Assignment 2 Report

CAB432 Cloud Computing – semester 2 2019

John Santias N9983244

Ka long Lee n9845097

2019

Contents

[Introduction 2](#_Toc22826327)

[Development 2](#_Toc22826328)

[Stage 1: 2](#_Toc22826329)

[Stage 2: 2](#_Toc22826330)

[Stage 3: 3](#_Toc22826331)

[Stage 4: 3](#_Toc22826332)

[Stage 5 (Extra feature): 3](#_Toc22826333)

[API and Packages 3](#_Toc22826334)

[Twitter API 3](#_Toc22826335)

[IBM Watson Natural Language Understanding API 3](#_Toc22826336)

[MongoDB 3](#_Toc22826337)

[Compromise 4](#_Toc22826338)

[ChartJS 4](#_Toc22826339)

[Use Cases 4](#_Toc22826340)

[Technical Description 6](#_Toc22826341)

[Cloud Architecture 6](#_Toc22826342)

[Server Architecture 7](#_Toc22826343)

[Client Side 7](#_Toc22826344)

[Server Side 7](#_Toc22826345)

[AWS Elastic Load Balance 8](#_Toc22826346)

[AWS Auto-Scaling Group 8](#_Toc22826347)

[AWS EC2 Server 8](#_Toc22826348)

[Docker 8](#_Toc22826349)

[MongoDB 9](#_Toc22826350)

[Scaling and Performance 9](#_Toc22826351)

[Scaling Up 10](#_Toc22826352)

[Scaling Down 11](#_Toc22826353)

[Testing and Limitations 12](#_Toc22826354)

[Test Plan 12](#_Toc22826355)

[Possible Extensions 13](#_Toc22826356)

[Scaling Twitter Stream 13](#_Toc22826357)

[D3JS Graphs 13](#_Toc22826358)

[Appendix 14](#_Toc22826359)

[Appendix A – Cloud Architecture design Iterations 14](#_Toc22826360)

[Appendix B – Deployment instructions 15](#_Toc22826361)

[Pre-Requisites 15](#_Toc22826362)

[Load Balancer 17](#_Toc22826363)

[Launch Configuration 17](#_Toc22826364)

[Auto-Scaling Group 18](#_Toc22826365)

[Appendix C – Test Cases 18](#_Toc22826366)

# Introduction

Tweetery aims to provide users with a visual representation of the emotional value of tweets on a search or trending topic. Utilising the Twitter API for getting tweets into the web server, cleaning the tweet body and performing emotional analysis with IBM Watson API. The average emotion scores of the chosen topic is displayed on a graph rendered on the client side.

When the user searches for a topic, the query is sent to the node server for further processing. The server uses the search query to get tweets from Twitter API. The received tweets are then combined and sent to IBM’s Watson Natural Language Understanding API to analyse the emotions. These results are stored on a remote global Redis Cache server which can be returned to users who searches for the same topic. Results are also stored on MongoDB for long term storage and can be copied to the cache upon user request. The node server will repeat this processing for each of the last seven days.

Between the client and node server, and the node server and Redis cache server, is a load balancer to assist in handling high volume of traffic that can be generated by popular trending topics and queries. The load balancer assists in the scaling of our servers.

# Development

Tweetery will be developed in stages in order to have tight grip on the scope of the application and reduce the number of potential issues towards the end of development. Difficulties experienced in each of the stages are further explained in `*Difficulties faced`* section.

## Stage 1:

Stage 1 involved setting up the skeleton of the project and generating the keys for the Twitter and IBM API. The skeleton for the node application consisted of two folders, one dedicated for the client, the other for the server. Dependencies were also installed for the node application to process API calls. As expected by the end of this stage, the application was able to call the Twitter API for trending topics and tweets based on single/multiple queries for emotional analysis using IBM’s API.

## Stage 2:

Stage 2 involved creating the front-end UI to display components such as the trending topics list, a navigation bar, search bar and a graph for the processed emotional analysis. It was ensured that client side was calling the correct server endpoints for heavy processing.

## Stage 3:

Stage 3 involved setting up two storage services for the node application to store and retrieve data. This included setting up a Redis cache server and a mongoDB database. The node application was connected to both services, then stored and retrieved data based on a defined Schema. Afterwards, testing and bug fixing was conducted followed by dockerising the application.

## Stage 4:

Stage 4 involved deploying to the cloud with further testing and bug fixing. Monitoring the application was an additional step to this stage to find the most optimal way for ensure auto-scaling. This involved setting up CloudWatch alarms and monitoring the application as it was being used.

During this monitoring process it was discovered that the application was not producing enough CPU load for scaling, so we tried to move the redis server from being a global cache to a docker container image linked to our node server. We also tried to increase the processing load by adding an NLP library and sorting functions. This issue is further explained in `Scaling and Performance` section.

## Stage 5 (Extra feature):

We attempted to add a new feature that would analyse the emotion of real time tweets and plot new results on the graph every 3 three seconds. We discovered that scaling the twitter stream was a problem when multiple users hit it with different search queries. The twitter streaming endpoint only allows one connection at a time. We decided to keep this as an extra feature however only allowing one request at a time.

# API and Packages

## Twitter API

<https://developer.twitter.com/en/docs.html>

Twitter is a social network platform for users to share messages, images and videos on a feed. Messages can be categorised by hashtags (#) and searches can be conducted on those hashtags or keywords. Tweetery uses this API to retrieve tweets based on a hashtag or keyword.

## IBM Watson Natural Language Understanding API

<https://www.ibm.com/watson/service/natural-language-understanding>

The IBM Watson Natural Language Understanding API is one of IBM’s Watsons machine learning API. It is used to process advanced test analysis and extract metadata from content such as concepts, entities, keywords, categories, sentiment, emotions, relations, and semantic roles. Some of which can provide a score for the tone or a list of words that have great meaning for the user.

Tweetery uses this API to analyse the emotion of the tweets gathered and return a score between zero and one. Given the highest score is the emotion analysed in the content.

## MongoDB

<https://www.mongodb.com> & <https://mlab.com>

MongoDB is a cross-platform document-oriented database program. Classified as a NoSQL database program and uses JSON-like documents with schema. mLab was used to host MongoDB databases on a cloud provider (AWS, GCP, Azure). Tweetery uses two tables to store emotion analysis, .

## Compromise

<http://compromise.cool/>

Compromise is a JavaScript library for processing natural language. Tweetery uses utilises compromise to clean tweets, removing invalid or unusual characters. This increases the accuracy and the likelihood of running a proper analysis from IBM’s API.

## ChartJS

<https://www.chartjs.org>

ChartJS is a community-maintained project for developers to visualise data in eight different ways. Each of them animated and customisable. Tweetery uses this library to visualise the emotional scores of the given topic.

