WordFreq Project Report

Brief Summary of the WordFreq

WordFreq is a Golang application that can receive a text file and return the ten most frequent words, including the corresponding numbers of occurrences, of the file. To make the WordFreq productive, we need to implement the application on the AWS cloud environment. The Figure.1 shows the basic process of the WordFreq application implemented on AWS.

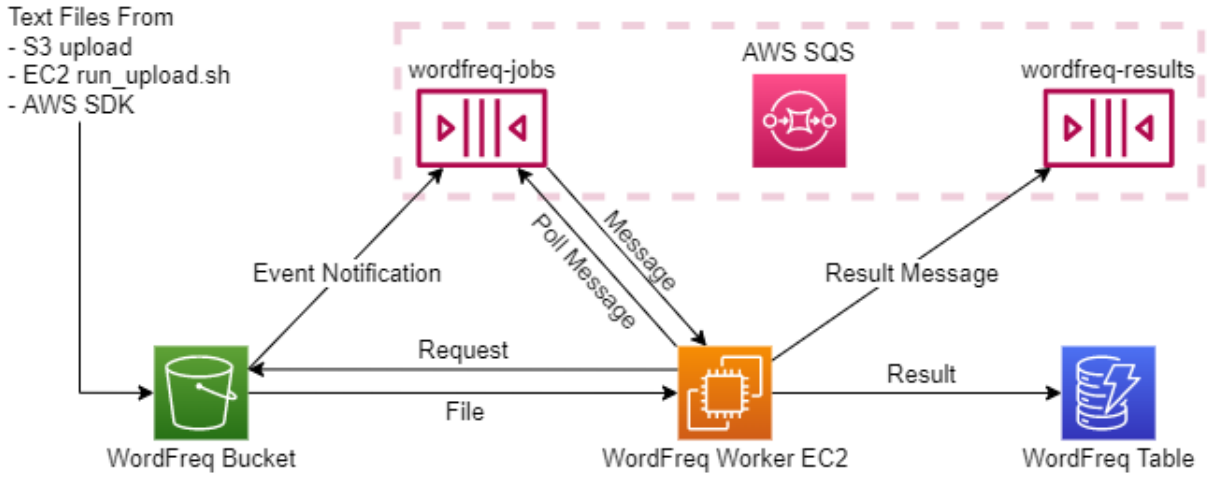

Figure.1 Basic process of the WordFreq on AWS

The origin text files should be uploaded to the specific AWS S3 bucket. When a test file uploads to the bucket, the bucket will save it as an object and send an event notification message, which contains the object location information, to the 'wordfreq-jobs' SQS message queue. The WordFreq Worker EC2s are the core component. The EC2s continuously poll messages from the wordfreq-jobs queue. By parsing the receiving message, the EC2s can locate and request the whole file from the S3 bucket and then process the file with the WordFreq application. The EC2s will send the result into the WordFreq DynamoDB table and the wordfreq-results queue.

Load Testing

After the WordFreq application is wholly implemented on the AWS, we can upload some text files to the S3 to test whether the application is well worked or not. By load testing, the cloud architecture issues may be exposed, and the application's performance can be measured. In this session, the test files attributes will be introduced initially, and then the automatic testing script will be illustrated.

The test files were collected on the Canterbury Corpus page. The chosen test files are around 2 MB to 4 MB in size. To meet the load testing demand, I have duplicated the files into multiple copies, and there are 50 files in total. Besides, to simulate various scenarios, different sizes and numbers of files have been prepared by a Python script. The three prepared files groups' screenshots are shown in the Figure.2.

(a) 10 big files (b) 100 small files (c) 50 normal files

Figure.2 The three text files set

Uploading the same test files onto the AWS S3 bucket is time-consuming. That is because the S3 bucket and the SQS queues need to be purged manually before uploading the files. In addition to the time, the application performance is difficult to be measured accurately. To test the application efficiently and accurately, I wrote the automatic test scripts by using Python SDK. The auto testing includes three scripts. The first script is the autoTest.py which can automatically purge the bucket and upload test files located in a specific directory. The getResultMsg.py is used after the process of the test files. The script can receive deleted messages from the wordfreq-results queue, and the received messages will be restructured and stored in a new CSV file. The resulting message contains the start time of the task and the result message sending time, and the difference between the two timestamps is the task's runtime. The runtimeParse.py is used to work out the runtime from the CSV file. The three script files have been sent to Github: https://github.com/theodoretsui/LSDE_coursework_AWS_autotest.

After all the configuration, let us run the auto load testing script to do the test. The files' group I used is the 50 normal files. As shown in Figure.3, at the beginning of the test, the S3 bucket and the SQS wordfreq-jobs queue had files and messages inside. When we ran the autoTest.py, the console showed the uploading logs (Figure.5). The bucket and the queue were automatically purged by running the script (Figure.4).

When the "messages available" and the messages "in flight" were reduced to zero, all the tasks were completed, and then we could run the getResultMsg.py script. A result file name needed to be assigned in the console, and the received messages would be shown in the console and stored in a CSV file (Figure.7). Finally, we could run the runtimeParse.py script to calculate the runtime from the result CSV file. The console showed the total runtime of the load testing and the number of duplicate tasks, working instances, and failed messages. As we can see in Figure.8, the total running time of the load testing was 820 seconds, and the number of failed messages was 28.

