Cardiff University

Final year project

Improving the Responsiveness of Autoscaling Systems

Gregory Nichols - C1304478

supervised by
Omer Rana

moderated by
Hantao Liu

May 5, 2017

Acknowledgements

I would like to thank Blurrt for kindly letting me do this project with their blessing and use their hosting resources.

I would also like to thank my supervisor Omer Rana for providing me helpful feedback and advice throughout my project.

1 Introduction

1.1 What is Blurrt

Blurrt is a social media analytics company which specialises in the sentiment and emotional analysis of social media posts, with a specific focus on Twitter. It is capable of scoring posts for their positive or negative sentiment, their emotion based on one of 8 possible emotions and the gender of the person posting as well as other characteristics such as "Blurrt Score" a combination of both volume and sentiment. Collected posts are then displayed on the web front-end in a dashboard with charts, maps and other graphics making the data interactive and easily accessible.

This project has been done in affiliation with Blurrt, who have the existing system which I aim to improve on.

1.2 Utility of the Project

In the Blurrt back-end there is a simple monitoring system which monitors the size of a queue of objects waiting to be parsed and as its size reaches above a threshold a request is triggered to the hosting company to create a new virtual machine running several parsing processes. Once the queue has reduced in size the system recognises that there is less demand and downscales the number of virtual machines accordingly, in order to not waste money unnecessarily. This system, while functional, was very slow to scale up and down, it was also very difficult to upgrade and patch and it was rather costly to run as the server hosting company (Rackspace) is not very cheap.

This is important because Blurrt relies heavily on being a real time social media analytics company and any delay to data being visible to clients is directly impacting one of the core selling points of the product. Currently if a large collection is created on a very popular topic, such as X Factor, the queues can become backlogged with several thousand messages and it can take a considerable amount of time to scale up to the necessary size. This delay can get so considerable at times that one of the developers has to manually create new parsing servers because the scaling system is not fast enough to keep up with the increased demand placed upon it.

It is also important because the lack of automation and difficulty of patching has led to patches not being implemented for the parsers as regularly as they should be and in some cases forgotten about by accident, leading to data being incorrectly parsed as their codebase is not up to date with the rest of the system.

1.3 Aims of this Project and Report

This project aims to research and implement an improved system of automatic scaling for a continuous parsing process which has a varying level of demand placed upon it. I shall to create a system which scales faster, improves the ease of implementing updates, improves the monitoring system and if possible reduces the cost of running the servers.

This will therefore aim to improve the system in a number of ways. Firstly, to alleviate the time taken to create the parsing processes, making the system more responsive to increased demand and keep our delay for real-time analytics as low as possible. Secondly, to increase the granularity of the scaling by reducing the amount of processing power added by scaling the smallest amount, this will help to prevent 'sawtoothing' and reduce wasted resources. Thirdly, aim reduce the costs associated with running the parsers by reducing the amount of resources required over time and through cheaper hosting solutions. Fourthly, to implement a system which is easier to deploy updated code onto.

In order to demonstrate the achievement of the stated aims this project will:

- Identify potential hosting and software solutions.
- Analyse the options and pick what I believe to be the best one to form the basis for my implementation.
- Migrate the existing parsers to the chosen system, implementing any technologies required to make the parsers work on the new system.
- Develop an improved monitoring system that uses a predictive algorithm and can be more easily updated and extended in the future.

- Then analyse the difference in performance in terms of parser creation and destruction compared to the old system, the cost of running the new system compared to the old system and the time taken to perform a certain amount of work on the new system compared to the old system.

Through my report I shall clearly explain the current architecture of the system and how it functions. I will analyse potential technologies, software solutions and hosting solutions to solve these problems I've identified. I shall then explain why I chose what I believe to be the best of these potential solutions; a Docker for AWS based architecture. I shall then implement a comprehensive scaling management system which is capable of polling and triggering scaling events based on queue size. I will perform a performance analysis of the original system demonstrating the performance in terms of; speed of scaling up and scaling down, speed to process a set number of messages from one queue to the other and the costs for running the system. I shall then compare and contrast that to the performance provided by the new system I have implemented.

1.4 Important Outcomes

I implemented a cohesive and comprehensive scaling and monitoring system for Docker for AWS in node.js which is capable of monitoring the size of queues, analyse the monitored queue size through a rules based scaling approach and trigger scaling events. The triggered scaling events then gracefully scale both the number of instances and the containers on those instances to account for the increased computing power available to the Docker swarm, while also avoiding any unexpected exiting of containers due to lack of resources. This can be caused either by failure to wait to scale up until after the addition of new instances or for not scaling the number of containers down before the destruction of instances.

