# WORD CLOUD

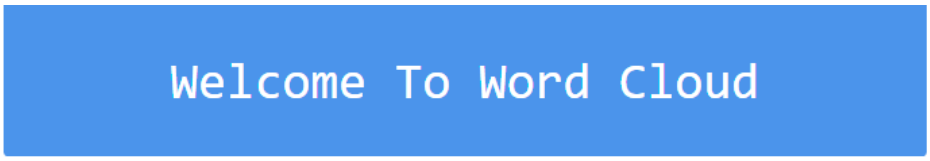
A simple word cloud website.

Dara Heng n9652159  James W. Flannery n8326631

# Table of Contents

## Introduction

The word cloud website allows the users to determine which words appear most frequently given each tweet streaming from Twitter website. Top one hundred words are nicely shown in a cluster within the main page and each word size is determined by its number of appearances during twitter streaming process. Top ten words are displayed in a table with number frequency included. Moreover, a sentiment analysis is done with all of the incoming tweets and the overall is display above the top 10 table.





Sentiment Analysis: Happy

Top 10 Words

| Word: | love | just | 2017 | Will | when | time | 171101 | happy | people | today |
|---|---|---|---|---|---|---|---|---|---|---|
| Frequency: | 347 | 346 | 284 | 282 | 281 | 263 | 246 | 240 | 222 | 204 |

Image 1. Picture of Word Cloud website

## Description of the application architecture

The word cloud website is constructed of three mechanisms such as, tweet-stream, processor and server. To begin with, tweet-stream is the component that uses one of twitter API functionalities called stream to get each tweet on live-stream. Once each tweet is received, it is being post to the second component which is the processor.

The processor gets tweets and put them into one of AWS storage called Amazon Relational Database Service (RDS). Before each tweet is being pushed into RDS, various formatting process is needed to be done such as, breaking down tweets into words and making sure that each word is being filtered appropriately. Afterwards, each word is being pushed into RDS.

Lastly, the third component is called server. It acts a result page that displays each validated word given by each tweet during the streaming process. This page contains a cluster of top one hundred words that appear the most and a table that shows the top ten words with number frequency included. Interestingly, the size of each word in the cluster is based on its number of appearances.

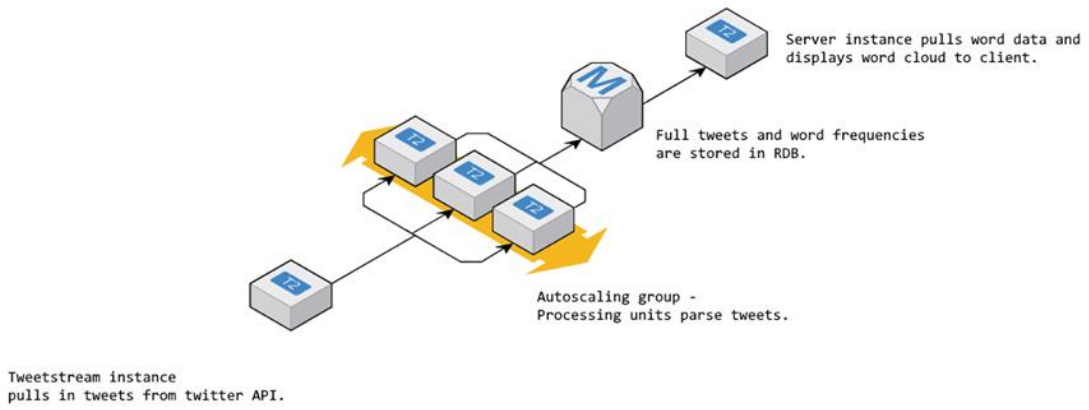



Figure 1. Application architecture

## Description of Services

Different types of API are being used in three components because each component has its unique features. Information about the APIs will be provided in the following sections.

### Services in Tweet-stream component

Two unique services are being used in this component:

- *Request*: this package is designed to make it easy for establishing http calls. It has a variety of features that you can use. In this project, we are using one of its feature called POST. It provides the ability to transfer data via post request to the specified URL.
    - https://www.npmjs.com/package/request
- *Twitter*: this service provides developers access to its data i.e. tweets and other relevant information relating to tweets. It comes with many features that can be made useful with various tasks. In this project, we are going to be using one of its features called Twitter Streaming. This provides the ability to filter the full Twitter firehouse and only receive the data we interested in. For the purpose of this project, we are interested in getting all tweets regardless of what it is, except when it is undefined.
    - https://developer.twitter.com/en/docs/tweets/filter-realtime/overview/powertrack-api

### Services in Processor component

This component operates with a lot more services than other components because the process of filtering each tweet is being done in the Processor.