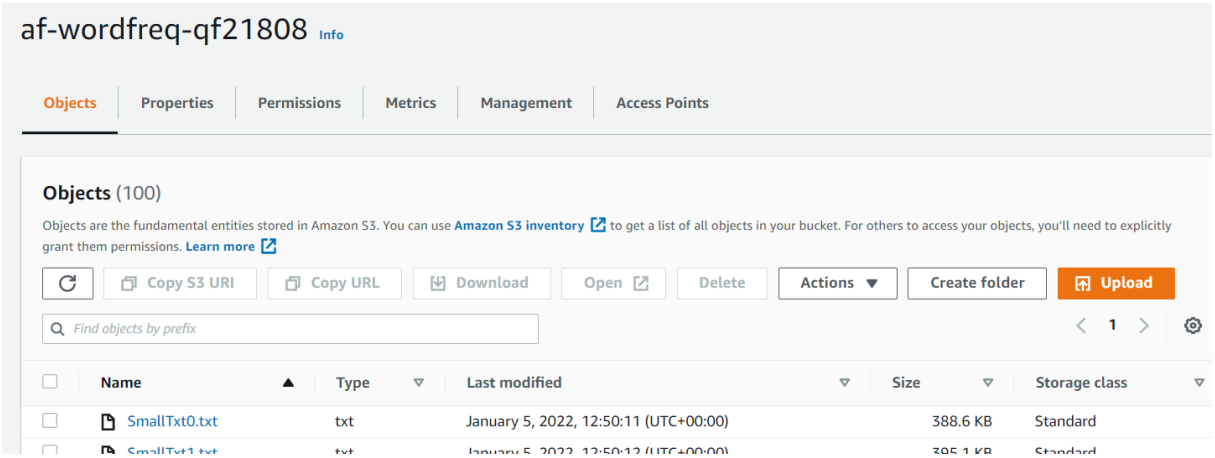

Figure.3 The S3 bucket before running the load testing script

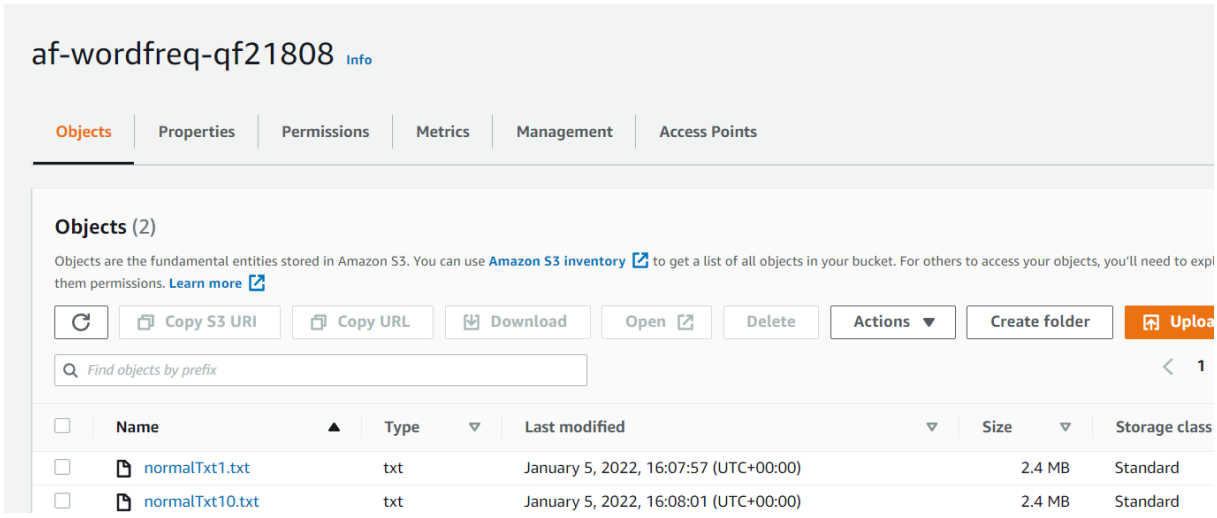

Figure.4 The S3 bucket after running the load testing script

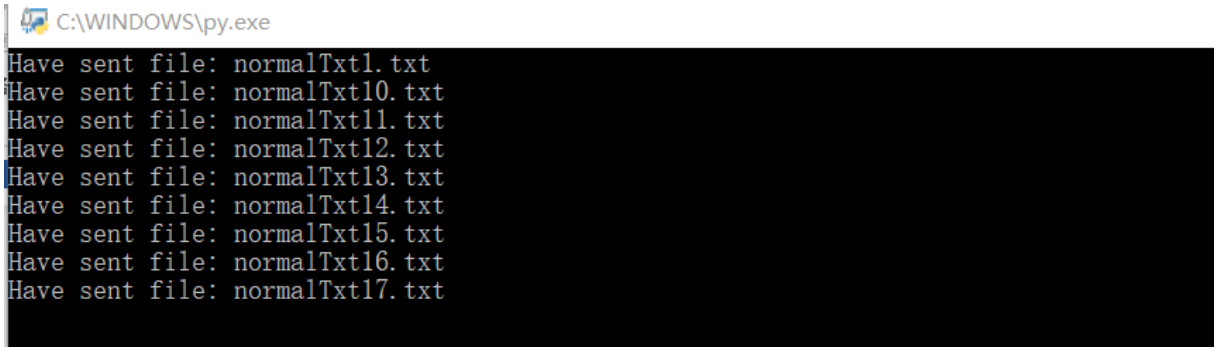

Figure.5 The logs shown in the console

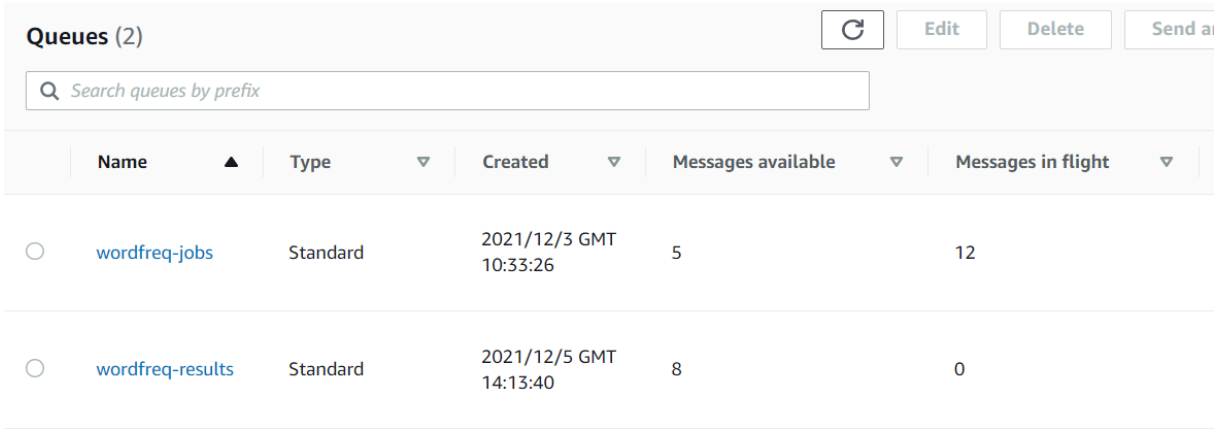

Figure.6 The SQS queues during the load testing

Figure.7 The console when running getResultMsg.py

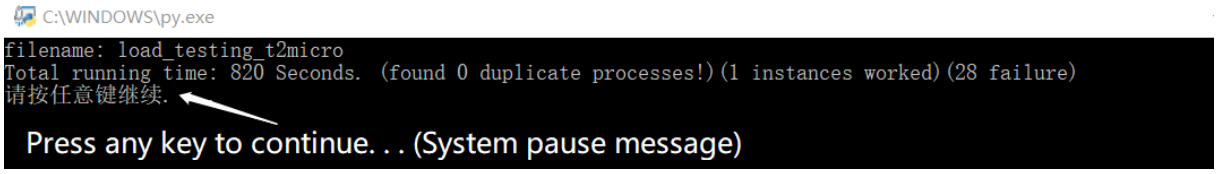

Figure.8 The running time and relative messages

To figure out why there were so many failed messages, I checked the CSV file. As shown in Figure.9, the status message of the failed task showed that the message did not exist or was not available. Besides, the text file that the failure task was working on had been done successfully at an earlier time. Therefore, the failures indicated that the message queue, which should have configurations to prevent duplication, was not configured properly. To fix the duplication, we should modify the message visibility timeout. After consulting with Dr Zheng, the Unavailable Timeout is set in the WordFreq program, so we cannot edit it on the AWS SQS website console. The argument can only be modified in the "run_worker.sh" file. In this file, environmental variables are collected by the Wordfreq program, and they are worked as configurations. Therefore, "export MessageVisibilityTimeout='150'" was added into the "run_worker.sh" file and the available message timeout is set into 150 seconds now. Finally, I made a new AMI called "wordfreq-img" to store the latest version of the instance image.

