

Part 2

In part 2, we are going to scale an application on AWS. We are going to install the application on the cloud, design and implement auto-scaling, perform load testing, and finally secure and optimize its architecture.

We begin by giving a brief summary of how the application works. The application is called "WordFreq" which is used to return the top ten most frequent words found in a text file. First, we upload the text files we want to analyze in an S3 bucket, which is a service that is used to store data. The S3 bucket then sends a notification to the SQS, which is a message queueing service. There are two queues used for the application, one is wordfreq-jobs which is used for the analyses to be done with the location of the text files, and one for the processed analyses with the top ten results. When there is a message in wordfreq-jobs an EC2 instance is being "triggered" to analyze the text file. An EC2 instance is a service that provides compute capacity on the cloud, in our example the application runs on an Ubuntu Linux EC2 instance. The output of the analysis is being stored in a DynamoDB, which is a NoSQL database table. The application has a ten-second delay after an analysis is finished before it begins with the next one.

To install the WordFreq application we followed the instructions that were available to us when this coursework was published. Hence, we are not going to extend on this matter.

We continue with the design and implementation of auto-scaling. In the beginning, we create an auto-scaling launch configuration, giving it a name, selecting the AMI we created when we were installing the application, an instance type (t2.micro on our occasion), and the security group we created during the installation of the application, choosing the key pair we downloaded during that time as well, and finally enabling EC2 instance detailed monitoring within CloudWatch. After that, we proceed to create an auto-scaling group. We begin by choosing a name (wordfreq-autoscaling-group) and selecting the launch configuration we just created. Then, we choose our VPC, all the available subnets (six in total), and we enable the "group metrics collection within CloudWatch" option. We select the desired and minimum capacities equal to 1, the former because we don't want more than one instance running when there are no jobs available due to cost-effectiveness reasons, and the latter because we need always at least one worker instance available to process when the application architecture is live. We choose the maximum capacity to be three, as our coursework mentions to have a maximum of three instances running at once. We finish our auto-scaling group creation by adding a tag of "Name" with the value of wordfreq-autoscalinginstance, so the instances that are created by auto-scaling have that specific Name. To terminate the most recent instance created, we navigate to our wordfreq-autoscaling-group → details tab → edit advanced configurations → Termination policies: Newest instance. Next, we create CloudWatch alarms, so when a specific event is taking place to trigger an alarm. We create two CloudWatch alarms, one for scaling-out and one for scaling-in. The following configuration is the one that returned the best time when we were testing the auto-scaling configurations. For the scaling-out alarm, we choose the SQS metric NumberOfMessagesReceived for the wordfreq-jobs queue with the following parameters:

- Statistic: Sum
- Period: 1 minute
- Whenever NumberOfMessagesReceived is: Greater/Equal
- than...: 1
- Alarm name: wordfreq-messagesreceived-scaleup

Similarly, we choose the same metric for scaling-in, with the only differences:

- Whenever NumberOfMessagesReceived is: Lower/Equal
- Alarm name: wordfreq-messagesreceived-scaledown

To finish with the design and implementation of auto-scaling we need to create policies for scaling-out and scaling-in to our auto-scaling group with the alarms we just created. We select wordfreq-autoscaling-group and under the tab "Automatic scaling" we add our first policy. For our scale-out policy, we select:

- Policy type: Simple scaling
- Scaling policy name: wordfreq-jobs-messagesreceivedpolicy-scaleuppolicy
- CloudWatch alarm: wordfreq-messagesreceived-scaleup
- Take the action:
 - Add 2 capacity units
- And then wait: 0 seconds before allowing another scaling activity

For our scale-in policy, we select:

- Policy type: Step scaling
- Scaling policy name: wordfreq-jobs-messagesreceivedpolicy-scaledownpolicy
- CloudWatch alarm: wordfreq-messagesreceived-scaledown
- Take the action:
 - Remove
 - 1 capacity unit when $2 \geq \text{NumberOfMessagesReceived} > 1$
 - 1 capacity unit when $1 \geq \text{NumberOfMessagesReceived} > -\infty$

That was the last step needed to design and implement auto-scaling, so we now proceed with the load testing.

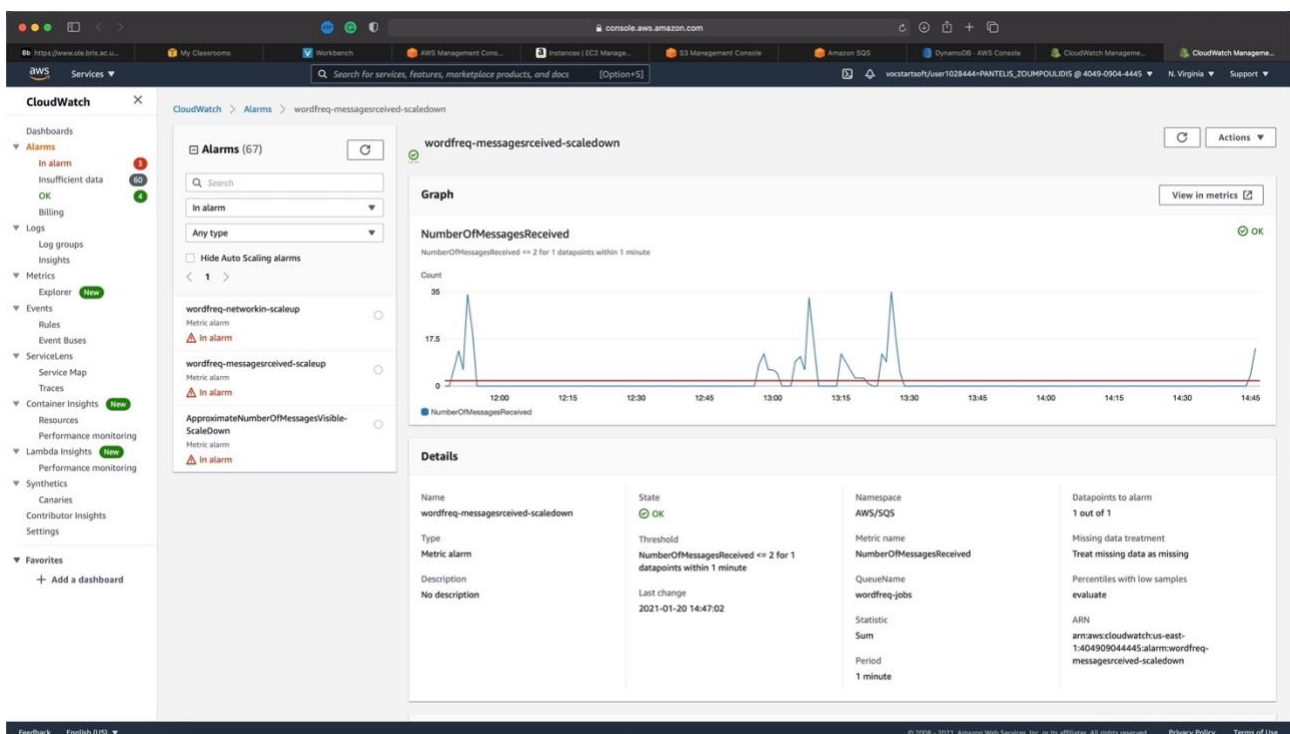
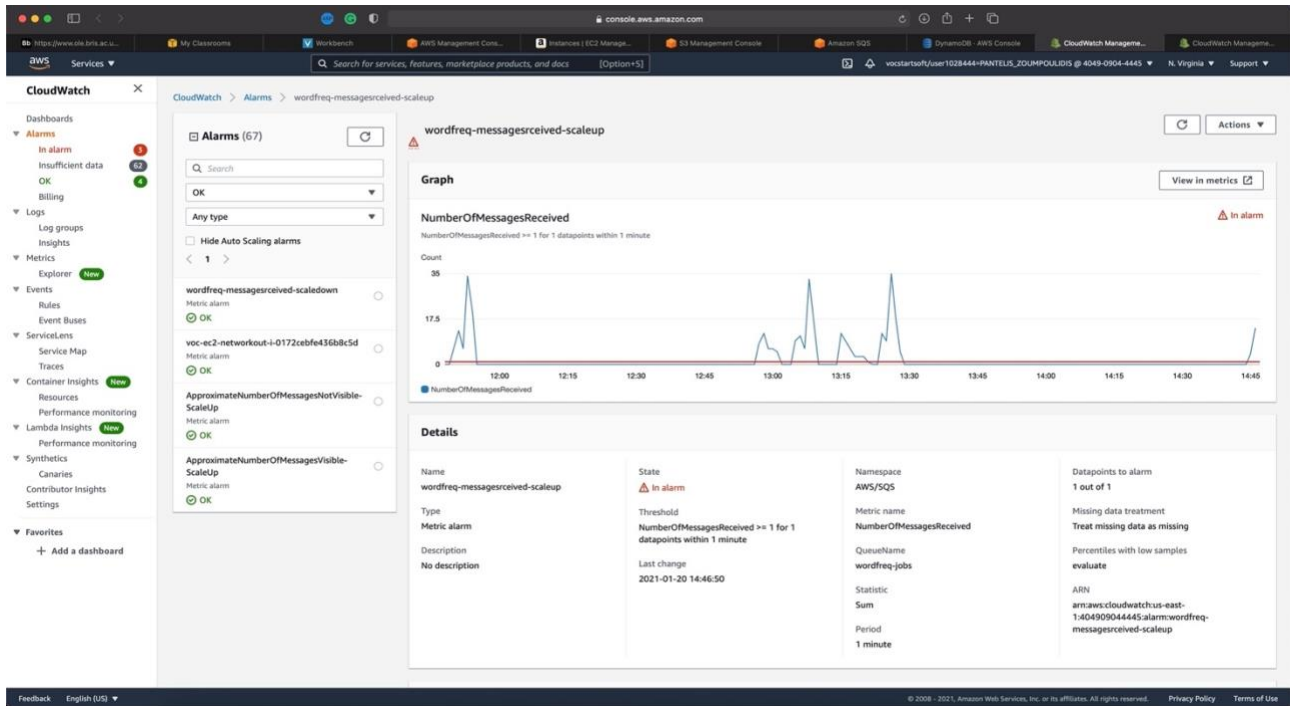
In order to test auto-scaling, we download the free e-book Anthem by Ayn Rand in .txt format, make 49 copies of it, and uploaded all these copies together as our data for analysis. Each file has a size of 128KB, so the total size of all books is 6.2MB. First, we will present the results of the auto-scaling configuration that returned the best time, while later we will briefly show some results from other configurations that we did during the optimization. The best time that we achieved was seven minutes and ten seconds, from the time we “hit” upload, and the time the last auto-scaling instance was terminated. Below are shown the timeframes of the key events that took place and their screenshots during the auto-scaling processing.

