

# Task 1

It may be advantageous for JPMorgan Chase to move to Amazon Web Services. For as long as they utilise the services, they just have to pay for what they use. They don't have to spend a lot of money on servers or data centres. They are exempt from the requirement to purchase, maintain, and upgrade certain physical assets, such as data centres. They just have to pay for what they actually use and only when they use resources. They are able to exchange capital expenses for variable expenses in this manner. They can save a tonne of money on technology this way. They may concentrate on the fundamental objectives of their company, like improving customer service, instead of having to worry about maintaining data centres, servers, and other physical assets. Even on the variable costs, they will benefit from massive economies of scale, which will lower the variable costs for them. JPMorgan will no longer have to speculate about the capacity of their infrastructure. It will need a few minutes' notice to scale up or down as needed. The company's speed and agility can be significantly increased by moving to an AWS cloud environment because new IT resources can be used quickly whenever needed. Developers can experiment and develop applications with significantly lower cost and time, these will increase the agility of the company. JPMorgan can quickly develop new financial service applications or new features to existing applications and test them. JPMorgan is a multinational company and has world-wide operations. AWS can greatly help them in this aspect. AWS will allow them to deploy their application in multiple AWS regions around the world with just a few clicks., with lower latency and better experience for their customers. In this way, they go global in few minutes.

AWS is a secure cloud based platform offering global cloud-based products. This gives JPMorgan a broad set of choices. These services work together like building blocks. JPMorgan can select services from these different categories to develop new innovative applications. Aligning with their vision of creating new innovations in the financial services industry using technology.

They can use Amazon EMR for trading analytics. AWS Lambda and Amazon Elastic Kubernetes services for risk calculations. JPMorgan needs to use analytics to better serve their customers. They can build a modern artificial intelligence platform on AWS using Amazon SageMaker. Amazon SageMaker will enable them to create a platform to rapidly test and train machine learning algorithms. They can build a highly modern and secure AI platform for rapid experimentation. Such algorithms can be used in use cases like a robo-advisory service where they can incorporate natural language processing to make client interactions more personalized. In risk analysis they can use advanced machine learning models to increase accuracy and have a more comprehensive view of risk. They can invest in cloud data warehousing technology with Amazon Redshift to scale analytic capabilities more effectively in a modern environment. JPMorgan can use AWS Trusted Advisor to provide real-time guidance that will help JPMorgan provision their resources following AWS best practices. It will check JPMorgan's entire AWS environment and give them real-time recommendations with regards to cost-optimization, performance, Security, Fault tolerance, Service limits.

JPMorgan can find out an estimate of the money required to build a solution by using the AWS Pricing Calculator which can estimate monthly costs and identify opportunities to reduce monthly costs. It is vital for JPMorgan to maintain and strengthen the trust of their customers and secure their data. With AWS, JPMorgan do not need to think about the

security of physical assets like data centres. AWS is responsible for Hardware and software infrastructure, Network infrastructure and virtualization infrastructure. Besides that AWS also provides services to secure data stored in the cloud. JPMorgan can use AWS SHIELD, which is a managed distributed denial of service protection service that safeguards applications running on AWS. They can use AWS Identity and Access Management (IAM) services to handle authentication and to specify and enforce authorization policies to specify which users can access which services.

JPMorgan holds financial data and financial data is very sensitive. Now JPMorgan is directly responsible for the security of the customers data and must protect their privacy and confidentiality. If AWS or any other cloud service provider fails to maintain the security of the infrastructure and data breaches occur then JPMorgan will be responsible for it and may have to pay fines and/or compensation to the customers and in the process they will lose trust of the customers. In the case, a data breach occurs there will be huge loss and risks associated with it like the possibility of threats. Cloud infrastructure is often vulnerable to cybersecurity attacks and often to a single point of failure. There is a possibility that the cost of shifting to cloud may outweigh the potential benefits. When shifting to cloud, there is a need for app refactoring and a need for skilled cloud computing specialists or need to train the employees in usage of cloud services and resources. It might face vendor lock-in. There is a chance of non-compliance with rules and regulations. Financial data is sensitive and according to the law certain types of sensitive data must not be given to a third party or service provider and especially if the provider is headquartered in a different country. JPMorgan is considering to adopt to cloud, especially AWS. Now this will eventually result in a lack of direct control over the infrastructure and recovery will depend on the promptness of the cloud provider's actions, in this case AWS. JPMorgan will have a reduced control over the performance, quality and outcomes. Business processes can get suspended due to downtime of the cloud and if internet connectivity goes out, then JPMorgan will not be able to access any of the applications, server or data. So it requires constant and high-speed internet connection as a lot of bandwidth is required to download and upload large documents, resulting in additional costs for JPMorgan.

## Task 2

### **A brief summary of how the application works**

Our application is known as wordfreq. Wordfreq counts the number of each word in a text file and returns the top ten most frequent words found in a text document and can process multiple text files sequentially. Our application uses two s3 buckets dc-wordfreq-nov23-uploading and dc-wordfreq-nov24-wordfreq, as the name suggests one is used for uploading and storing original text files. To process our files, the user needs to copy the files from the uploading bucket to the processing bucket. The processing bucket has upload notifications or SNS enabled such that when a file is uploaded a message notification is added to a Wordfreq sqs queue and an email will be sent to the user. The application uses two kinds of sqs queues, one for holding notification messages of newly uploaded files, which are also known as jobs to be performed by the application, so it is Wordfreq-jobs queue and another one is the wordfreq-results

queue to hold messages of the “top 10” results of the processed jobs. These results are stored in dynamoDB, a NoSQL database.

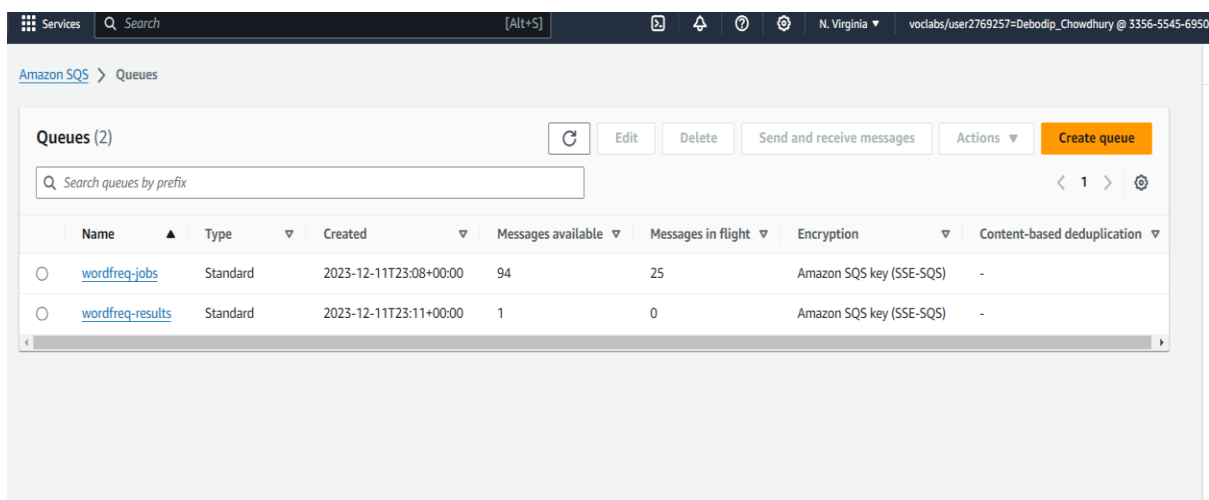
## Design process to architect the scaling behaviours

We architect and implement autoscaling functionality for the word frequency architecture, so that the application can scale up or down due to demand. Our objective is to upgrade the system so that it can work with high workloads like processing of many large files. First, I created an image for the instance and then launched a template. Then based on the launch template, I created an autoscaling group. We set load balancing to no and enable detailed monitoring with cloudwatch. We set the desired and minimum capacity to 1, this will keep costs minimum and maximum capacity to 6, to scale up when needed. Then we created two cloudwatch alarms for both removing and adding instances. To ensure that our autoscaling group does not add instances every two minutes, we later on set our period to 3 minutes, such that `ApproximateNumberOfMessagesVisible >= 2` for 1 datapoints within 3 minutes. We choose the `ApproximateNumberOfMessagesVisible` in the `wordfreq-jobs` queue to be processed. It is an indicator of the number of messages to be processed in the `wordfreq-jobs` queue. This is where messages are to be picked by the application and get processed. These alarms get triggered when a threshold is breached and cause instances to be launched or terminated. We specified an instance to be launched when there are greater than or equal to 2 messages to scale up and make data processing faster and an instance to be removed when there are less than 2 messages to save costs. This makes sure that instances are launched quickly.