I was also able to re-architecture the existing virtual machine based parsing system into a containerised solution using docker for AWS. I was also able implement additions to the parser codebase to allow for the parsers to make use of a new queuing system Amazon's Simple Queue Service instead of the previous system RabbitMQ. From this I was able to demonstrate a reduction in time taken to scale by 59% as well as reducing the costs, barring the very worst case scenarios, while also leaving the potential for at least 39% reductions through changing the instance type used. I was also able to increase the speed of parsing by 58% from a combination of switching to AWS from Rackspace and upgrading to a newer version of PHP. This was due to the container based architecture allowing for the software of different aspects of the system to be at separate versions.

2 Background

2.1 System Architecture Introduction

The focus of this project shall be the back-end of the system, with a specific focus on the parsers as that is currently the part of the system which has autoscaling in place and it is also the part which lends itself most to the benefits that come from autoscaling. The back-end of the Blurrt system consists of a data collection system with 3 main components. Firstly, a collector which takes a stream of objects from the chosen social media platform and places it onto a parsing queue, next there are a number of parsers which take an object off the parsing queue, normalises it, scores it and then places it onto the storage queue, then a batching process takes groups of objects and places them into the database, which is an Elastic cluster, in one go. There is a monitoring process which monitors the parsing queue every minute and if it reaches above or below certain thresholds it creates a new parsing VM running several parsing processes.

This scaling system is being used in order to prevent large build ups of messages on the queue which were occurring before when the automated scaling system was not in place, this was due to the increased demand we were getting from certain clients. In order to account for the work of a large client which wished to collect on an TV show, for example X Factor which has high demand irregularly, we needed a system in place which would scale up when the queue got too large without the developers having to manually create and destroy servers to account for the demand.

2.2 Technologies Overview

2.2.1 Collectors and GNIP

Blurrt collects data from many different social media platforms, all of them have their own collector which is able to access the data API from that specific platform, process the data and place it onto a parsing queue. There are collectors and parsing queues for twitch, youtube, facebook, instagram and multiple twitter APIs.

In this project I will focus on GNIP as the data source, this is the private data API from Twitter which is Blurrt's main data source, you send off specific rules in the form of keywords with booleans and it provides you all tweets which match those rules in real time. I decided to focus on GNIP in this project as this is where the vast majority of Blurrt's data comes from and so has the greatest need to scale with incoming demand.

2.2.2 Json

Javascript object notation or json is a common data format similar to CSV. It is human readable text consisting of attribute-value pairs to form a data object.

2.2.3 Parsers

The parsers are the main workhorse of the back-end of the Blurrt system, they are responsible for taking messages off the parsing Queue, normalising the format of the message so messages from all platforms are in the same format and scoring them for 8 different emotions (happiness, sadness, anger, disgust, fear, love, thankfulness and confusion). It also scores whether the tweet is overall positive or negative as well as whether the creator of the tweet was male or female and several other metrics. The detailed methodology of how this is done is a company secret and is not necessary for the purposes of this project, what I can say is that the tweet object is given as a parameter to the natural language processing system. This NLP system is a collection of algorithms which score the tweet based on analysis of the text content and append to the tweet json our emotion and sentiment scores as well as Blurrt's other enrichments such as gender. After it has finished parsing a message it marks it as completed so that no other parser can take it and then the parsed json objects are placed onto the storage queue.

There are different parsers for different collection sources however the GNIP parsing process is written in PHP and runs on PHP 5 on the servers, this is because it needs to be able to run on the same server as the front-end which uses a framework only compatible with PHP 5. Each parsing process runs continuously as they are essentially a never ending while loop.

2.2.4 RabbitMQ

RabbitMQ is an open source queuing system which handles messages being stored and taken off either individually or in batches. These messages can be of any data type as the queue itself is completely opaque to the data type being placed on it, it uses the AMQP protocol which states that messages will not be permanently removed from the queue unless it is marked as processed when it has been finished with. RabbitMQ itself can run in either a clustered mode which consists of multiple servers with load balancing (which automatically distributes demand) to handle high message throughput or a standalone mode, which is how it is run at Blurrt. The Blurrt system has many different queues all running on this standalone server, each collection source has its own parsing queue assigned to it where messages which come from the collector are placed. There are also multiple storage queues where the parsers place messages onto to be stored in the database by the batchers, these storage queues are not collection source dependent and are instead based on which database we wish to store the data in as we have multiple Elastic clusters.

2.2.5 Batchers

The batchers exist to take messages off of the storage queue and group them together into bundles of up to 100 messages or the number of messages placed on the queue within 30 seconds and put them into the specific database they are assigned to. This system was implemented because without it we were placing messages into the database one by one which was causing stress on it, these batchers act as a buffer to keep down the number of storage operations.