StartTime	TextName	Status	StatusMsg	SentTimestamp
2022-01-06T10:46:29.195349	normalTxt28.txt	success		1641466117316.00
2022-01-06T10:47:54.568687	normalTxt28.txt	failure	Failed to update job messages's visibility timeout, InvalidParameterValue: Value AQEBXR13mLsVsn/9lNQEpfx/f6UUdTDqy7YBlUsaORg5GQNBcSo3Zfc9xMAtjU2AY0+67EkoMy9XjT9mnHlUWcWqY5Vj5JKlDUoaKoQXBo4McvouHfu4dpLzC2QkDPiVJtSxk3KjDUTO4ykp0lYuTjk76fCkp9pZa17R5E3UFSM+TdvUKMzaJ4lwEwTpFUSdClwBGeduadQplsBwZAAaoeuh28NiF1DFcGLdTwVtgZ7/7ysAaAl+rWD1T1ojL1yLRGeJYjK0bsgv7sZ/gV7ivQZwjjCOcVaF31E+1bMGqYGVLuK9HmJkdkvaJeqqCHMFhOETaz9PYukh+EDDgCsKx7SGc1OvojFcrHrra81PuzjCFs2lOpRghk6+4mQrl1FqDNZs for parameter ReceiptHandle is invalid. Reason: Message does not exist or is not available for visibility timeout change. status code: 400, request id: fa0fd554-9028-50f5-af1d-8f045ba5cb1c	1641466199881.00

Figure.9 Failure message in the result CSV file

After modifying the sh file, let us do the load testing again. The Figure.10 shows the result. There was no failure, and the total running time was reduced by 55.6% to 527 seconds.

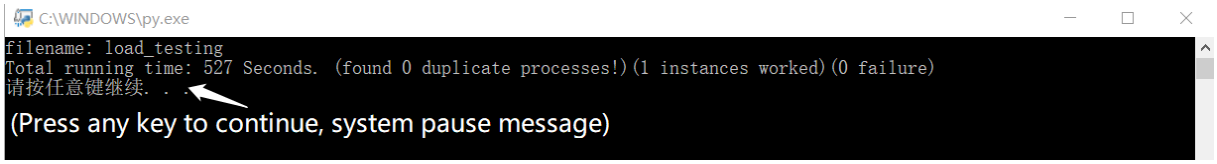

Figure.10 The running time after modifying the visibility timeout

In this session, I have created the three groups of test files. Then, the auto testing scripts have been written to make the load testing more efficient and accurate. Finally, I have tried to use the automatic load testing and successfully figured out the issue of the failed tasks.