## Socket.io

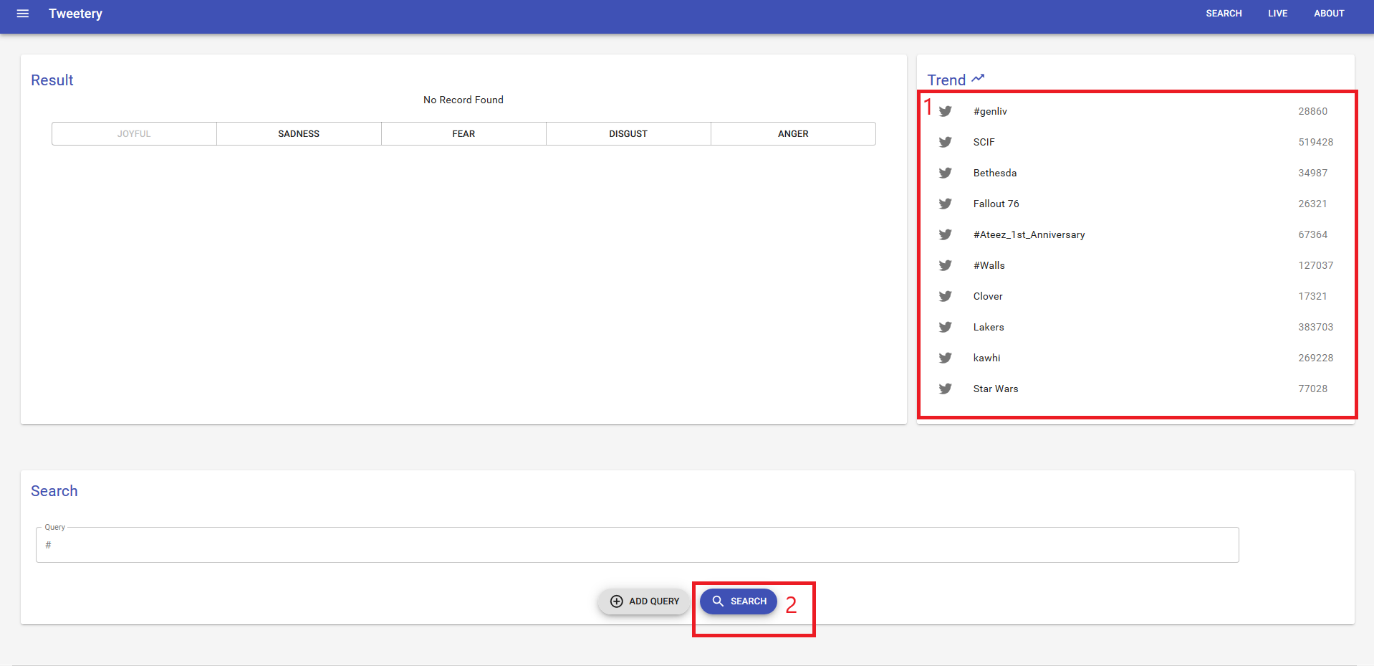
<https://socket.io/>

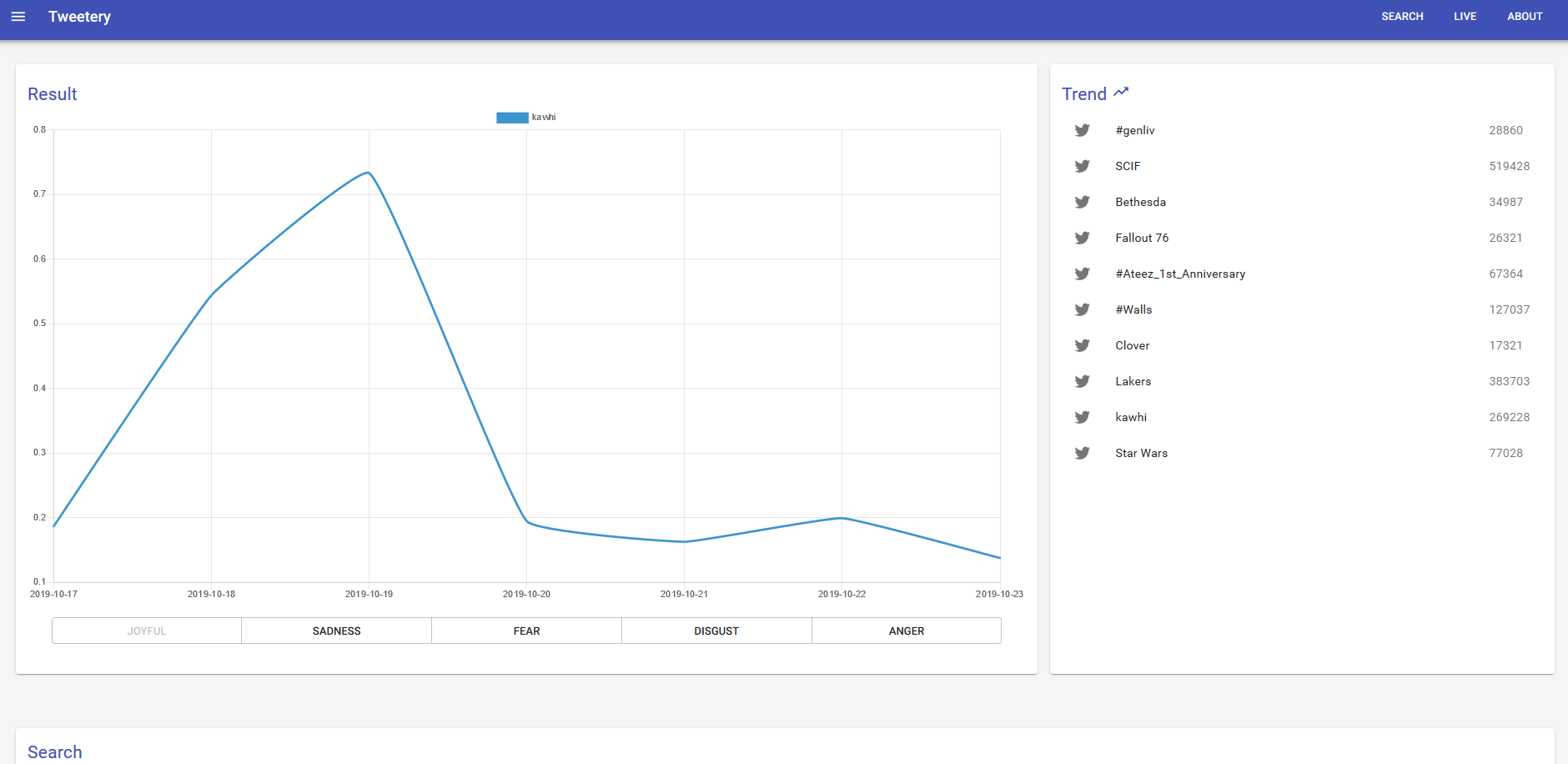
Socket.io is a JavaScript library for building real time web applications. It enables real time, bi-directional communication between web clients and server’s application. Tweetery uses this library to achieve real time analytics of tweets.

# Use Cases

*As a user, I want to see a list of trending topics and select one to see the average emotional stats.*

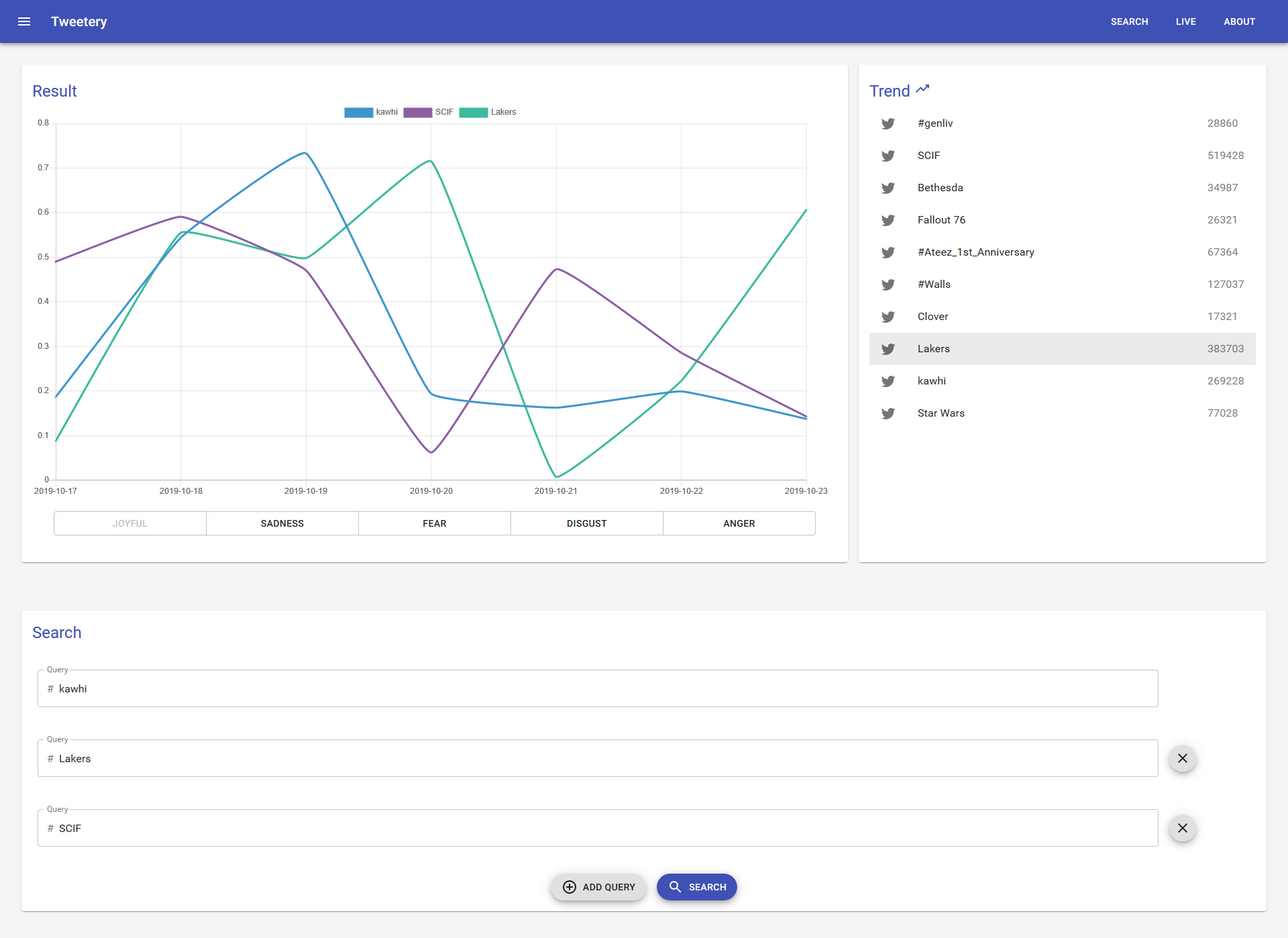
The user lands on the index page of Tweetery. The user selects a trending topic from the box on the right and selects search to initiate the request and view the results.





*As a user, I want to select or search multiple trending topics to compare the emotions of those topics.*

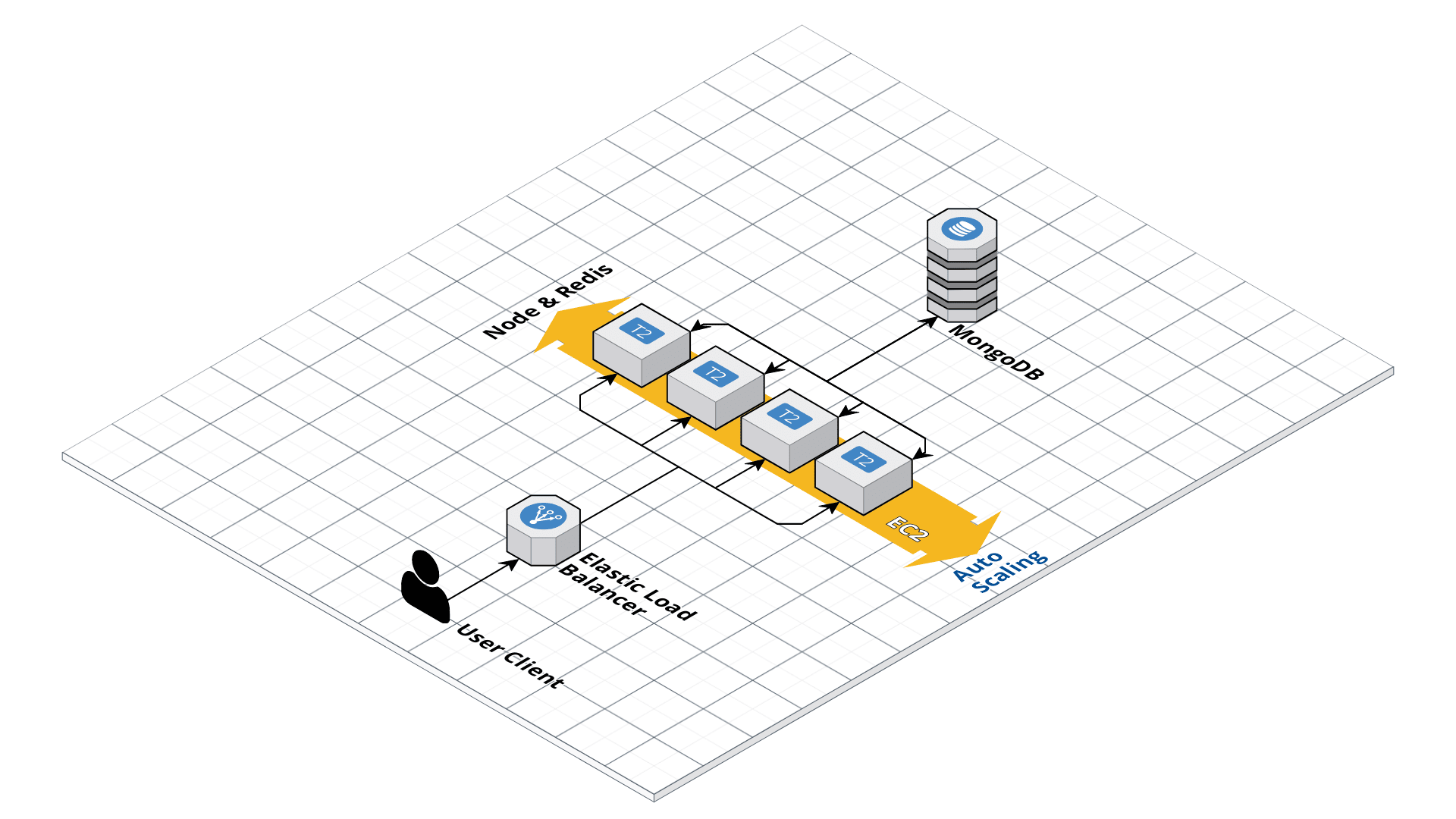
User selects multiple topics and initiates the search. Results will show one line for each query.