- *Sequelize*: this is a promise-based Object-relational mapping (ORM) that supports a variety of databases such as, Postgres, MySQL, SQLite and Microsoft SQL server. In this project, the type of databases we are using is MySQL that is being host from Amazon Relational Database Service (RDS). This service provides the ability to do transaction support, relations, read replication and more. In this component, it is using this service to create and store tweets into AWS storage.
    - https://www.npmjs.com/package/sequelize

- *MySQL2*: This server is a MySQL client for Node.js that supports various features such as, prepared statements, compression, etc. In order to use Sequenlize with MySQL, this packge is needed to install into the project.
    - https://www.npmjs.com/package/mysql2
- *Natural*: this is a general natural language service for Node.js that is used to do interesting analysis on words or sentences. It provides a variety of features such as, Tokenizers, String Distance, Stemmers, Classifiers, etc. In this project. We are using Tokenizers to split each tweet into an array of words.
    - https://www.npmjs.com/package/natural
- *Stopword*: this service is a node module that provides the ability to remove stopwords from an input text. This is being used in a part of the filtering process in this project to identify and remove unnecessary words (stopwords) from each tweet.
    - https://www.npmjs.com/package/stopword
- *Sentiment*: this is a service from Node.js module that uses AFINN-165 wordlist and Emoji Sentiment Ranking to perform sentiment analysis on arbitrary blocks of input text. In this project, it is simply used to determine the attitude of every individual tweet.
    - https://www.npmjs.com/package/sentiment

### Services in Server component

This component is implemented to generate a simple website that nicely displays word cloud.

- *Sequelize*: information about this service has been mentioned above. This component is using Sequelize to pull back all words and their frequencies from RDS in order to display to the website.
- *MySQL2*: To use Sequenlize with MySQL, this package is needed to install into the project.
- *D3.js*: it is a JavaScript library that uses data to do document manipulations. It provides the ability to bring life to data by using HTML, SVG, and CSS. In this project, we are using a resource provided by D3.js called Word Cloud layout to implement word cloud to our website.
    - https://github.com/jasondavies/d3-cloud

## Development strategy

There are 4 stages on the development process. To start with, we started working on the project locally. This means we ran the program locally via IntelliJ Webstorm IDE and used MySQL localhost database as the persistence. Once we got everything working correctly locally, we began to configure the program to use the database from Amazon called RDS. Afterwards, we launched the processor component into AWS instance and built the instance into AWS image (AMI). Lastly, the load balancer was generated with the AMI we just created and then linked the load balancer to auto scaling launch configuration.

# Technical description of the application

## In-depth discussion of the architecture

Twitter-stream component has two main functionalities. First, it is using Twitter API to stream live-tweet. Before each tweet is being stored, Twitter-stream checks whether the tweet and its user is defined or undefined and only the defined one gets stored. Second, each tweet it receives is then sent to the specified URL by making a post request.

The processor component handles the filtering process on incoming tweet. Firstly, each tweet is being run through regex to remove all unnecessary characters that are identified as emojis. Secondly, each tweet is split into words using Tokenizers feature provided by Natural and then a stop word detection is created to remove stop words that appear in each tweet. Once stop words are removed, each word is being filtered with the following conditions: (1) a word must have a length greater than three. (2) a word must have a length less than seven. (3) a word must not contain a string “http”. (4) a word must not contain a string of “https”. Once the filtering process is completed, each word that passes the conditions I have mentioned is pushed into the AWS storage called RDS. When word is

being pushed, the program detects if the word is already existed in the storage. If the word is found, then the program will increment the number frequency of the word. If the word is not found, then the program will create and store the word into a new row. Moreover, the processor also run sentiment analysis on every incoming tweet to determine the tweet user's emotions.

The server component generates a simple website that displays the top one hundred words in a cluster of words with the help of D3.js resource. In order to do so, firstly, it pulls the all the data e.g. words and frequencies from RDS. Then, the data is fed into D3.js to create word cloud layout. In addition, the website displays the top ten words along with their frequencies below the cluster.

## The technology used

The three mechanisms below are run by Node.js which is an open source server framework that uses JavaScript on the server. It uses asynchronous programming style. Additionally, they rely heavily on node package manager (npm). A NPM named Express is used in Processor and Server component. We use a tool from Express called Express-generator to create an application skeleton for these components. We mostly ignore the default modules provided by express-generator. For Tweet-Stream component, Express-generator is not required since it is just a simple Node.js that is used to stream tweet and send a post request to the processor.

The following modules are included by default with express-generator:

- Body-parser: Node.js body parsing middleware. (Body-parser, n.d.)
- Cookie-parser: this service is used to parse Cookie header and populate req.cookies with an object keyed by cookies names. (cookie-parser, n.d.)
- Debug: A tiny JavaScript debugging utility modelled after Node.js core's debugging technique. (debug, n.d.)
- Ejs: embedded JavaScript templates. This module appears in Server component because it is being specified into express-generator. (ejs, n.d.)
- Jade: This is a high-performance template engine heavily influenced by Haml and implemented with JavaScript for node and browsers. This module appears in Processor component by default because we have not specified it to ejs. (jade, n.d.)
- Express: This is a Node.js framework that allows JavaScript to be used outside the Web Browsers, for creating web and network applications. (express, n.d.)
- Morgan: This is HTTP request logger middleware for node.js. (morgan, n.d.)
- Server-favicon: This is a middleware for serving a visual core that client software, like browsers, use to identify a site. (server-favicon, n.d.)