Auto Scaling Strategy

Dynamic auto scaling is essential for a resilient application. Proper scaling strategy makes the application work faster and more cost-effective. The wordfreq scaling strategy will be illustrated in two parts in this session. Initially, the concerned CloudWatch metrics will be discussed, and then, the scaling out and scaling in strategy will be explored.

To implement a dynamic auto scaling group, we need to identify proper CloudWatch metrics initially. The basic WordFreq application includes four AWS services, and two of them, SQS and EC2, are the core part of the application. The metrics of the two services can be the critical metrics for automatic scaling. When talking about the EC2 metrics, the CPU utilisation is always the key metric. However, during the 100 small files load testing, the CPU utilisation has no manifest fluctuation as the Figure.11 shows, and thus, it is not a suitable metric.

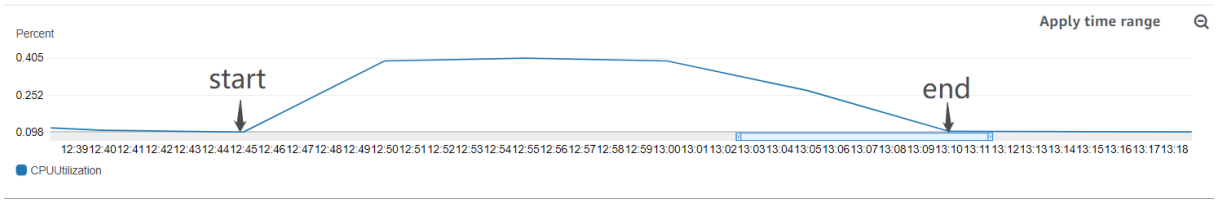

Figure.11 Instance CPU utilisation during the load testing

Luckily, three metrics in SQS are perfectly matched for the application. The number of messages sent of the wordfreq-jobs queue monitors the notification messages sent from the S3 bucket to the queue. The "approximate number of messages visible" and the "approximate number of messages not visible" of the queue monitor the number of available messages and messages in flight of the queue, respectively. Figure.12 presents the three metrics data of the wordfreq-jobs queue during the load testing, and the red line indicates that there is only one working instance.

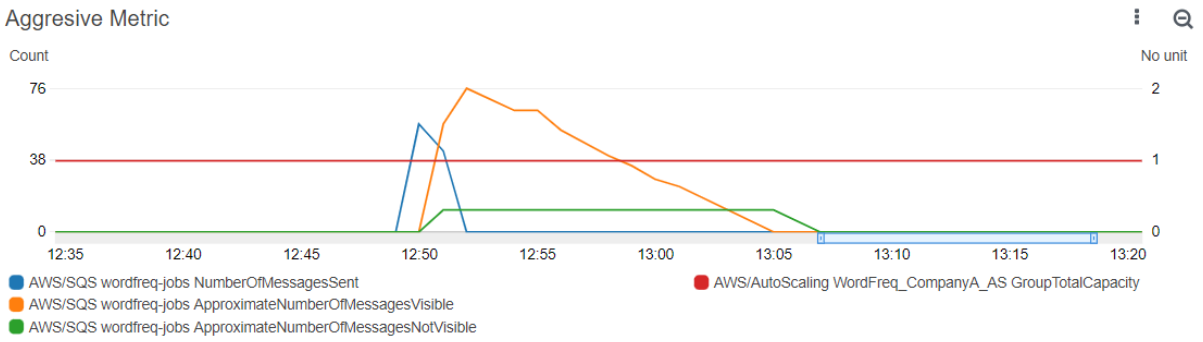

Figure.12 SQS queue metrics during the load testing

At the beginning of the testing, the number of messages sent (blue line) spiked from zero to a peak and then quickly dropped back to zero. As the first metric rose steeply, so did the number of available messages (orange line), but the second metric decreased slowly, unlike the first metric. Turning to the last metric, the approximate number of messages not visible (green line) reached a certain number (12 messages not visible) and did not go down until the end of the testing. The green line also indicated that a working instance could only hold 12 tasks at a time.

The first two metrics, the blue line and the orange line, are suitable for scaling out metrics since they reflect the demand of the tasks. However, the approximate number of messages visible is a better choice because it shows the actual congestion situation in the message queue. The last metric, the approximate number of messages not visible, is a perfect choice for the scaling in trigger. The metric will not drop to zero until the instances finish the tasks. Taken together, the approximate number of messages visible has been chosen to be the metric for scaling out trigger, while the approximate number of messages not visible has been selected to be the metric for the scaling in trigger.

I am now considering the scaling out and scaling in strategy in detail. Initially, the scaling out and the scaling in policies were simply configured as follows. When the approximate number of messages visible is greater than 12, add 1 instance, and then wait 60 seconds; when the approximate number of messages not visible is less than 6, remove 1 instance, and then wait 60 seconds. The reason for the threshold is 12 is that a working instance can only hold a maximum of 12 tasks at a time. Finally, to satisfy the given situation, the auto scaling group's minimum capacity is 1, the desired capacity is 1, and the max capacity is 6, but it can be set to a larger value. After configuring the auto scaling group, I conducted a load testing by using the 100 small files to examine the performance of the auto scaling group. As shown in Figure.14, the new instances were launched when a vast number of tasks flooded into the message queue (Figure.13). The running time was 533 seconds, and the scaling policies worked successfully (Figure.15). Nevertheless, the scaling operation was imperfect and could be more resilient.