2.2.6 Elastic

Elastic, formerly known as Elasticsearch is a distributed database written in Java optimised for speed of querying and retrieval, all data stored in it is in json format and you query it through a REST API. It is ran as a cluster with master nodes and worker nodes or as a standalone single node cluster, we use it in clustered mode in order to increase speed.

At Blurrt we have several different Elastic clusters one for development, one for ordinary production data and one for high performance production data for high paying clients. This is not directly relevant to our project as the parsers do not interact with this in any way however it is important to understand the system.

2.2.7 Cron

Cron is the unix method of scheduling commands to be executed at regular intervals and is normally used for system maintenance or administration. At Blurrt a bash script executed minutely via cron is used to check the queue size and send off the requests to scale up or down, which while simple is not a recommended way of using cron as it is making calls to other servers. It also makes the monitoring script hard to update as the cron system and its configuration file, known as a crontab are not version controlled.

2.2.8 Monitoring Script

This is the bash (unix command line) script executed via cron on a minutely basis, the code for which you can see in appendix A. This gets the current CPU usage, the current tweet collection rate and the current tweets not parsed by sending a request to rabbit queues (running on the localhost) querying the queue size. It then sends the results of those queries to the server stats server, which is the graphite server used for creating graphs of resource usage and other system information. The last 7 lines of the script are responsible for taking the tweets not parsed value and if it's greater than a threshold, which is currently set to 2000, send a request to scale up and if it is less than 500 send a request to scale down.

2.2.9 Graphite

Graphite is a method for graphing time series data in real time, you send requests containing the a metric name of your choice and the current value for that metric and it plots that data as real time data on a graph viewable via a web frontend. It is used at Blurrt in order to monitor the health of the system with metrics such as CPU usage of servers, queue sizes and elastic cluster size, these are all collected from the servers in question via cron script.

2.2.10 Rackspace

Rackspace is the hosting platform used at Blurrt, it allows for the creation of virtual machines from vanilla linux distribution images or from premade images based on running servers. It then allows you to choose the server's specifications based on a choice of predefined general purpose images. Another feature is the ability to create autoscaling configurations based on selected a server image and provides custom URLs to use as web hooks. Rackspace also allows for the creation of remote storage drives and load balancers as well as other typical cloud hosting provider features.

Creating an autoscaling group provides individual autoscaling hooks which are URLs to either scale up or scale down for that group. The Rackspace autoscaling groups also manage the maximum and minimum numbers of servers for each group, the image to be used for each group and the cooldown periods which prevent multiple scalings happening within that time.

2.2.11 AWS

Amazon Web Services or AWS is the hosting platform I ended up choosing to use and it has a wide variety of different products available, the most important for this project is the EC2 instance service which allows for the creation of cloud virtual machines it refers to as instances. You can choose the operating system on the instance as well as the specifications of the instance from an available selection configured either for burst or sustained processing. The configuration settings for instances allows for the creation of autoscaling groups much like Rackspace but they do not

use web hooks in order to scale, you have to either use the AWS software development kit or use their metric scaling based on things such as CPU utilisation on the instance or of another of their services. Another technology used during this project provided by AWS is ECR which stands for EC2 container registry, this is Amazon's own container repository can be used as private remote container storage that machines running docker can access in order to quickly download container images instead of using the more public official container repository Docker Hub. It was used in this project as, due to secrecy reasons, Blurrt did not wish to place their codebase on a remote server out of their direct control.

2.2.12 Docker

Docker is a common method to simplify interacting with and running software containers, providing tools to create, manage and execute them. Containers are "a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. Available for both Linux and Windows based apps, containerised software will always run the same, regardless of the environment." [13].

Docker is often run in swarm mode, a swarm is a collection of nodes which are used to run services as opposed to individual containers, "when you run Docker without using swarm mode, you execute container commands. When you run the Docker in swarm mode, you orchestrate services." [14]. Nodes are machines running Docker which are participating in a swarm, there can be either manager nodes, which both send tasks to worker nodes and execute tasks, or worker nodes which merely execute tasks. A service is the definition of the tasks to execute on the nodes, it is the primary root way of executing commands on the swarm system. When a service is created you specify the container image and the commands you wish to run inside the container. In order to execute multiple containers across the swarm you use services in replicated mode, when this occurs the swarm manager distributes a the specified number containers with the specified command across the nodes based upon the number of containers you wish to have [14].

2.3 Technical System Architecture Overview

In figure 1 you can see a diagram outlining the architecture of the current system. At the start the collection feed, in the case of this project from GNIP, comes into the system and the tweet objects which are in json format are placed onto the parsing RabbitMQ queue.

