CAB432 Cloud Computing
Cloud Project - Twittersphere

Prepared for: Associate Professor Jim Hogan
Prepared by: Nikom Dupuskull n9013482
Thanat Chokwijitkul n9234900

Queensland University of Technology (QUT)
31 October 2015

INTRODUCTION

The name of the application is “Twittersphere”, which refers to the universe of Twitter and its habits. This application is a cloud-based query processor and sentiment analyser based on live tweets. Multiple hashtags (keywords emphasised by users) can be submitted to the application and will become a live filter against the inflow of the Twitter messages, monitored by the Twitter Streaming API, the public stream endpoint which streams the public data flowing through Twitter. Any tweet that passes through the filer will be extracted and analysed, then the result will be displayed on the screen, including the filtered messages.

This application can be used to detect emotions and evaluate opinions in the content that people tweet specified by a collection of hashtags. It can qualitatively investigate public’s thoughts and opinions on a particular subject using quantitative scale. In terms of real usage, streaming inflow Twitter messages filtered by a set of hashtags can be used to get opinions or feedback on a particular topic. For instance, live Twitter streaming at any event is one of the efficient way to collect feedback and questions. By referring back to the collected data and performing sentiment analysis, it can give some useful insides of what should be improved and what are particularly appreciated.

Nevertheless, another crucial aspect of this project is to demonstrate the elastic scalability of the application using the Azure cloud service. Hence, it is necessary that the system must be able to generate sufficient workload in order to meet this requirement. As a result, the main computational demand must be based on the number of concurrent queries (hashtags) being processed across the Twitter feed and sentiment analysis.

The application has been deployed via a Docker container, sitting on top of an Azure Linux VM and can be accessed via twittersphere.cloudapp.net.

TECHNICAL DESCRIPTION

APPLICATION ARCHITECTURE

This section delineates the architecture of the application, technologies used in both client and server sides, along with how the application has been developed.

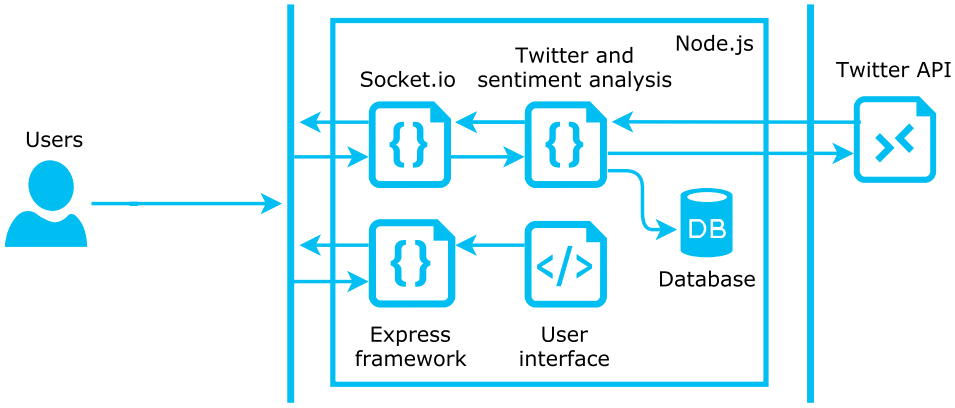

Figure 1: The application architecture diagram

The above diagram illustrates the architecture of the application. When a connection is established, the application will start listening for a specified query event on the socket. When a collection of hashtags is submitted, the application will create a new stream instance using a Twitter client and start listening for events on the stream. Each tweet that passes through the filter will trigger the socket to emit a new data object containing an incoming tweet to the client. At the same time, the client will be listening for the tweet event and update the user interface.

In addition to streaming and emitting tweets to the client, each tweet will also be analysed by the Sentiment module and stored in the local database. At this stage, each message should be successfully classified whether it is positive, neutral or negative by the sentiment score. The system also calculates the mean value of the overall sentiment scores and store the result in another database. The sentiment analysis result then will be emitted to the client and visualised using Epoch, a data visualisation library, to create a real-time graph and display it on the user interface.

Server-side Architecture

Node.js and the Express framework are the main technologies used to build the web server. The major advantage of using Node.js as a server-side language is that it allows developers to create JavaScript isomorphic applications, which are applications that run JavaScript in both client and server sides (Node.js, 2015). However, Node.js alone still lacks of adequate capability to create fully-functional web applications. This is where the Express framework becomes a crucial part in the development process. It is a backend web application development framework written in JavaScript, which contains the common functionalities for single-page, multi-page, hybrid mobile and web applications, and also can be utilised to build application programming interfaces (APIs) or Node.js modules (Wodehouse, 2015).