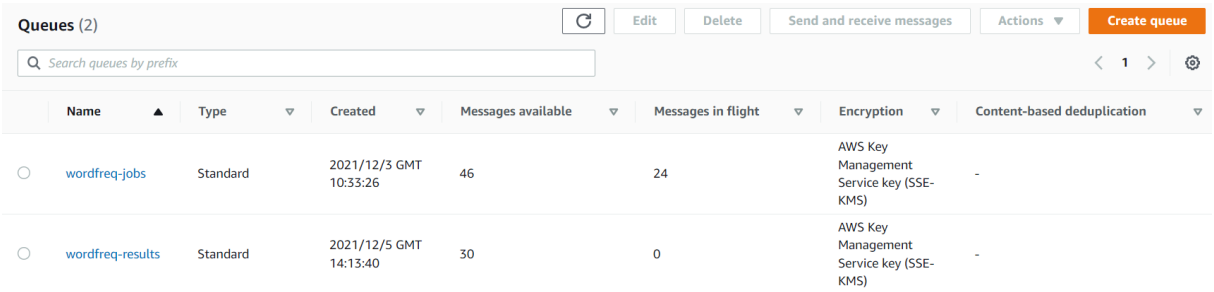

Figure.13 SQS queues during load testing

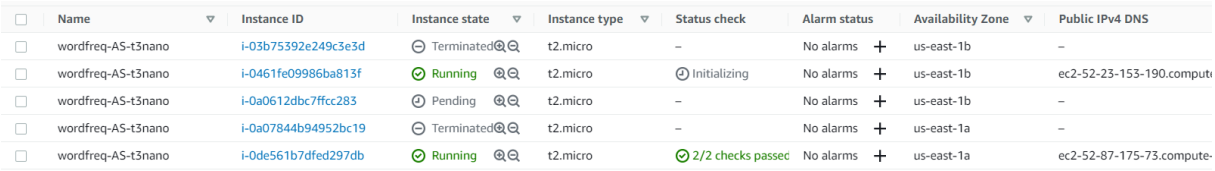

Figure.14 EC2 instance console during load testing

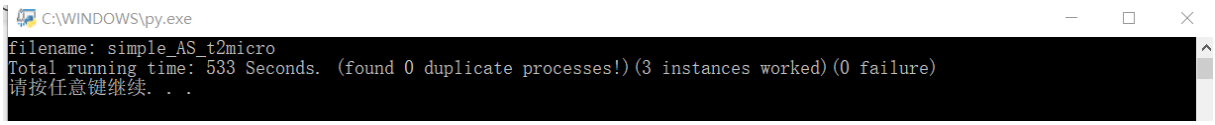

Figure.15 Auto scaling group load testing result

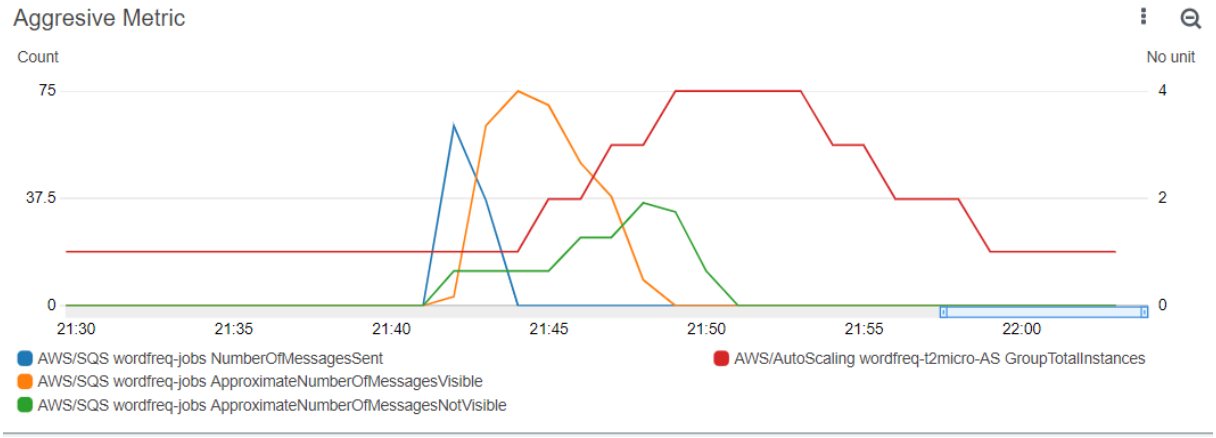

Figure.16 Metrics of SQS queue and auto scaling group during load testing

To improve the auto scaling group's performance, I changed the auto scaling policy to step scaling. The step scaling can set multiple thresholds to meet different demands. The detailed scaling out policy states as follows, and all the policy is spaced by 12. When the approximate number of visible messages is greater than or equal to 12 and lower than 24, the scaling group add 1 instance; when the metric is greater than or equal to 24 and lower than 36, and so forth. As for the scaling in policy, the removal strategy is too slow in the test. Therefore, the new policy is when the approximate number of messages not visible, remove 100% capacity in the group. The two policies' screenshot is shown in Figure.17.Then, I conducted a load testing by using the 100 small files to examine the performance of the auto scaling group. As shown in the aggressive metric graph, both the scaling out and scaling in results were faster than the simple scaling group. The total running time for the testing was 381 seconds, which was 40% less than the result of the simple scaling group.