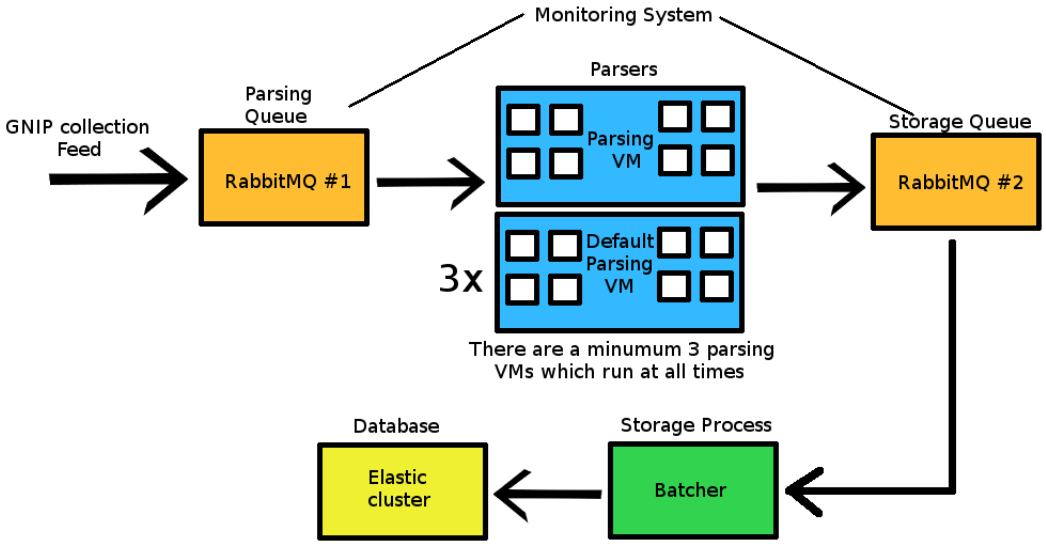


Figure 1: The Architecture

The parsers take the tweet objects off the parsing queue and parse them before placing them onto the storage queue. At all times there are minimum of 3 parsing VMs for the GNIP parsers set by the autoscaling group, each containing 8 parsing processes. These parsing VMs are created using the 8 GB General Purpose v1 virtual server on Rackspace which has 8 vCPUs, 8GB of RAM, 160GB of diskspace and 1.6Gb/s disk network. There is a hard limit set to the autoscaling

group these are part of with a maximum of 15 of these servers allocable for GNIP parsing. These parsing VMs are created with a specialised parsing image (which is an Ubuntu 14.04 image with the minimum software required to execute the parsing process). This image is created using the Rackspace web portal's "image creation from running server" feature manually every patch.

The monitoring system monitors the parsing queue and if it exceeds the sizes mentioned in section 2.2.8 of above 2000 and below 500 it sends a request to an autoscaling webhook URL which contains an API key corresponding to one of the autoscaling groups that has been created. Each webhook has a separate cooldown and there is a 3 minute cooldown to scale up and a 10 minute cooldown to scale down.

Parsed messages are placed onto another queue, this time for storage, which is monitored by a separate monitoring system running on cron on the storage queue server. This does not impact the scaling of the parsers in any way it is simply so we can be aware if there is a backlog for storage.

The batcher then takes messages off the storage queue and places them into the elastic cluster in groups in order to prevent I/O load on the cluster.

2.4 Existing Material

Virtual machines and cloud computing are very important technologies for web applications in order to provide easy hosting without requiring managing the hardware and operating systems yourself. Because of their popularity there have been several papers written on improving the deployment time of virtual machines, Risto et al [20] worked on the transfer of images to virtual machines and increasing the speed using different transfer methods such as multicast or BitTorrent, this relies on having access to empty virtual machine hosts without an image attached to them. Another paper by Gaochao et al [15] which proposes a rapid virtual machine deployment strategy using compression and multicast, this relies on being the owner of the hosting service in order to implement the required improvements in infrastructure. There have also been management algorithms proposed such as CloudFlex [18], which proposes a way to manage virtual machine images in autoscaling across many different cloud providers and reduce costs by not deleting images until the end of the hourly charge period. There have also been systems proposed using machine learning in order to predict when incoming demand will occur and scale up in advance to offset the delay in creating virtual machines such as the paper by Sadeka Islam et al [17], this is a very interesting idea and relevant to the work I am doing on my project in terms of improving autoscaling speeds. In terms of comparing different hosting services there has been little work done directly comparing their scaling speeds for example this paper by Mehran et al [16] compares the architectures of autoscaling on several hosting services but not the time taken to scale, rather it proposes a general purpose model for these scaling actions. I was able to find information in a 2010 paper [19] on just Microsoft's Azure hosting service which stated that there was about a 10 minute delay when creating virtual machines from specific images, this data may now be out of date as the infrastructure Azure is based off of is likely to have improved substantially in the 7 years since that paper was written.