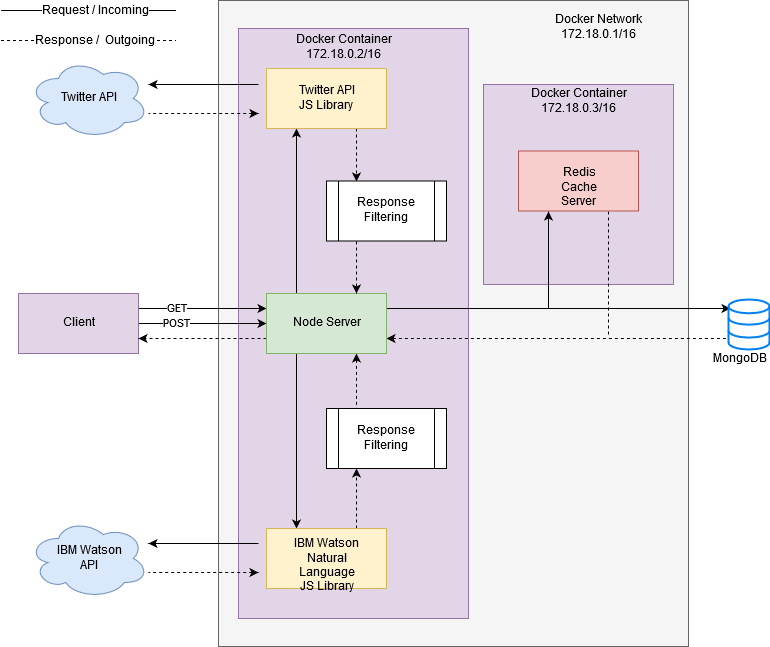
# Technical Description

Tweetery utilises one EC2 t2.micro server achieve its tasks in a stateless manner. Inside this server runs two docker containers inside of a bridged network. One running NodeJS application to process user requests, and the other running a redis server to cache data. In addition to this, MongoDB is utilised to provide persistence.

## Cloud Architecture



## Server Architecture



# Client Side

The client can initiate two types of request, a GET and a POST request. When a user requests to access the application, a GET request is sent to the server. The server returns the index page with a nav bar, a list of trending topics and a search bar. When the user initiates a search, the form body sends a POST request to the server for processing.

The server handles the query sent from the form, processes the API calls with the query and displays the results in a graph. The client uses the ReactJS framework to render the page and information being sent to the user.

Users can click on the Stream button to active the stream mode. When stream mode is active, the client-side application will constantly connect to the server receiving the real time analysis data of the query instead of the data recorded in the pass seven day. The server will send the analysis data to the client in every ten second. The web application then will plot the new data on the line chart. Finally, the client will disconnect to the server when user close or refresh the page.

# Server Side

The node server manages the incoming client request and outgoing responses. When it receives a GET request, the server renders and returns the home page with a nav bar, search bar and trending topics. There is no heavy processing in this request because the client hasn’t searched for anything. The client is only requesting the home page.

The heavy processing is initiated when the server receives a POST request containing the search queries. The server processes one query at a time. Using a query to search tweets for each of the last seven days. By hitting the twitter search endpoint, it returns a JSON object with tweet data stored in an array. Each element containing the tweet message, user information, time, date etc. At this point the server extracts the tweet messages, combines them and sends it to IBM’s API for emotional analysis. The result of this analysis is pushed to a JSON object with the date and query, then stored in the Redis cache for short-term storage and mongoDB for long-term storage. This information is also rendered and sent to the client for display. This process is repeated for each search query.

The server runs in a stateless manner, relying on storage services and APIs for storing and searching information. In every search initiated, the server always checks if the data exists on Redis and mongoDB before running a new search on the APIs.

The server also provides real time analytics services for clients. Socket.io is implemented in this project using WebSocket protocol for constantly sending real time analysis data to the client application. When stream mode is active in the client-side application, the server will then assign the client in specific “room” according the query name the client provided. Room, in Socket.io, refers to an arbitrary channel that clients can join and leave. It helps server to send the latest emotion analysis of the query to multiple users who are interest in this query. For instance, there are two users requiring the real time emotion analysis of tweets related to Donald Trump. The server will automatically create a room named as Donald Trump and assign these users in the room when one of these users sending the request at the first time. After that the server will constantly broadcast the latest emotion analysis data to all the users in the room every 5 second. The server only processes the data analytics once, then sends the data to multiple users in the room. It dramatically reduces the workload in the server. Finally, the server will automatically check the room usage in every minute. If there is no user in the room, the room and the data analytics process in the room will be removed.

## AWS Elastic Load Balance

An AWS Elastic Load Balancer automatically distributes incoming application traffic across multiple targets. Tweetery utilises a load balancer to balance the traffic between all the instances in the Auto-Scaling group.

## AWS Auto-Scaling Group

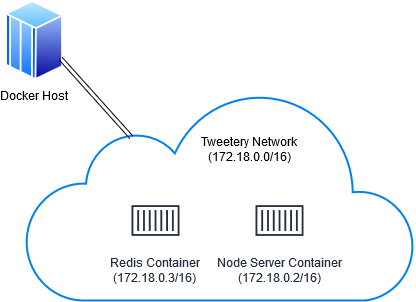
An AWS Auto-Scaling group creates EC2 instances dynamically based on the defined conditions. It maintains the number of instances by performing periodic health checks on the instances in the group. Tweetery utilises this to ensure consistency by scaling when server’s resources are overloaded.

## AWS EC2 Server

All of the servers are launched as a t2.micro server in the Asia Pacific (Sydney) region. Each of the server runs a custom bash script on boot to check for the latest version of the docker image. If there’s a new version the server will pull and run it. This was used throughout development, allowing me to make changes locally on my machine, create and push the docker image and reboot the server.

## Docker

The Node and Redis servers are separated into its own containers and linked to each other for storage. These containers are run inside of an independent network (172.18.0. 0/16) connected to the default network bridge for ease of access and minimising multiple port forwarding rules. By default, public users can connect to the node application through port 5000 and the node application can access the cache through port 6379.



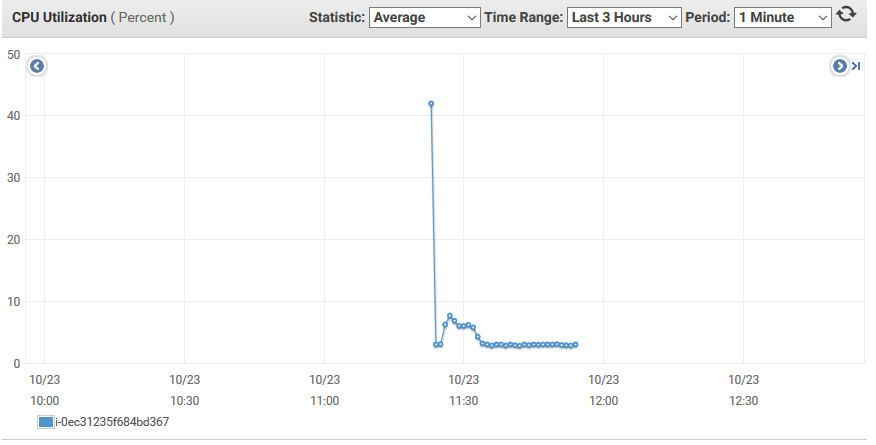
## MongoDB

MongoDB was remotely hosted through mLab. Tweetery utilises a NoSQL database to write and retrieve records. Collections ‘Trends’ to store trending data, and ‘Emotions’ to store emotional data. MongoDB was chosen due to the ease of use and documented-oriented storage suitable for our JSON data.