## Issues encountered

The first issue we encountered is the program was not getting enough loads. Fortunately, we were able to fix this problem by changing the way we are getting each tweet. Before, we were only getting the tweets that came with language specifying as English. This caused the program to ignore the tweets that did not come English language specified in the settings because not many users bother providing this information. We fixed this issue but making sure that we are getting all the tweets regardless of their language.

We did not expect the second issue to happen until one of my teammates discovered that he ran out of credits and could not work on AWS anymore. Luckily, my account is still functioning with a warning that I still have 50% credit left on my account. We are still not sure what might have caused his AWS credits to run. There are several things that might cause this issue. First, he left one instance running for a few days. Second, he was using cloud monitoring. Third, he was doing a lot of experimenting with auto-scaling with multiple instances running at once.

## Scaling and performance

There are several steps that must be taken to make the program scalable. Firstly, the processor component needs to be created into an instance on AWS and then it gets stored into an Amazon Machine Image (AMI).

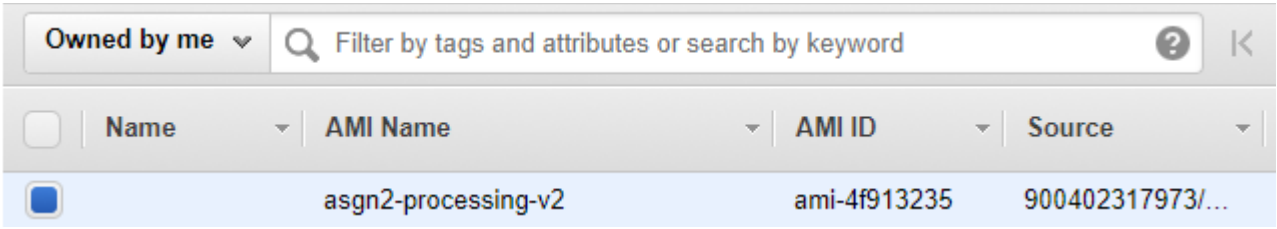



Image 2. The processor instance is created into an AMI.

### Load Balancer

Afterwards, a load balancer from AWS needs is created and the correcting setting must be configured in order for the load balancer to perform accordingly. The image below is the configuration of load balancer for this project. There are a few key things that I would like to point out about the setting that are crucial towards the server. Firstly, under load balancer section, the Port Configuration must be configured with 80 (HTTP) forwarding to 3080 (HTTP). Secondly, under health check section, the Ping Target needs to be set to HTTP:3080/ with 10 seconds timeout and 60 seconds interval because the processor needs to process a significant number of loads.

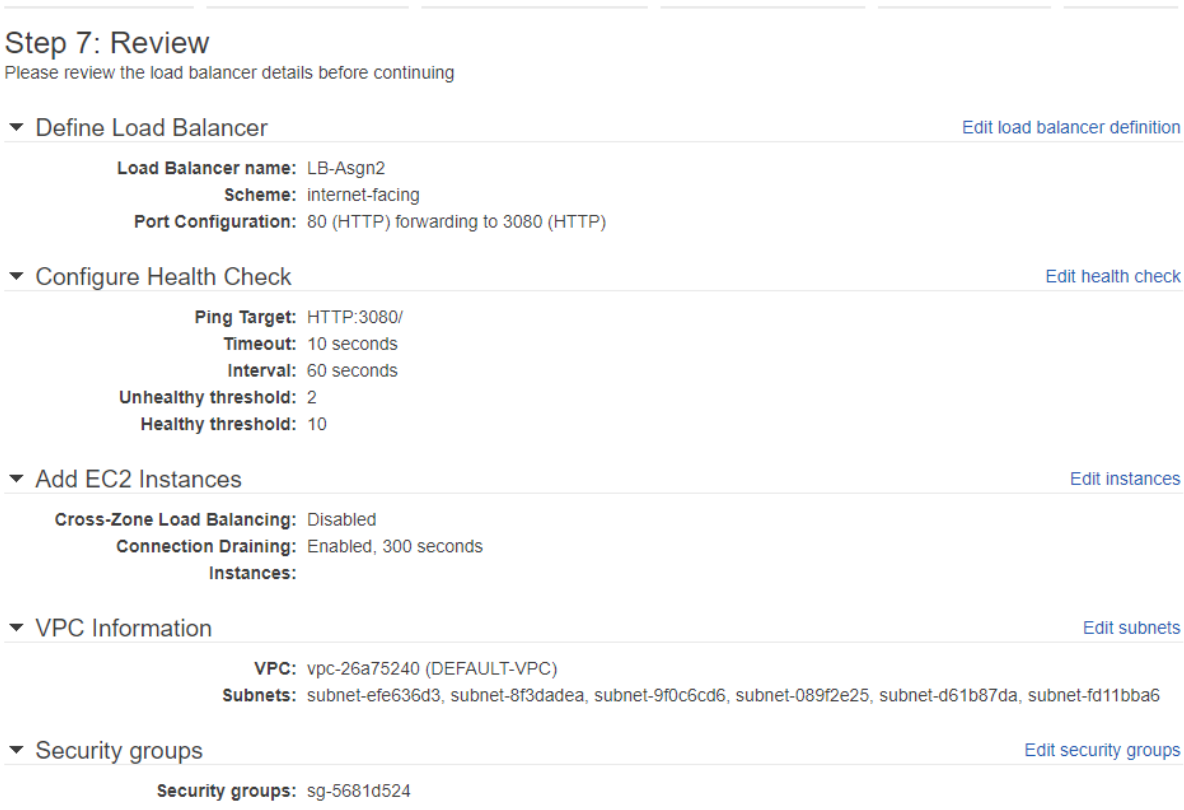



Image 3. Overall settings of Load Balancers

### Launch Configuration (Auto Scaling)

Once the load balancer is successfully created, a launch configuration for Auto-Scaling is created and the created AMI needs to be pointed into Auto-Scaling launch configuration.