There has not been, from my research, any work done on how to re-architect a system in order to reduce the initial deployment time in an autoscaling system from the client side rather than the hosting side, which shall be the focus for this project. This is important because many start-ups or other small companies cannot afford the time or cost required to host their own servers, which would allow them to implement a hosting side solutions and instead want a way to implement faster autoscaling with the existing hosting companies and software solutions available. Some of the papers such as Sadeka Islam et al [17] have useful concepts and others such as Hill et al [19] have useful data for my project.

3 Specification & Design

3.1 Hosting Technologies

I researched several different hosting and software solutions before settling on using a container based architecture in order to solve my problem. The initial choice was whether to continue down the path of pre-created virtual machine images, implement some form of start-up script which is ran when the new virtual machine is created or implement a container based system.

3.1.1 Custom Virtual Machine Images

This solution would involve keeping the idea of manually creating virtual machine images and manually creating the image each patch. This would go against my aim of improving the ease of implementing updates and isn't ideal as it has on multiple occasions been forgotten that these images need to be updated. Also it would not solve the inefficiency in that the servers have to already be running parsing processes when the image is created in order for them to parse. Furthermore the performance gains that come from this would only then only come from cutting down the image by reducing the required amount of data on the disk, switching to a smaller version of Linux which takes up less disk space such as Alpine [1] or from the underlying improved infrastructure from switching to the new service compared to Rackspace. I decided this would not be a suitable way of improving the performance as it would not implement enough of the required aims of my project.

3.1.2 Provisioning System

The other option would be to spool up blank servers and then run a script on them using a tool such as chef, ansible or puppet, which are responsible for managing the automatic deployment of software. These are able to run the configuration required on the server after the sever has been created and then start the parsing processes. This would definitely meet the aims to improve the ease of deployment, as it would be relatively easy to write a script which would work automatically for each new release as all of the software is version controlled, this would also make it very easy to create up many servers as required. However these system can be very slow to run, with ansible potentially taking as long as 20 minutes to execute on some systems while smaller configurations can take between 5 and 8 minutes. I decided this was unacceptable as it would not meet my deployment speed aims.

3.1.3 Container Based Solutions

The other option would be a container based solution using something such as Docker swarm or Kubernetes, this would allow for easy deployment as it would be a case of creating and pushing the image to a Docker repository (which could potentially be automated with a CI system) and then having the swarm pull from it when the size increases. It is also very fast as creating new containers is done in a matter of seconds rather than minutes [21] taken for normal virtual servers. This is not without its drawbacks however, in order to scale up the computing power of the docker cluster you would have to add virtual machines to it which would require either setting up a system which automates the adding of the new virtual machines to the cluster or using a solution which manages this. There are several docker based hosting solutions such as Carina from Rackspace [7], Google Container Engine [3] and Docker for AWS [4] which all aim to simplify the creation and management of swarms meaning less time is required to set up the swarm.

3.1.4 Platform as a Service Solutions

Platform as a service (commonly known as PaaS) was invented in 2005 but was popularised by Google in 2008. PaaS takes the idea of infrastructure away from the development process by handling the hosting, deployment and scaling all for you, this responsibility it given to the company you choose as your PaaS provider. This would have the advantage of meaning that I would have to focus only on the re-architecture required to set it up on the new system and simplifying the deployment process, as it would remove a lot of the development required in terms of setting up the new infrastructure correctly. There are several popular PaaS providers such as Amazon's Elastic Beanstalk [6], Google's App Engine [2] or Heroku [9].

3.1.5 Hosting Choice

This project went away from using custom built virtual machine images as I believed this was not substantially differentiated from the existing architecture and as such would not bring about any improvements in terms of architecture except from potential improvements brought from switching to a different hosting provider. It also does not solve the problem of updates and migration in any way.

I also chose to move away from using a provisioning system as I did not believe there would be substantially increase in speed compared to a simple virtual machine image. This is because these systems are not set up for speed but rather for ensuring that you can create correctly configured servers from identical starting point, as such I did not think it would meet my aim of reducing the start up time of the parsers.

I chose not to use a platform as a service solution because whilst it would be a way of solving the problem, due to the fact I wish to demonstrate I have improved the current monitoring and scaling system and the speed of deployment in response to this monitoring and PaaS is responsible for doing this it would not demonstrate enough work for this project. Placing it on a PaaS solution would take away the handling of the scaling from me and places it in the hands of the provider. Also I am doubtful whether the re-architecture required to place the system, which requires several hundred megabytes of various libraries in order to achieve the required functionality, would be conceivable in the time frame as PaaS solutions focus on small applications in the tens of megabytes.

It was chosen instead to use a container based solution, this was because containers are much faster to deploy than virtual machines and are able to handle deployment management through the use of a repository. There has also been little to no work done on getting autoscaling working within a swarm which combines both scaling the number of containers and the underlying number of nodes within the swarm in one go. As such this will be a research topic of interest even if there are no scaling speed gains yielded and I will still get the advantage of having an improved deployment architecture.