The following list is the essential modules used in the development process of the application.

Express

Express is the most essential module when developing web application with Node.js. As mentioned before, even though Node.js is a powerful server-side scripting language which contains some basic HTTP functions, whereas those functions cannot efficiently be used build a fully-functional web application. This Sinatra inspired web development framework becomes an effective tool to provides a robust set of features for web applications without obscuring Node features (Express, 2015).

Socket.IO

Socket.IO is an event-driven JavaScript library for real-time applications. It enables bi-directional communication between client and server. it contains two parts, including the server library that can be utilised using Node.js and the client library that runs on a browser (Socket.IO, 2015). In this application, Socket.IO acts as an event-based communication channel between server and client for sending and receiving data in real time.

Twit

Twit is an asynchronous Twitter client library which supports both the REST and Streaming APIs (Tezel, 2015). However, the application only uses the Streaming API due to some limitations of the REST API.

Sentiment

Sentiment uses the AFINN-111 word list as a classifier to perform sentiment analysis (Sliwinski, 2014). However, the module can be trained by overriding the existing words and values or adding a collection of new words. Therefore, the AFINN-165 word list and AFINN-emoticon-8 also have been used in this application to train its classifier to enhance its functionality. The following table is a set of sample results when using the Sentiment library for sentiment analysis.

Sentence	Score	Interpreter
I love Twitter.	3	Positive sentiment (love: 3)
I hate Twitter.	-3	Negative sentiment (hate: -3)
Twitter is very cool, I love it.	4	Positive sentiment (cool: 1, love: 3)
Twitter is irritating, I dislike it.	-5	Negative sentiment (irritating: -3, dislike: -2)
Twitter is an online microblogging service.	0	Neutral sentiment

Table 1: Sentiment analysis results

DiskDB

DiskDB is a lightweight disk based JSON database. Data is stored in a JSON format and can be interacted with a MongoDB-like API (Ravulavaru, 2014). According to the specification, it is not necessary to store the entire search history in a database, whereas persistent is one of the required aspects. As a result, this lightweight database has been utilised to store the data, including tweets and sentiment results, only when the

application remains active. DiskDB identifies different sessions using socket IDs. Event though DiskDB is persistent, but if a particular session is revoked by its end user, all data in that session will be deleted.

Morgan

Morgan is very useful in development since it logs all the request details (GET, POST, PUT and DELETE) onto the terminal console (Express, 2015).

Embedded JavaScript Template (EJS)

EJS has been used to render the main page instead of using the “sendFile” function, which needs the system to identify the root and current directory. However, the “sendFile” function is still used to handle any invalid route, by redirecting users back to the main page.

Client-side Architecture

The application is a single-page web application. HTML and CSS are the default markup and stylesheet languages in developing the user interface, and JavaScript is the main scripting language used on the client side. The design layout of the application is based on the Material Design standard web page layout.

The front-end of this application uses Bower as a package manager, which depends on the Node package manager (npm). The following list is the essential Bower components used in the development process of the application.

AngularJS

AngularJS is a JavaScript framework for single-page applications. One of the benefits of using AngularJS is that it can automatically synchronise data from the user interface with JavaScript objects through 2-way data binding (AngularJS, 2015). It has been heavily used for server-side communication in this application. By using AngularJS with Socket.IO, it becomes a combination that makes real-time communication between client and server plausible.

Epoch

Epoch is a general purpose real-time data visualisation library based on Data-Driven Documents (D3) (Fastly, 2015). Generally, Epoch by itself can be utilised without AngularJS. Therefore, in order to use Epoch with AngularJS, a custom directive is required. This application uses an AngularJS directive wrapper for Epoch called “ng-epoch” for real-time data visualisation.

Mocha

Mocha is a simple, extensible and fast JavaScript test framework running on both server and client. It can be used for unit and integration testing (Mocha, 2015).

Supertest

Supertest is a uni test suite developed using SuperAgent, a small progressive client-side library. It provides services for testing HTTP requests (Vision Media, 2015).

Should

Should is very expressive framework-agnostic unit testing module. It keeps all the tests organised and makes error messages more readable and helpful for developers (ShouldJS, 2015).

DEPLOYMENT ARCHITECTURE

The application has been implemented on top of the Microsoft Azure Virtual Machine (IaaS Cloud). In case of deployment, it has been deployed via a Docker container on top on Ubuntu Server VM on Azure.

Microsoft Azure

The application has been implemented on top of the Azure Linux Virtual Machine (Ubuntu Server 15.04), a scalable compute infrastructure in the cloud. The following diagram demonstrates a simple architecture when deploying an application via the Azure Linux VM.