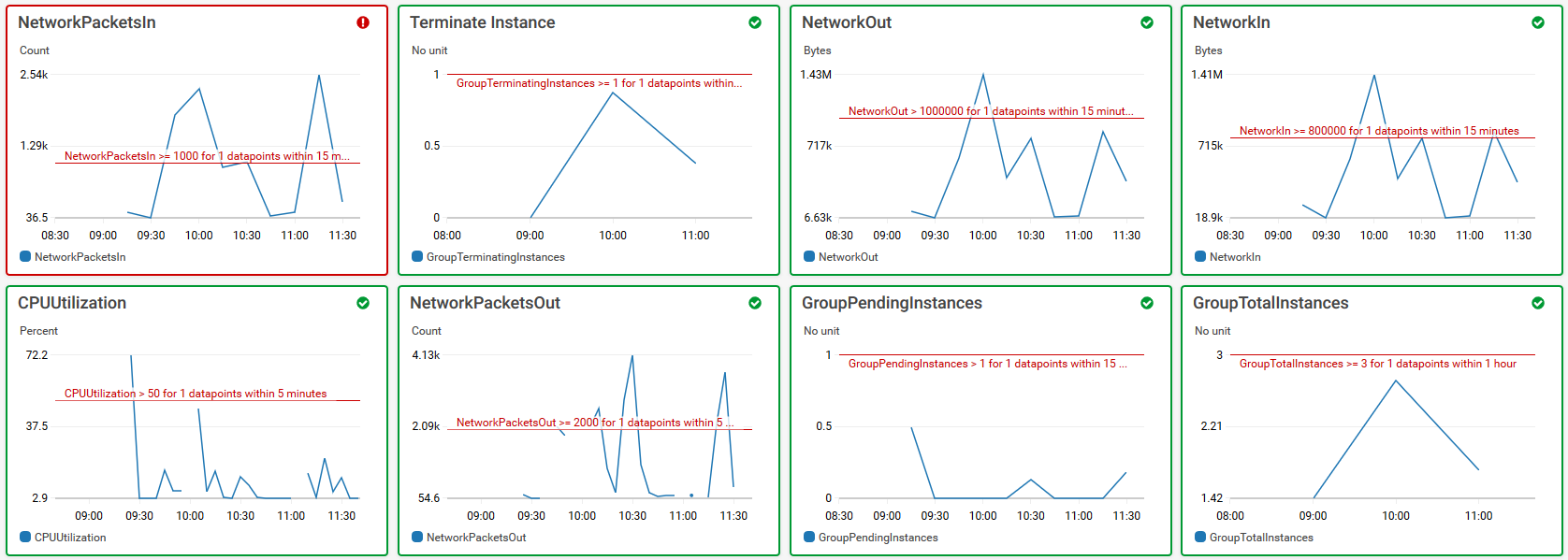
# Scaling and Performance

The auto scaling group has been configured to scale based on the size of network traffic coming in from the load balancer. This was chosen over CPU utilisation because server processing was very low and wasn’t ideal to scale when less than 10% of the CPU was used.

Thorough investigation went into the metrics used to scale the application. CPU utilisation never hit peaks that warranted extra server capacity as it managed to handle client request quickly. At the launch of a new instance, the server would utilise around 40% of the CPU. During this time the server would start the docker containers and establish new connections to the storage services. The usage would drop to 3% after everything was set up. We found that initiating search with new queries would slightly increase the processing to 6%. This was due to the information not available on our storage services and the server would have to process API calls, structure the data, store it and return to client. New information takes more time to process. If the same information was requested, then the server can retrieve the information from the cache and database resulting in lesser processing and quicker response time.



We attempted to increase the CPU usage by adding sorting and cleaning functions, and an NPM package such as NLP Compromise. As a result, there was no increase and finalised the decision to scale based on incoming network traffic. This analysis was conducted by setting up alarms in AWS CloudWatch to monitor the scaling group, server behaviour, and configuring the scaling policies based on what was observed.

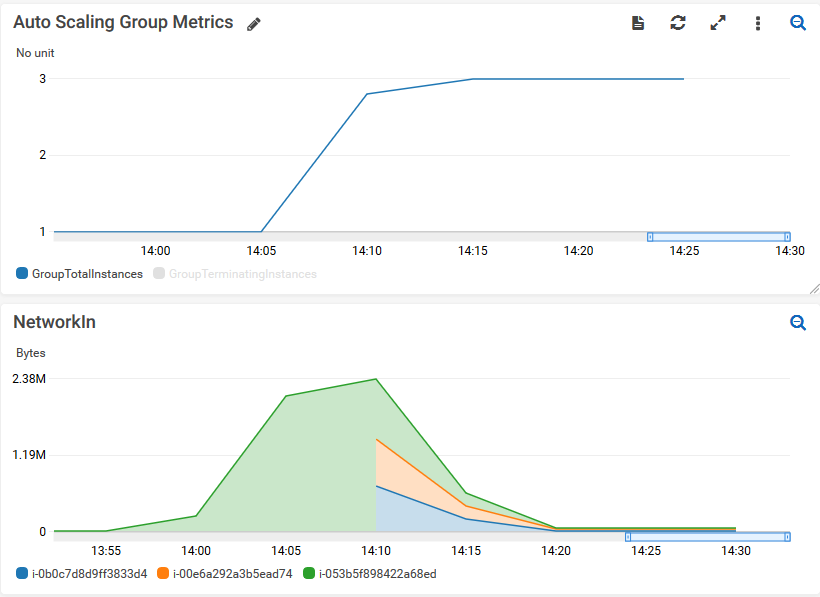


(write up about the robustness of node, and the stateless could be why cpu utilisation is so low)

For the purpose of this assignment, scaling policy based on incoming network traffic was the best way to demonstrate scaling up and down, as it could be controlled by the client. The trending topics and analysed data and be requested multiple times, resulting in an increase of traffic which initiates scaling.

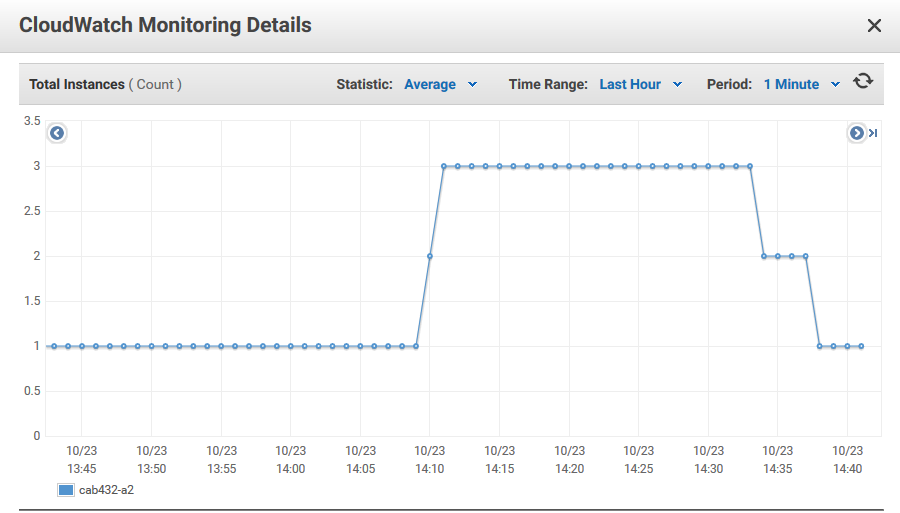
## Scaling Up

To increase the size of the auto scaling group, a policy was created based on the incoming network traffic. If the network traffic was greater than 1.5 million bytes for a period of 60 seconds, then it would add one instance to the auto scaling group.



## Scaling Down

To decrease the size of the auto scaling group, a policy was created based on the incoming network traffic. If the network traffic was less than 1.5 million bytes for a period of 60 seconds, then it would remove one instance of the auto scaling group.



# Testing and Limitations

When Tweetery was first deployed, it was expected that the application would scale based on CPU utilisation. When this was not the case, more time was invested into adding more processing (functions and libraries) and analysing the cloud infrastructure to determine the best method for scaling. As a result, error handling, restructuring code, code commenting, and bug fixing was conducted to create a robust application.

## Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| **TASK** | **Expected Outcome** | **Result** | **Screenshot/s (Appendix)** |
| Retrieve Home | Display the search box | **PASS** | **C – 1A** |
| Display the trending topics | **PASS** | **C – 1B** |
| Search tweets | Search button sends query | **PASS** | **C – 2A** |
| Graph displays | **PASS** | **C – 3B** |
| Graph displays Legend | **PASS** | **C – 3C** |
| Graph displays Labels | **PASS** | **C – 4A** |
| Graph displays Date | **PASS** | **C – 3D** |
| Selected topic adds to search box | **PASS** | **C – 5A** |
| Add row for search query | **PASS** | **C – 8A** |
| Delete search row | **PASS** | **C – 8A** |
| Initiate search with more than one topic | **PASS** | **C – 6A, 7A** |
| Multiple queries displays on the graph | **PASS** | **C – 7B** |
| Handle Twitter API response error | Application continues running, display error |  |  |
| Handle IBM Natural Language Understanding API response Error | Application continues running, display error |  |  |
| Stream live tweets | Live tweets analysed and displayed on the graph |  |  |
| Stream doesn’t write after close |  |  |
| API request errors shown to the user without crashing the application |  |  |
| Data to and from Redis | Node server stores data in Redis by specified key | **PASS** | **C – 9A** |
| Node server retrieves data from Redis by key specified | **PASS** | **C – 10A** |
| Data to and from MongoDB | Node server stores data in mongoDB | **PASS** | **C – 11A** |
| Node server retrieves data from mongoDB by key specified | **PASS** | **C – 12A** |
| Load Balancer | Load balancer health check | **PASS** | **C – 13A** |
| Auto Scaling | Auto scale up | **PASS** | **C – 14A** |
| Auto scale down | **PASS** | **C – 14A** |

# Possible Extensions

## Scaling Twitter Stream

Although the twitter stream was implemented, there is still room for improving this feature. One solution is searching all the tweets based on all the trending topics and store it in the database. From there the server could query the database, buffer returned data and display to user.

Applying this solution would possibly lead to restructuring our cloud architecture and the node application itself. Thus, resulting in more time developing, investigating and observing AWS configurations (load balancer, scaling policy).

## D3JS Graphs

The application current uses ChartJS to display the data in a line graph. D3JS could be substituted in to display data in a more attractive way that engages users. The variety of charting options is far greater.

## CICD Pipelines

# Appendix

## Appendix A – Cloud Architecture design Iterations

Our initial design separated the Redis Server from our node server. Our Redis server was going to be used as a global cache and scales when the CPU reaches the threshold. This approach would be more accurate in terms of the data requested and reduce the number of queries on the database. The downside to this was that it would take longer to search Redis.

We investigated this architecture and finalised the decision to combine both servers. This was due to the number of EC2 instances launched. AWS allows users to run up to five of the same instance type and choosing a different type would cost money. We realised that the data would be stored on the cache for a short period of time and are optimised to use less space. This approach would make the data on each server inconsistent, however this can be eliminated with the use of MongoDB.

