checks the frequency of each function called at regular intervals. Based on the LRU (Least Recently Used) algorithm, the priority queue worker will update the reference list to obtain a new function priority list. Then after next time the worker receive the requests, which will be sorted referred to the new reference list.

This is our dynamic priority queue worker design. In our design, the worker not only considers the effectiveness of the platform's workreference priority list), but also ensures the profitability of the platform(benefit priority), and also tries its best to make most users get the best service(Time Priority).

VI. OpenFaaS Structure and Extend Priority Queue Worker

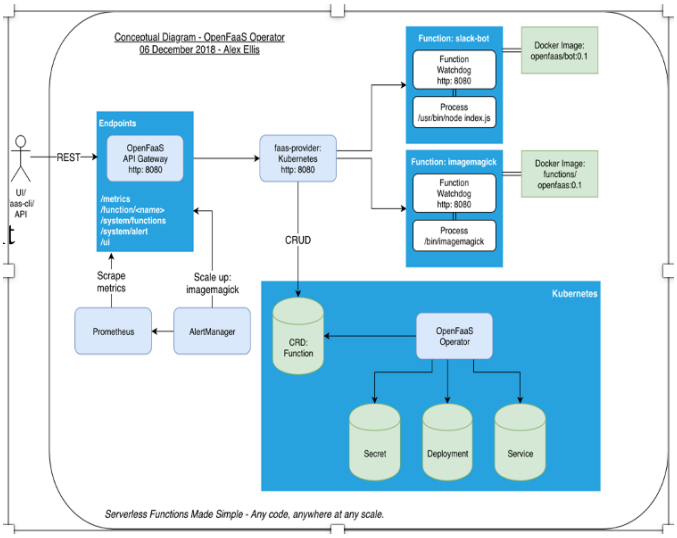

Fig. 4. OpenFaaS Overview

A. Overview of OpenFaaS Platform

According to the Figure 4, we can briefly outline how the openFaaS Platform works. In detail, When OpenFaaS Platform starts, the platform will initialize a Nats-queue in Gateway. Meanwhile, OpenFaaS will subscribe to queue information. Then when a new request arrives, it is posted on Nats-Queue. OpenFaaS will get the request and put it into the queue buffer channel and wait for execution in next step. After this step, openFaaS will extract the request/function from the buffer channel in order, and paired with a certain function provider, then starting and executing the computation work. Finally, the platform will send response back to users after execution.

B. OpenFaaS Gateway Structure

Now, we already have simple overview of functionality of OpenFaaS Platform. But remember that, our project goal is to define and extend a priority queue worker for platform. We can easily know that we only need to focus on how to replace the queue buffer in Gateway by a priority queue sorted by certain demand. Therefore, we know we should focus on the functionality of gateway Structure in OpenFaaS, understanding where can we extend our priority queue worker.

According to the Figure 5, we can understand that:

- Generate asynchronous queue nats queue, and establish a connection between gateway and nats-queue, getting "NC" connection.
- MakeQueuedProxy function puts the received request into a structure
- Queue gets the request coming previous stepcompiles it into string with json.marshal, and stores it in "out" variable.
- Then get the connection "NC" in step 1, and publish queueName and "out" variable to Nats-Queue.
- OpenfaaS will subscribe Nat-queue in Queue worker,
- When Queue worker calling subscribe function, it will pass in a processMsg method. This method will process the request received through the subscription. In detail, processMsg will unmarshal the incoming message and regain the request.

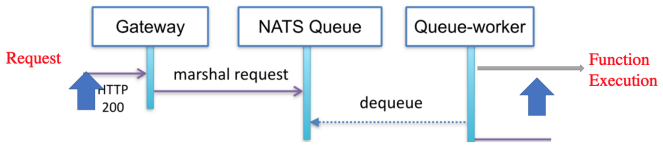

Fig. 5. Gateway Structure and Extend/Deploy the Implementation

C. Extend Priority Queue Worker into Platform

As I mentioned in previous section 3, we plan to implement the priority queue worker by modifying the message processing function after subscription , and extend our the designed priority queue worker in the middleware before the request enters the OpenFaaS gateway.

1) Method one: For the first method, as we explained above about the OpenFaaS gateway structure. We know that there is a processMsg function in the gateway, which is used to process/extract the requests in the queue buffer in order. If you need to replace this queue buffer with a priority queue sorted by a certain demand, you need to modify this function. In theory, this is a good way to solve this priority problem. However, when we tested and deployed this method, there were some limitations and obstacles.

First of all, when we deploy and test functionality in docker on local computer system, we found that there are a big execution latency when the platform do the simple computation work, sometimes it will spend a few seconds to get the response. In this case, we thought that will the execution latency become a particular larger if we deploy the priority queue worker into OpenFaaS Gateway? Because the OpenFaaS not only needs to prioritize the queue buffer, but also needs to update the function priority list within a certain period. There will be much more works than before, which will affecting the efficiency of platform's execution.