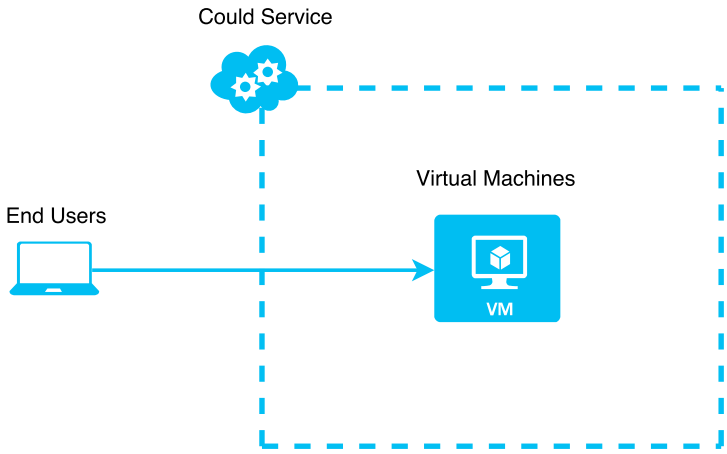

Figure 2: The deployment architecture diagram

At this stage, the application is running inside the Docker container on top of the Azure Ubuntu Server VM. Users can access the web application by using the cloud service's DNS name. As illustrated in the diagram, a load balancer is not required in this case and scaling is also not possible since there is only one VM running in the cloud service.

If an application is running instances of Web Roles or Worker Roles, it would be much less time consuming to accommodate more workload by adding or removing instances. In case of Virtual Machines, however, the methodology must be different due to its infrastructure. When scaling an an application running Virtual Machines, the provisioned machines in the availability set will be turned on or turned off instead of creating a new machine. The following diagram demonstrates how the application running Virtual Machine can be scaled in the IaaS Cloud.

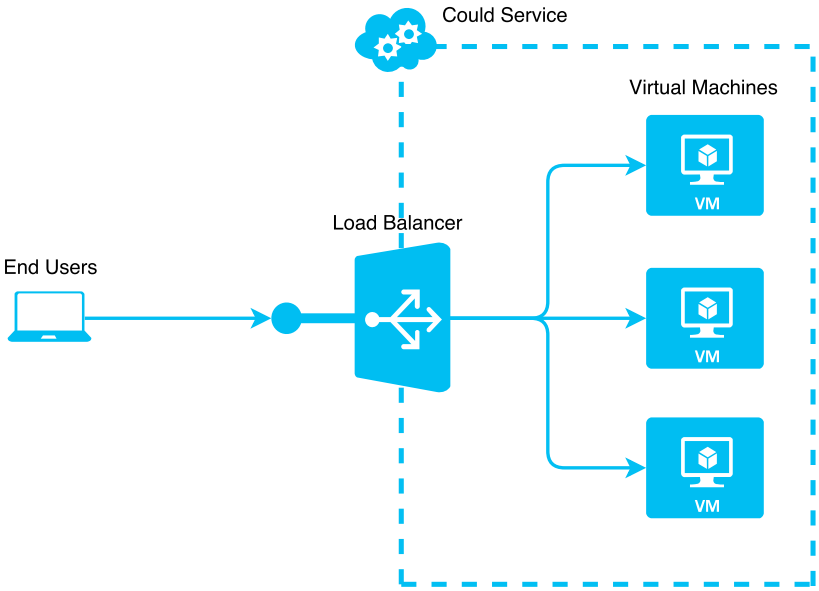

Figure 3: The deployment architecture diagram (autoscaling)

The diagram represents a load-balanced endpoint that is shared among three machines which are in a load-balanced set. A load balancer is required in this case in order to evenly distributed incoming traffic across multiple machines when the scale-up action occurs.

As illustrated in the diagram, three Virtual Machines that serve the same purpose have been added to the availability set. These machines will be turned on in the scale-up action and turned off in the scale-down action, but one of the machines should be initially turned on, others can be initially turned on or turned off. The scaling protocol can be specified based on the average CPU usage or the number of messages in queue. For this application, the scaling protocol is specified based on the average CPU usage. For instance, if the average CPU usage is greater than 70%, another machine in the same availability set will be automatically turned on, and when the percentage is below 50%, one of the machine will be turned off.

Autoscaling Configuration

The average percentage CPU usage is the main parameter used to configure autoscaling for this application. If it increases above or below a specified thresholds, Virtual Machines will be turned on or turned off from the availability set.