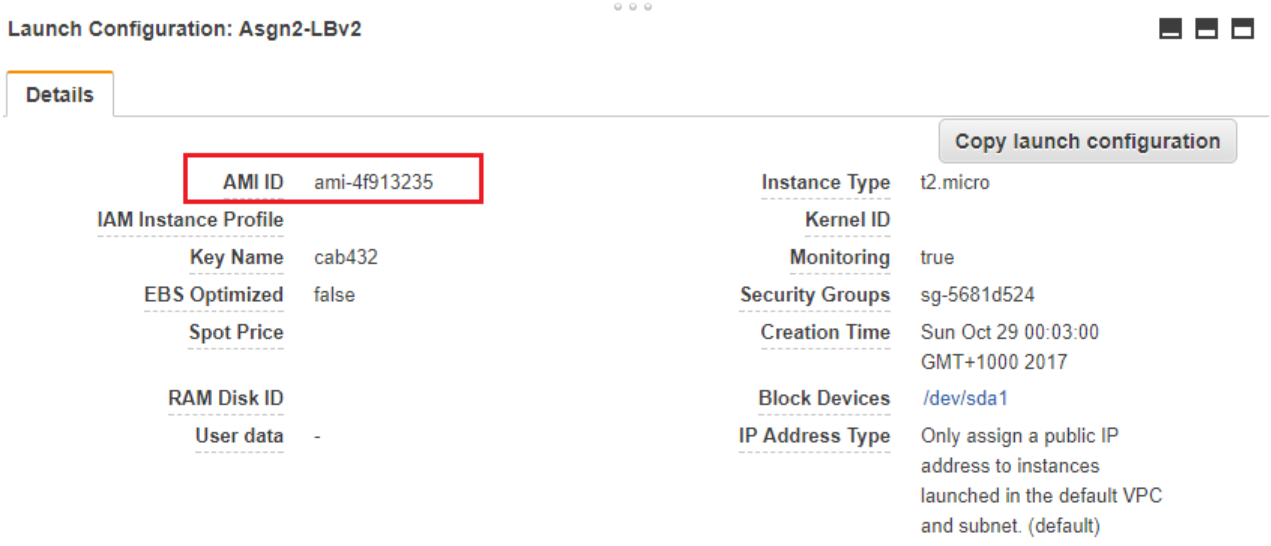



Image 4. Overall settings of Auto Scaling Launch Configuration

## Auto Scaling Groups

Once the launch configuration is created and configured successfully, Auto Scaling Group is needed to be made to specify some policies for the server to scale appropriately. There are several key points that we need to carefully consider when configuring the Auto-Scaling Groups. To begin with, it is necessary to specify the minimum and maximum group size. If you fail to do so, you will end up with only a single instance with no scaling. It is also compulsory to configure the scaling policies appropriately to make the web application scalable and more information about scaling will be shown under Auto Scaling Policies section.