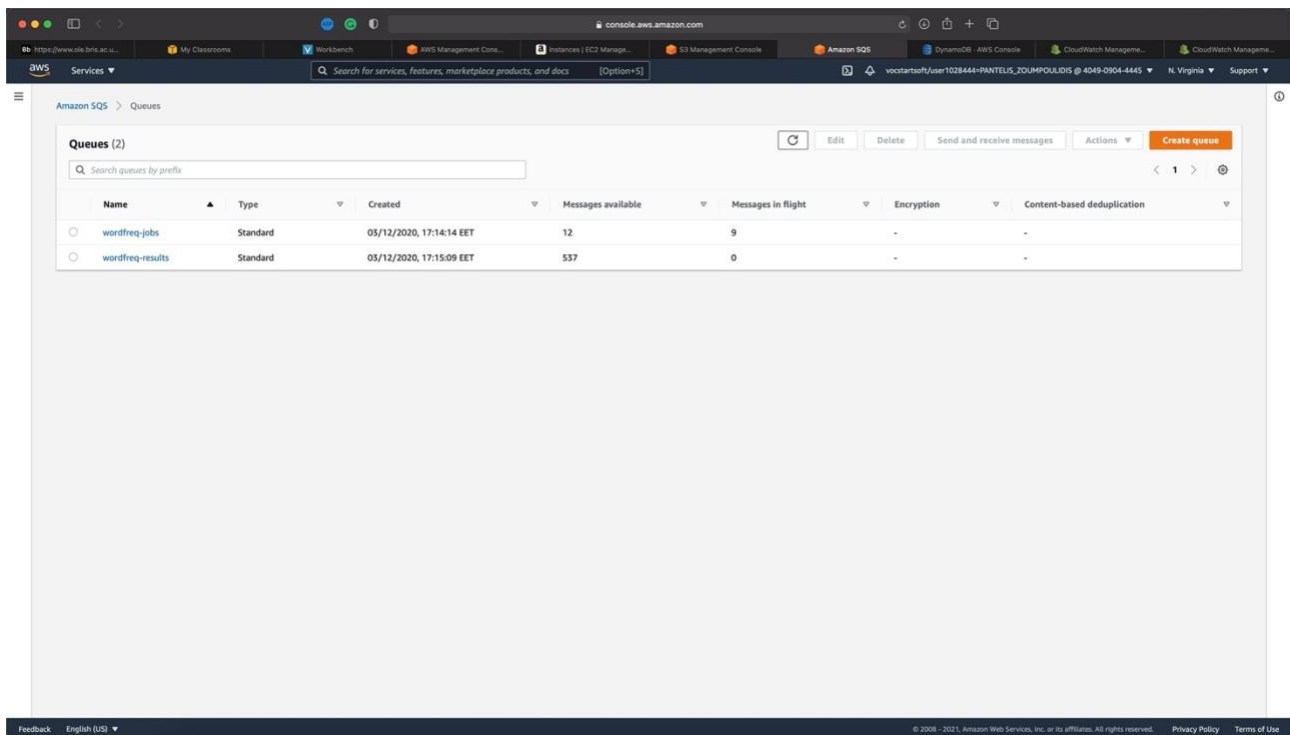
1. (0:00) Hit upload and started the stopwatch.

The screenshot shows the AWS Management Console interface for an upload operation. At the top, there's a progress bar indicating 2% completion. Below this, the 'Upload: status' section provides a summary of the upload. The summary table shows the destination as 's3://pe-wordfreq-dec20', with 1 file successfully uploaded (127.7 KB, 2.00%) and 0 files failed (0.00%). Below the summary, the 'Files and folders' section shows a list of 50 files (6.2 MB total). The files are named 'a.txt' through 'aa17.txt', all of type 'text/plain' and size '127.7 KB'. The status for all files is 'Pending'.

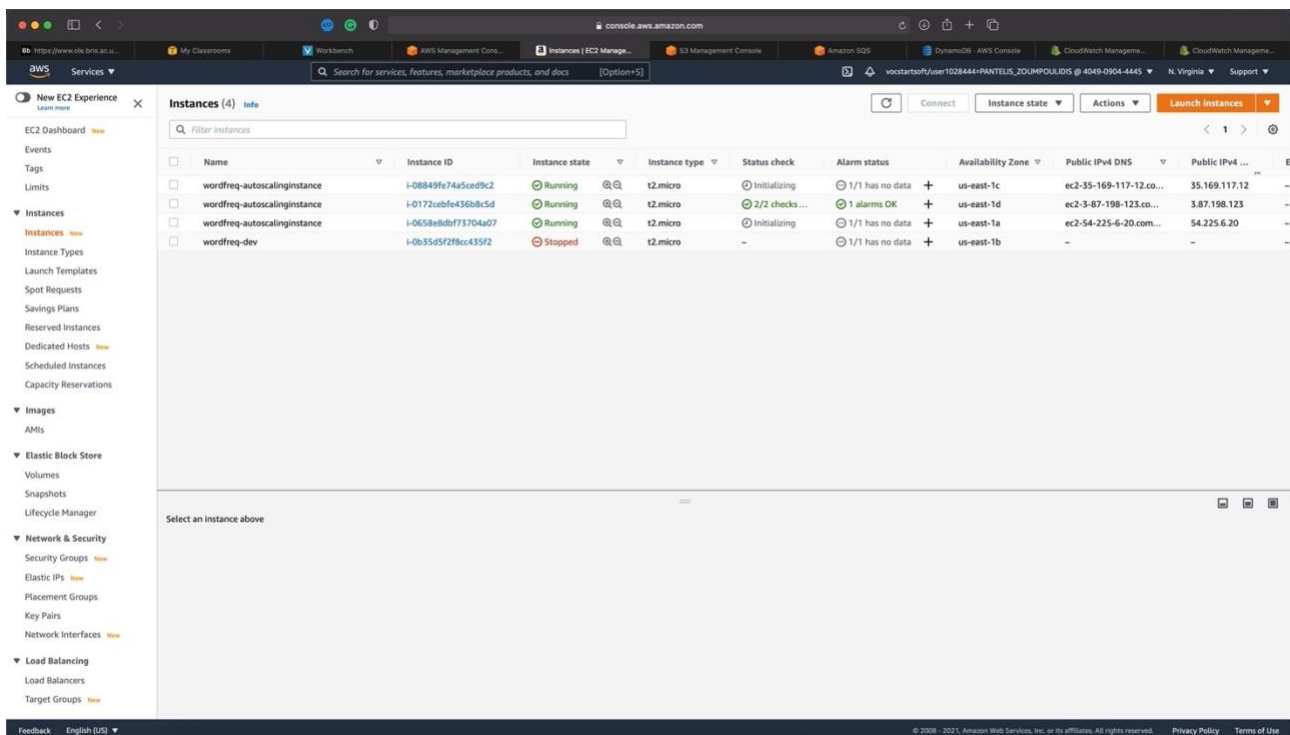
Name	Folder	Type	Size	Status	Error
a.txt	-	text/plain	127.7 KB	Pending	-
aa.txt	-	text/plain	127.7 KB	Pending	-
aa10.txt	-	text/plain	127.7 KB	Pending	-
aa11.txt	-	text/plain	127.7 KB	Pending	-
aa12.txt	-	text/plain	127.7 KB	Pending	-
aa13.txt	-	text/plain	127.7 KB	Pending	-
aa14.txt	-	text/plain	127.7 KB	Pending	-
aa15.txt	-	text/plain	127.7 KB	Pending	-
aa16.txt	-	text/plain	127.7 KB	Pending	-
aa17.txt	-	text/plain	127.7 KB	Pending	-

2. (1:25) CloudWatch alarm “wordfreq-messagesreceived-scaleup” for scaling-out went off, while “wordfreq-messagesreceived-scaleup” for scaling-in went on. Twelve messages were available at that time in the wordfreq-jobs queue, whilst nine were in flight. By the time the first CloudWatch alarm went off, eight text files were already processed.