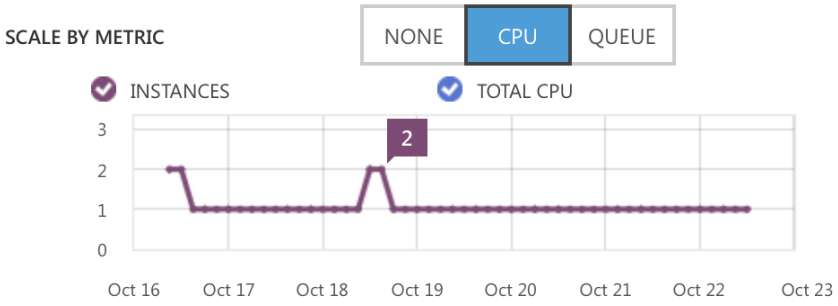

Figure 4: Scale by metric (average CPU usage)

Autoscaling can be configured within the Scale section in a cloud service using the Azure Management Portal. By setting the metric to CPU, the application will be automatically scaled based on the average percentage of the CPU resources that have been used.

Figure 5: Target CPU usage

This slide bar can be used to specified the number of VMs that can be used. The bar on the left specifies the minimum number of VMs and the bar of the right specifies the maximum number of VMs that can be used in the scale-up action.

Figure 6: Target CPU usage

This figure specifies the range of average percentage of CPU usage. if the average CPU usage is higher than the maximum rate, another machine in the same availability set will be automatically turned on. On the other hand, if the percentage is lower than the minimum rate, one of the machines will be turned off.

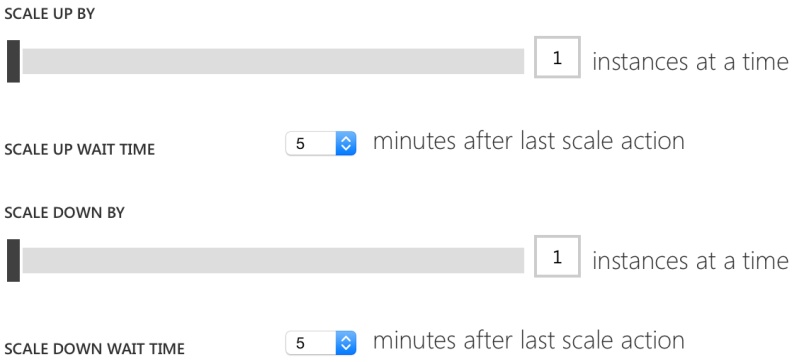

Figure 7: Number of instances to be turned on or off when scaling

These slide bars are used to specify the number of VMs to be turned on each time the application is scaled up and also the number of VMs to be turned off when the application is scaled down. The scale up and scale down wait time can also be configured with the dropdown list below the sliders.

After completing the configuration process, the application should be automatically scaled up or scaled down according to the parameters that have been set in the cloud service.

Docker

Docker has been utilised for the deployment of this project. In order to deploy the application to the IaaS Cloud, a Docker image needs to be built using a Dockerfile, a simple text file containing a sequence of commands to assemble an image. The Dockerfile used in building a software image for this application consists of the following commands:

The latest version of Ubuntu Server on Azure is 15.04. Therefore, the command below tells Docker which operating system will be used to build an image.

```
FROM ubuntu:15.04
```

Identifying the author of Dockerfile.

```
MAINTAINER Thanat Chokwijitkul & Nikom Dupuskull
```

Updating all the packages in the base operating system and install some of the basic applications, including Node.js and Node Package Manager (npm).

```
RUN apt-get update
RUN apt-get install -y nodejs npm
```

Managing an internal path by making a symbolic link between files.

```
RUN ln -s /usr/bin/nodejs /usr/bin/node
```

In addition to npm, this application also needs the Grunt Command Line Interpreter (Grunt CLI) and the Bower package manager for all the client-side components.

```
RUN npm install -g grunt-cli bower
```

Copying the folder containing the source files into the local image.

```
COPY ./Twittersphere /src
```

Installing all the dependencies, including Node modules and Bower components.

```
RUN cd /src; npm install
RUN cd /src/public/libs; bower install --allow-root
```

Telling the Docker container to listen to a specified port (8080) during run time.

```
EXPOSE 8080
```

Setting the default command to execute when creating a new Docker container.

```
CMD ["nodejs", "/src/server.js"]
```

The Dockerfile needs to be located in the root directory on top of the folder containing all the source files. In order to build a software image from the Dockerfile, the following command must be executed:

```
$ sudo docker build -t alexenriquent/twittersphere .
```

Where “alexenriquent” is the DockerHub username and “twittersphere” is the application name. After building the image, the application can be run using the following command:

```
$ sudo docker run --restart=always -p 80:8080 -d alexenriquent/twittersphere
```

As mentioned before, “alexenriquent” is the DockerHub username and “twittersphere” is the application name. The -p option will map the external port to the internal port and the -d option will deamonise the application and make the container run in the background. Another additional command is the --restart=always command, which will automatically restart the container after restarting the VM.