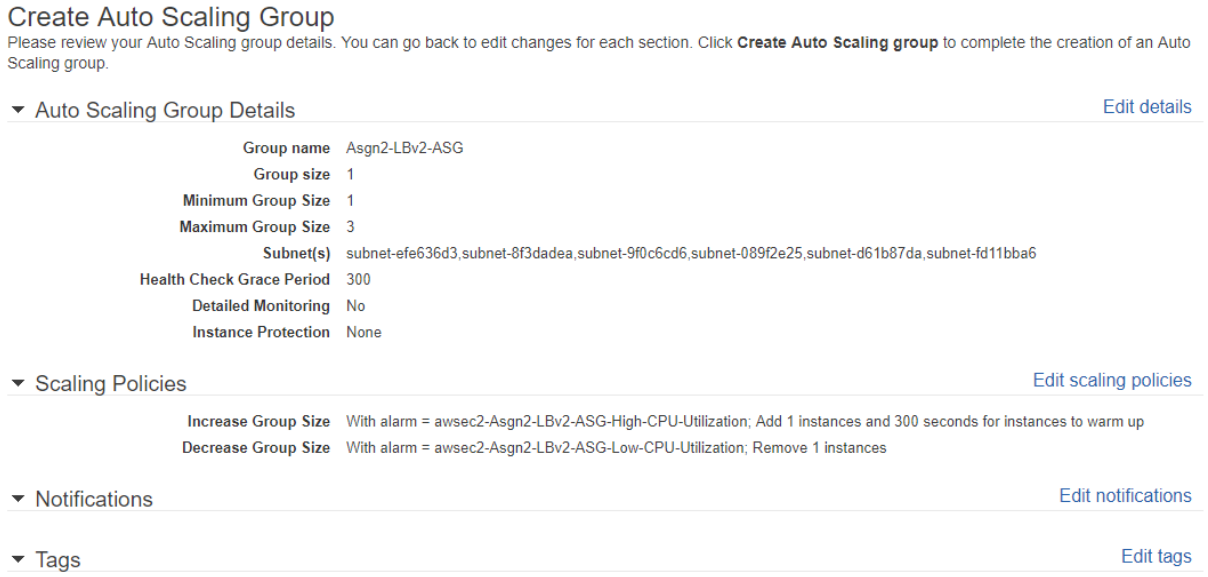



Image 5. Overall settings of Auto Scaling Groups

## Auto Scaling Policies

This table shows the policy and alarm for auto scaling increase group size. When the average CPU usage of the server reaches 50% or above for at least 1 consecutive period of 1 minute, it then creates a new instance with the default 300 seconds of instance warm up after each step.

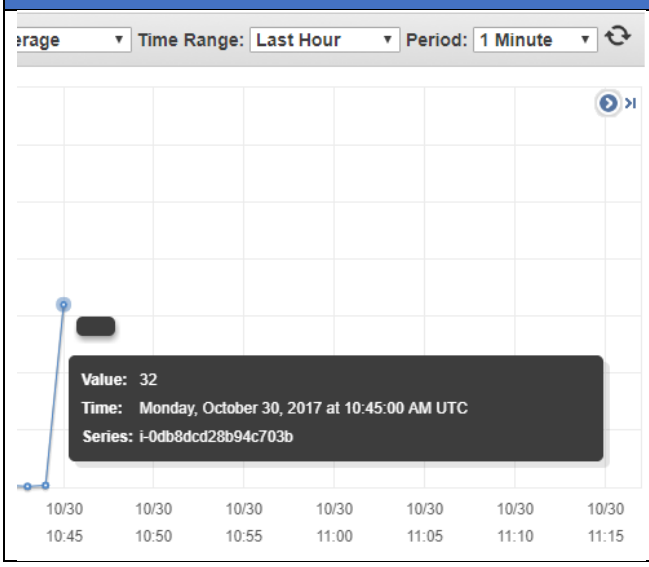
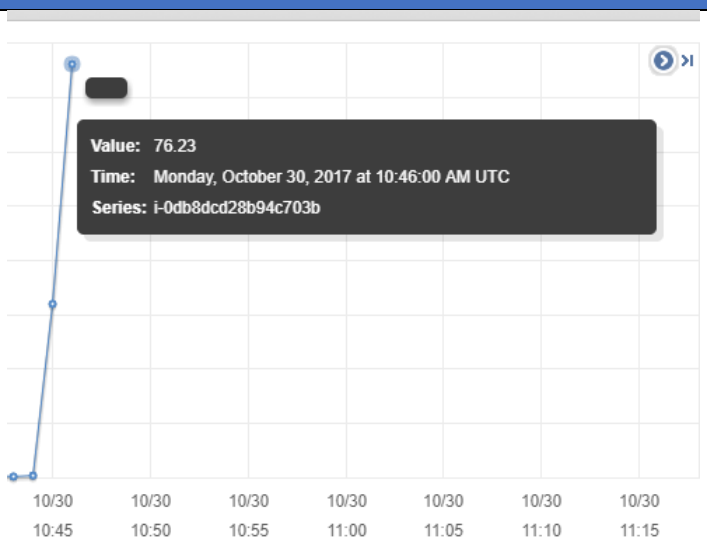
| Policy | Alarm |
| --- | --- |
| Increase Group Size<br>Name: Increase Group Size<br>Execute policy when: awsec2-Asgn2-LBv2-ASG-High-CPU-Utilization Edit Remove<br>breaches the alarm threshold: CPUUtilization >= 50 for 60 seconds<br>for the metric dimensions AutoScalingGroupName = Asgn2-LBv2-ASG<br>Take the action: Add 1 instances when 50 <= CPUUtilization < +infinity<br>Add step<br>Instances need: 300 seconds to warm up after each step | Send a notification to: No SNS topics found...<br>Whenever: Average of CPU Utilization<br>Is: >= 50 Percent<br>For at least: 1 consecutive period(s) of 1 Minute<br>Name of alarm: awsec2-Asgn2-LBv2-ASG-High-CPU-Utilization |