3. (1:45) The additional two auto-scaling instances were up and running.



```
Large-Scale Data Engineering (LSDE) (TBI) — ubuntu@ip-172-31-68-33...
Last login: Tue Jan 19 11:31:52 2021 from 46.12.251.26
ubuntu@ip-172-31-68-33:~$ cd lsde-wordfreq-app/
ubuntu@ip-172-31-68-33:~/lsde-wordfreq-app$ ./run_worker.sh
Job Message queue starting
Worker 0 starting; CTRL-C to quit
Processing message c485c753-5ab6-4a85-9a4f-d5f46437ce83
Worker 0 received job c485c753-5ab6-4a85-9a4f-d5f46437ce83
Received job result c485c753-5ab6-4a85-9a4f-d5f46437ce83
Successfully processed job c485c753-5ab6-4a85-9a4f-d5f46437ce83
Deleted message, c485c753-5ab6-4a85-9a4f-d5f46437ce83
Processing message 4b448b86-444f-4338-a3cc-4895b988683c
Worker 0 received job 4b448b86-444f-4338-a3cc-4895b988683c
Processing message 6fbccae-7693-4a5f-833c-f8d81158121
Processing message 378ee2e-3a52-4638-8728-83a61a28ade3
Worker 0 received job 6fbccae-7693-4a5f-833c-f8d81158121
Received job result 4b448b86-444f-4338-a3cc-4895b988683c
Successfully processed job 4b448b86-444f-4338-a3cc-4895b988683c
Deleted message, 4b448b86-444f-4338-a3cc-4895b988683c
Processing message 4b448b86-444f-4338-a3cc-4895b988683c
Worker 0 received job 378ee2e-3a52-4638-8728-83a61a28ade3
Received job result 6fbccae-7693-4a5f-833c-f8d81158121
Successfully processed job 6fbccae-7693-4a5f-833c-f8d81158121
Deleted message, 6fbccae-7693-4a5f-833c-f8d81158121
Worker 0 received job 4b448b86-444f-4338-a3cc-4895b988683c

Large-Scale Data Engineering (LSDE) (TBI) — ubuntu@ip-172-31-28-192...
Introducing self-healing high availability clusters in MicroK8s.
Simple, hardened, Kubernetes for production, from RaspberryPi to DC.
https://microk8s.io/high-availability
48 packages can be updated.
23 updates are security updates.
Last login: Thu Dec 3 15:47:48 2020 from 31.285.55.236
ubuntu@ip-172-31-28-192:~$ cd lsde-wordfreq-app/
ubuntu@ip-172-31-28-192:~/lsde-wordfreq-app$ ./run_worker.sh
Job Result Collector starting.
Job Message queue starting
Worker 0 starting; CTRL-C to quit
Processing message e52d554e-c68d-44c8-b292-51254b968dce
Worker 0 received job e52d554e-c68d-44c8-b292-51254b968dce
Processing message 5aac3f4f-4817-4995-9482-c9c4ac5ea881
Worker 0 received job 5aac3f4f-4817-4995-9482-c9c4ac5ea881
Received job result e52d554e-c68d-44c8-b292-51254b968dce
Successfully processed job e52d554e-c68d-44c8-b292-51254b968dce
Deleted message, e52d554e-c68d-44c8-b292-51254b968dce
Received job result 5aac3f4f-4817-4995-9482-c9c4ac5ea881
Successfully processed job 5aac3f4f-4817-4995-9482-c9c4ac5ea881
Deleted message, 5aac3f4f-4817-4995-9482-c9c4ac5ea881

Large-Scale Data Engineering (LSDE) (TBI) — ubuntu@ip-172-31-49-1:~$
Management: https://landscape.canonical.com
Support: https://ubuntu.com/advantage
System information disabled due to load higher than 1.0
Introducing self-healing high availability clusters in MicroK8s.
Simple, hardened, Kubernetes for production, from RaspberryPi to DC.
https://microk8s.io/high-availability
59 packages can be updated.
41 updates are security updates.
Last login: Thu Dec 3 15:47:48 2020 from 31.285.55.236
ubuntu@ip-172-31-49-1:~$ cd lsde-wordfreq-app/
ubuntu@ip-172-31-49-1:~/lsde-wordfreq-app$ ./run_worker.sh
Job Result Collector starting.
Job Message queue starting
Worker 0 starting; CTRL-C to quit
Processing message 88fc1848-8fad-41a5-b2c7-47b5b9d88d83
Worker 0 received job 88fc1848-8fad-41a5-b2c7-47b5b9d88d83
Received job result 88fc1848-8fad-41a5-b2c7-47b5b9d88d83
Successfully processed job 88fc1848-8fad-41a5-b2c7-47b5b9d88d83
Deleted message, 88fc1848-8fad-41a5-b2c7-47b5b9d88d83
```

4. (2:45) Upload of the fifty text files was finished.

s3.console.aws.amazon.com

Upload succeeded
View details below.

Upload: status

The information below will no longer be available after you navigate away from this page.

Summary

Destination	Succeeded	Failed
s3://pz-wordfreq-dec20	50 Files, 6.2 MB (100.00%)	0 Files, 0 B (0%)

Files and folders | Configuration

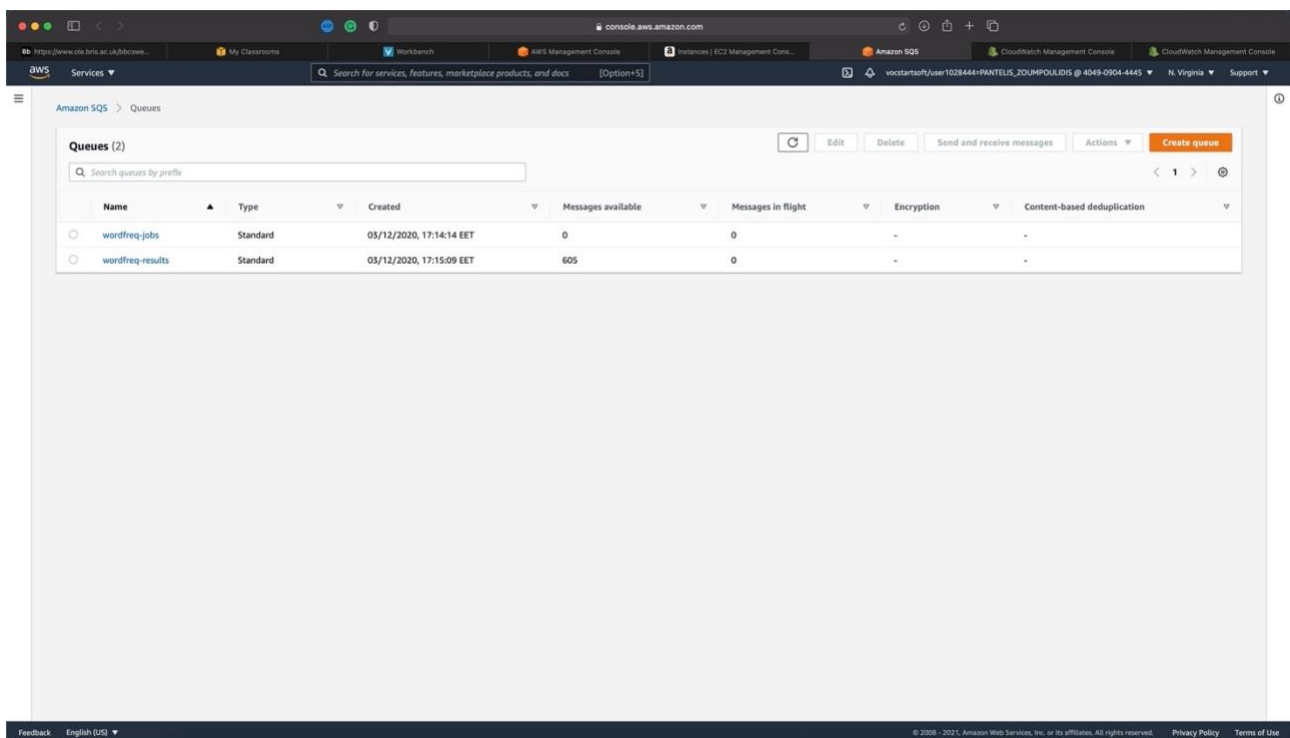
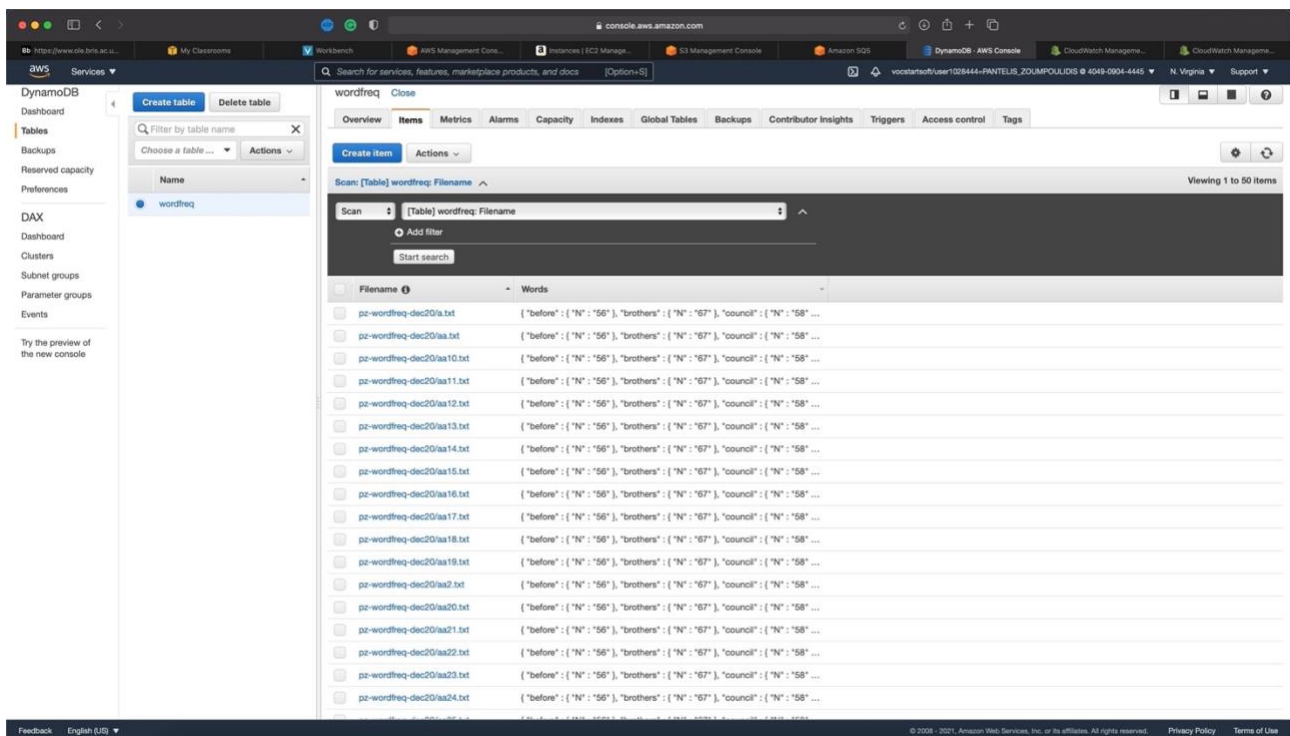
Files and folders (50 Total, 6.2 MB)