## Testing and results:

We have set up our autoscaling infrastructure. Now we perform load testing on it so that it works. For this we uploaded 120 large text files to our uploading bucket and then copied them to our processing bucket, to test the behaviour of my application and have taken all the necessary screenshots. The autoscaling policies and alarms work smoothly and fast. The below screenshots prove that:

The screenshot below shows the SQS Queue page showing message status:



The screenshot shows the Amazon SQS console interface. At the top, there's a search bar and navigation links. Below, the 'Queues (2)' section displays a table with two queues. The first queue, 'wordfreq-jobs', has 94 messages available and 25 in flight. The second queue, 'wordfreq-results', has 1 message available and 0 in flight. Both queues are Standard type and use Amazon SQS key (SSE-SQS) for encryption.

Name	Type	Created	Messages available	Messages in flight	Encryption	Content-based deduplication
<a href="#">wordfreq-jobs</a>	Standard	2023-12-11T23:08+00:00	94	25	Amazon SQS key (SSE-SQS)	-
<a href="#">wordfreq-results</a>	Standard	2023-12-11T23:11+00:00	1	0	Amazon SQS key (SSE-SQS)	-

Figure 1: screenshot of SQS Queue page showing message status

The screenshot below shows the files being successfully copied to the processing bucket:

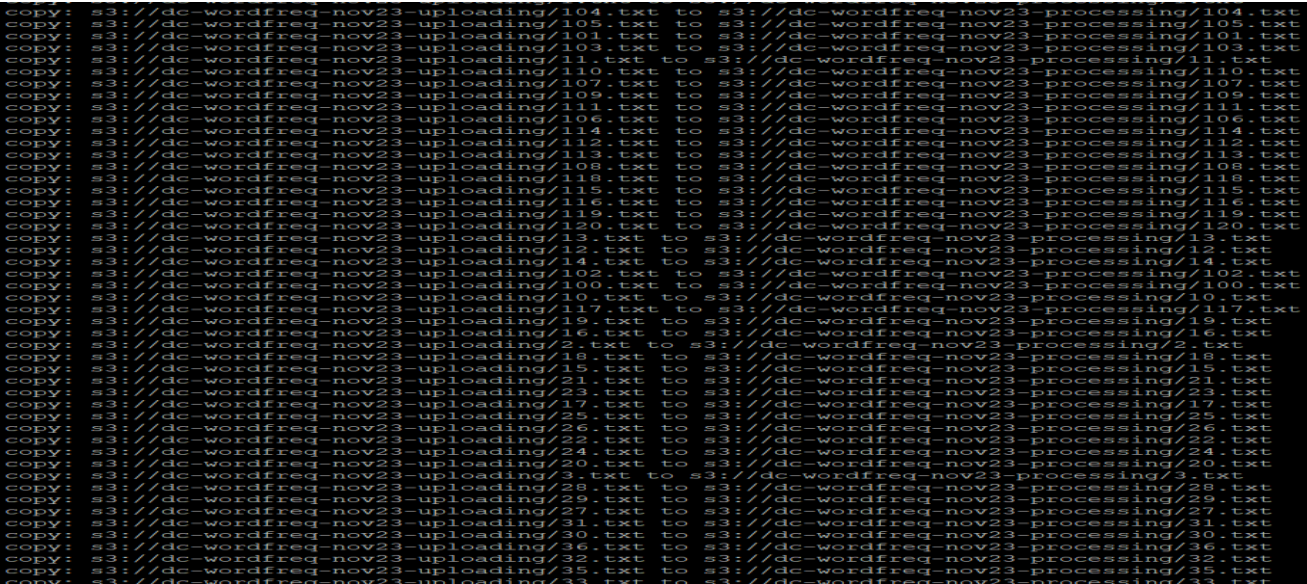


Figure 2: screenshot of files being copied

The screenshot below shows the file has been copied to the processing bucket:

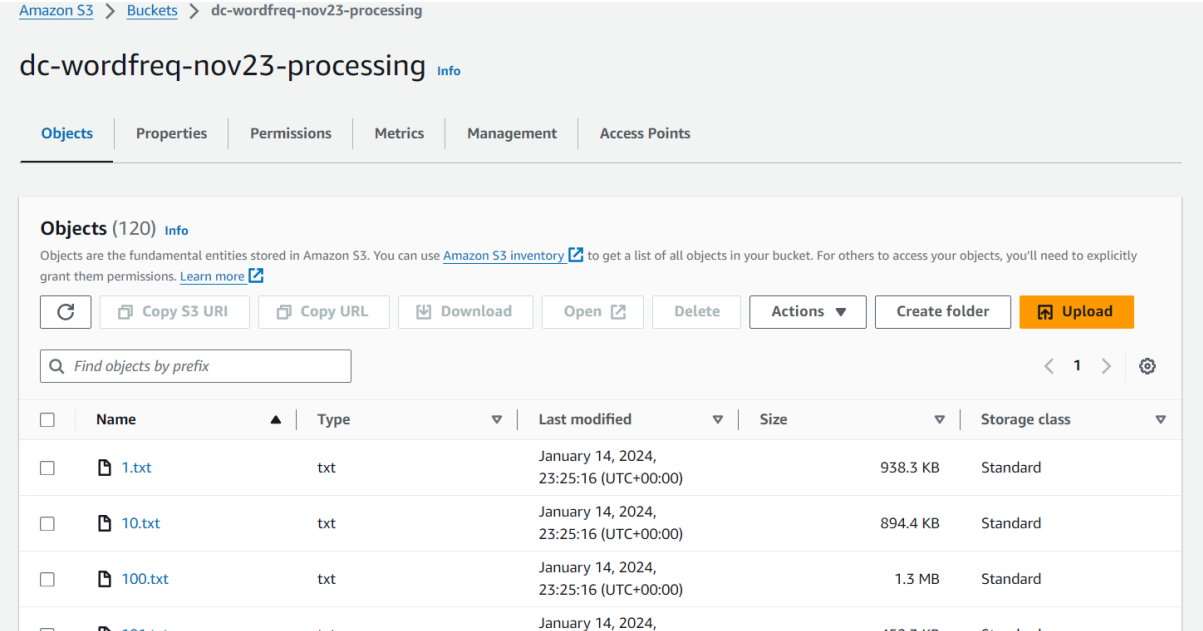
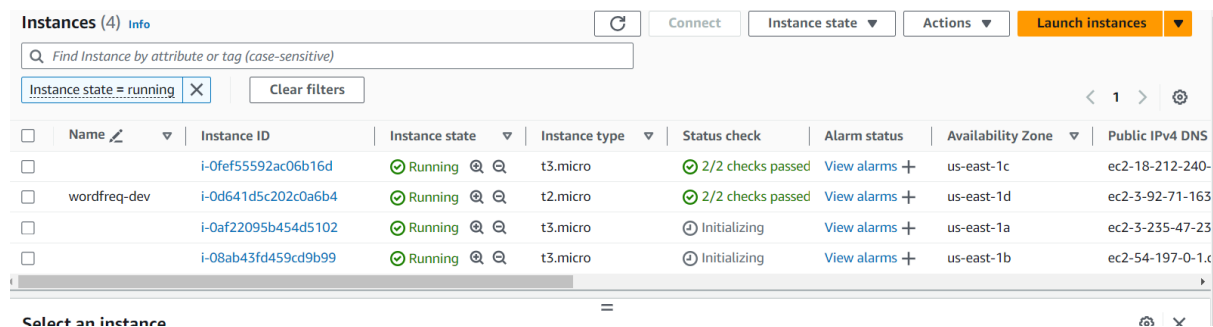