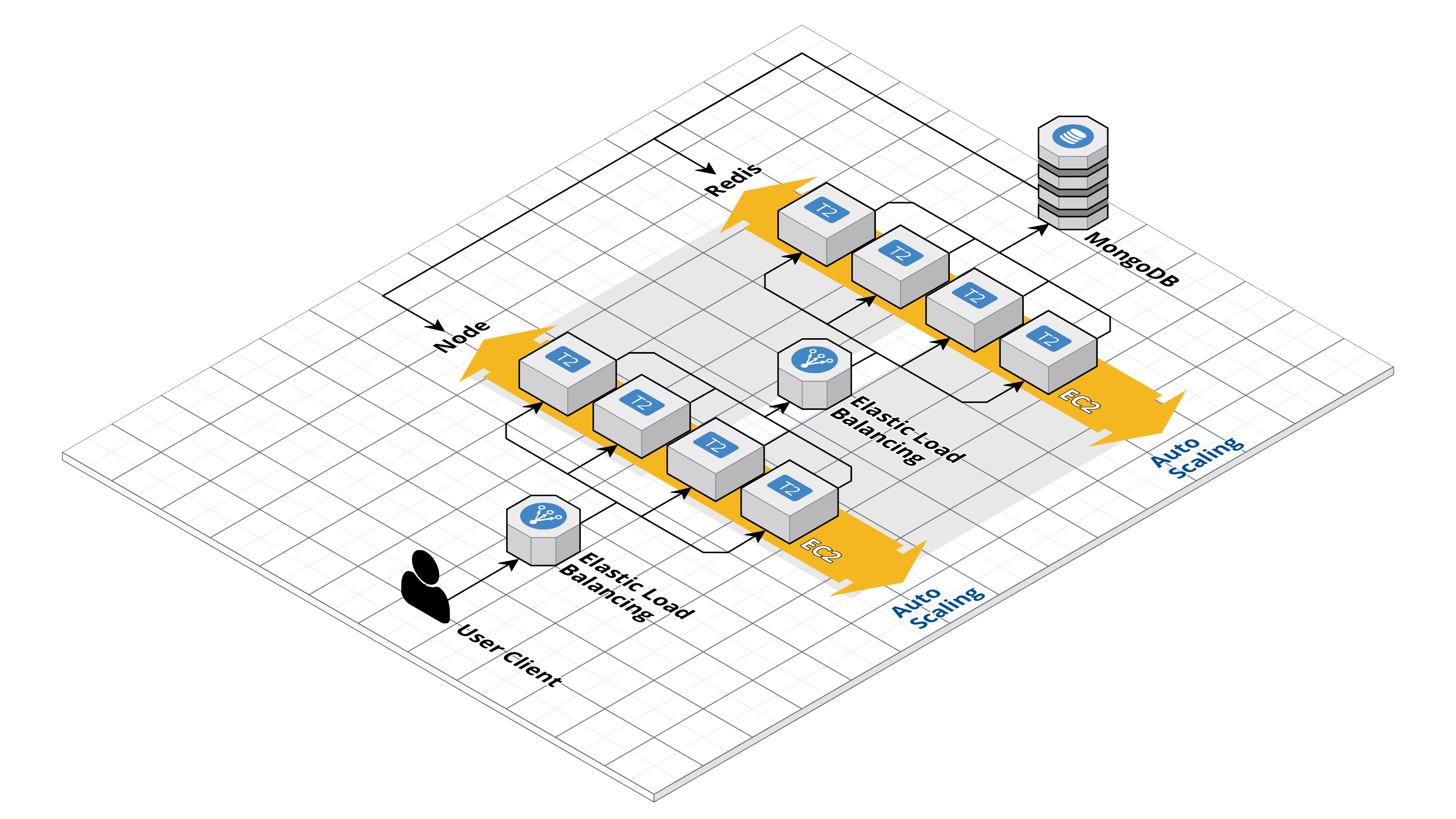


Figure - First Iteration

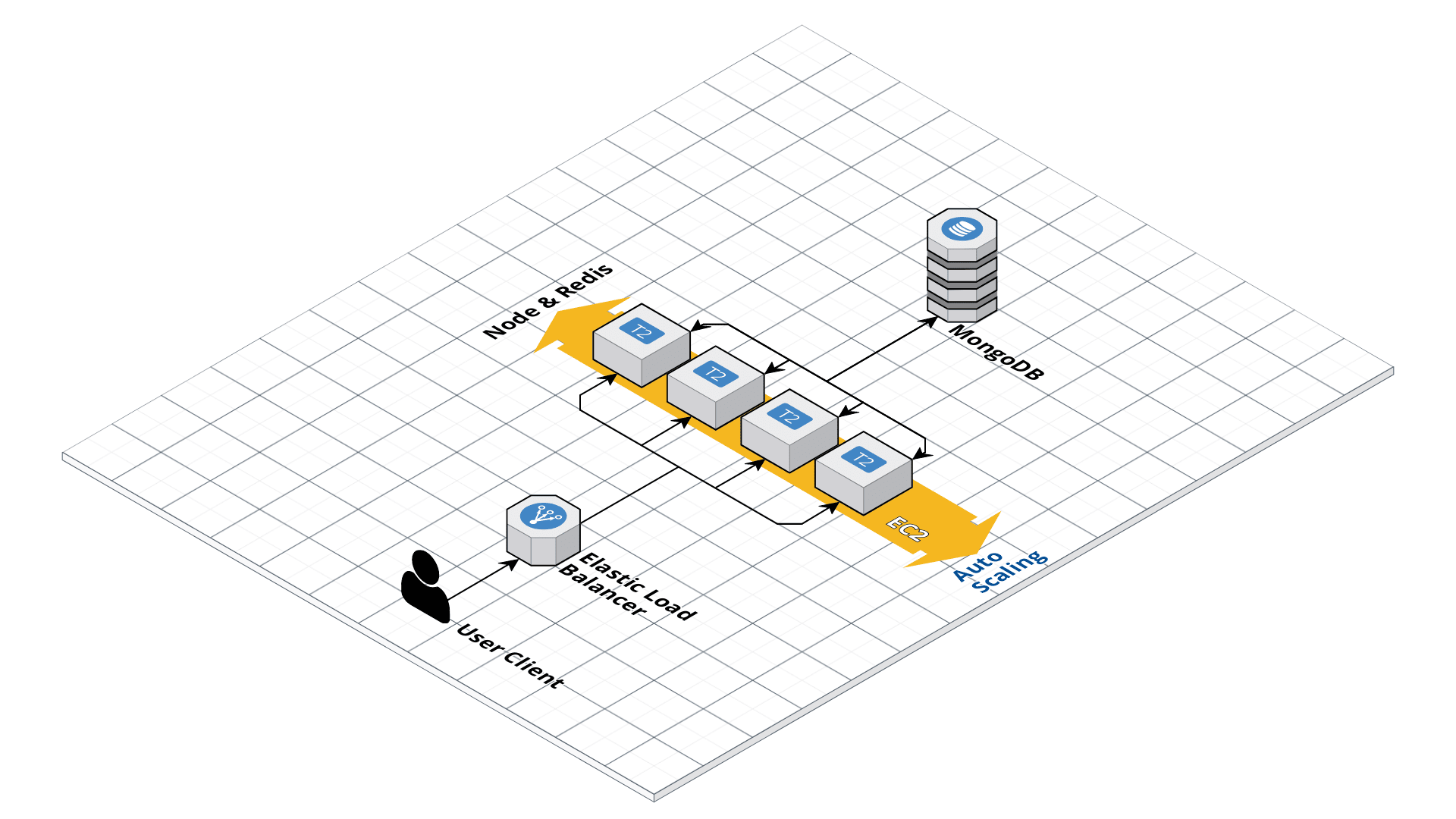


Figure - Second Iteration

## Appendix B – Deployment instructions

### Pre-Requisites

**Security Groups**

It is required that three security groups are created. One for running the application and cache, another for the load balancer, and one for the auto-scaling group. This is configured in the Security Groups section of the AWS EC2 management console.

1. **CAB432-Server**
   1. Inbound Port 22 SSH – TCP Protocol – Any Source
   2. Inbound Port 5000 – TCP Rule – Any Source
   3. Outbound – Open All Traffic
2. **Load-Balancer**
   1. Inbound Port 80 HTTP – TCP Rule – Any Source
   2. Outbound – Open All Traffic
3. **Auto-Scaling-Group**
   1. Inbound Port 22 SSH – TCP Protocol – Any Source
   2. Inbound Port 5000 – TCP Rule – Any Source
   3. Outbound – Open All Traffic

**AMI**

An Amazon Machine Image (AMI) is needed for the auto scaling group. Create an EC2 t2.micro instance with Ubuntu 18.04 and apply the security group **CAB432-Server**.

1. Install docker and configure it to start on boot:

sudo systemctl enable docker

1. Install docker compose. This tool is used to define and run multi-container docker applications. Once installed, create a new file docker-compose.yml in /home/ubuntu/ with the following:

version: '2'

services:

    node-server:

        image: asianjohnboi/tweetery:latest

        ports:

          - "5000:5000"

        networks:

          tweetery-network:

            ipv4\_address: "172.18.0.2"

        restart: unless-stopped

    redis-server:

        image: redis

        ports:

          - "6379:6379"

        networks:

          tweetery-network:

            ipv4\_address: "172.18.0.3"

        restart: unless-stopped

networks:

  tweetery-network:

    driver: bridge

    ipam:

      config:

        - subnet: 172.18.0.0/16

1. Run docker-compose. This will set up a new docker network, then pull and run the images inside that network.
2. Next create another file called startup.sh in the same directory. This bash script file will be used to pull the latest version of the tweetery image (if available).

#!/bin/bash

docker rm ubuntu\_node-server\_1 --force

docker pull asianjohnboi/tweetery:latest

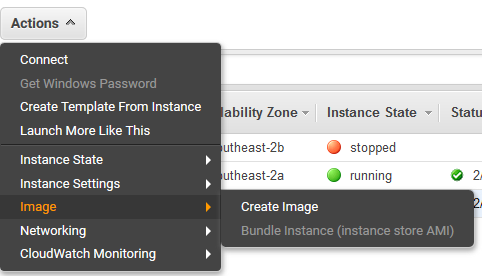
docker run --name ubuntu\_node-server\_1 --restart=unless-stopped \

            --network=ubuntu\_tweetery-network \

            --ip=172.18.0.2 \

            -p 5000:5000 asianjohnboi/tweetery:latest

1. Create a rule to execute to execute this file on boot.
   1. Run crontab -e
   2. Add @reboot sudo /home/ubuntu/startup.sh >> /home/ubuntu/output.log at the end of the file
   3. Save and reboot.
   4. Open the output.log file to view the outcome of executing the bash script.
   5. Check if the docker container is running.
2. When everything is running as expected, create an image of the running EC2 instance from AWS EC2 management console.