Find by name

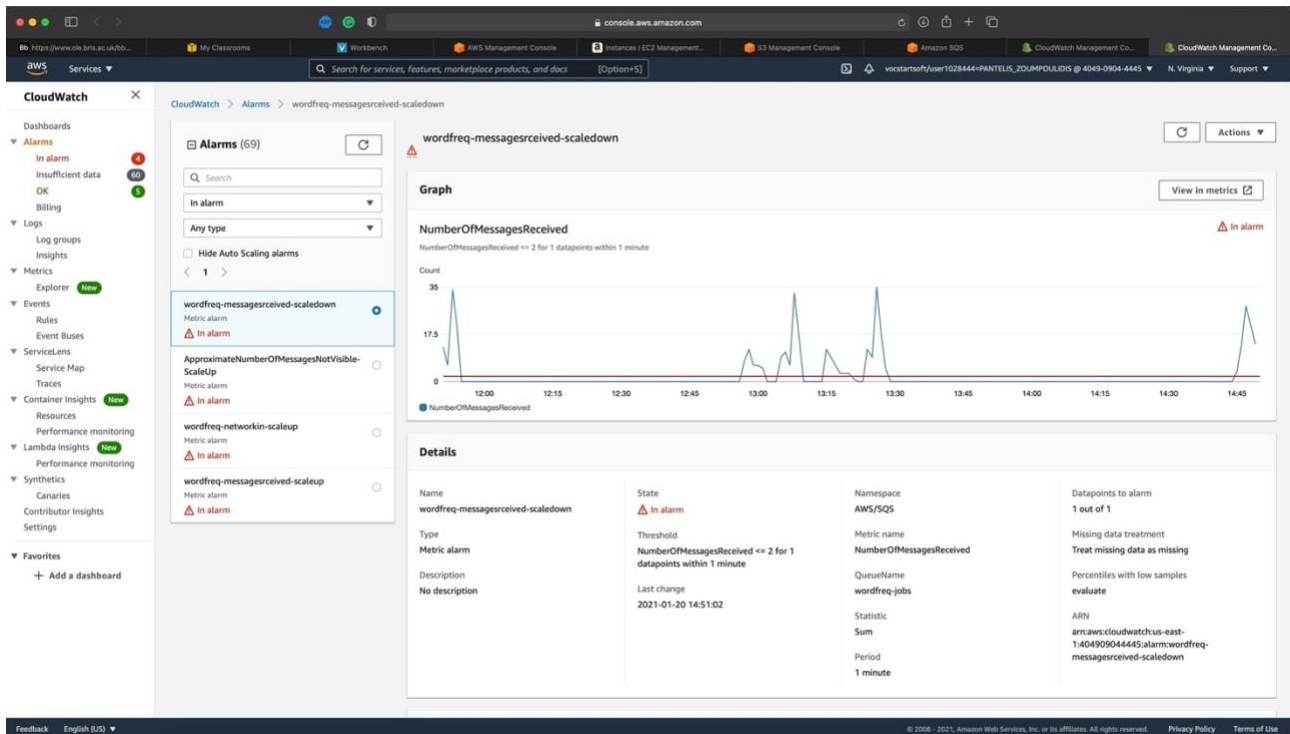
Name	Folder	Type	Size	Status	Error
a.txt	-	text/plain	127.7 KB	Succeeded	-
aa.txt	-	text/plain	127.7 KB	Succeeded	-
aa10.txt	-	text/plain	127.7 KB	Succeeded	-
aa11.txt	-	text/plain	127.7 KB	Succeeded	-
aa12.txt	-	text/plain	127.7 KB	Succeeded	-
aa13.txt	-	text/plain	127.7 KB	Succeeded	-
aa14.txt	-	text/plain	127.7 KB	Succeeded	-
aa15.txt	-	text/plain	127.7 KB	Succeeded	-
aa16.txt	-	text/plain	127.7 KB	Succeeded	-
aa17.txt	-	text/plain	127.7 KB	Succeeded	-

Feedback English (US) © 2008 - 2021, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

5. (4:52) All fifty text files were processed successfully.



6. (5:25) CloudWatch alarm “wordfreq-messagesreceived-scaledown” for scaling-in went off and the first auto-scaling instance started shutting down.



The screenshot shows the AWS EC2 console. The 'Instances (4)' page is displayed, showing a table of instances. The table has columns for Name, Instance ID, Instance state, Instance type, Status check, Alarm status, Availability Zone, Public IPv4 DNS, and Public IPv4 The instances listed are wordfreq-autoscalinginstance, wordfreq-autoscalinginstance, wordfreq-autoscalinginstance, and wordfreq-dev. The wordfreq-dev instance is in a 'Stopped' state, while the others are 'Running'.

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...
wordfreq-autoscalinginstance	i-08849fe74a5ced9c2	Running	t2.micro	2/2 checks ...	1 alarms OK	us-east-1c	ec2-35-169-117-12.co...	35.169.117.12
wordfreq-autoscalinginstance	i-0172cebfe436b8c5d	Running	t2.micro	2/2 checks ...	1 alarms OK	us-east-1d	ec2-3-87-198-123.co...	3.87.198.123
wordfreq-autoscalinginstance	i-0658e8dbf73704a07	Shutting-down	t2.micro	-	1 alarms OK	us-east-1a	ec2-54-225-6-20.com...	54.225.6.20
wordfreq-dev	i-0b35d5f2f8cc435f2	Stopped	t2.micro	-	1/1 has no data	us-east-1b	-	-

7. (6:28) Second auto-scaling instance started shutting down.

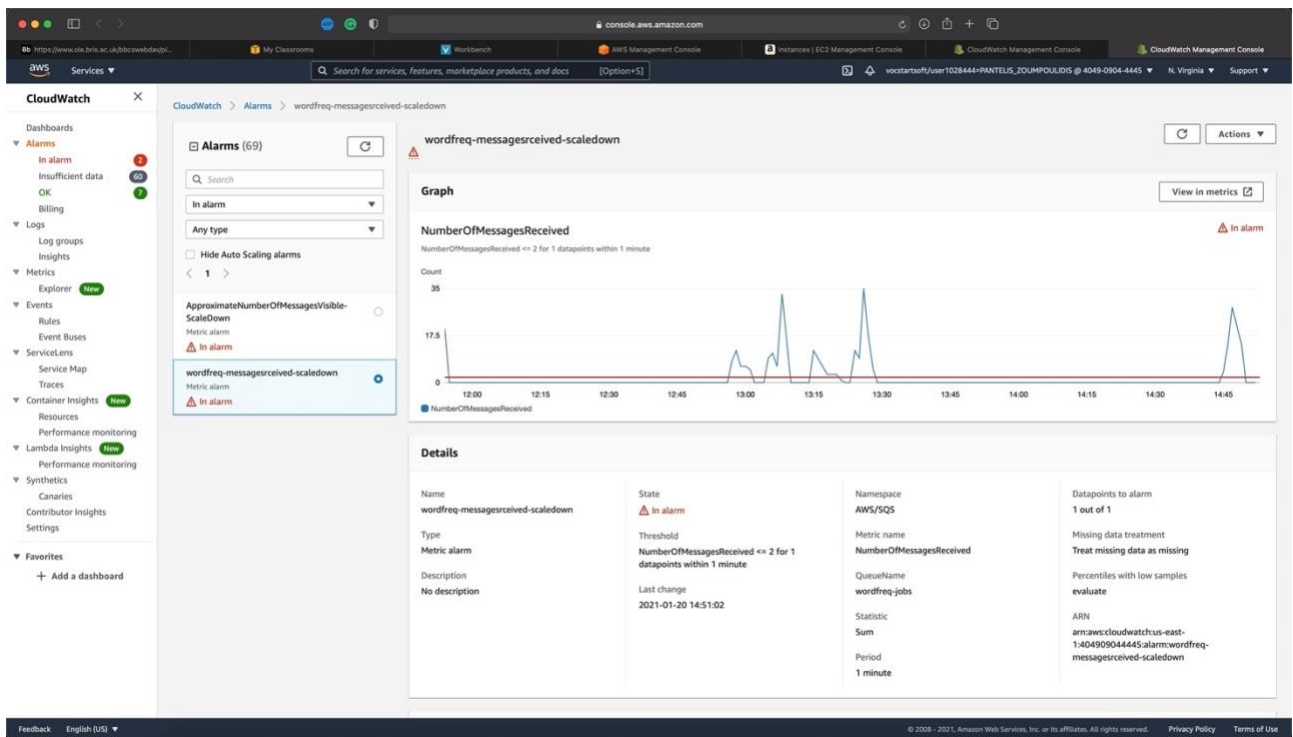
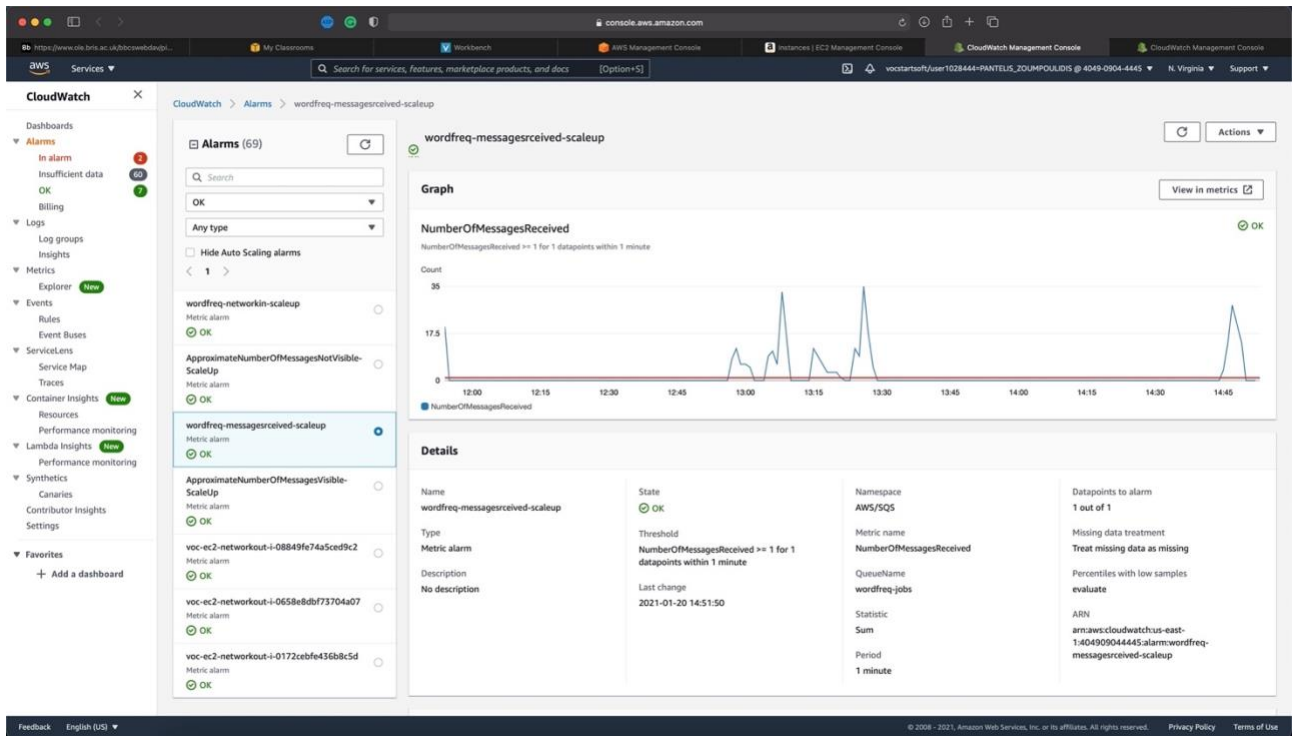
The screenshot shows the AWS Management Console 'Instances' page. The table below represents the data visible in the console:

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...
wordfreq-autoscalinginstance	i-08849fe74a5ced9c2	Shutting-down	t2.micro	-	1 alarms OK	us-east-1c	ec2-35-169-117-12.co...	35.169.117.12
wordfreq-autoscalinginstance	i-0172cebfe436b8c5d	Running	t2.micro	2/2 checks ...	1 alarms OK	us-east-1d	ec2-3-87-198-123.co...	3.87.198.123
wordfreq-autoscalinginstance	i-0658e8dbf73704a07	Terminated	t2.micro	-	1 alarms OK	us-east-1a	-	-
wordfreq-dev	i-0b35d5f2f8cc435f2	Stopped	t2.micro	-	1/1 has no data	us-east-1b	-	-

8. (7:10) Both instances were terminated. Note that the most recently instances were terminated, while the oldest was keep running. Both CloudWatch alarms are back at their normal state (scale-out: OK, scale-in ALARM).