SCALING AND PERFORMANCE

As the amount of workload increases, an application may require additional resources to enable it to perform its operation or handle more traffic. Autoscaling becomes a crucial part in dynamically allocating resources required by the application to deal with the increasing workload along with minimising run time costs. For this application, the autoscaling policies have been configured using the Azure Management Portal. Due to the fact that the application is implemented on top of the Azure Linux VM, it is not possible to create new instances or delete existing instances like Web Roles or Worker Roles. Instead, autoscale will turn on or turn off VMs in the availability set either due to the scale-up or scale-down action.

The scaling strategy used for this application is metrics-based autoscaling, which reacts to the average percentage of CPU utilisation over the past hour. For instance, if the average CPU usage is higher than the maximum CPU rate, one or more VMs will be automatically turned on. On the other hand, if the average CPU usage is lower than the minimum CPU rate, one or more VMs will be automatically turned off.

However, because the metric used in the autoscaling service is average CPU utilisation across all of the VMs over the last hour, if the amount of workload suddenly grows over the maximum rate in a short period of time, autoscale will not be triggered. The system will take approximately 60-minute running average to scale (Microsoft, 2015). Even the scale up and scale down wait time can be configured to 5 minutes, but that cannot confirm that a scale action will be triggered immediately after 5 minutes waiting time.

The following section is an example of how the application can be autoscaled on Microsoft Azure (with 2 VMs).

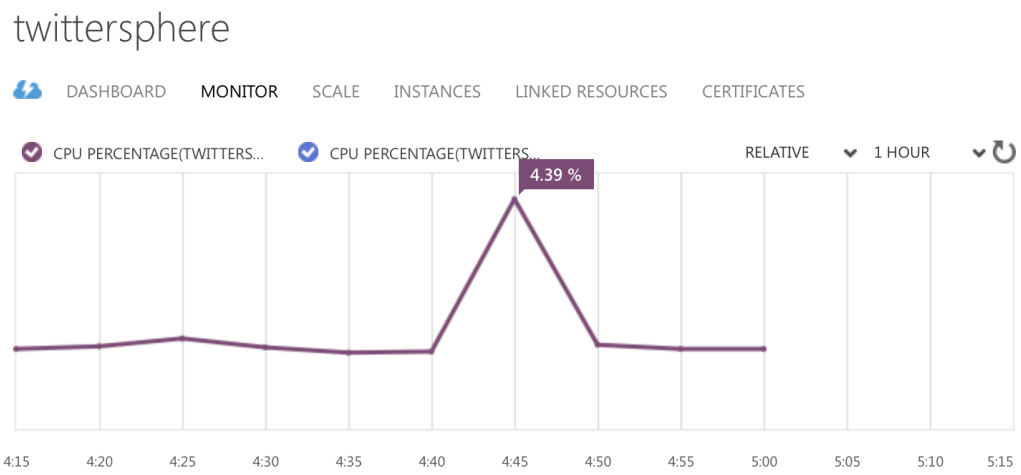

Figure 8: Real-time CPU usage

The above figure shows the real-time CPU usage of two VMs, namely “Twittersphere01” and “Twittersphere02”. The purple line and blue line represent the CPU usage of Twittersphere01 and Twittersphere02 respectively. However, only the CPU usage of Twittersphere01 was displayed since Twittersphere02 had been initially turned off.