Second, it is difficult to deploy and test in docker on the local computer system. Because when deploying openFaaS in docker, docker directly pulls the OpenFaaS image from dockerhub. Therefore, we cannot modify the source code directly in docker. In our thinking, what we can do is that we should modify the OpenFaaS source code locally and build our own docker image, push it to dockerhub, then change the docker yml file, and finally implement a new OpenFaaS deployment in docker. After two weeks's attempts. We still feel this is a big challenge for us. Because we should spend much time on studying and understand how to modified docker source file. This is not our goal for this project. So we change our mind and focus on second method.

2) Method Two: For method two, referring to Figure 6, you can find that we design and extend a middleware to achieve the functionality of priority queue worker. Unlike the previous approach, users will directly send the request to the openFaaS Gateway. Instead, all users will send their requests to this middleware structure. In this structure, according to the priority queue worker mechanism we design, this structure will sort all the requests received within a certain period. Finally, according to the result of the list reordering, it will send the request to the openFaaS Gateway by order.

The advantage of this approach is that since we have not made any changes to the platform, all the prioritization work has been completed before entering the platform. In this case, the platform can always maintain its own operating efficiency without additional execution delay. The disadvantage is that an intermediate layer is established between the user and the platform, and the priority sorting work is implemented in the intermediate layer. Therefore, compared with the OpenFaaS platform without this priority queue worker, there is a higher transmission delay between the user and the platform.

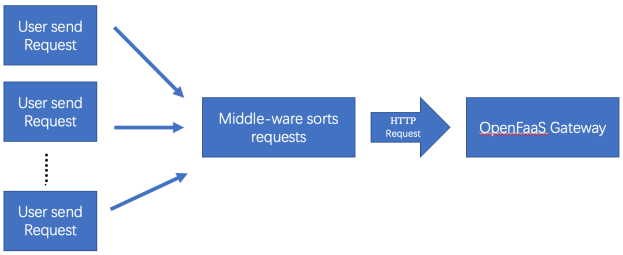

Fig. 6. Method 2 Structure(extend a middle-ware Structure)

VII. Experiment and Evaluation

In this section, we will focus on how to implement our design for priority queue worker. As I mentioned before, for the first method of deploying our design in subscriptions in the openFaaS gateway, we achieved this goal through the Go language and tested it on docker in the local Mac OS system. However, as I mentioned earlier, it is difficult to properly modify the default file and deploy our design in docker in a short time. Therefore, we will not explain the implementation in more detail, and will focus on the second method of extending the design in the middle-ware structure. More than that, in the second subsection, we will explain the evaluation and analysis for our design from different three aspects: Energy consumption, latency and fairness.

A. Implementation

For the second method, we directly use python language and RPC (Remote Procedure Call) protocol [13] to achieve our implementation. In this regard, we have few important parts to mention.

First of all, as I mentioned in the previous section, the priority queue worker should contain a feature priority list that references the current existing features in the OpenFaaS platform. Therefore, the middleware structure should update the function list within a certain period of time to avoid missing new functions in the platform. The newly added functions contain the lowest priority in our priority queue worker because no user has used them before. The priority will be updated or increased based on how often certain functions are called. And the rest of implementation is the same as I mentioned in algorithm design.

Second, in order to facilitate testing and evaluation, we use our local MacOS system and deploy openFaaS image in docker to do this experiment. we set the user's request format to json format, which contains information about the functions they requested and the tokens they paid (benefit priority). At the same time, we deployed 10 different simple functions in docker's openFaaS platform for experiment. Then,

within a certain period of time, we established 10 different user requests to send to the middle-ware structure, after reording referring to designed priority mechanism, forwarded the requests by order to openFaaS to next step's function execution. Meanwhile, this middle-ware structure will wait for a response from openFaaS after completing its execution and forward the response back to each user.

This is our implementation for our design. Then,in next subsection, we will explain the experiment analysis and performance evaluation for our design.

B. Evaluation and Analysis

For this subsection, we set up experiment to test and evaluate the performance of priority queue worker from 3 different aspect: Execution Latency, Time Consumption and Fairness. And we will explain more in detail in next few paragraphs.

1) Execution Latency: We set up the experiment to record the execution latency in priority queue worker. In our experiment, In a certain period of time, we make our middle-ware structure receive and process 10, 20, 50, 100200500 requests from users respectively, and calculate the time consumption complete the priority queue sorting by our algorithm design. In total, we tested it 10 times and averaged the average results obtained. The analysis results are shown in Figure 7 below. According to the execution delay result shown in the figure, it only takes 1 second to process 200 requests and complete the priority sorting. Therefore, we can conclude that even if we process 100 requests or even more requests at the same time, it will not cause a lot of delay and only use a small amount of time. This shows the feasibility and efficiency of the middle-ware structure. Although the final result is not yet known, if we deploy the priority work program in the subscription function of the OpenFaaS gateway. We can imagine that the execution delay in the middle-ware structure is likely to be much smaller than the deployment delay in the subscription function. Since the OpenFaaS structure needs to perform different functions at the same time, they will take up a lot of memory to perform. By the way, all our experiments are currently run and tested on the local MacOS system. We believe that if we test on the cloud platform, we will get better results.