Figure 3: proof of files copied to s3 processing bucket

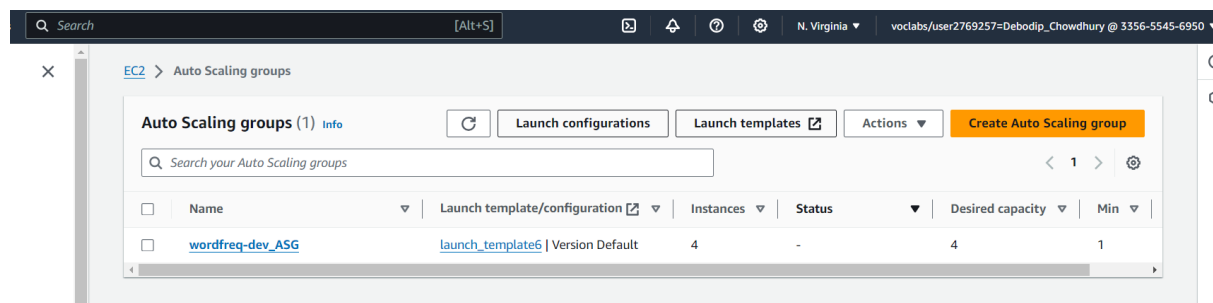
The screenshot below shows us the ec2 instance page:



	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
<input type="checkbox"/>		i-0fef5592ac06b16d	Running	t3.micro	2/2 checks passed	View alarms +	us-east-1c	ec2-18-212-240-
<input type="checkbox"/>	wordfreq-dev	i-0d641d5c202c0a6b4	Running	t2.micro	2/2 checks passed	View alarms +	us-east-1d	ec2-3-92-71-163
<input type="checkbox"/>		i-0af22095b454d5102	Running	t3.micro	Initializing	View alarms +	us-east-1a	ec2-3-235-47-23
<input type="checkbox"/>		i-08ab43fd459cd9b99	Running	t3.micro	Initializing	View alarms +	us-east-1b	ec2-54-197-0-1.c

Figure 4: Screenshot of EC2 instance page showing launched instances during this process.

The screenshot below shows us the autoscaling group page showing instance status:



	Name	Launch template/configuration	Instances	Status	Desired capacity	Min
<input type="checkbox"/>	wordfreq-dev_ASG	launch_template6   Version Default	4	-	4	1

Figure 5: AutoScaling group page showing instance status

The screenshot below shows us the emails received from Amazon S3 Notification:

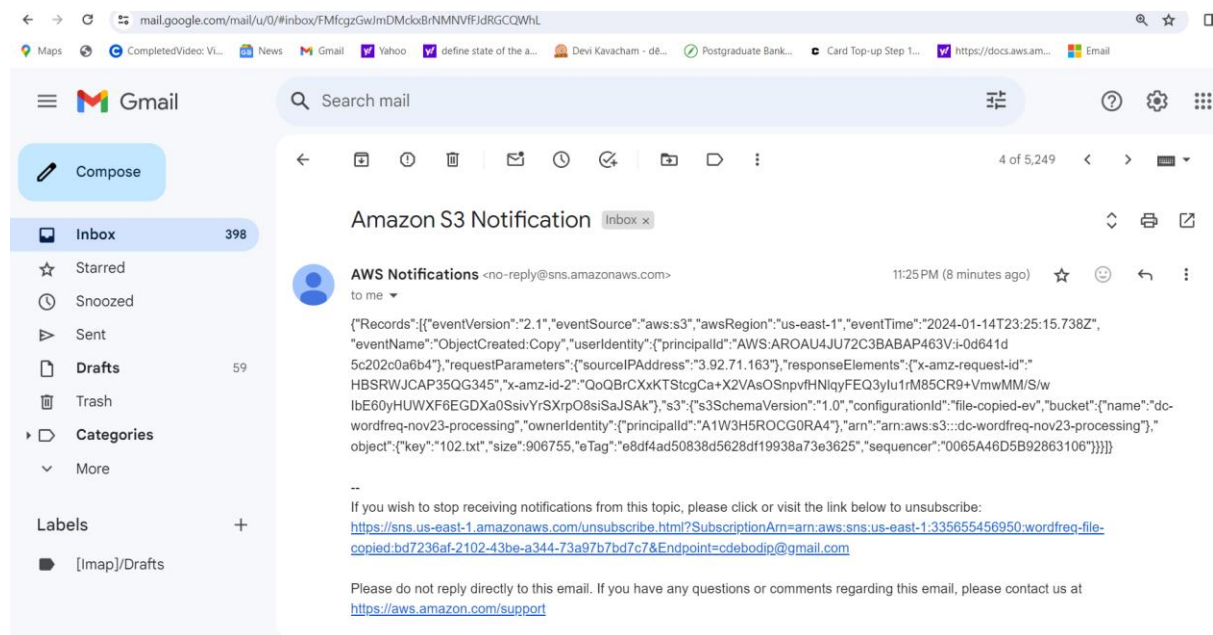


Figure 6: screenshot of the emails received from Amazon S3 Notification



The screenshot below show us the email received about launching of instance due to alarm:

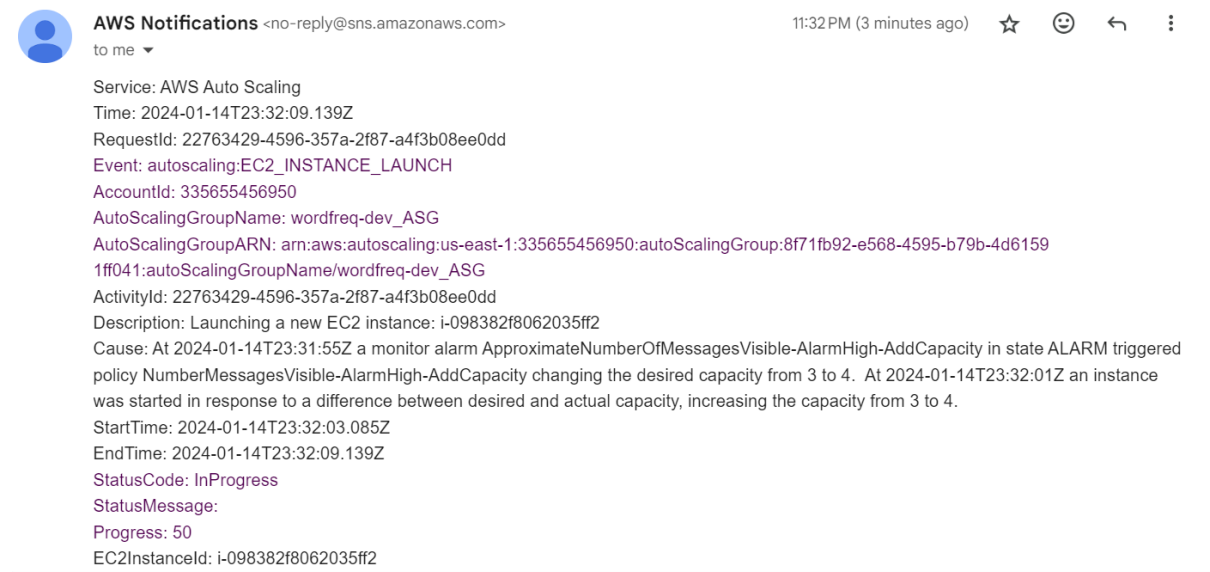


Figure 7: screenshot of email about launching of instance

The screenshot below show us the email received about termination of instance due to alarm:

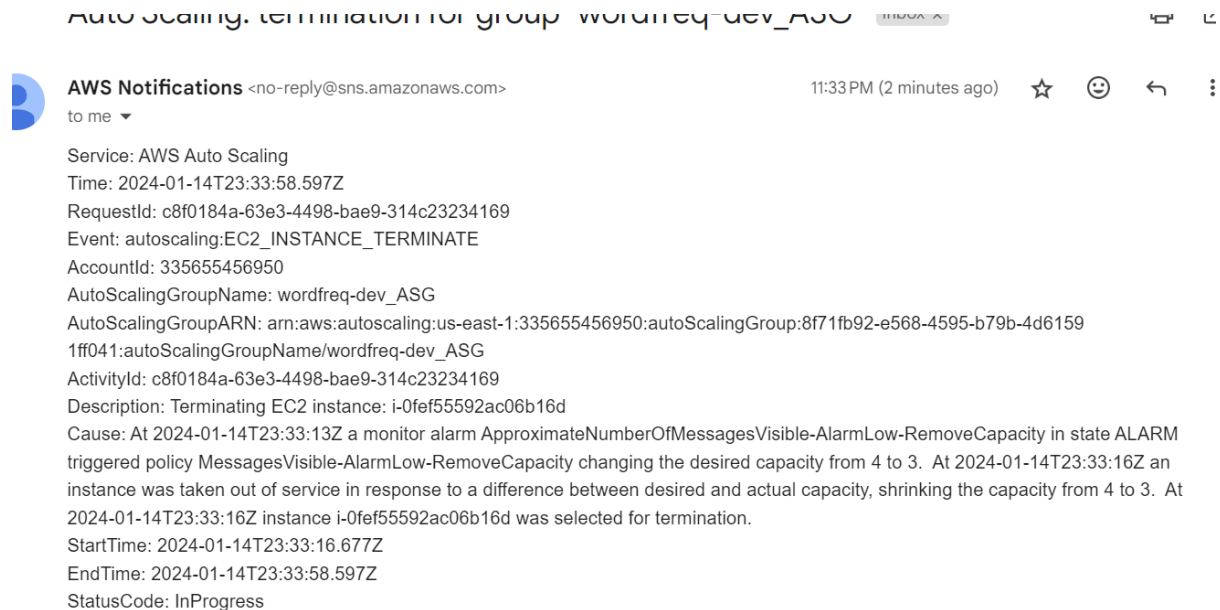


Figure 8: Screenshoot of email about termination of instance

We try to Optimising our autoscaling group, We do experiment for different setting like different combinations of threshold values and period values.

We choose the remove threshold as  $<2$  and add capacity threshold as  $\geq 2$

The processing time is the time required to publish all results in the nosql database. We measure time using the stopwatch, the time required for the results to appear in the database table

Time: 7 minute 30 seconds

We choose the remove threshold as  $<3$ , and add threshold as  $\geq 6$  and then measure the time

Time: 7 minute 37 seconds

We chose the remove threshold as  $<4$  and add threshold as  $\geq 10$ , and then measured the time.

Time: 7 minute 35 seconds

We chose the remove threshold as  $<8$  and add threshold as  $\geq 10$ . We record the processing time as given below:

Time: 7 minute 38 seconds

All the above experiments were with period = 1 minute, now we would like to increase the period of both alarms to 2 minutes

We chose the remove threshold as  $<8$  and add threshold as  $\geq 10$

Time: 7 minute 14 seconds

I repeat the same experiment above, but this time I set the remove instances alarm to 1 minute

We chose the remove threshold as  $<8$  and add threshold as  $\geq 10$

Time: 7 minute 37 seconds

Before this, the default cooldown was 60, we change it to 120

Time: 8 minute 01 seconds

It seems decreasing the period of the remove alarm somewhat increases the processing time as remove alarm acts quickly (within one minute). So we now increase the remove alarm to two minutes.

We chose the remove threshold as  $<2$  and add threshold as  $\geq 2$

Time: 7 minute 14 seconds

So fastest processing time achieved is 7 minute 14 seconds

Then we try using a few different instance types and we record the time each runs so that later on we can easily calculate the price of each experiment

We carry out the experiment for Instance type: T2.Large

For this instance type, we had to use launch\_template2

Processing time : 8 minutes 27 seconds

Instance 1:

9 minute 5 seconds

StartTime: 2024-01-17T19:35:04.290Z

EndTime: 2024-01-17T19:35:46.132Z

Instance 2:

10 minute 9 second

Instance 3:

8 minutes 32 second

We now repeat the experiment for Instance type: T2.Medium

For this instance type, we had to use launch\_template4

Processing time: 8 minutes 31 seconds

Instance 1:

10 minute 41 second

Instance 2:

6 minute 47 seconds

We now repeat the experiment for Instance type: T3.Micro

For this instance type, we had to use launch\_template6

Processing time: 7 minute 33 seconds

Instance 1:

9 minute 53 second

Instance 2:

9 minute 8 second

Instance 3:

9 minute 58 second

Now we Calculate the cost for each experiment

Cost for experiment 1

Instance 1:

- Rounded Runtime: 10 minutes

- Cost for Instance 1: \$0.015467



Instance 2:

- Rounded Runtime: 11 minutes
- Cost for Instance 2: \$0.0170136

Instance 3:

- Rounded Runtime: 9 minutes
- Cost for Instance 3: \$0.0131008

T2.micro:

- Rounded Runtime: 9 minutes
- Cost for T2.micro: \$0.001813

Total Cost:  $\$0.015467 + \$0.0170136 + \$0.0131008 + \$0.001813 = \$0.0473944$

So total cost for experiment 1 is \$0.0473944

Cost for experiment 2

Instance 1:

- Rounded Runtime: 11
- Cost for Instance 1: \$0.01746666667

Instance 2:

- Rounded Runtime: 7 minutes
- Cost for Instance 2: \$0.00881866667

Instance 3:

- Rounded Runtime: 9 minutes
- Cost for Instance 3: \$0.013228

T2.micro:

- Rounded Runtime: 9 minutes
- Cost for T2.micro: \$0.001813

Total Cost:  $\$0.01746666667 + \$0.00881866667 + \$0.013228 + \$0.001813 = \$0.04133533334$

So total cost for experiment 2 is \$0.04133533334.

Cost for experiment 3

- Rounded Runtime: 10 minutes
- Cost for Instance 1: \$0.001733333333

Instance 2:

- Rounded Runtime: 10 minutes
- Cost for Instance 2: \$0.001733333333

Instance 3:

- Rounded Runtime: 10 minutes
- Cost for Instance 3: \$0.001733333333

Another Instance:

- Rounded Runtime: 8 minutes
- Cost for Another Instance: \$0.00154666667

Total Cost:  $\$0.001733333333 + \$0.001733333333 + \$0.001733333333 + \$0.00154666667 = \$0.00674666666$

So total cost for experiment 3 is \$0.00674666666

So we see that in terms of the processing types having instances with greater cpu and/or memory does not fasten the processing times of our application and also increases the cost greatly.