When it came to assessing the available container based solutions I chose to go with Docker for AWS. This was because ,while the company currently uses Rackspace and as such would prefer to keep their infrastructure on the same platform for simplicity sake, Carina is currently in beta and as such it is not suitable for any production ready solution as it can be taken down at any time and service is not guaranteed, indeed as of the 18th of April this service is currently down as the beta has ended. Another option was Google Container Engine, this is Google's hosted solution using it's Docker container management system Kubernetes. This differs from another popular way of managing groups of containers Docker Swarm by not using the normal Docker CLI or API but by instead creating its own interface and concepts [10]. After discussing this with my supervisor at Blurrt and with my project supervisor I decided it would be best to make use of Docker Swarm as it is a simpler framework and is officially supported by Docker themselves, it is also considered easier to set up compared to Kubernetes so making personal clusters to test would be easier. Another reason that factored into this decision was my supervisor at Blurrt was more willing to use AWS as a hosting solution than Google Container Engine as it is more widely used. This lended itself to Docker Swarm as Docker swarm is recommended for use on AWS and is best supported there whereas Kubernetes is best supported on Google Container Engine.

3.2 Scaling System Design

With the hosting and architecture being chosen my focus shifts towards how scaling would work in this system. The Docker for AWS set up template [4] creates autoscaling groups for both the Manager nodes (those nodes in a swarm responsible for controlling the other worker nodes and from which commands are executed) and the worker nodes. These scaling groups are responsible for scaling the number of underlying EC2 virtual machine instances which are part of the swarm, they are also responsible for the cooldown period between which the scaling operations are allowed to occur, which prevents multiple scaling operations from occurring within the time period specified.

In my design I would wish to have a system which gracefully scales up the size of the swarm and then scales up the size of the service afterwards, this would prevent containers being created prematurely and causing container crashes due to lack of resources. It would also want to scale down the service before scaling the size of the swarm down in order to prevent the same problem occurring when the swarm recreates destroyed containers on the remaining instances. I also would wish to have a scaling down cooldown period which is longer than the scaling up cooldown period. This would be so that it is capable of scaling up quickly and then scaling down slowly in order to ensure that if the queue does empty and cause the machines to scale down it doesn't do so too rapidly causing the queue to fill up again. AWS as a hosting service does not allow for separate upscaling and downscaling cooldowns, unlike Rackspace and as such this would need to be implemented manually in the monitoring process. Also due to the way on-demand pricing works, which would be most suitable pricing for an application which wishes to scale up and down

as demand increases, in an ideal system it would be beneficial to prevent destruction of servers until the hour long period is over in about to end. This would be to prevent being charged extra if you were to scale up again within the hour. Instead you could simply scale the size of the service in order to reduce the number of containers that would normally sit on the swarm and leave the EC2 instances up until the end of the the hourly allocation at that point you could either delete them or reuse them if the demand had increased by scaling the service size back up.

3.3 Monitoring Design

There is currently a bash script which runs by cron used for monitoring, which is hard to maintain and update. Instead I wish to move this to another system of my creation which has greater control over when to scale, how to scale and is capable of keeping a record of the previous queue size values. I would also like to be able to send other data to the graphite server such as the node count, which is done via the AWS sdk available for many programming languages but not bash as it's a scripting language and if I wished to do so would require replicating the functionality myself. There are several programming languages I could use in order to implement this, the main ones which are used at Blurrt would be PHP which I am relatively fluent in or node which I am less so. I decided to go with node despite having less experience in it as there are other monitoring scripts such as the slack bot, which displays the API status, that are written in node and so it would make sense from an architectural standpoint to have these in the same language.

Initially when I was first researching how I would go about implementing the monitoring system in order to see if there was a web hook system similar to that of Rackspace I could not find such a system. What I found were AWS' built in autoscaling configurations which only scale off of metrics of the server itself, so for example if CPU usage is high for a given period of time e.g "add an extra server when CPU utilisation is over 80% for 300 seconds.". This would have made it awkward as you are only able to impact it through the metrics such as CPU usage which obviously depend on the number of containers running and not the queue as the queue acts as a buffer for increased demand. I did not believe that it would be impossible to make use of this with docker however, as increasing the number of containers on the swarm will increase the utilisation of resources of the servers running it, therefore leading to an increase in the number of servers in total. While not ideal, as if there is too much RAM utilisation on the server the container exits, this was my initial idea of how to potentially solve the scaling problem.