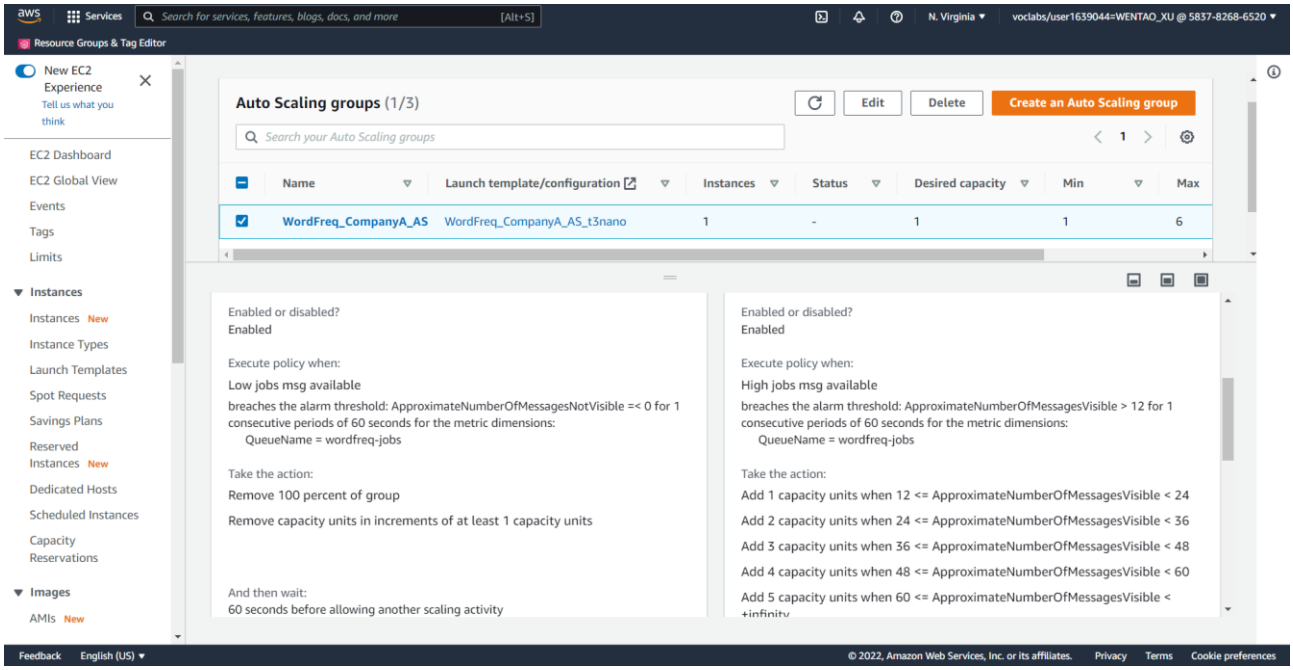

Figure.17 Modified auto scaling policies

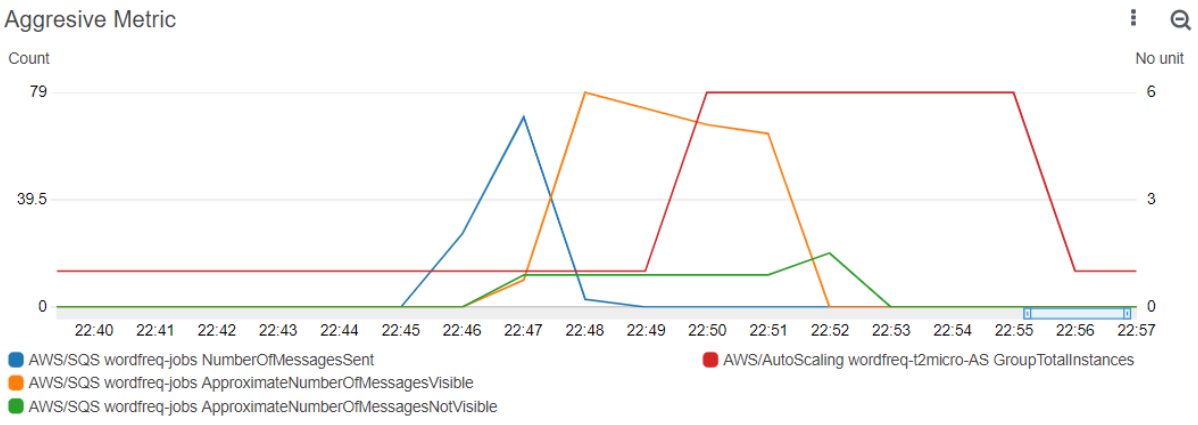

Figure.18 Selected metrics during modified auto scaling group's load testing

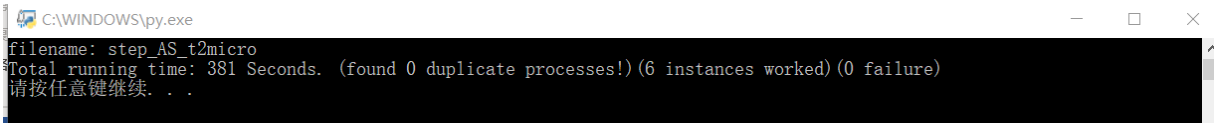

Figure.19 Modified auto scaling group load testing result

In this session, I chose the approximate number of messages visible and the approximate number of messages not visible for the auto scaling triggers. Then, the simple scaling strategy and the step scaling strategy have been implemented and compared. The result of the comparative shows that the step scaling strategy can effectively reduce the load testing running time.