1. The image can then be terminated.

### Load Balancer

Create a load balancer from the AWS EC2 Management console.

1. Select Application Load Balancer
2. Configure load balancer
   1. Name the load balancer
   2. Select internet-facing scheme
   3. Address type of ipv4
   4. HTTP listener on port 80
   5. Select two availability zones
3. Do not configure any security settings
4. Add **Load-Balancer** security group
5. Configure routing
   1. Create new target group and name it
   2. Target type is instance
   3. Using HTTP Protocol port 80
   4. Leave health checks by default.
6. Do not register any targets.
7. Review and create Load Balancer

### Launch Configuration

Create a launch configuration from the AWS EC2 Management console.

1. Select your recently created AMI.
2. Use the t2.micro instance type
3. Name the configuration and enable CloudWatch detailed monitoring.
4. Skip additional storage.
5. Select **Auto-Scaling-Group** security group.
6. Review and Create launch configuration.

### Auto-Scaling Group

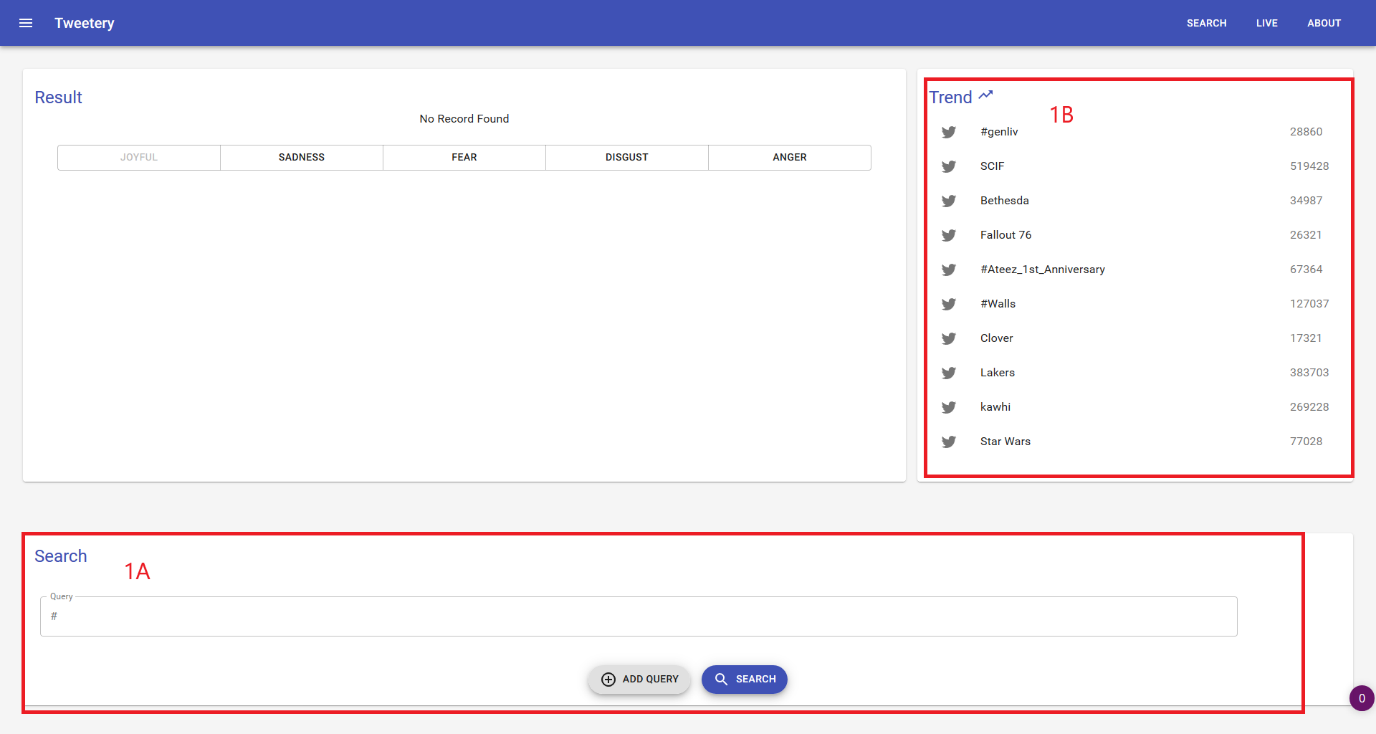
Create an Auto scaling group from the AWS EC2 Management console.

1. Select launch configuration.
2. Name the scaling group.
3. Start the group size with one instance.
4. Select a subnet (Must be in the same availability zone with the load balancer).
5. Click on advanced details
   1. Select ‘Receive traffic from one or more load balancers’ and add target group
   2. Health Check Grace Period of 60 seconds
   3. Enable CloudWatch detailed monitoring
6. Use scaling policies to adjust the capacity of the group
   1. Scale between 1 and 3 instances.
   2. Change metric type to Average Network In (bytes)
   3. Target value of 1,500,000
   4. Instances need 5 seconds to warm up after scaling
7. Review and Create auto scaling group.