## Architectural description

Now, we try to optimise our wordfrequency architecture. We should use Amazon DynamoDB for storing metadata and intermediate results. We should enable DynamoDB for cross-region replication to enhance availability. We should turn on point-in-time recovery so that the data is backed up automatically and can be restored to any second in the preceding 35 days. This is called PITR. We should use S3 for data storage. We should configure versioning on s3 bucket to track changes and recover from accidental deletions. S3 is a cost effective We should use Lambda for periodic backups. Store input data and results We should encrypt data in transit from s3. We can use Amazon s3 glacier for archiving older data like older input files. Using amazon ec2 spot instances. Usage of ebs snapshots is a

good way to create regular data backups. We can create lambda functions to process the data when a file is uploaded to s3. We should use instances with T3.micro this will be more cost effective, this will slow down the data processing performance of the application as our processing does not need to be immediate. In order to increase availability, we should monitor the system for key performance indicators and configure our system to trigger an automated recovery whenever a threshold is breached. We can use Amazon CloudWatch to collect and track metrics, set alarms and then implement AWS AutoScaling to implement automated recovery. We should monitor metrics such as CPU utilization, memory utilization, incoming and outgoing network traffic, storage input/output, storage capacity, Database read/write latency, Database connections. We need to test our recovery procedures by simulating failure scenarios. We can create lambda functions to automate the failure scenarios, for instance by writing lambda functions that terminate EC2 instances or modifies security group rules. We should scale horizontally to increase aggregate workload availability. This reduces the impact of single point of failure on the overall system. So we will use AutoScaling groups and an elastic load balancer to distribute incoming traffic across multiple instances. We will use our elastic load balancer to perform health checks and replace unhealthy instances. We will use Amazon CloudFront to distribute content globally. We need to use the cross-region replication to ensure data availability when the entire region becomes unavailable. We should deploy our AWS resources across multiple availability zones. This ensures if one availability zone experiences an issue, other availability zones can continue to operate. We should replicate our database across multiple regions or multiple availability zones to ensure that data instance remains available even if one instance fails. With regards to data performance processing, we will use the auto scaling to automatically adjust the number of EC2 instances based on demand. If we use a load balancer it will increase the data processing performance as it distributes the network traffic and prevents overload on a single server. This ensures high availability and cost efficiency. Our system will be cost-effective as we are using one EC2-micro instance. During auto scaling we will keep the desired capacity as low as possible like one or two instances and ensure that EC2 instances are launched quickly and terminated soon after use. We can ensure these by having suitable scaling out and scaling in policies, allow more frequent monitoring and collection of required metrics and having less cooldown period. In auto scaling group, we should use spot instances that will optimize cost savings. Additionally, we can use redundant EC2 instance to remove a single point of failure and thus increase redundancy. Instead of using one large EC2 instance, we should use smaller instances. It is wise to set up Billing alerts to try to avoid over-reaching our estimated budget, I will use AWS Trusted Advisor. We use a VPC and divide it into many subnets. We use AWS Identity and Access Management. Users should only have permissions necessary to perform their tasks. We should encrypt data in rest and at transmit. We should use virtual private cloud and network access lists. Security groups should be used. We should use multi-factor authentication. I will use AWS Key Management Service. Multiple network interfaces for high availability as in event of failure traffic can be redirected to the instance in another availability zone. This works with elastic load balancer to distribute traffic across multiple IP addresses. I will implement a NAT gateway. A NAT gateway is service that enables instances in a private subnet within a virtual private cloud to connect to the internet and prevents inbound traffic initiated by connections outside the VPC. To further enhance the security we should create a Network ACLs, and it is a virtual firewall that controls traffic at the subnet level. We will create VPC. We will use VPC endpoints as they help reduce NAT gateway charges and improve security by accessing S3 directly from the VPC so that we can have our own virtual network to launch AWS resources. VPC allows us to have full control over our network environment and enhances security as they allow us to use security groups and network access control lists to control inbound and outbound traffic to and from the instances. We

will have private subnets to secure backend resources that don't need public access. Below is architecture diagram for vpc:

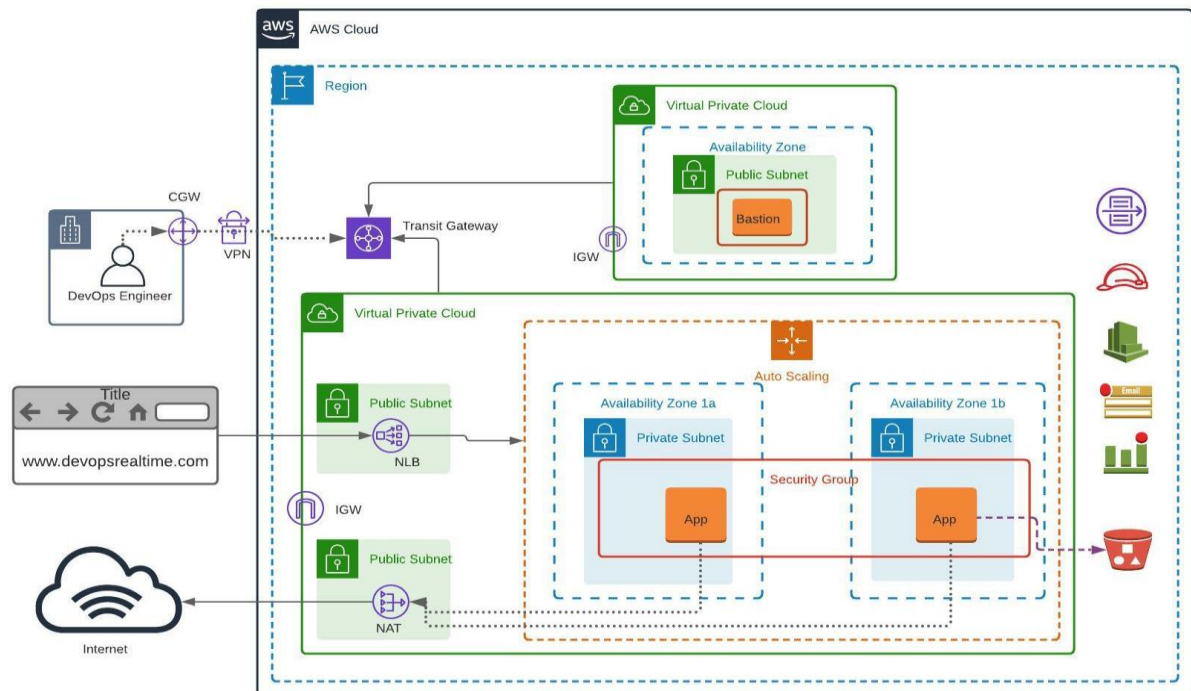


Figure 9: vpc architecture diagram

We have implemented and explored most of the aforementioned to our application. We use the amazon DynamoDB to store the result of processing files because DynamoDB is a nosql database and our results has an unstructured formatmaking it unsuitable for noSQL databases. We enabled DynamoDB for cross-region replication to enhance availability. We enabled cross-region replication for our table. For this we had to shift to on-demand capacity mode under the "Capacity" tab. Then we replicated our word-freq table to Europe(London). I have set the encryption option to Owned by Amazon DynamoDB for simplicity. I have enabled point in time recovery for our dynamodb database. This is illustrated by the screenshot below:

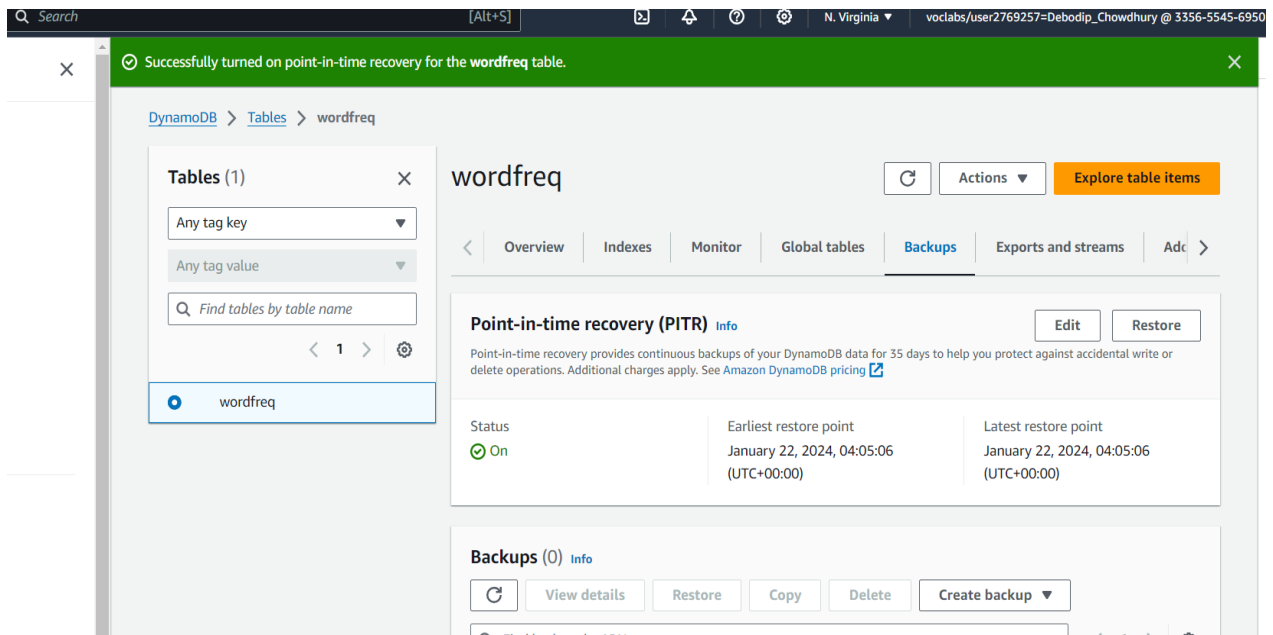


Figure 10: point-in-time-recovery(PITR) enabled

I have created S3 buckets to store our uploaded files and named it dc-wordfreq-nov23-uploading. In S3 buckets, we explored using dual-layer server-side encryption with AWS key management service keys. This protects our objects with two separate layers of encryption but I did not choose that to save costs. Then I explored creating my spot instances by checking the request spot instances under the “Advanced Details” section. But I am not permitted to launch spot instances so couldn’t launch it. I have created an ec2 snapshot of volume 30 GiB, called snap-0f648825a4579f5d5. I have explored how to create a lambda function to attach to the S3 processing bucket using the lambda service in the AWS management console. I did not implement it as I do not have permission to change application’s functionality or code. I have enabled detailed monitoring with cloudwatch in the autoscaling group. I have created an application load balancer and attached it to our autoscaling group. We named it wordfreq-app-loadbalancer. For this we created our own wordfreqloadbalancerTargetgroup.

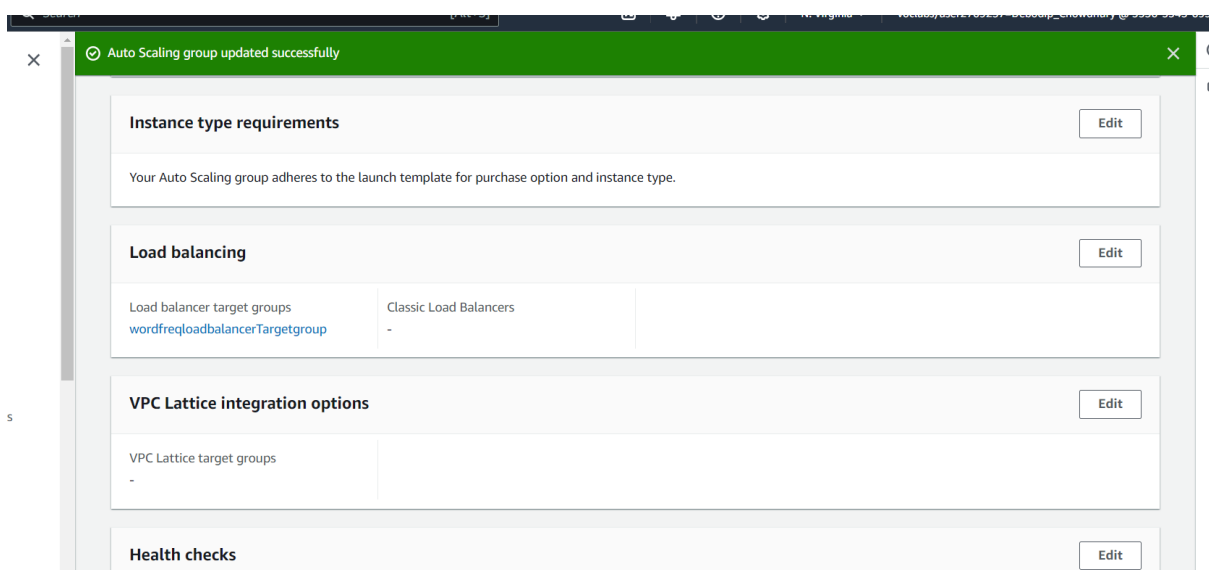


Figure 11: Autoscaling group with the load balancer added

We created our wordfreq-vpc. We set the NAT gateways to be 1 per availability zone and we used vpc end points. We set private subnets to 3. We set VPC endpoints to S3. Below is the preview of our wordfreq-vpc:

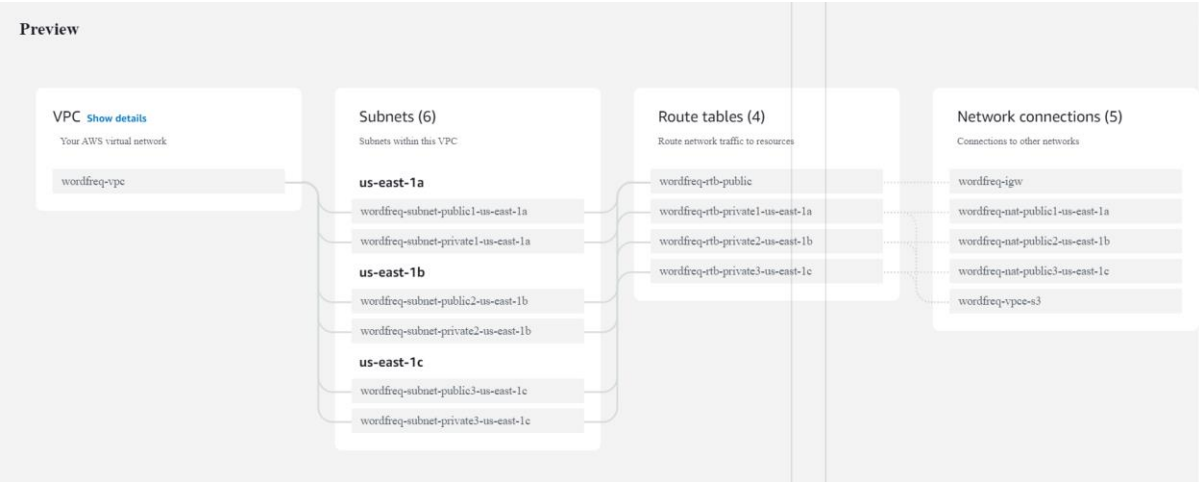


Figure 12: wordfreq-vpc preview

We can view our subnets like below:

Subnets (12) Info						Actions	Create subnet
Find resources by attribute or tag							
<input type="checkbox"/>	Name	Subnet ID	State	VPC	IPv4 CIDR		
<input type="checkbox"/>	wordfreq-subnet-private1-us-east-1a	subnet-0a8d1e614c095a489	Available	vpc-057efb186dab9bdc3   wor...	10.0.128.0/20		
<input type="checkbox"/>	wordfreq-subnet-private3-us-east-1c	subnet-0b72d26a37debb292	Available	vpc-057efb186dab9bdc3   wor...	10.0.160.0/20		
<input type="checkbox"/>	wordfreq-subnet-private2-us-east-1b	subnet-09b4cc682eaf5e20	Available	vpc-057efb186dab9bdc3   wor...	10.0.144.0/20		

Figure 13:subnets in our vpc

I have implemented a network ACL for wordfreq-vpc.

I have explored and implemented using Amazon cloudfront to distribute network globally with low latency and high transfer speeds. I chose our elastic load balancer as origin so that our application can serve dynamic content and also it ensures that cloudfront is routed to healthy dynamic servers. I set it to use all edge locations based on best performance and enabled security protection from AWS Web application Firewall. In the cloudfront settings I also turned on the SQL protections to block malicious request patterns that attempt to exploit sql databases like SQL injection.

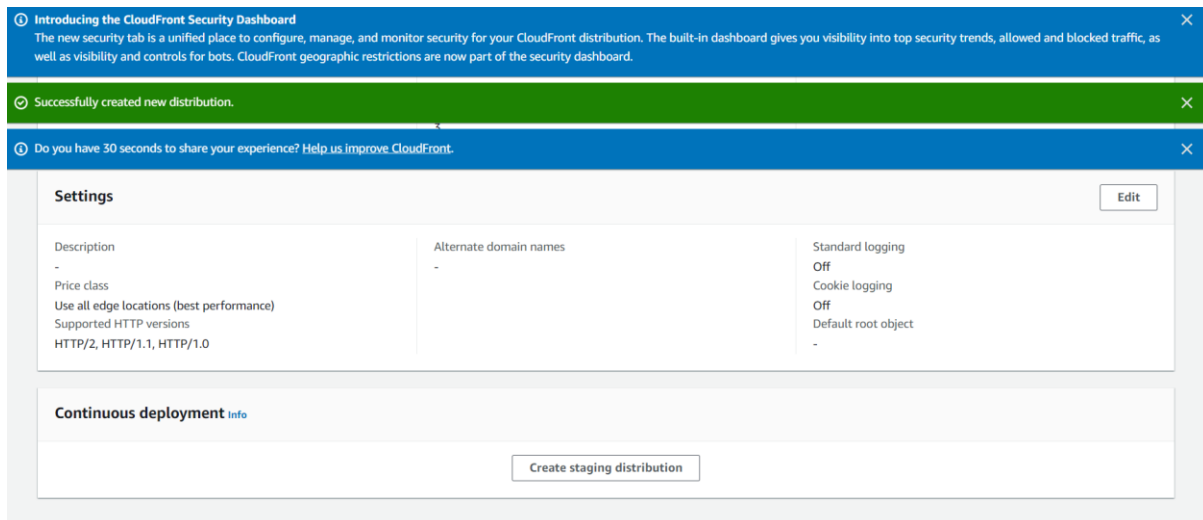


Figure 14: Amazon cloudfront distribution

Amazon cloudfront is a content delivery network that help improve the performance of my application without the need for large scale infrastructure.

We have implemented our aws autoscaling group and named it word-frq-dev-ASG. We explored creating IAM users and groups but were not permitted to do that.

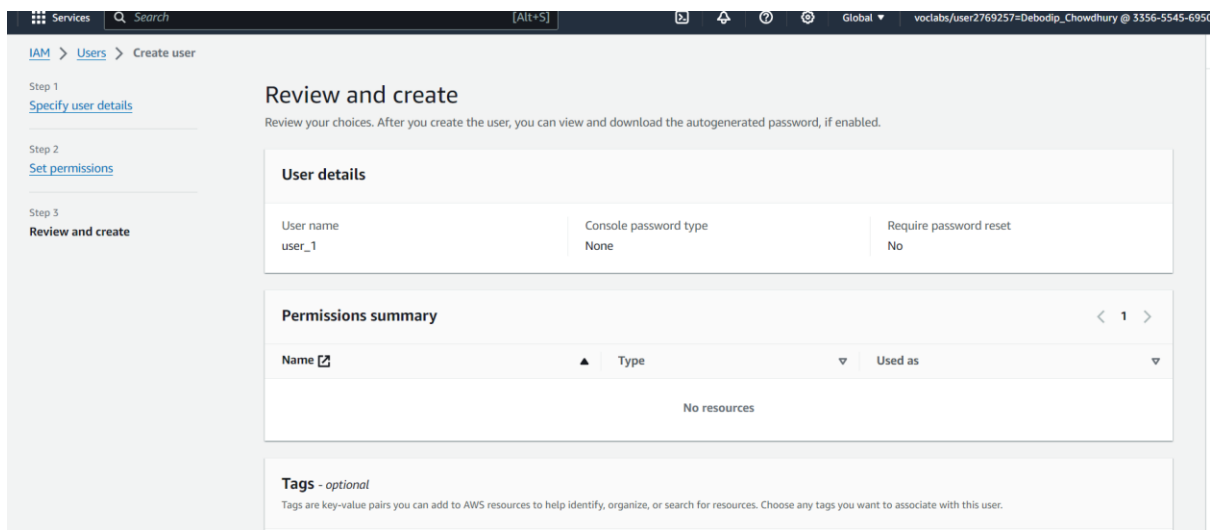


Figure 15: IAM

We could not create multi-factor authentication as we were not permitted to create IAM user.

Also we tried to explore budgets to set up billing alerts but could not access that.

We have explored S3 glacier and set lifecycle policies in dc-wordfreq-nov23-uploading-bucket and set storage class transition to Glacier Instant Retrieval and set to 36 days after which objects become non-current to make our application efficient for occasional use. We named our lifecycle rule as transition\_old\_data. The below image illustrates this:



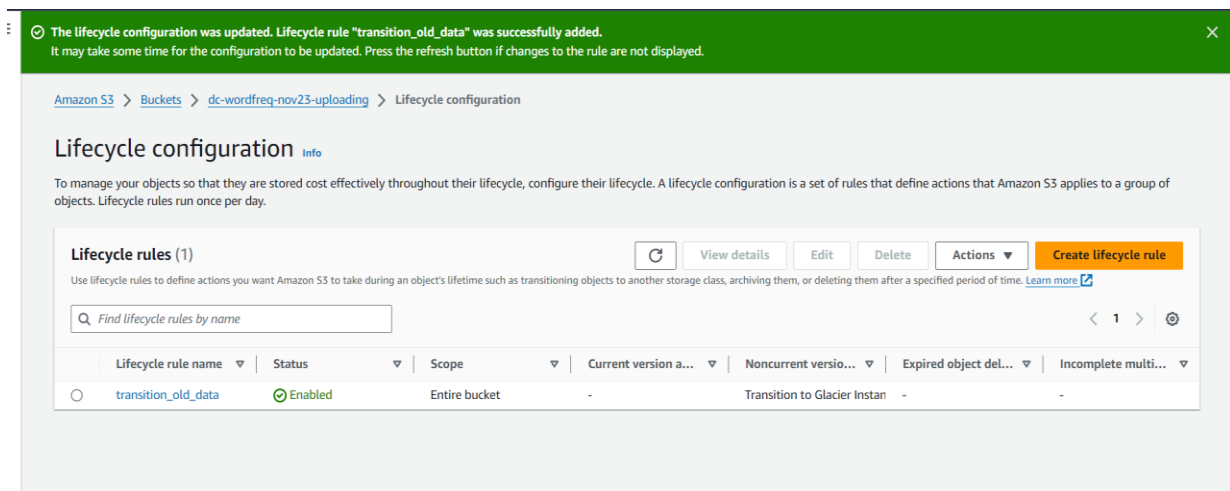


Figure 16: our lifecycle rule

In my dc-wordfreq-nov23-uploading-bucket I have enabled versioning to keep regular backups of data. This allows us to preserve, retrieve and restore very version of every object stored in our Amazon S3 bucket. The below image illustrates this:

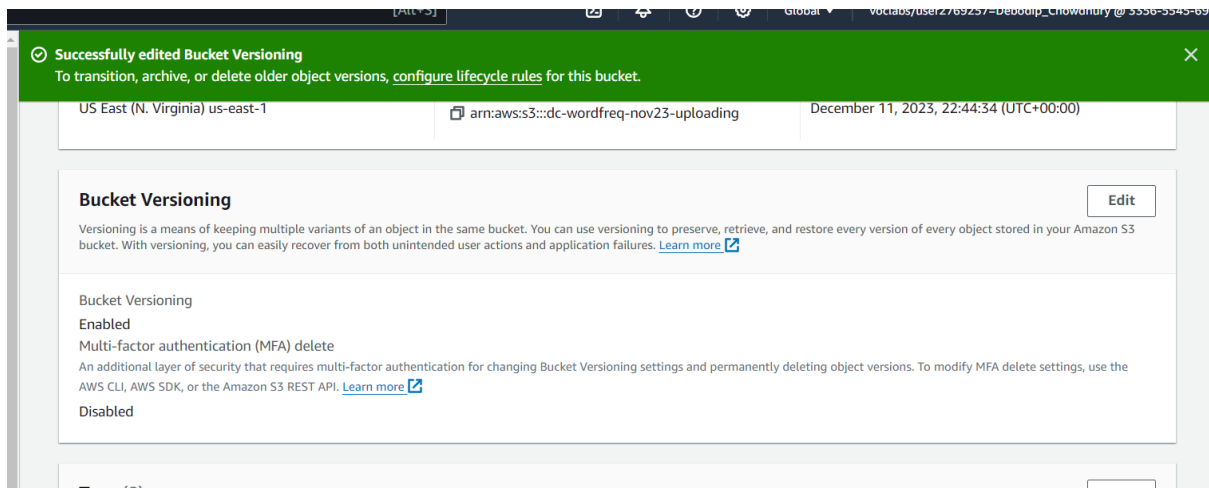


Figure 17: bucket versioning enabled

Because we are not permitted to access billing information for this account, we were unable to determine if the decisions we took and the extra changes we introduced were optimised in terms of the price or billing, like which resources is contributing to how much extra cost and explore saving opportunities. We tried to explore the billing and cost management dashboard as shown below:

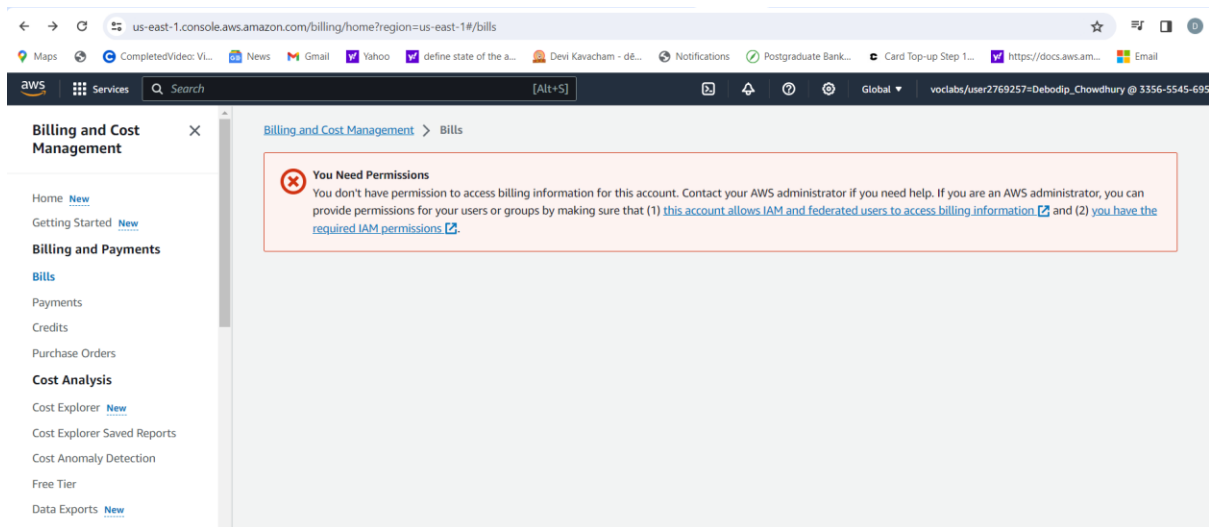


Figure 18: billing and cost management

## Issues

I faced an issue during the experiments, I had to decide how to measure the processing times I used a stopwatch to measure the processing time and I also had to make decision on what to measure. I decided my endpoint to be when all the results of the processing the files have appeared in the dynamo table. This gives us an accurate picture of the file processing time.

## Further Improvements

For this particular processing task, we have some other alternatives. There is AWS EMR. EMR means 'Elastic Map Reduce'. We can launch a Hadoop cluster using AWS EMR. We can write a mapper that will read every line of the text file, split them into tuples like (something, 1) and output tuples for each word. These pairs are then sorted so that all the values pertaining to a given key end up in the same node before the Reduce step. Then we write the reducer which takes multiple outputs of the mapper function and combines them. Then we put our data or text file in the Hadoop Distributed File System (HDFS). It is used to scale a single Hadoop to large number of nodes while ensuring high data storage reliability. Then using the Hadoop streaming Jar file we run the Map and reduce function on the distributed data. We can modify the Map and reduce function to return the top ten most mentioned words instead of returning all words. Using AWS EMR will greatly improve Scalability allowing us to dynamically add or remove nodes. AWS EMR takes care of managing the cluster enabling us to focus more on focusing. The integration with Hadoop distributed file system for distributed storage increases data storage reliability. AWS EMR can rapidly process large amounts of data like 'Big Data'. We consider another way to do this task. We can use Apache Spark to perform these data processing task. Spark has in-memory processing capabilities that can significantly speed up iterative algorithms like the MapReduce function. It has a rich ecosystem that includes libraries for machine learning. We create a Spark cluster on AWS EMR. After launching this cluster, check 'inbound rules' for SSH traffic, if there isn't any we add rule to allow SSH traffic. Then we connect to master node via SSD and start a new pyspark shell session and import data into Spark dataframe. We

implement the mapping and reducing steps using Spark transformations and actions. In both Hadoop and spark we have the ease of using rich ecosystems like machine learning, graph processing. We get to use unified processing engine. Both Spark and Hadoop will handle the scalability and cluster management for us. We can integrate with AWS EMR and in both of these we get the advantage of parallel processing making our data processing task very fast compared to the wordfreq.

**AWS academy username: [cdeboodip@gmail.com](mailto:cdeboodip@gmail.com)**

**AWS academy password: Dc\*141201**