EC2 Instance Type Experiment

AWS provides various types of EC2 instances, and the performance of the instance type may have a different impact on the application's performance. In this session, the result of the batch experiment that compares different EC2 instance types will be presented, and I will choose the most cost-effective one to be used as the default type of instance in the following session.

The batch experiment adopted the control variable method. The number of vCPU and the memory size were adjusted in the experiment. The running time was taken as the deciding metric for the experiment. To prevent the error caused by the CloudWatch, I did not use the auto in the experiment, and all the experiments were worked by one instance. To further reduce the error, three parallel experiments were conducted for each type of instance.

Table.1 provides the results obtained from the batch experiment. The t3.nano, t3.large, and t3.small have the same vCPU performance but different memory sizes. From the average running time of t3.nano, t3.large, and t3.small, it is apparent that the bigger memory size of the instance has no improvement for the testing. The types with larger memory sizes even cost more time on the testing. Compared to the results of t2.small and t3.small, a higher number of vCPU can slightly improve the application's performance.

Table.1 EC2 instance type experiment result

Instance Type	vCPU	Memory/ G	Test Files Group	Total Running Time/ s			Average Time/ s
t3.nano	2	0.5	50 Normal Files	522	522	522	522
t3.large	2	8	50 Normal Files	529	528	528	528.3
t3.small	2	2	50 Normal Files	527	529	526	527
t2.small	1	2	50 Normal Files	532	530	530	530.7

These results indicate that vCPU performance positively impacts application performance, while memory size has no effect. To sum up, the t3.nano, which has relative better CPU performance and lower price, is chosen to be the default type of instance for the following sessions.

Cloud Architecture for Company A

Company A wishes to utilise the WordFreq application for the large-scale processing of text files. The application needs to be implemented on the AWS cloud and meets the company’s requirements. The section below describes the complete cloud architecture for Company A, and the final cloud architecture diagram is shown in Figure.20. For clearer description, the session will be split into four parts, VPC network setting, auto scaling group configuration, non-VPC services setting, VPC endpoint setting, and encryption.

Before setting other services in AWS, it is necessary to build up a proper VPC networking environment. Company A requires a very secure and highly reliable application. Therefore, the VPC should include multiple available zones and private subnets. In the complete architecture, the VPC named wordfreq-VPC is set up in the us-east-1 region with a size /20 IPv4 CIDR block (10.0.0.0/20). In the VPC, two private subnets and one

public subnet are created. The private subnets are in available zone us-east-1a and available zone us-east-1c respectively. The public subnet is in available zone us-east-1c. To fully implement the public subnet, the internet gateway and the route table need to be configurated. As shown in the diagram, the internet gateway has been created, and the route table for the public subnet has also been modified.

So far, the basic VPC and subnets have been created, and the following part will discuss the auto scaling group. For application security, the two private subnets have been allocated to the auto scaling group. Consequently, the auto scaling group can launch the instances in both available zones to satisfy the high availability requirement. In addition to high availability, the auto scaling group also provides the application with high resilience. Through the auto scaling group healthy check, failed instances will be rebooted automatically. Besides, to ensure the cost-efficiency, the group use the cheap but effective type of instance, t3.nano. Last but not least, the auto scaling group is in the private subnets, and there is no way to connect the instances in the group. To connect to the instances, we need to create a bastion instance in the public subnet, and the inbound rule of the auto scaling group should add a rule that allows the bastion instance to connect the instances in the group with SSH.

In addition to the auto scaling group, the S3, SQS, and DynamoDB are indispensable components. Previous sessions have discussed the basic configurations of the three services, and the configurations can directly inherit from the sessions. Nevertheless, for better administration and security, the services access policies should be modified. Unluckily, the learner lab does not allow me to adjust the IAM roles and create new accounts. In future modification, child accounts need to be created for the company to use the application because it is not secure to use the root account. And the IAM rules and the key should be allocated to the child accounts in order to permit them to access S3 and DynamoDB.

Let us move on to consider the VPC endpoint setting. The S3, SQS, and DynamoDB are all not included in the VPC, and they normally need to be accessed by the internet, which is not a very secure way to transmit data. The VPC endpoint is the securer way for instances in VPC to access the services owing to no public internet exposure. The S3 endpoint and DynamoDB endpoint are both gateway type endpoints. The two endpoints create two gateway and edit the private subnets' route table. Therefore, the instances in the private subnets can access the services by the related VPC endpoints. In addition, the SQS endpoint is an interface type endpoint. The SQS endpoint interfaces are like instances located in the private subnets, and each interface allocates an IP address of the subnet. Besides, the interfaces need to open port 443 (HTTPS) to the VPC (10.0.0.0/20) by editing the interface security group.

This session has illustrated most of the architecture, and the last part is the encryption. Having set the private subnets in the VPC, the working instances are secure now. However, the S3, SQS, and DynamoDB are not in the private subnets. The services have potential risks for Company A. Because of this, the services need to be encrypted, and the AWS Key Management Service is perfectly suitable for the application. In the WordFreq application cloud architecture, a customer-managed key named wordfreq-encrypt has been created, and the services have been set to be encrypted by the key. During architecture construction, the S3 bucket could not send messages to the encrypted queue, nor could the working instances access it. After learning on the AWS knowledge centre, I understood that the key policy needs to be edited to allow the S3 and the EC2 instances to use the key[1].

In summary, the cloud architecture contains not only the fundamental components for the services but also the extremely secure networking environment. In addition, to further improve the resilience and performance, the well-designed auto scaling group is also included. Besides, the AWS cloud encryption services are utilised in the S3, SQS, and DynamoDB to make the architecture impregnable. Last but not least, although the learner

lab has no permission to modify and create IAM rules, the child account is also considered in the description. The complete cloud architecture is perfectly suitable for Company A.