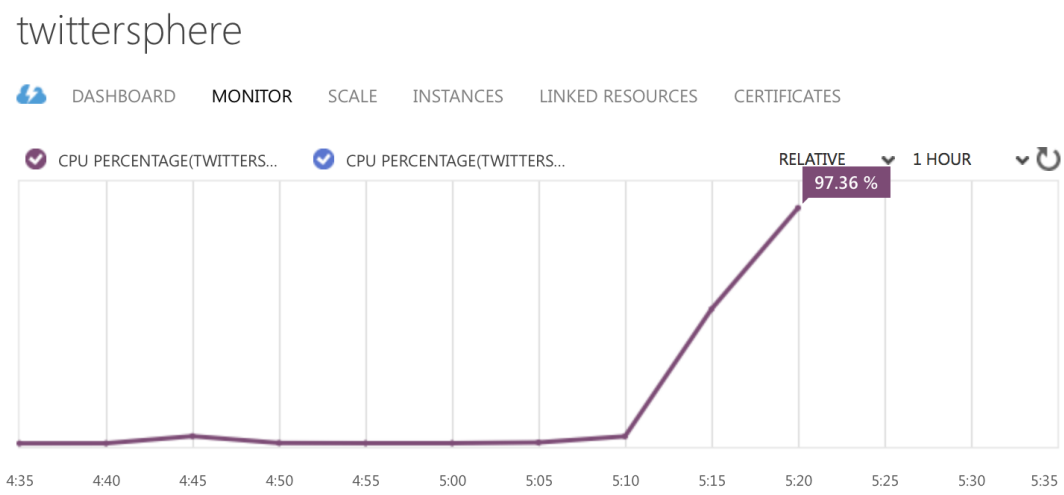

Figure 9: Real-time CPU usage

The CPU usage of Twittersphere01 kept increasing and eventually went higher than the maximum CPU rate, which was 70% at that time. At this stage, the autoscale status of the Twittersphere cloud service was “Autoscale reduced your costs by up to 50%”, which indicates that Twittersphere02 still remained deallocated.

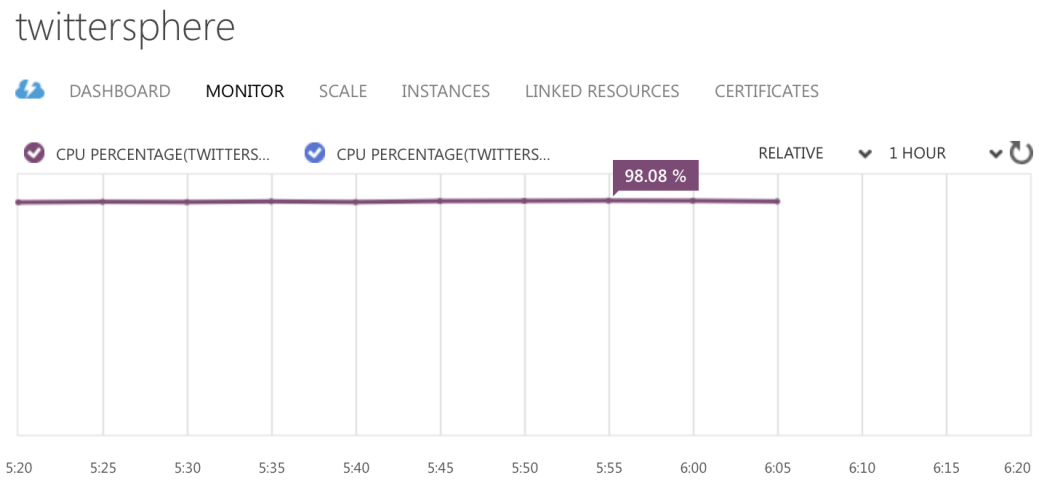

Figure 10: Real-time CPU usage

The CPU usage of Twittersphere01 remained almost stable over the time period. The average CPU usage kept increasing from 74.88% to 93.57%. The autoscale status remained the unchanged.

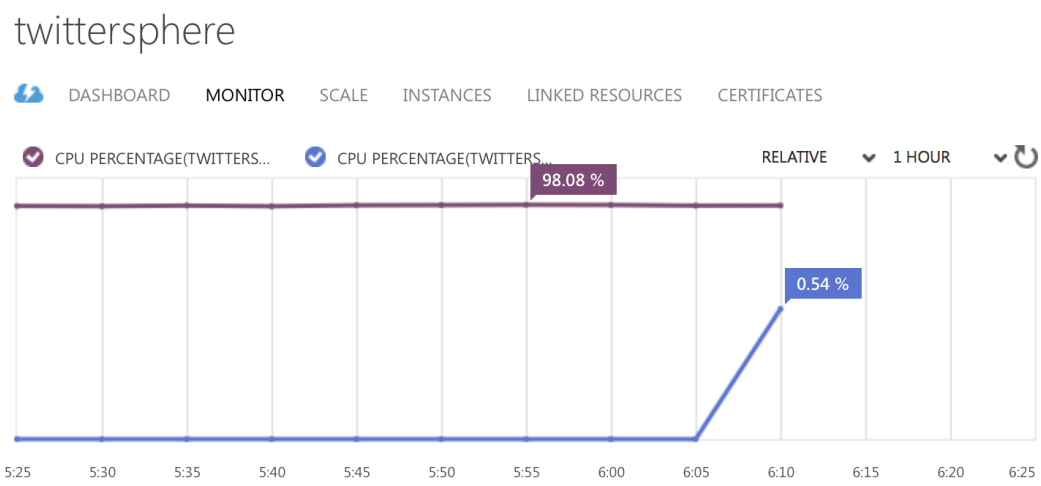

Figure 11: Real-time CPU usage

At the percentage of average CPU usage of 97.73%, Twittersphere02 was automatically turned on, which means the application was successfully scaled up. The total time taken before scaling up was approximately 45 minutes. The autoscale status became “Autoscale is configured and running”.