The screenshot shows the AWS Management Console 'Instances' page. The table below represents the data visible in the console:

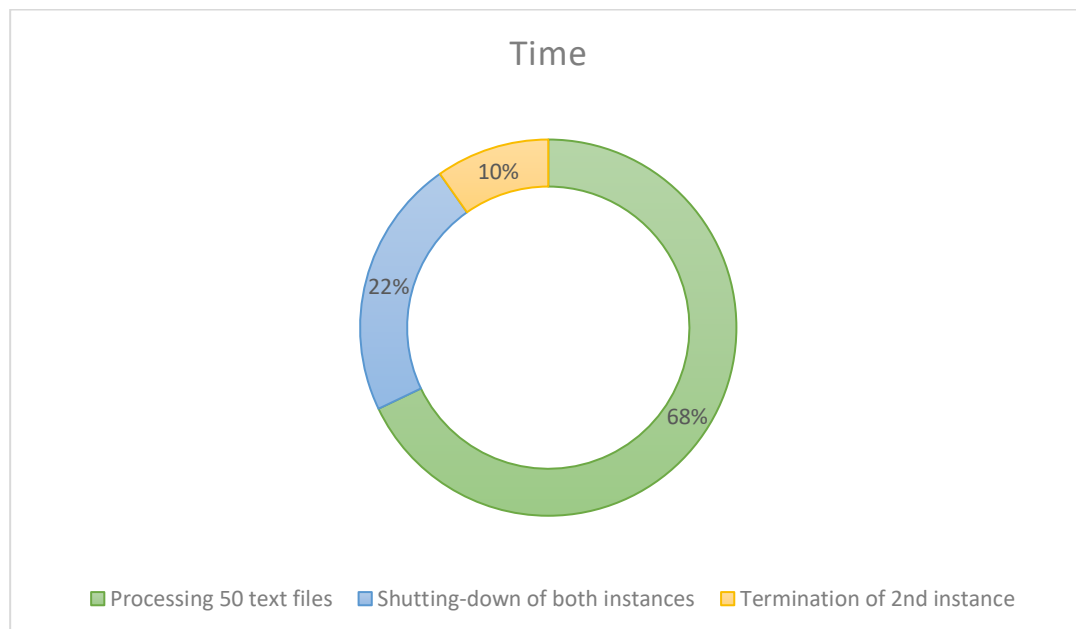
Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...
wordfreq-autoscalinginstance	i-08849fe74a5ced9c2	Terminated	t2.micro	-	1 alarms OK	us-east-1c	-	-
wordfreq-autoscalinginstance	i-0172cebfe436b8c5d	Running	t2.micro	2/2 checks ...	1 alarms OK	us-east-1d	ec2-3-87-198-123.co...	3.87.198.123
wordfreq-autoscalinginstance	i-0658e8dbf73704a07	Terminated	t2.micro	-	1 alarms OK	us-east-1a	-	-
wordfreq-dev	i-0b35d5f2f8cc435f2	Stopped	t2.micro	-	1/1 has no data	us-east-1b	-	-



We can see that we have a fully-functional auto-scaling configuration with all the functions requested working perfectly.

- The maximum number of instances are three
- When we perform scaling-in the most recent instances are terminating
- There is always one worker instance available to process messages
- All the messages are processed correctly with no remaining messages in the wordfreq-jobs queue
- The instances are terminated one-by-one in a small amount of time and not immediately when they are not required

Our auto-scaling configuration takes 4:52 minutes to process 50 text files, about 68% of the whole time, while the termination of the unused instances takes 32% of that time. The time between the finish of the processed files and the shutting down of the first auto-scaling instance takes 32 seconds, the shutting down of the second auto-scaling instance takes 1:35 minute, and the termination of both auto-scaling instances takes 2:18 minutes, giving us the total of 7:10 minutes.



We achieved these times using a t2.micro instance, which consists of 1 vCPU, 1GB of RAM, and 8GB of SSD storage. In order to check if we can achieve better times with a more powerful instance, we tested the same auto-scaling configuration, with the same 50 text files, but with a t2.xlarge instance which consists of 4 vCPUs, 16GB of RAM, and 8GB of SSD storage. We define key times to be the time until all files are processed successfully and the time all instances are terminated. In the first, we achieved 4:50 minutes, a time faster by 2 seconds from our t2.micro, but in the second, we achieved 8:02 minutes, a time slower by 52 seconds. Even though it was faster at processing the text files, the difference of time is so little that it isn't worth using these types of instances. This is caused by the ten-second-delay after each processed file, and also the low computing power needed for each one. Below, there is a screenshot showing three different instances we tried to test for auto-scaling, the first is t2.micro, the second is t2.xlarge, and the third is c5d.xlarge. We were not able to use the latter because of our AWS educate account restrictions.

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...
wordfreq-autoscalinginstance	i-0f3b846fde4671e4f	Terminated	t2.micro	--	1 alarms OK	us-east-1c	--	--
wordfreq-autoscalinginstance	i-0f34c7e69c00f256f	Terminated	t2.micro	--	1 alarms OK	us-east-1c	--	--
wordfreq-autoscaling-power	i-087be0862b9421bb8	Terminated	t2.xlarge	--	1 alarms OK	us-east-1c	--	--
wordfreq-autoscaling-power	i-081801eb146540059	Terminated	t2.xlarge	--	1/1 has no data	us-east-1e	--	--
wordfreq-autoscaling-power	i-068a6f45a54120d5	Pending	t2.xlarge	--	No alarms	us-east-1e	ec2-52-3-240-239.co...	52.3.240.239
wordfreq-autoscalinginstance-morepower	i-0436b146591ac403b	Terminated	c5d.xlarge	--	No alarms	us-east-1d	--	--
wordfreq-autoscaling-power	i-02e520bd25d30b674	Running	t2.xlarge	2/2 checks ...	1 alarms OK	us-east-1d	ec2-3-86-88-167.com...	3.86.88.167
wordfreq-autoscalinginstance	i-016c75053409fca0e	Terminated	t2.micro	--	1 alarms OK	us-east-1d	--	--
wordfreq-autoscaling-power	i-01b742e08ff44e18e	Terminated	t2.xlarge	--	1 alarms OK	us-east-1a	--	--
wordfreq-autoscalinginstance	i-091fb4cfa977e5cc	Terminated	t2.micro	--	1 alarms OK	us-east-1a	--	--
wordfreq-autoscaling-power	i-06f05c328819ddac	Pending	t2.xlarge	--	No alarms	us-east-1a	ec2-54-83-104-82.co...	54.83.104.82
wordfreq-dev	i-0b35d5f2f8cc435f2	Stopped	t2.micro	--	1/1 has no data	us-east-1b	--	--

So we keep the t2.micro as our working instance. The following auto-scaling configurations were also tested to instances like it.

Scale-in configurations:

- i. CloudWatch alarm: ApproximateNumberOfMessagesVisible
Auto-scaling policy: Simple scaling, add 2 instances when there are 1 or more messages visible
- ii. CloudWatch alarm: ApproximateNumberOfMessagesNotVisible
Policy: Simple scaling, add 2 instances when there are 1 or more messages not visible
- iii. CloudWatch alarm: NetworkIn
Policy: Simple scaling, add 2 instances when the EC2 instance receives more than 20000 bytes

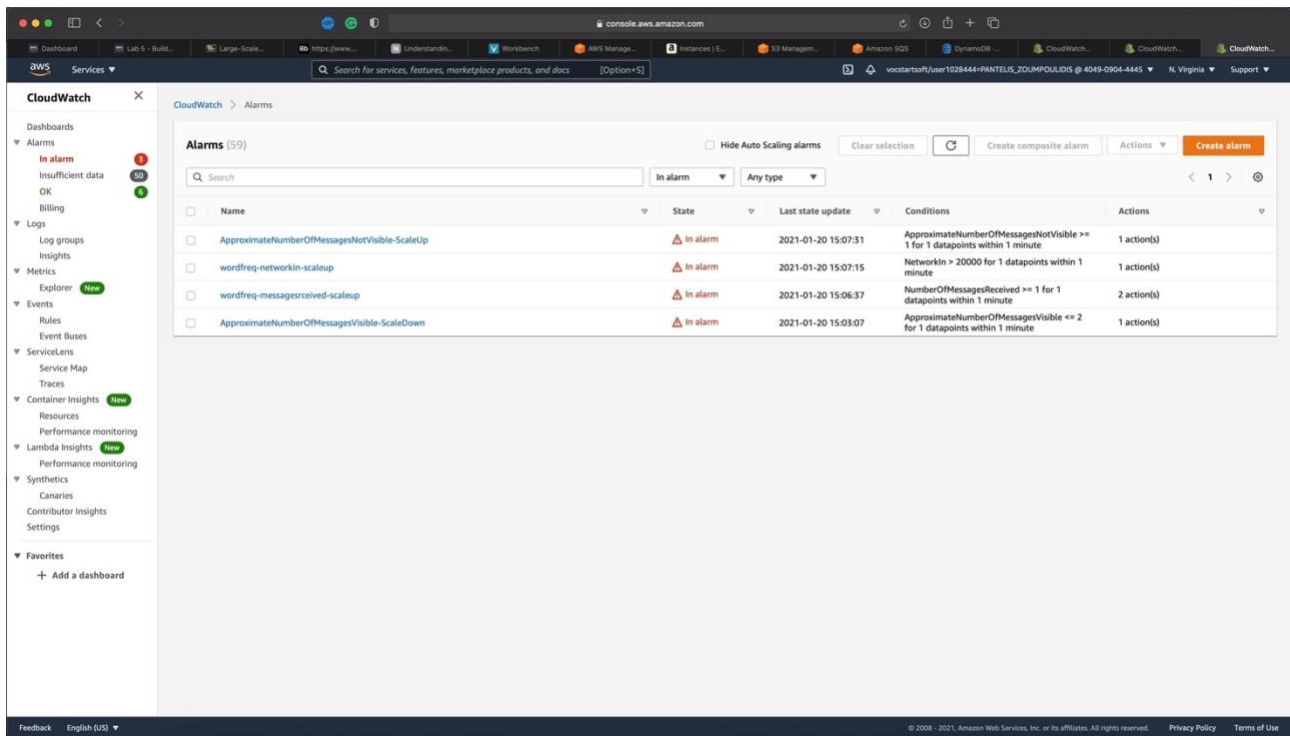
Scale-out configuration:

- i. CloudWatch alarm: ApproximateNumberOfMessagesVisible
Policy: Step scaling, remove 1 instance when there are 2 visible messages and another 1 when there is 1 or less message visible

Table 1 Times recorded with different scale-out-scale-in configurations

Scale-out \ Scale-in	ApproximateNumberOfMessagesVisible		MessagesReceived	
	Processing	Terminating	Processing	Terminating
ApproximateNumberOfMessagesVisible	7:01	9:42	6:50	8:54
ApproximateNumberOfMessagesNotVisible	5:05	7:40	5:02	7:20
NetworkIn	5:12	-	5:18	7:39

In the previous table, we can see the times that we recorder with all the different configurations we applied. Need to mention that the first scale-out configurations had a Step scaling policy but they were slower by at least a minute, so the simple scaling by adding 2 instances at once is way more productive for our cause. MessagesReceived was the best CloudWatch alarm for scaling-in to use by a difference of a few seconds. With the processing times we recorded for the scale-in policies, we are certain that MessagesReceived was the best CloudWatch alarm for scaling-out as well. The screenshot below shows a difference of almost a minute between the time that other alarms triggered.

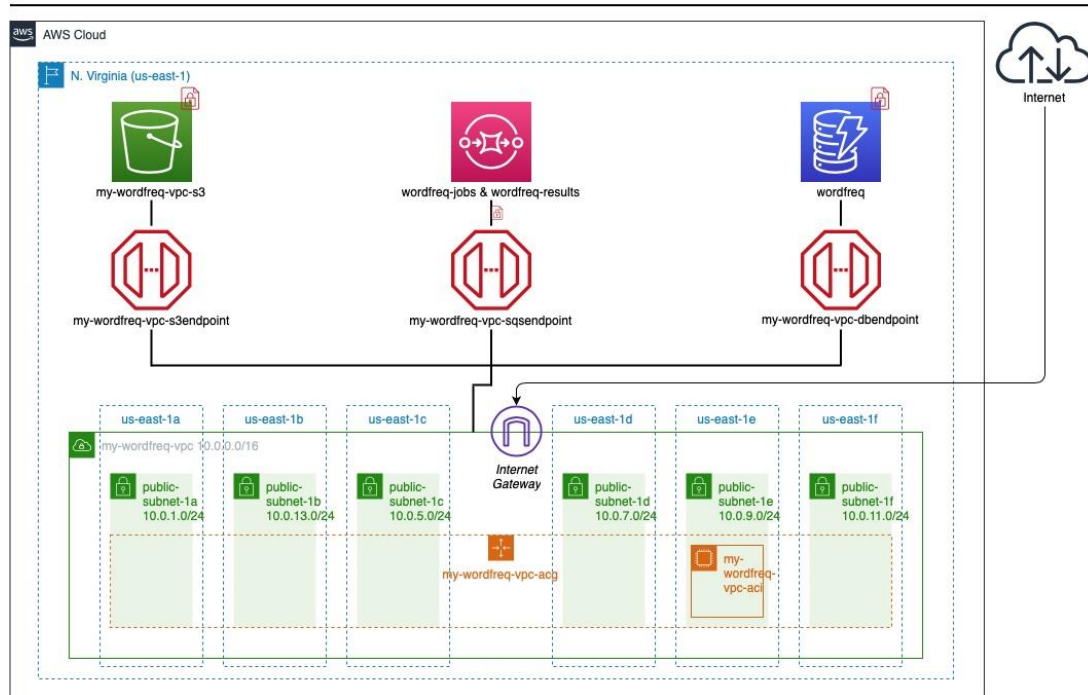


The screenshot shows the AWS CloudWatch Alarms console. The left sidebar contains navigation links for Dashboards, Alarms, Logs, Metrics, Events, Rules, ServiceLens, Container Insights, Lambda Insights, Synthetics, and Favorites. The main panel displays a list of alarms under the heading 'Alarms (59)'. The list includes columns for Name, State, Last state update, Conditions, and Actions. All four alarms shown are in the 'In alarm' state.

Name	State	Last state update	Conditions	Actions
ApproximateNumberOfMessagesNotVisible-ScaleUp	In alarm	2021-01-20 15:07:31	ApproximateNumberOfMessagesNotVisible >= 1 for 1 datapoints within 1 minute	1 action(s)
wordfreq-networkin-scaleup	In alarm	2021-01-20 15:07:15	Networkin > 20000 for 1 datapoints within 1 minute	1 action(s)
wordfreq-messagesreceived-scaleup	In alarm	2021-01-20 15:06:37	NumberOfMessagesReceived >= 1 for 1 datapoints within 1 minute	2 action(s)
ApproximateNumberOfMessagesVisible-ScaleDown	In alarm	2021-01-20 15:03:07	ApproximateNumberOfMessagesVisible <= 2 for 1 datapoints within 1 minute	1 action(s)

Next, we secure and optimize our application Architecture. We start by creating our own VPC in Region us-east-1, with six different public subnets and an internet gateway. Also, we created VPC endpoints to establish private connections between our VPC and the AWS services that we use. Because of the coursework's word restrictions, we are not going to mention step-by-step how we created our VPC. The following diagram represents the VPC that we've created.

my-wordfreq-vpc



We begin our optimization with the *resilience and availability of our application against component failure*. We have four different components that we need to make sure that are constantly working, EC2 instance, S3, SQS, and DynamoDB, as services like auto-scaling and CloudWatch are available 24/7. When we implemented auto-scaling, we chose all availability zones, and the desired as well as the minimum capacity to be 1. The first means that the auto-scaling can start instances in all six availability zones of us-east-1 (N. Virginia) that our account is located at, while the second means that when there is no EC2 instance running due to an unfortunate event such as a component or data center failure, the auto-scaling configuration will launch a new EC2 instance in a normal operating Availability Zone of the ones that we selected. Furthermore, EC2 and system status checks are always enabled by AWS when using auto-scaling. One more step that we take is by selecting the Details tab under our wordfreq-autoscaling-group and at Health checks section and entering a Health check grace period of 60 seconds which ensures that our configuration will check in the first minute of running if there is at least one EC2 instance running normally. Amazon S3 is designed for 99.99% availability, which translates to 351.96 days availability over a year, and a 99.999999999% (11-9s) durability of stored objects in it. SQS is designed in a way that no availability zone failure can make its messages to be unavailable. DynamoDB is designed to be high-available and data-durable as well. It automatically replicates the stored data in various Availability Zones within an AWS Region. So, we can easily understand that our system already offers high resilience and availability of our application against component failures.

Following this, we need to enhance our system's *security for data protection and against unauthorized access*. It was impossible to apply almost everything we tried because of AWS educate account restrictions. We begin by protecting our EC2 instances. We use AWS Systems Manager so we can install important security updates on our virtual machine. We select Patch Manager and then Configure patching. We insert the following configurations:

1. Instances to patch

Enter instance tags

Tag key: Name

Tag value: wordfreq-autoscalinginstance

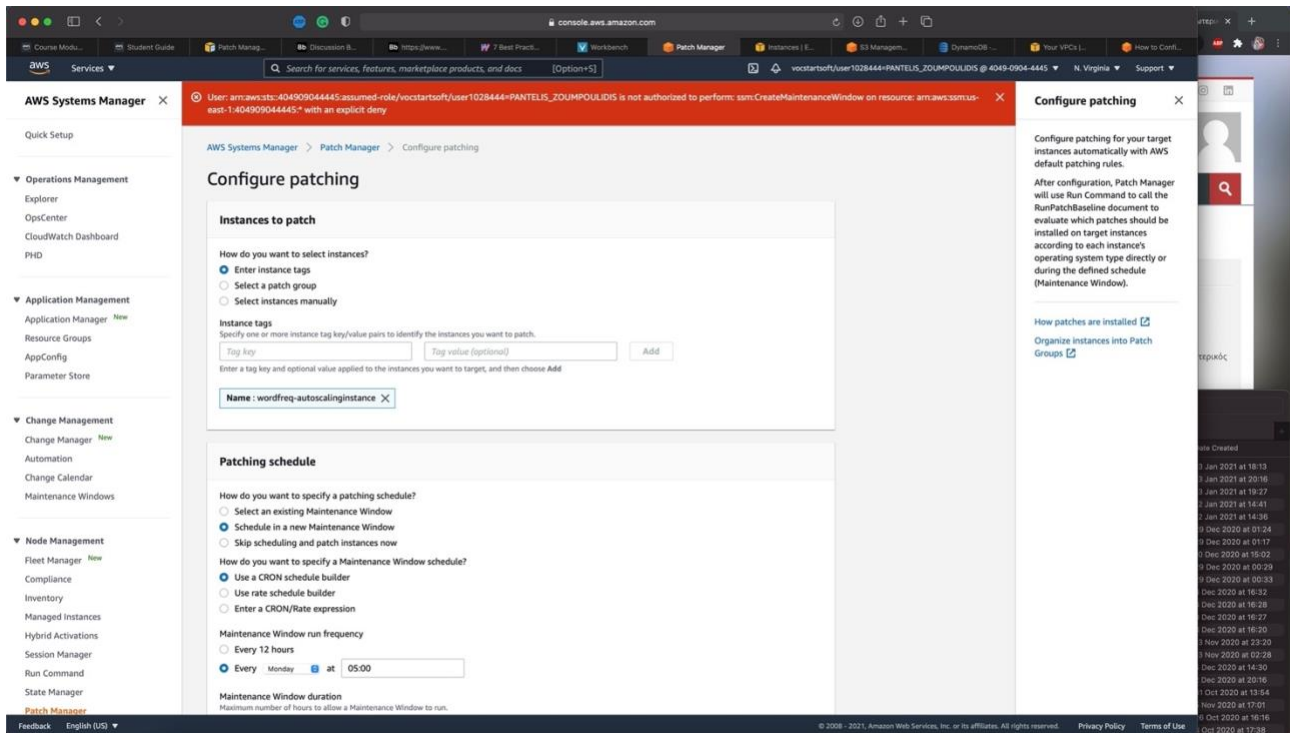
2. Patching schedule

Schedule in a new Maintenance Window

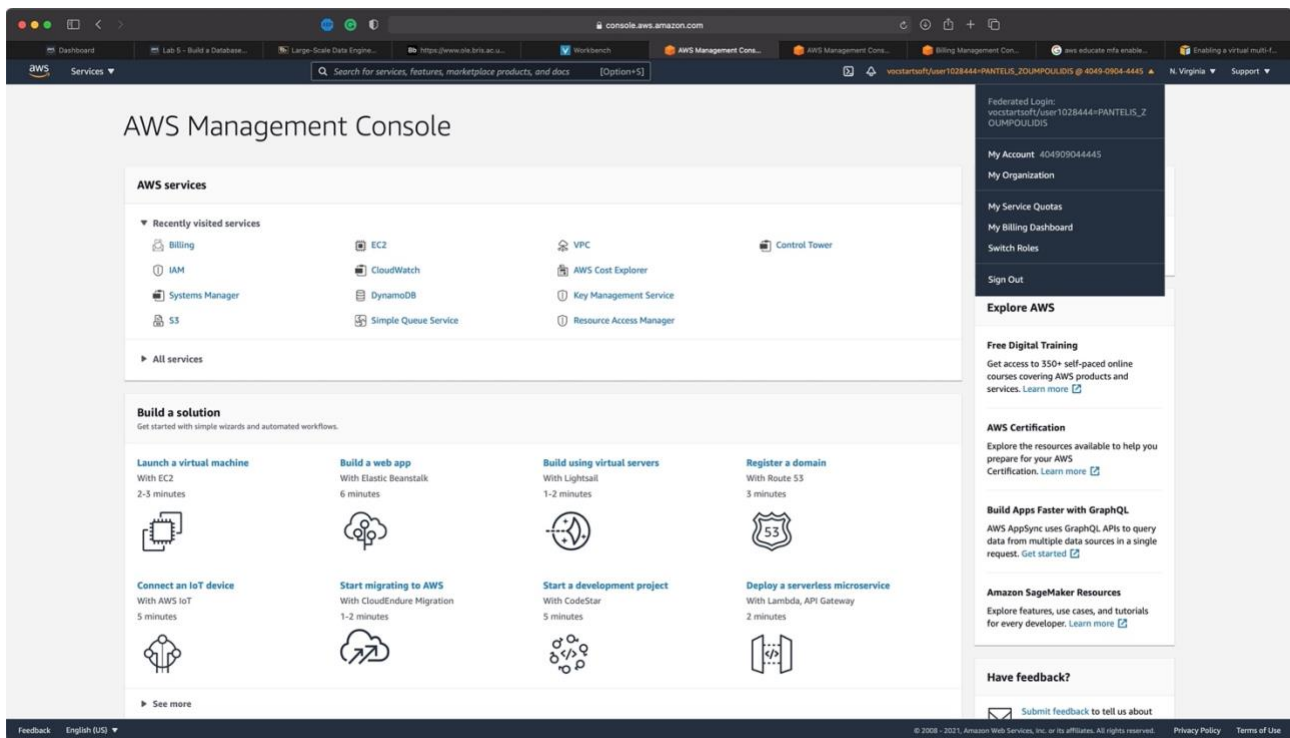
Use a CRON schedule builder

Every Monday at 05:00

Unfortunately, we were not able to configure the patching as shown below.



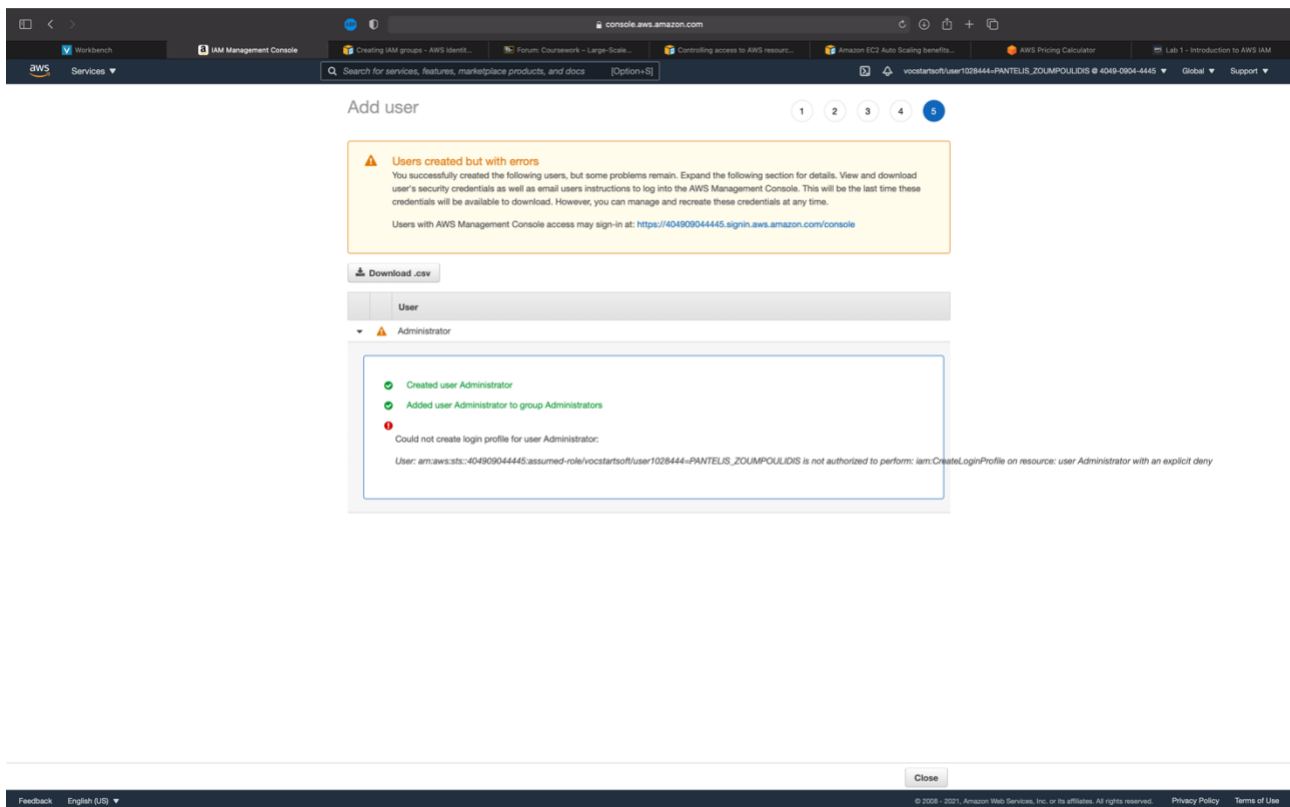
Carrying on, we need to secure our AWS account, so no one can take possession of our data, delete them, or use our resources. They need our username and our password or access keys, therefore we need to add another layer of protection called MFA (Multi-Factor Authentication). The steps are the following: select our username in the upper right corner → click on My Security Credentials → Activate MFA where we choose our MFA configurations. We were not able to do this either, as we don't have the option of Security Credentials in our account.



Another thing AWS recommends is to create a new administrator user and never use our root AWS account for managing it. Additionally, it recommends us to use accounts with as few privileges as possible required to achieve tasks. We continue by creating an Administrators group and then a new User, whom we will add to the previous group. We search IAM from the search bar.

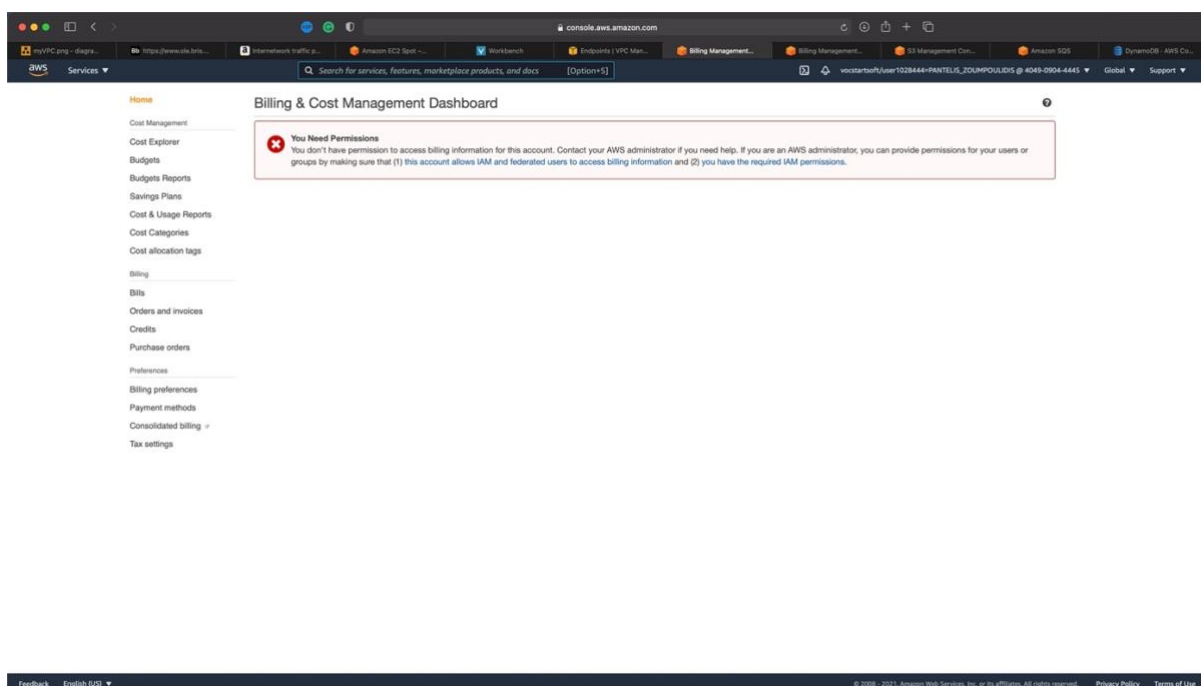
1. Create Administrators group:
Groups → Create New Group → Group Name: “Administrators” → Attach Policy: Administrator Access → Create Group
2. Create User and add him to Administrators group:
Users → Add user → User name: “Administrator”, Access type: AWS Management Console access → Console password: Custom password: type in a custom password → Add user to group: Administrators → Key: “Name”, Value: “Administrator” → Create User