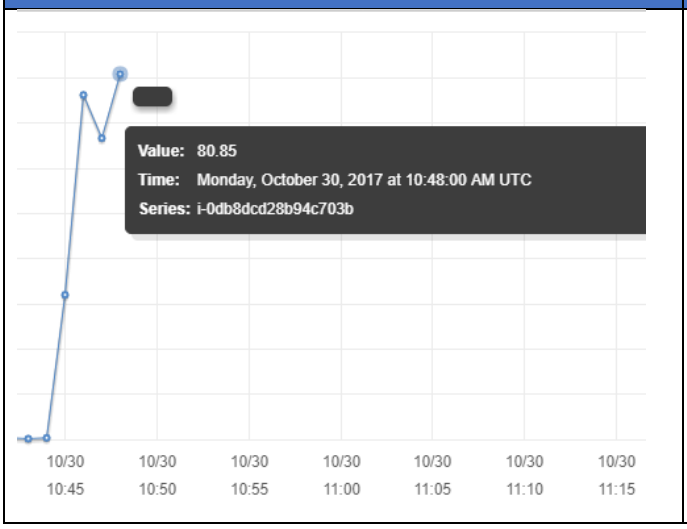
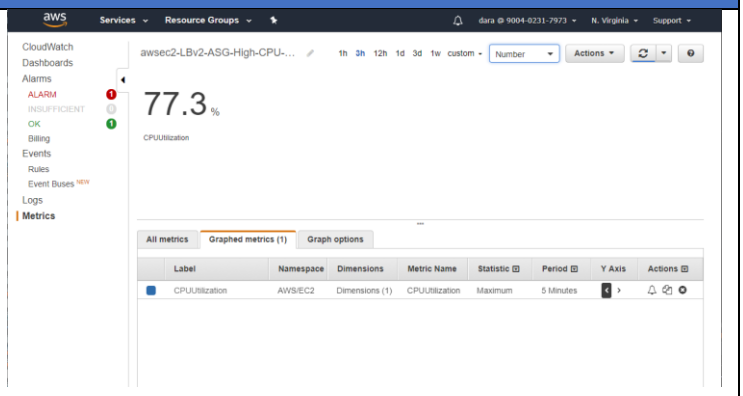
This table shows the policy and alarm for auto scaling decrease group size. When the average CPU usage of the server drops to 40% or below for at least 1 consecutive period of 1 minute, it then removes a new instance.

| Policy | Alarm |
| --- | --- |
| Decrease Group Size<br>Name: Decrease Group Size<br>Execute policy when: awsec2-Asgn2-LBv2-ASG-Low-CPU-Utilization Edit Remove<br>breaches the alarm threshold: CPUUtilization <= 40 for 60 seconds<br>for the metric dimensions AutoScalingGroupName = Asgn2-LBv2-ASG<br>Take the action: Remove 1 instances when 40 >= CPUUtilization > -infinity<br>Add step | Send a notification to: No SNS topics found...<br>Whenever: Average of CPU Utilization<br>Is: <= 40 Percent<br>For at least: 1 consecutive period(s) of 1 Minute<br>Name of alarm: awsec2-Asgn2-LBv2-ASG-Low-CPU-Utilization |

## Performance

The CPU usage of the server grows to around 30% in the first 30 seconds and then peaks to around 80% after 2 minutes. After 3 minutes, the CPU usage gradually decreases to around 77% and further to around 70%.

| 30 Seconds | 1 Minute |
| --- | --- |
|  |  |

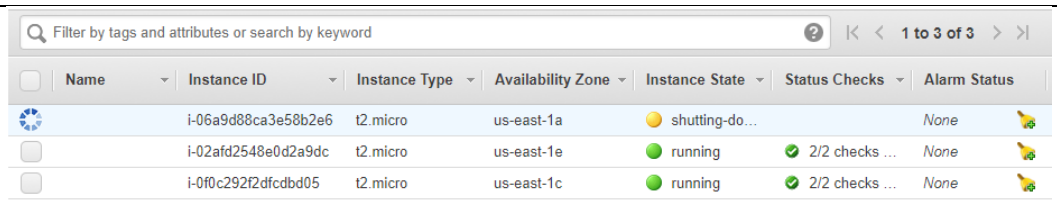
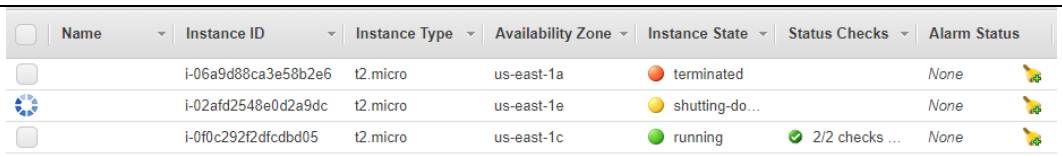
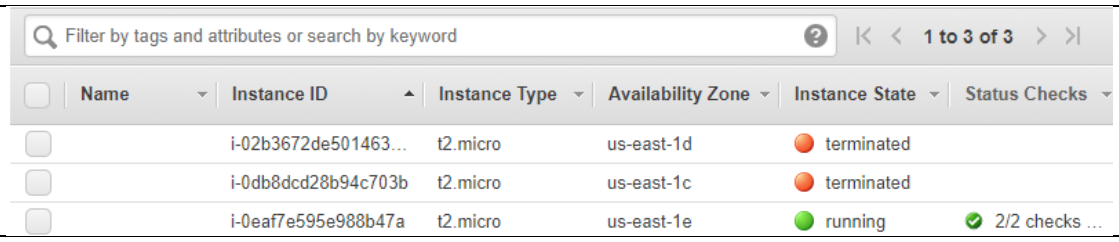
| 2 Minutes | After 3 Minutes |
|---|---|
|  |  |

Keep in mind that the performance varies depending the number of tweets the processor receives at that period of time. Sometimes, it might take up to about 4 to 7 minutes to get the server to reach its peak performance and 2 to 4 minutes to go above the auto scaling policy.