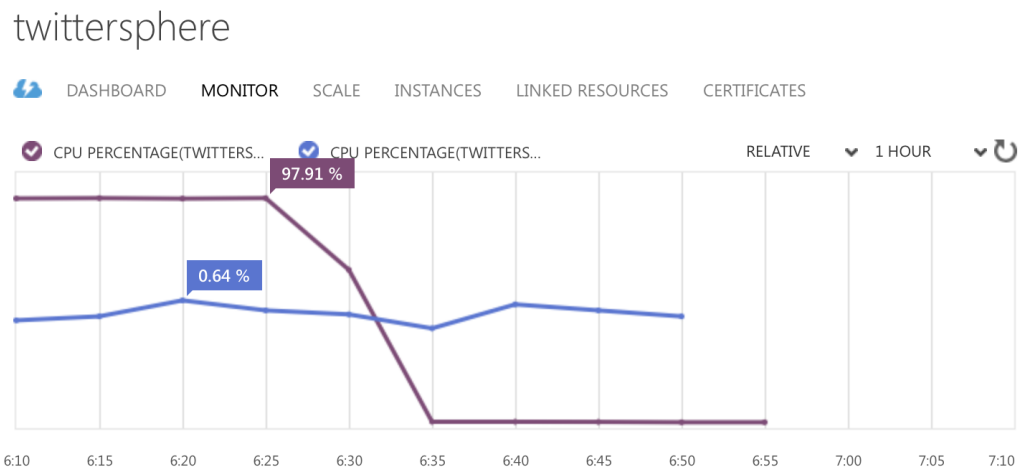

Figure 12: Real-time CPU usage

The application continued running for another 10 minutes before being terminated (removing the search including its history and statistics). According to the graph, the CPU usage of Twittersphere01 rapidly decreased from 97.91% to 2.76% in 10 minutes, then remained stable for another 25 minutes until the average CPU usage became 37.69%, which was lower than the minimum CPU rate (50%). At this stage, Twittersphere02 was automatically turned off and the autoscale status became “Autoscale reduced your costs by up to 50%” again. This means the application was successfully scaled down and also concludes that autoscaling for an application running Virtual Machines functions correctly based on the average percentage of CPU utilisation over the last hour.

TESTING AND LIMITATIONS

This section delineates the testing framework used in the application and limitations of the development process and functionality.

Functionality Testing

Functionality testing is a quality assurance process to ensure that the application functions as expected based on its functionalities and use cases.

ID	Purpose	Expected/Actual	Pass
1	Ensure that the search functionality works correctly when input a single hashtag.	E: The hashtag should become a filter for incoming tweets and the sentiment analysis results should display on the screen. A: The hashtag becomes a live filter for incoming tweets and the sentiment analysis results display on the screen. However, the number of tweets depends on the popularity of the hashtag.	✓
2	Ensure that the search functionality works correctly when input multiple hashtags separated by commas.	E: The hashtags should become a filter for incoming tweets and the sentiment analysis results should display on the screen. A: This functions as expected and the number of tweets is higher. However, the sentiment result may vary due to the sentiment score of each keyword.	✓
3	Ensure that the content panel functions as expected.	E: While streaming, the content panel should display incoming tweets and keep shifting old tweets in case that the number of tweets is higher than 10. A: The content panel functions as expected. However, as the number of hashtags increases, updating and shifting are also accelerated.	✓
4	Ensure that the Sentiment Analysis graph functions as expected.	E: While streaming, the number of tweets and the average sentiment score should be continuously updated in real time and the graph should be updated according to the average sentiment score of the set of hashtags. A: The number of tweets and the average sentiment scores are updated in real time and the live line graph is continuously updated in almost real time. This is acceptable due to the animation delay of the Epoch library.	✓

ID	Purpose	Expected/Actual	Pass
5	Ensure that the Aspect Analysis graph functions as expected.	E: While streaming, the number of positive, neutral and negative tweets should be continuously updated in real time and the graph should be updated according to these data. A: The number of positive, neutral and negative tweets are continuously updated in real time. Addition of these three number will result in the total number of filtered tweets in real time. The live line graph is continuously updated in almost real time. This is acceptable due to the animation delay of the Epoch library.	✓
6	Ensure that the CPU usage gauge functions as expected.	E: Before streaming, the initial percentage CPU usage should be very low. While streaming, it should increases as the number of tweets goes up. A: The gauge is continuously updated according to the real-time percentage of CPU usage. The percentage is very low at first, then increases as as the number of tweets goes up.	✓