2) Total Time Consumption: In this experiment, we set up an experiment to record the total time it takes from the user sending a request to the user getting a response. We also tested on the original openFaaS platform and the platform with an extended middleware structure. The user request contains one of 10 different simple function requests in the OpenFaaS platform, and the execution time of each function is almost the same. As we did in previous subsection's experiment, in a certain period of time, we send the 10, 20, 50, 100, 200, 500 requests openFaaS directly or via middle-ware and record the total time consumption after user getting responses. In general, I will still test 10 times and take the average. The detailed results are shown in Figure 8. The "Priority Workers" column shows the total time spent after expanding priority workers. The "Original" column shows the results

Request	10	20	50	100	200	500
Priority Worker	0.343	0.52	0.94	1.13	1.425	3.24
Original	0.08	0.124	0.168	0.288	0.588	0.918
Execution Latency	0.214	0.351	0.679	0798	0.997	1.79

Fig. 7. Comparison Result between our design and original

of the original openFaaS platform. According to the analysis results shown in Figure 8, we can know that the total time consumption after expanding the priority queue worker is 3-4 times that of the original platform. This latency comes from two different aspects, one of main aspect is execution latency in priority queue worker, another one aspect is connection latency from user to middle-ware and from middle-ware to platform. In the future work, we will tried to optimized our design from these two aspects.

3) Fairness: In order to evaluate the fairness of the openFaaS platform, we use user evaluation and feedback for this analysis. We have established a questionnaire to clarify the original time sequencing mechanism of openFaaS and our designed priority worker mechanism, accomplished with a simple demo show function of both. We send this questionnaire and demo to our friend circle, asking for whether our design guarantees the fairness between user and openFaaS or not. In the end, we got feedback from about 35 people. These 35 people have different professional backgrounds, not just students and programmers. The detailed results are shown in Figure 8. Feedback from most people shows that our design has achieved certain results, and the fairness of the platform has been improved to a certain extent, which can meet the requirements of some users, but there are still some issues that need to be improved and considered.

Unfairness	Not helpful	Better than before	Absolutely fairness
1	8	20	6

Fig. 8. Fairness

In conclusion, based on our evaluation and analysis, we can know the effectiveness and feasibility of our design. However, there are still many issues that need to be considered and optimized such as security and privacy, execution time complexity etc. Therefore, in this project, we just gave an implementation and design direction, we will optimize our work if we have chance in the future.

VIII. Conclusion

In short, in order to meet the needs of most users and balance the fairness of all users and obtain benefits, we have considered several factors in the priority queue sorting, and put the three main factors of time priority, function frequency priority and revenue into consideration. Combining priority and priority queuing scheduling algorithms, a dynamic priority queuing mechanism is proposed to sort the priority order of requests in an attempt to meet the needs of most users. Based on our experiments and evaluations, we can know that our design is feasible, but still needs to be optimized, such as reducing connection latency. Also in future work, we can reproduce our first method and compare the performance with the second method. Based on this, and regrading other industries factors such as security issues, we can understand that which aspect should be on optimization.

IX. Personal Contribution

This is personal contribution for this project:

- Shengyu Chen: Design the Priority Queue Algorithm and Come up evaluation ideas. And write the main part of final report
- Bangyan Li: Write the main part of code for method two.And do the experiment for evaluation and write a few part of final report
- The rest of work, we both work on together whatever in demand analysis, search related materials, testing method one and figuring out the docker deployment issues.

References

[1] P. Mell, “The nist definition of cloud computing,” Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. 800-145, 2011.

[2] A. Fox, “Above the clouds: A Berkeley view of cloud computing,” Dept. Electr. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS, p. 2009, 2009, vol. 28, no. 13.

[3] I. Baldini, “Serverless computing: Current trends and open problems,” in Research Advances in Cloud Computing. Singapore: Springer, 2017, pp. 1–20.

[4] FaaS, ”What Is Serverless Computing?” https://www.cloudflare.com/learning/serverless/what-is-serverless/

[5] OpenFaaS, ”OpenFaaS - Serverless Functions Made Simple”https://docs.openfaas.com/

[6] OpenFaaS,”OpenFaaS modules-Chinese Instruction” https://jahentao.gitbook.io/openfass/4.-openfaas-zu-jian

[7] Priority Queue Designhttps://www.programiz.com/dsa/priority-queue

[8] ”Wikipedia: Priority queue”. https://en.wikipedia.org/wiki/Priority_queue

[9] ”Wikipedia: Serverless Computing”. https://en.wikipedia.org/wiki/Serverless_computing

[10] ”Wikipedia: Dynamic priority scheduling”. https://en.wikipedia.org/wiki/Dynamic_priority_scheduling

[11] ”Wikipedia: Dynamic priority scheduling”. https://en.wikipedia.org/wiki/Cache_replacement_policies

[12] ”Wikipedia: Dynamic priority scheduling”. https://en.wikipedia.org/wiki/Least_frequently_used

[13] ”Wikipedia: RPC ”. https://en.wikipedia.org/wiki/Remote_procedure_call#:~:text=RPC%20is%20a%20request%E2%80%93response,the%20application%20continues%20its%20process.