Further research led me to find a better potential way, using Amazon's Simple Queue Service, which while not the same as RabbitMQ we use currently could be used in a similar way. Amazon allows their built in autoscaling to make use of the size of the SQS queue which would be roughly analogous to the way the existing system worked, in that it monitored queue size and sent a request to scale up based on size. In order to do this you would need to use one of AWS Lambda hooks which trigger based off events. This is not without it's own problems however, as now increasing the number of instances is not linked in any way to the number of containers. If it was poorly implemented and I tried to create a system which monitored the queue size manually using the SDK it's possible that scaling of the instances could get out of line with the number of containers. For example if the time the service scaling system happens to poll is the exact time the queue goes above a scaling threshold but Amazon's built SQS scaling trigger does not poll at that moment and it then drops below the threshold again it could lead to either there being too many containers causing them to exit early or too few meaning money is wasted. Another way of implementing the service scaling which I realised would potentially be better is to implement a system which polls at regular intervals the number of nodes and then changes the service size in line with that. However, this would cause problems when scaling down as it would not be able to preempt the removal of the node and as such parsers will be recreated on the remaining nodes, leading to them running out of memory on the node and exiting until the polling system interval comes around and notices a node has been removed.

Later I realised when I was implementing the SQS system's Lambda hooks that it was possible to change the size of the scaling group using the AWS sdk. Using the SDK and changing the value of the autoscaling group's desired size property you are able to scale the size of the group to whichever size you choose in the API call, this causes new instances to be created or destroyed on demand. This enabled me to implement a more unified and coherent monitoring system where both the size of the swarm and Docker service are scaled by the same system in order to prevent polling differences causing them to become out of sync.

3.4 Initial System Design Diagrams

Before I realised that I was directly able to change the size of the autoscaling group through AWS lambda function I created diagrams explaining how I envisioned the system would work. These were then updated and changed when I realised the superior way of implementing the system.

3.4.1 Class Diagram

While Node doesn't strictly use classes I believed it would be beneficial to design a class diagram to give me a clear design from which I can work on and develop my project. In this diagram the classes represent the 2 different node files which would need to be written to get my project working.

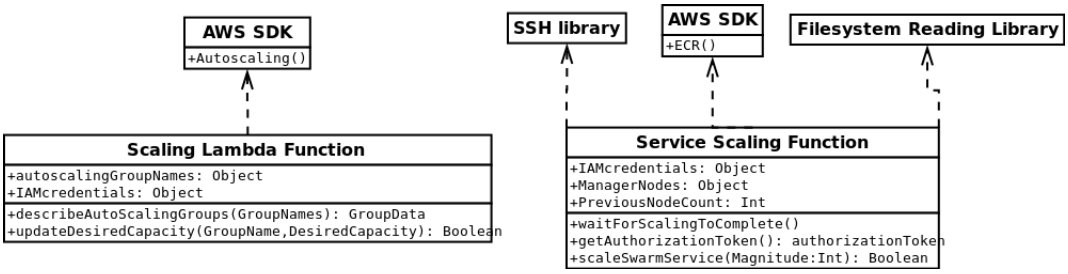


Figure 2: Class Diagram of the Initial Layout of the Monitoring and Scaling System

In this diagram you can see the 2 parts of my design of the monitoring system with the scaling lambda function file which is responsible for scaling the number of EC2 instances and the service scaling function file which is responsible for scaling the size of the service on the swarm.

The scaling lambda function file would trigger from a lambda hook based on the size of an SQS queue, so for example greater than 2000 would trigger it causing it to scale up and less than 500 would cause it to scale down. The file contains the attributes autoScalingGroupNames in the form of an object to allow for multiple configurations which reference either development or production scaling groups in the future. This is needed to pass to the methods describeAutoScalingGroups and updateDesiredCapacity in order to information about and scale the group. The IAM credentials are also stored as an object as the AWS sdk needs both ID and secret in order to allow for access to the autoscaling methods it provides. As for the functions it provides it has describeAutoScalingGroup which allows you to get the current size of the autoscaling group, needed in order to calculate what the new size will be. Then there is the updateDesiredCapacity method which takes the chosen group name and desired capacity and uses the autoscaling SDK to alter the size of the autoscaling group.

For the service scaling file I decided to go with the method which would require consistent polling of the swarm size discussed in 3.3 rather than the method that would that would look at the queue size as this would prevent the 2 from ever getting out of sync. The service scaling function file also contains the IAM credentials attribute, this time for the ECR container repository, this allows you to retrieve an access token allowing the swarm to download the container image to the new instances when it scales. It also contains 3 methods, the first is to waitForScalingToComplete which requires no arguments and would consist of a loop polling the size of the swarm, in order to do this it would need to sign into the manager node server by use of the filesystem reading library to read a private key stored on the service scaling function server and the SSH library to use that key to SSH into the server, once on there it would check the size using the "docker node ls" command and compare it to the value in PreviousNodeValue. Once the number of nodes has changed from what it was previously it would then trigger the next method getAuthorizationToken. GetAuthorizationToken makes a request using the AWS SDK to get a login for the ECR repository and returns it, this is then passed to scaleSwarmService. ScaleSwarmService would SSH in using the same manner as the first method and then execute a swarm scale command through the command line based on the parameter passed into it.