Unit Testing

The application utilises 3 Node modules for unit testing, including Mocha, Should and Supertest. Even though unit testing is not a huge part of the project, but it is necessary to ensure that all the routes function as expected. Since it is a single-page web application, the main route should direct to the main page, and other invalid route may also redirect to the main page. The unit testing within the application can be run using the “npm test” command, the results should be similar to the following:

```
Main route
    GET /
GET / 200 8.265 ms - 4473
      ✓ should return 200 (43ms)
GET / 200 2.324 ms - 4473
      ✓ should display content as "text/html"
    GET /test
GET /test 200 3.850 ms - 4473
      ✓ should return 200
GET /test 200 1.535 ms - 4473
      ✓ should display content as “text/html"

  4 passing (79ms)
```

Limitations

The application has utilised the Twitter Streaming API to gain access to Twitter's global stream of tweet data. By using the public stream endpoint, the system can retrieve the streams of public data flowing through Twitter filtered by a collection of hashtags. Even though polling and REST API rate limit are not worrisome issues in this case, the Twitter Streaming API still has its own limitations.

This application does not require users to login and use each individual's credential as a token for data streaming. Therefore, using only one set of credentials over and over again can reach the connection rate limit. According to the documentation related to the connecting to a streaming endpoint issue, Twitter will automatically close a streaming connection if a Twitter client establishes too many connections that exceed the streaming rate limit with the same credentials. In this case, the oldest connection connected to the stream endpoint will be terminated (Twitter, 2015). When the client is disconnected, it will attempt to reconnect to the stream endpoint immediately, if the reconnect fails, it will be slowed down. Even though the exact number of connections or attempt to reconnect that causes the rating limiting to occur has never been specified, but it is crucial not to increase the number of connections if a HTTP 420 response is received. This possibly causes the IP address to be blocked for a period of time (Ibid.).

POSSIBLE EXTENSIONS

Possible extensions here.

REFERENCES

AngularJS. (2015). AngularJS: HTML enhanced for web apps. Retrieved October 24, 2015, from AngularJS: https://angularjs.org

Express. (2015). Express: Fast, unopinionated, minimalist web framework for Node.js. Retrieved October 24, 2015, from Express: http://expressjs.com

Express. (2015). Morgan: HTTP request logger middleware for node.js. Retrieved October 24, 2015, from npm: https://www.npmjs.com/package/morgan

Fastly. (2015). EpochL Real-time. Retrieved October 24, 2015, from Epoch: https://fastly.github.io/epoch/real-time/

Kitterman, C. (2013). Autoscaling Windows Azure Applications. Retrieved October 24, 2015, from Microsoft Azure: https://azure.microsoft.com/en-us/blog/autoscaling-windows-azure-applications/

Microsoft. (2015). GitHub. Retrieved October 24, 2015, from Autoscaling guidance: https://github.com/mspnp/azure-guidance/blob/master/Auto-scaling.md

Mocha. (2015). Mocha: Simple, flexible, fun. Retrieved October 24, 2015, from Mocha: https://mochajs.org

Node.js. (2015). About Node.js®. Retrieved October 24, 2015, from Node.js: https://nodejs.org/en/about/

Ravulavaru, A. (2014). DiskDB: A Light Weight Disk based JSON Database with a MongoDB like API. Retrieved October 24, 2015, from npm: https://www.npmjs.com/package/diskdb

ShouldJS. (2015). Should: Test framework agnostic BDD-style assertions. Retrieved October 24, 2015, from npm: https://www.npmjs.com/package/should

Sliwinski, A. (2014). Sentiment: AFINN-based sentiment analysis for Node.js. Retrieved October 24, 2015, from npm: https://www.npmjs.com/package/sentiment

Socket.IO. (2015). Socket.IO: Docs. Retrieved October 24, 2015, from Socket.IO: http://socket.io/docs/

Tezel, T. (2015). Twit: Twitter API client for node (REST & Streaming). Retrieved October 24, 2015, from npm: https://www.npmjs.com/package/twit

Twitter. (2015). The Streaming APIs. Retrieved October 24, 2015, from Twitter Developers: https://dev.twitter.com/streaming/overview

Vision Media. (2015). SuperTest: Super-agent driven library for testing HTTP servers. Retrieved October 24, 2015, from npm: https://www.npmjs.com/package/supertest

Wodehouse, C. (2015). Express.js: A Server-Side JavaScript Framework. Retrieved October 24, 2015, from Upwork: https://www.upwork.com/hiring/development/express-js-a-server-side-javascript-framework/