# Testing and limitations

## Test plan and results

| Test Plan | Results | Result |
|---|---|---|
| After auto-scaling is configured. | Filter by tags and attributes or search by keyword — 1 to 1 of 1<br>Name \| Instance ID \| Instance Type \| Availability Zone \| Instance State<br>i-0db8dcd28b94c703b \| t2.micro \| us-east-1c \| running | Passed |
| Another instance appears after 3 minutes of average CPU usage going over 50%. | Filter by tags and attributes or search by keyword — 1 to 2 of 2<br>Name \| Instance ID \| Instance Type \| Availability Zone \| Instance State \| Status Checks<br>i-02b3672de501463... \| t2.micro \| us-east-1d \| running \| Initializing<br>i-0db8dcd28b94c703b \| t2.micro \| us-east-1c \| running \| 2/2 checks ... | Passed |
| The third instance appears after another 4 minutes of average CPU usage going over 50%. | Name \| Instance ID \| Instance Type \| Availability Zone \| Instance State \| Status Checks<br>i-02b3672de501463... \| t2.micro \| us-east-1d \| running \| 2/2 checks ...<br>i-0db8dcd28b94c703b \| t2.micro \| us-east-1c \| running \| 2/2 checks ...<br>i-0eaf7e595e988b47a \| t2.micro \| us-east-1e \| running \| Initializing | Passed |

| | | |
|---|---|---|
| One instance is terminated after another 2 minutes of average CPU usage below 40%. |  | Passed |
| Another instance is terminated after another 3 minutes of average CPU usage below 40%. |  | Passed |
| Lastly, it goes back to 1 instance. |  | Passed |

## Timeline of the testing

This is the timeline of the average CPU usage during testing phase. The tweet starts hitting the processor at 10/29 – 4:55. The CPU usage rises to around 20% and then peaks at approximately 79%. After about 8 minutes which is when the tweet stops hitting the processor, it drops from around 78% to around 27% and it goes backs to around 0.245%.

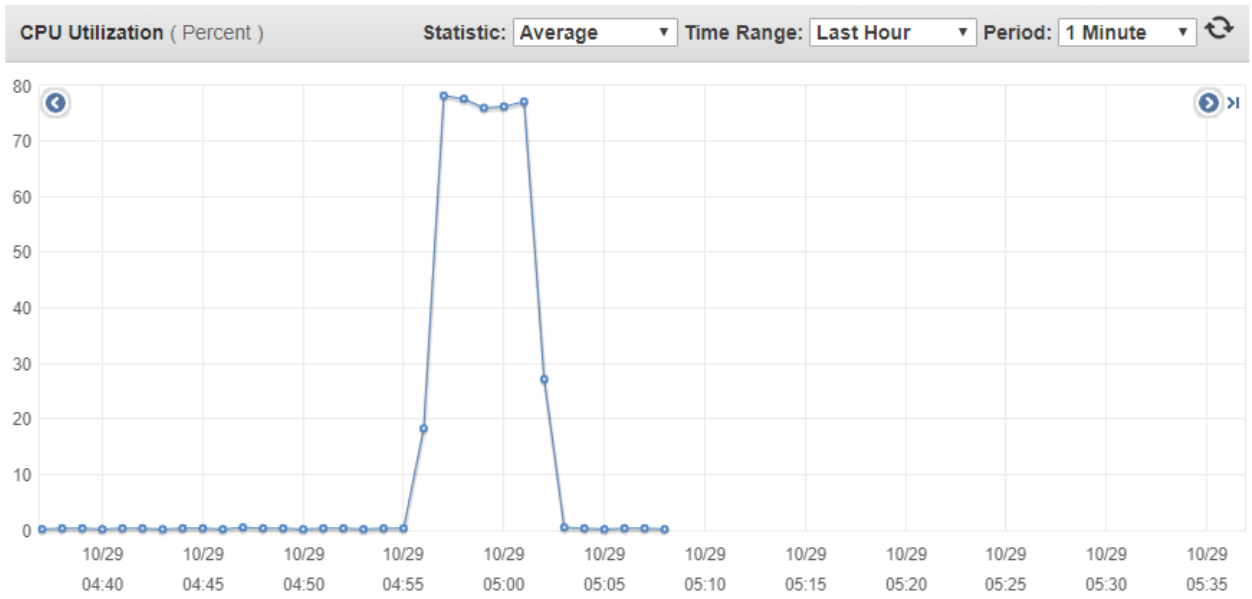



Image 6. Timeline of the CPU Utilization during testing.

### Compromises

Initially, one of the goals was to make the application scale up to 5 instances but during the implementation process, we found that more time and resources (AWS credits) are going to be required towards the web application in order to generate more loads. Fortunately, we managed to get the web application to scale up to 3 instances with the loads we have. Therefore, we decided to set the maximum number of auto scaling instances to 3 instances and we believe that it is sufficient enough to satisfy the auto scaling requirements.

### Limitations

RDS slow response – One of the limitations is that RDS does not provide a quick response when tweets are being pushed into the storage and pulled out to display on website. This results the website to slowly update the frequency of each word that get stored into RDS and being shown in top 10 words table that is used on the website. Data undefined error might occur when the web application is initialized with a new RDS because the response server is not instant.