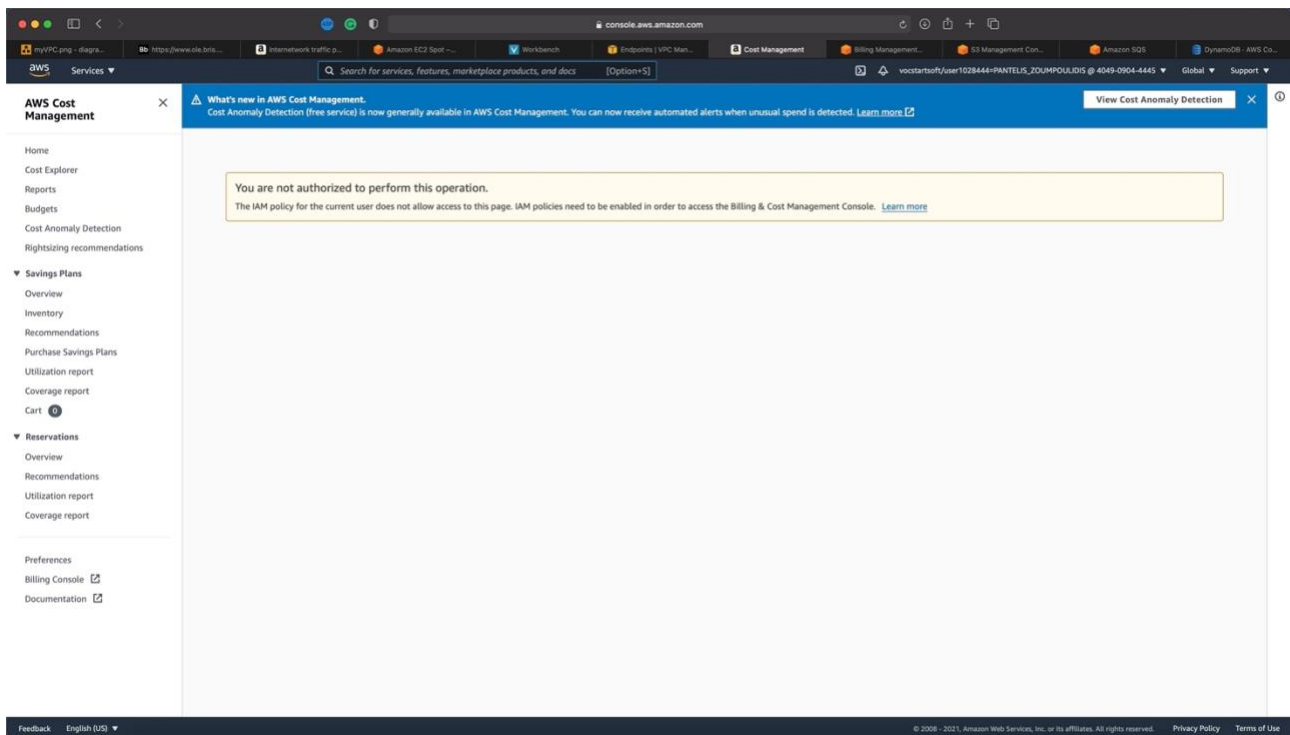
There is an option to assign an MFA device for our newest created user as well, but we don’t have the permissions to do so. We were able to create the Administrators group, the Administrator user, but due to our restrictions, it was not possible to create a login profile for our user, as you can observe in the next screenshot.



Moreover, when we created our VPC we also added an inbound rule with an SSH protocol allowing connections only from our PC's IP address. As mentioned before, we also created VPC Endpoints in order for our VPC to communicate privately with the AWS Services that we use. We enabled Default encryption for data in our S3 bucket (my-wordfreq-vpc-s3 → Properties → Default encryption → Enable, AWS Key Management Service key). In-transit encryption is enabled by default in Amazon SQS. The same applies to encryption in DynamoDB as well.

Additionally, we discuss how we could *ensure that our application is as cost-effective as possible*. Unfortunately, we are not able to access “My Billing Dashboard” and AWS Cost Management.





Without the metrics, it's difficult to figure out what we can do. Amazon S3 and Amazon SQS have standard pricing each month. DynamoDB has a standard charge for read and write requests, although the first 25GB stored per month in it are free. We could delete unused data from our database when we reach close to that point. There is not much we can do about those services. Other services like Amazon EFS or Amazon RDS are more costly than the ones we already use. One technique that we could do is to use EC2 spot instances. As mentioned by AWS documentation, we can save up to 90% compared to EC2 On-Demand prices. Also, reserving an EC2 instance for more than a year and paying upfront partially or fully, can also decrease cost up to 87%.

We continue, with the issues we had and how we managed to resolve them. While we were creating our new VPC we came across 2 issues. The first was that we couldn't connect to the first EC2 instance we created. The problem was that it didn't have a public IPv4 address, therefore we created an elastic IP for it. Then we created the AMI and enabled auto-assign IP address to our public subnets for the new instances in order for them to access the internet, so we, as well as AWS services, can access them. The second issue we had was the creation of NAT gateways because of our account's restrictions. Thus, we created VPC Endpoints that offer private communication between our VPC and AWS services.

console.aws.amazon.com

Dashboard Lab 5 - Build... Welcome, Pa... Weeks 8 - 11... Bb https://www.o... Workbench VPC Manage... Enable SSH C...

Services Search for services, feat [Option+S] vocstartsoft/User1028444=PANTELIS_ZOUMPOULIDIS @ 4049-0904-44 N. Virgini Support

There was an error creating your NAT gateway

You are not authorized to perform this operation. Encoded authorization failure message: 6jqyuArByh8Qn4DouZTdbrgu0iP-3iicwyHyCzn-Wwy3U-J19GwAUINf3jKVg2f3-3HifJ5h9T7myTRJZrEPDaltgUitfi5ynKYFa9Q2Hln9DfHR35Wj_-blR5d1WMcWicu_YBwLAMJgkeFLioHvdNw4YtFfQUqrAfoAJTsFFHWAYe7mVQuS0bNtZ-pdp-_1Kchx3VM9mrARJrkVymui5pbUtoLFsj59K-GJUE5p4AwRZD7wjH6tul0qD4soq9w6iIWg9VE90Lh2zWGNQdXEJHQt3Pf1pFrkZPBqlxXu6cOBTZJ4up4Sh43LcfvRV1bZtLSITQeSWDVk3rADaVkbA0Aihbq7wQu-eKZfztHT8YLe33vtDKT1R_kDfZ2qmYtIHwG6T2tyZGy0FwAx5-qUwFjxbXXEOzB9cmEyL80osPFABMhKjRBG-z2bvrqMGiTSV6DoQye8-t_B_PNyGBjSn9WK4crgSpguolWnMgGHlR_c-VFHWPDQ90m-legZKpVqLPPqIPt9cggDtSWs1Ylkk26Xg5Ls8BH9yto06kcAttyKzbxJDTHZJl1hwRAYNCj1kAloodA11r8eHC9QeUunnuzRJEm0Y9Lc29gkcbemZlaZIDv1KV5sB7cMjANqjgJmVMjSsBgomiXldtUMiMrYmd6kruNroH6Cl3PIBV0YZxSyOGnX-_rvTeSvM_xMYibGhOjatLJAGTbIAZrEtCsGyaJXJUth_3TnerVnjqFNrTZ9AyeG_kl1FdYJ3OLWtLn2ntwb-D0oqzgbcgWv2XcoR-3q8Jva9O1-ZzCHQxtw5XGc5M_tCc8-3JlwgqewqEOkGD55_kHp9-S_n7PDWngme7fPw69rvZ4g633sFLng0HQUEakZyXpuRHWZ0xHDQFzc27Cds0s-Q22-VCE1EZEZWQWNMB9A-SNpkSsGNH9Xnk4gtou9LSe3ErAh5onBJTogtODR2NqLwGhrhIOiBCEE9WFWCY1bOT4de4ZGG0oUH6d0XpAOp57ThGwaLcg50mFQ-Pskwy7992nSR1buGwHE5iLwtAv7ZzK4SsBPmA4PK0PW2fz2tNBQpUfZPq7PTmeBvB31NMPWdltCLtoq82W5SnnbnoR6BF_CnRZiGvLZu9zVlBCyOJJpZ1hPR8C4RgPaXKF872AGlQqZexz877PFilF55VDhaTo-1vJjkpM

VPC > NAT gateways > Create NAT gateway

Create NAT gateway [Info](#)

Create a NAT gateway and assign it an Elastic IP address.

NAT gateway settings

Name - optional
Create a tag with a key of 'Name' and a value that you specify.

The name can be up to 256 characters long.

Subnet
Select a public subnet in which to create the NAT gateway.

Elastic IP allocation ID [Info](#)
Assign an Elastic IP address to the NAT gateway.

Tags

A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

Key	Value - optional	
<input type="text" value="Name"/>	<input type="text" value="my-wordfreq-vpc-natgw"/>	<input type="button" value="Remove"/>
<input type="button" value="Add new tag"/>		

You can add 49 more tags.

Feedback English (US) © 2008 - 2021, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

The last thing in our report is mentioning *two alternative data processing architectures that would make our application to be more performant*. One of them is Hadoop MapReduce. MapReduce was first created by Google back in 2003, which more or less do the following. It takes as input data a large file, splits it into a number of segments, then machines map the data and write them in their local file stores, and finally, other machines sort them and take the unique values of these outcomes to combine them into one final result the user had requested. On our occasion, we could input large text files, and the machines simultaneously process them and give us the results we wanted in almost no time. It is also fault-tolerant, as one machine would fail, another would take its position and start the job from the beginning. Another data processing framework is Apache Spark. It uses Hadoop MapReduce as its foundation, but the key difference is that Spark is an in-memory processing engine, meaning that processes data in RAM rather than in disk, as Hadoop MapReduce does, making it 10 to 100 times faster than it. It is also fault-tolerant just like Hadoop's MapReduce. Both of these frameworks can be used on AWS using the service called AWS EMR.