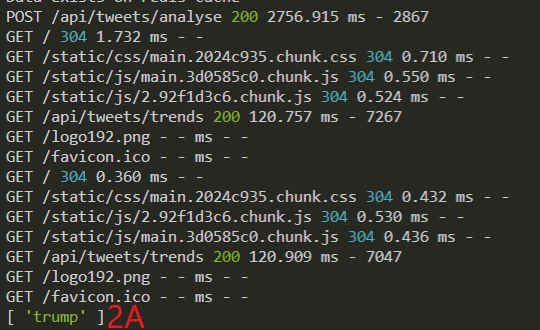
## Appendix C – Test Cases

1A – Display the search box

1B – Displays the trending topics



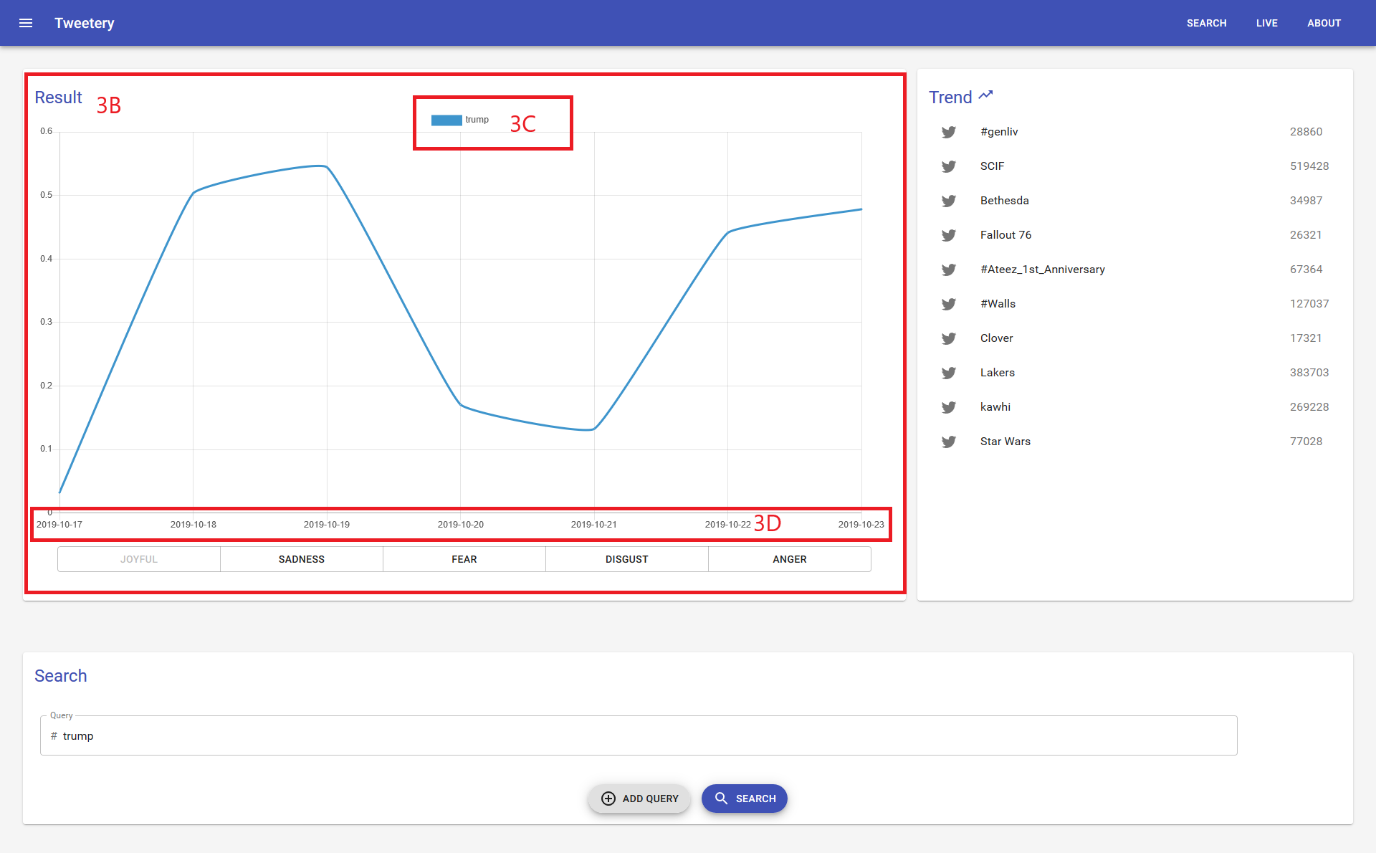
2A – Search button sends query



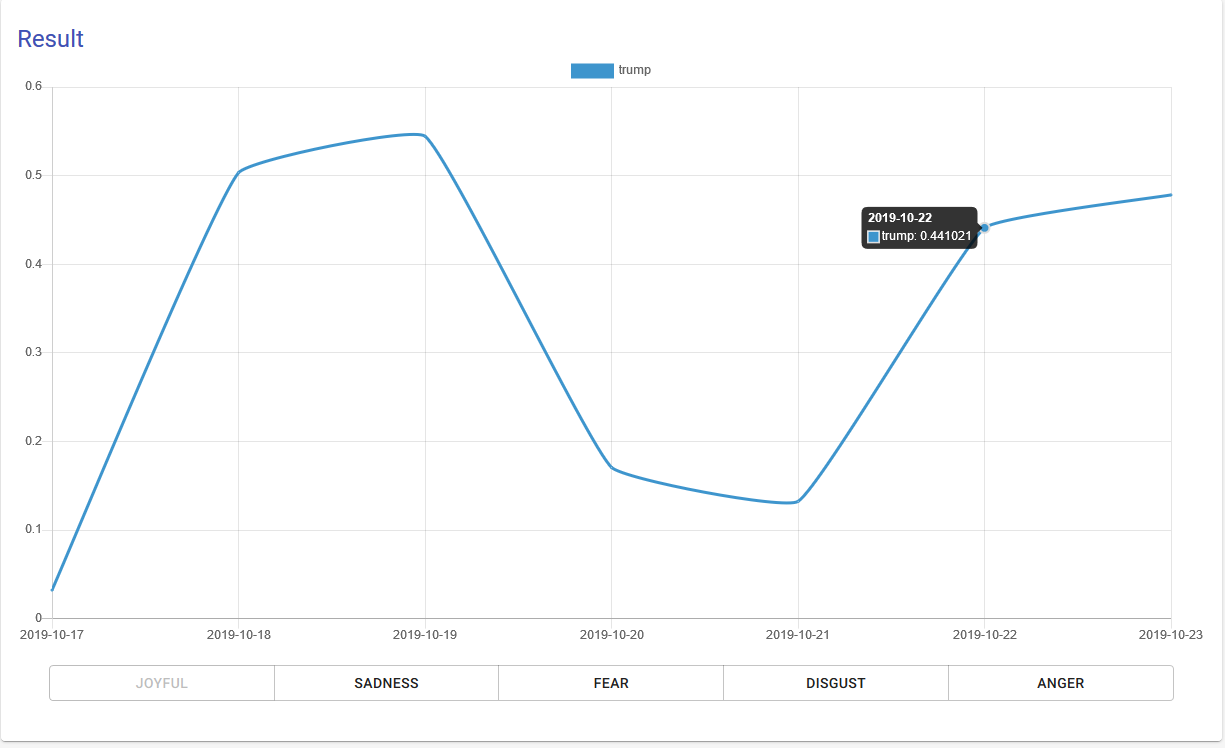
3B – Graph Displays

3C – Graph displays legend

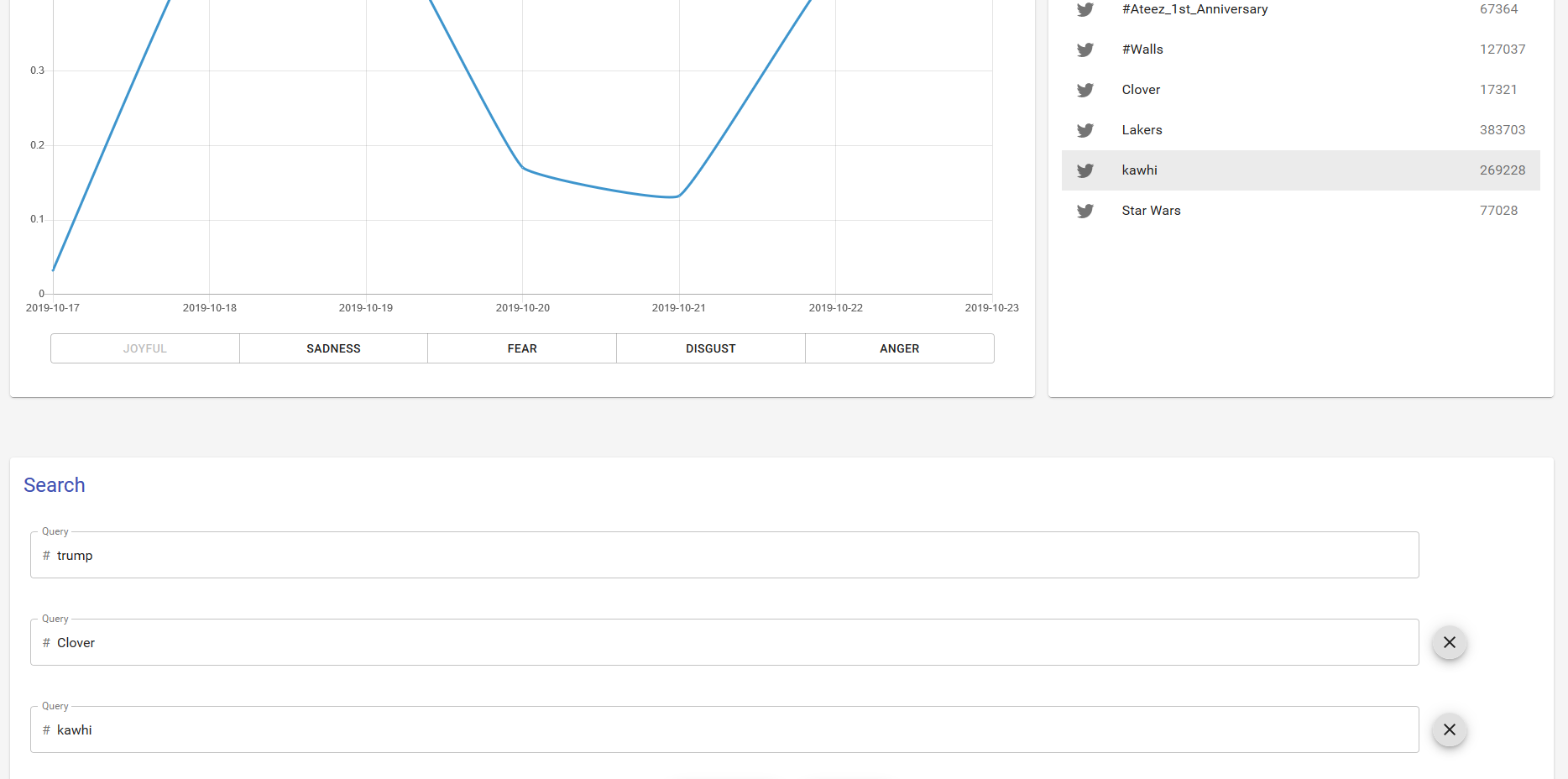
3D – Graph displays Date



4A – Graph displays labels

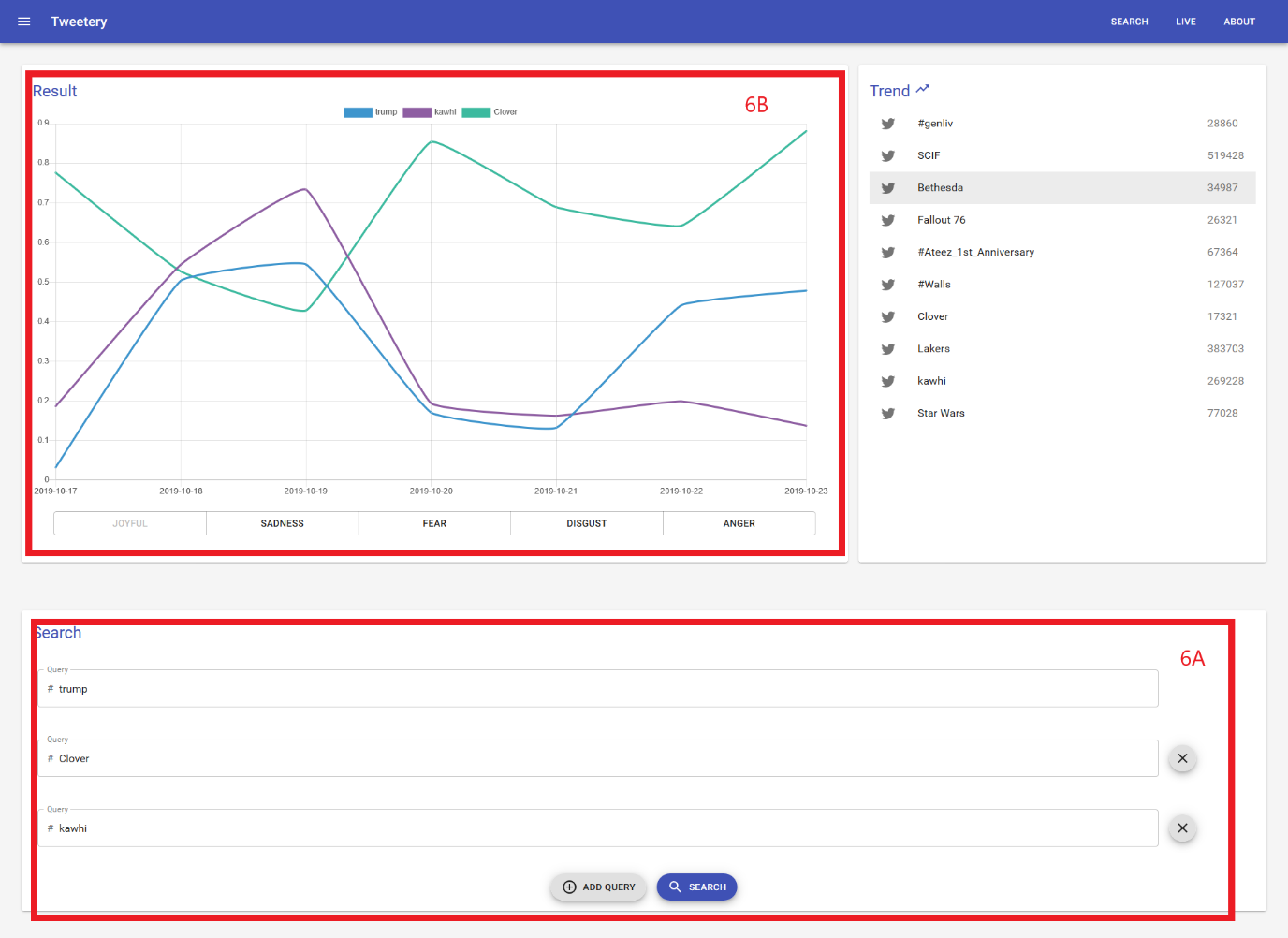


5A – Selected topics adds to search box



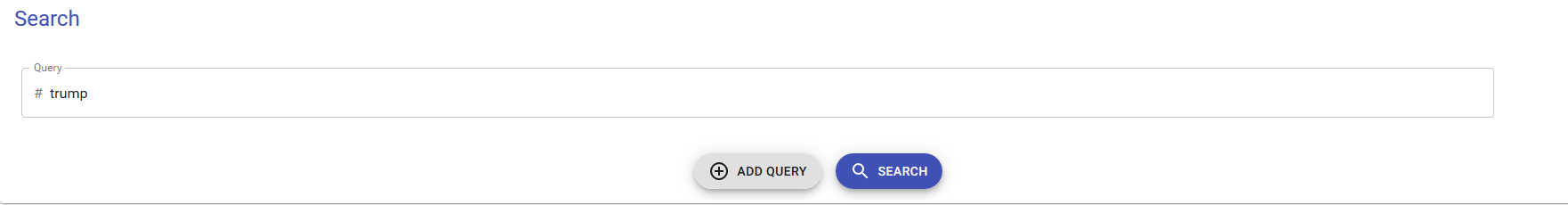
6A – Initiate search with more than one topic