AWS scaling policy – there a time interval before AWS scales up another instance. The minimum time interval is 1 minute. In a real word situation, this might cause some issues to the website when it is hitting its peak performance because AWS does not produce another instance straight away.

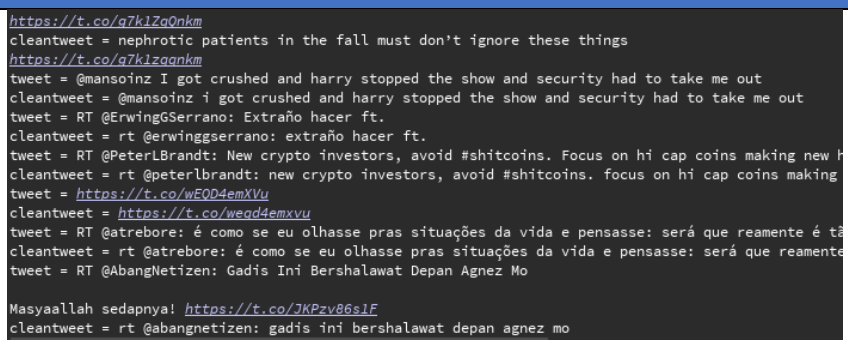
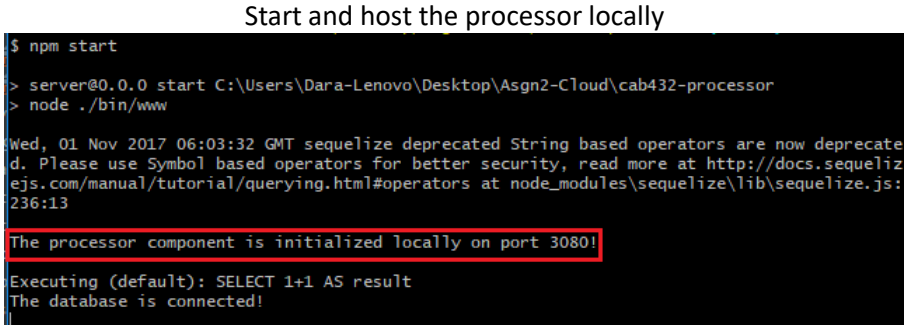
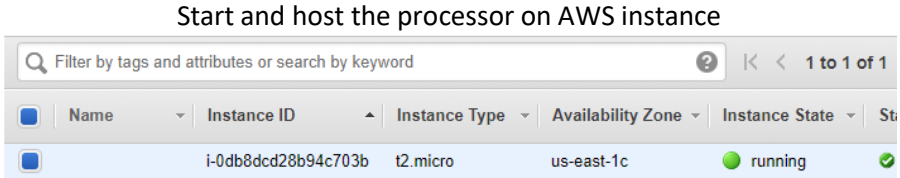
AWS Educator account – we decided to stop doing further testing with auto scaling once we found out that we are not going to have enough credits to carry through till the demonstration day. Fortunately, we managed to get the website to auto-scale appropriately according the assignment requirements, but we could have done more if there were more credits for the account.

## Possible extensions

One possible extension to add a search box to the website homepage that allows the users to provide their own keyword to see the tweets that contains the specified keyword. Since this assignment is heavily focused on generating loads and making the web application scalable, we decided to leave this out as a possible extension.

Second possible extension is to create a console box that display every incoming tweet live provided by the users specified keyword. This is a fancy feature to have and will make the web application looks more dynamic and user-friendly.

# Appendix: Brief User Guide

| Description | |
|---|---|
| To make the Twitter Stream runnable, it is required to provide several necessary keys for Twitter Streaming API such as, consumer key, consumer secret, access token key and access token secret.<br><br>The image on the right is showing the incoming tweets. |  |
| Before you run this component, you need to start the processor component to create a server for twitter stream component to do a post request. The processor filters each tweet and store them into RDS database. | Start and host the processor locally<br>  <br><br>Start and host the processor on AWS instance<br>  |
| The server provides a simple word cloud website that pulls the top 100 words from RDS and displays them into a cluster. A table of top 10 words are created with number frequency included for top 10 words.<br><br>An overall result of sentiment analysis is displayed above the top 10 table.<br><br>It can be clearly seen that the bigger words have more frequency than the words with less frequency. |  |

# References

*Body-parser*. (n.d.). Retrieved from npm: https://www.npmjs.com/package/body-parser

*cookie-parser*. (n.d.). Retrieved from npm: https://www.npmjs.com/package/cookie-parser

*debug*. (n.d.). Retrieved from npm: https://www.npmjs.com/package/debug

*ejs*. (n.d.). Retrieved from npm: https://www.npmjs.com/package/ejs

*express*. (n.d.). Retrieved from npm: https://www.npmjs.com/package/express

*jade*. (n.d.). Retrieved from npm: https://www.npmjs.com/package/jade

*morgan*. (n.d.). Retrieved from npm: https://www.npmjs.com/package/morgan

*server-favicon*. (n.d.). Retrieved from npm: https://www.npmjs.com/package/serve-favicon