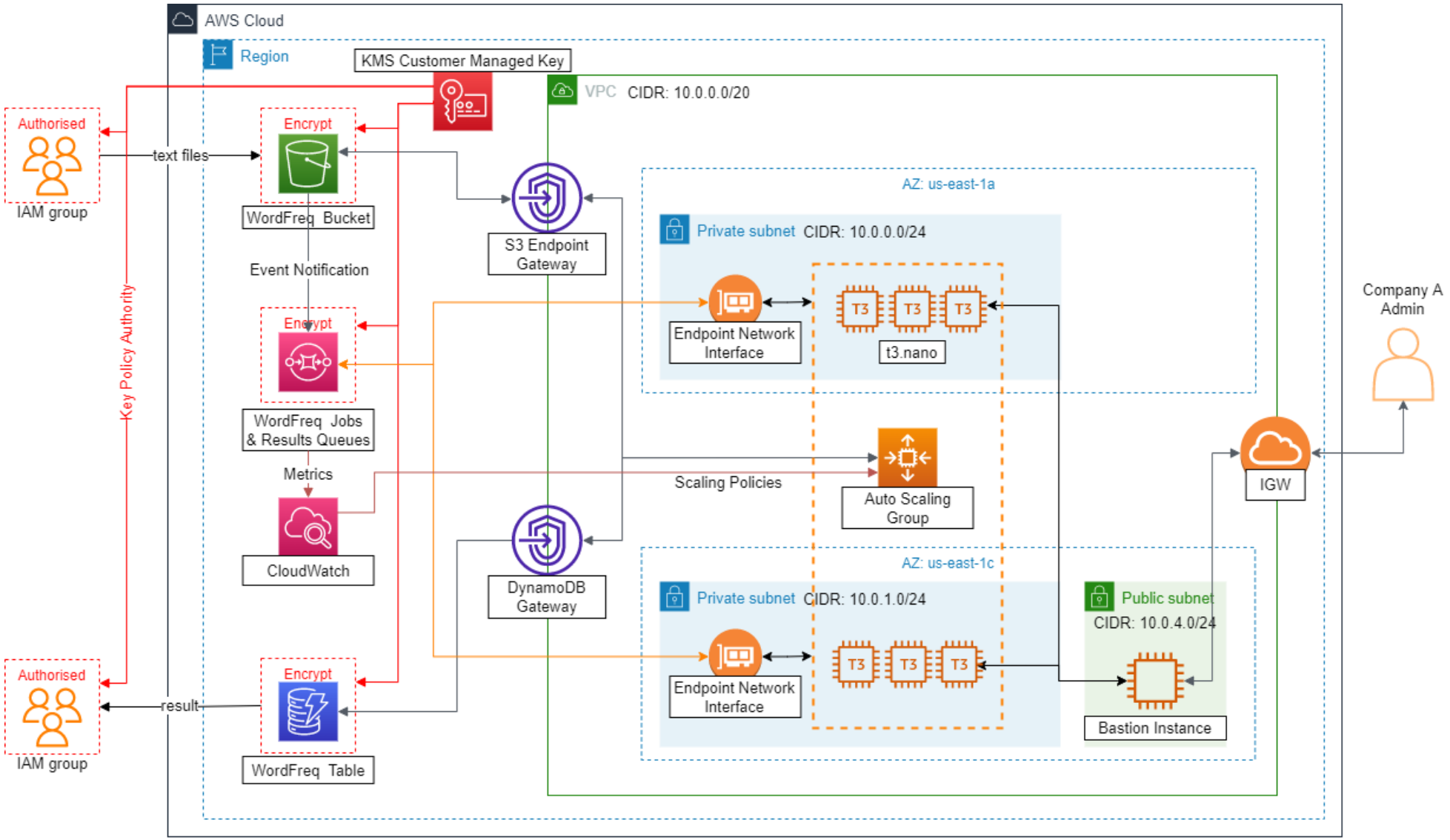

Figure.20 WordFreq architecture for company A

Future Improvements

This section will introduce two advanced data processing applications that can replace WordFreq applications to make the service more performant and robust. The first application is Hadoop MapReduce, and the second application is Spark.

The word counting application can be implemented by using Hadoop MapReduce. Hadoop is an open-source framework that provides distributed data processing tools, and MapReduce is a programming model for processing big datasets. The main advantage of MapReduce over WordFreq applications is distributed processing. Distributed processing can improve the processing performance and ensure high availability. In addition, the MapReduce is easier to code than the WordFreq by using Hive or Pig. If the word counting application is implemented by using Hadoop, it will have the ability to process extreme big files (gigabyte level) in real-time.

Spark is also a distributed data processing application. Like Hadoop, Spark can also conduct large-scale data processing via the MapReduce model. Therefore, Spark has all the advantages of Hadoop. Moreover, unlike Hadoop, the data will be stored in memory during the processing workflow, saving much time in reading and writing operations. Due to this feature, the processing speed in Spark is much higher than the Hadoop when the processing task requires a lot of iterations or I/Os. The word counting application does not require a high demand for iterations. However, if the company uses the application as a component in their workflow, Spark's benefits will come to the fore.

Reference

[1] AMAZON WEB SERVICE, Why aren’t Amazon S3 event notifications delivered to an Amazon SQS queue that uses server-side encryption? [online]. AWS, 2021. [viewed 03/01/2022]. Available from: https://aws.amazon.com/premiumsupport/knowledge-center/sqs-s3-event-notification-sse/