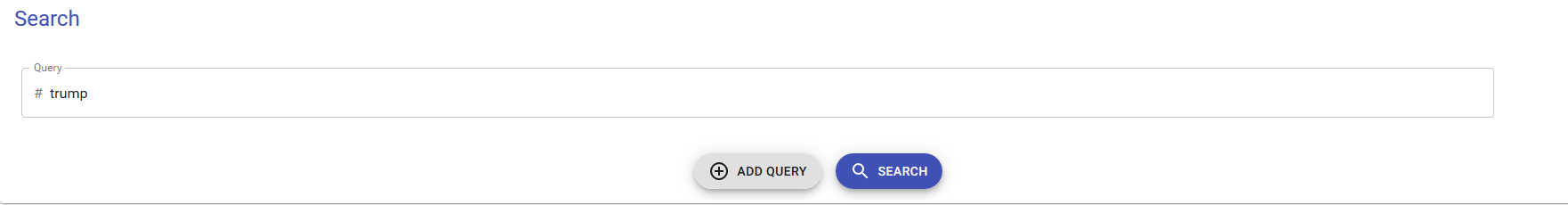
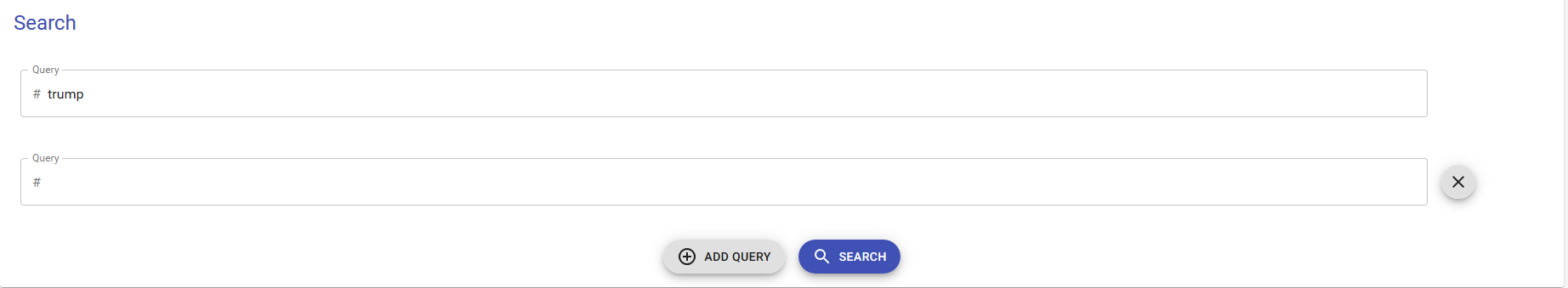
6B – Multiple queries displays on the graph

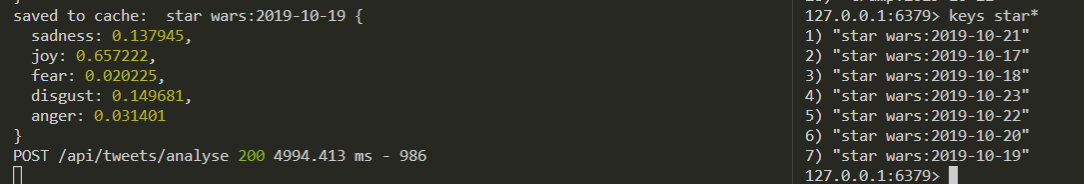


7A – Initiate search with more than one topic

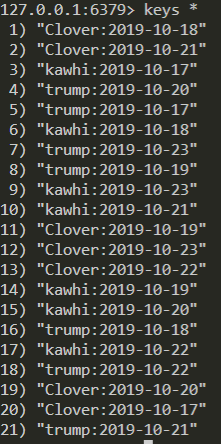
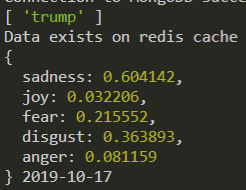


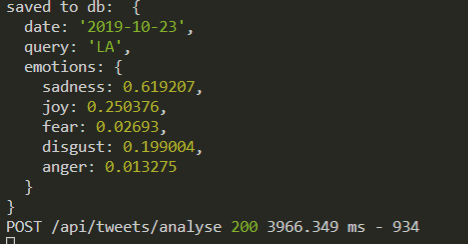
8A – Add for search query, Delete search row

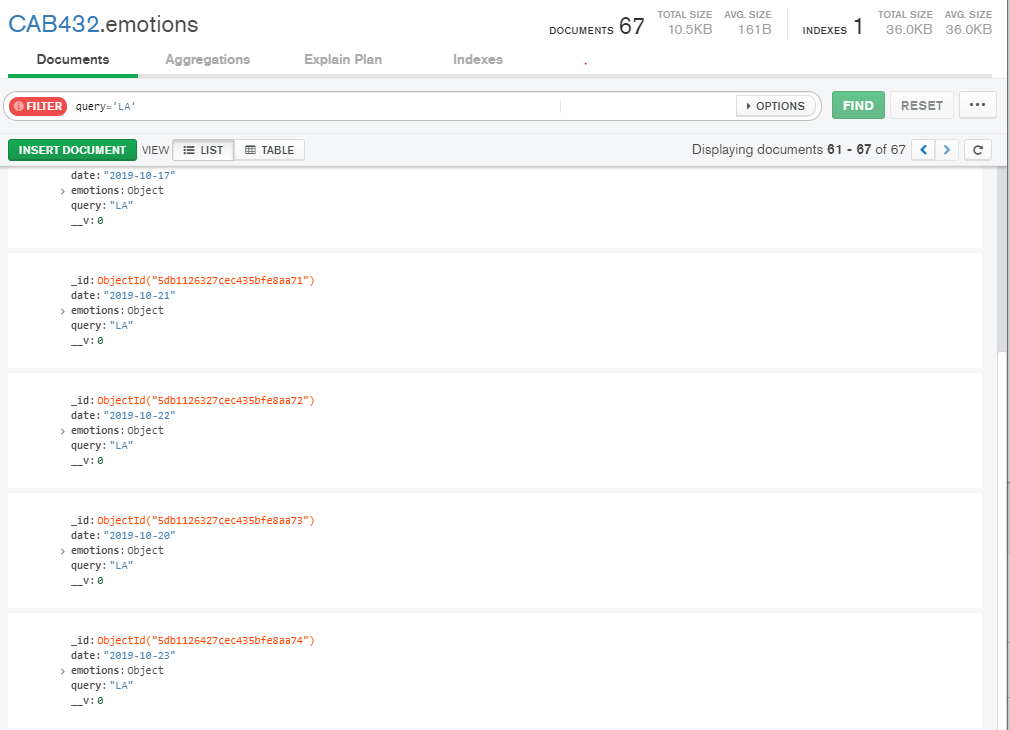


9A – Node server stores data in redis with specified key

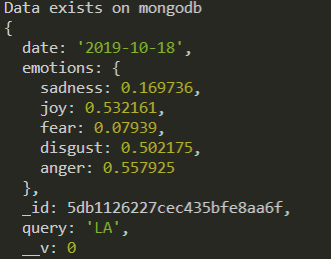
10A – Node server retrieves data from Redis by key specified. In this case (“trump:2019-10-17”)



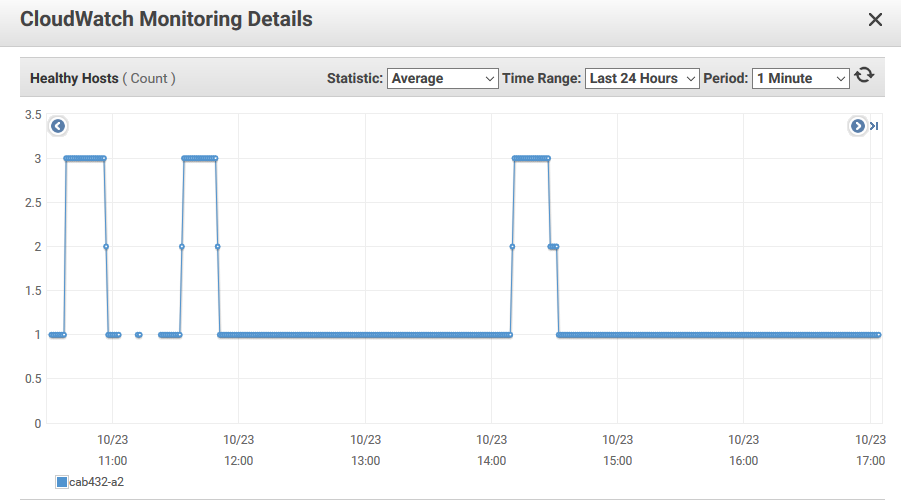
11A – Node server stores data in mongoDB 



12A – Node server retrieves data from mongoDB by key specified



13A – Load balancer health check



14A – Auto scale up, Auto scale down