3.4.2 Component Diagram

I also created a component diagram (figure 3) to display more clearly the parts of the system. It would also show how the parts which I would not be writing code for such as AWS interact with the system. I also believe this diagram shows quite clearly how the instance, docker swarm, parser hierarchy will work in reality and which components of the system will interact with which aspect of this hierarchy.

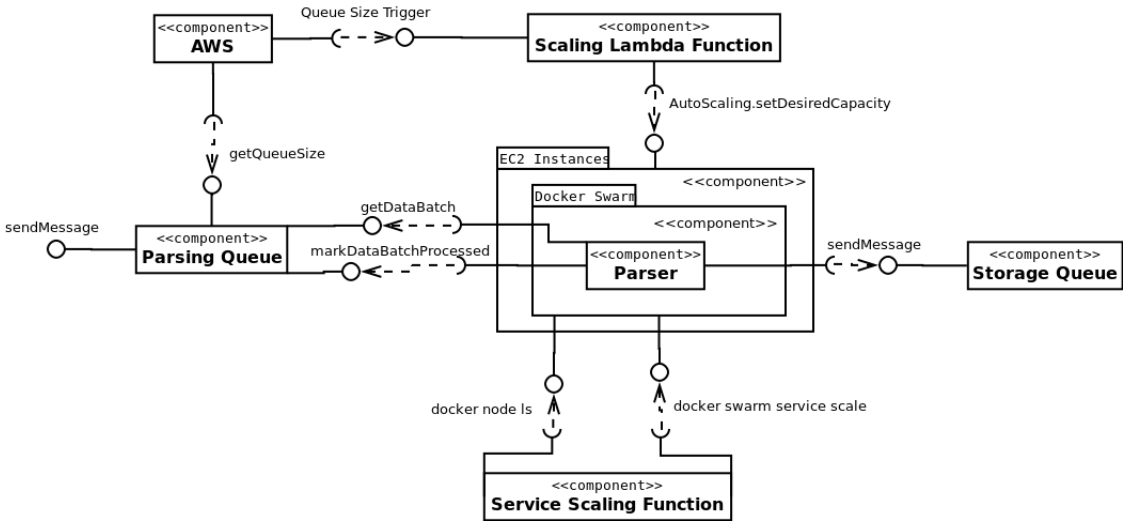


Figure 3: Component Diagram of the Initial Layout of the Monitoring and Scaling System

3.5 Final System Design Diagrams

When I realised I was able to implement the system without the use of lambda functions I decided it would be best to re-architecture the system. I created a state diagram because as I now have greater control over each step of the scaling process the states would need to be more clearly defined and follow on from each other. After I completed the project I updated the existing class diagrams and component diagrams to better reflect the finished product.

3.5.1 State Diagram

This figure 4 was created in order to envision how I thought the flow of the system would occur every time the system checks the queue size.

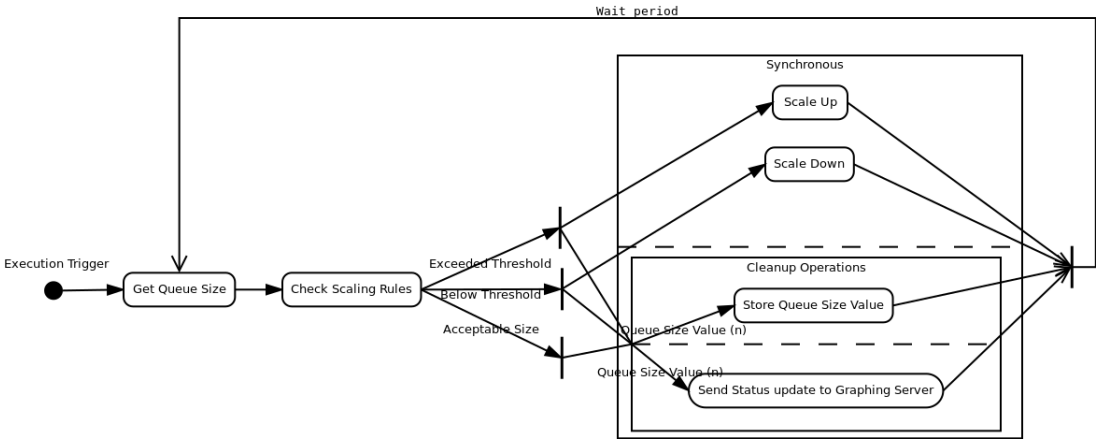


Figure 4: State Diagram of Monitoring System

I started by envisioning a polling system which checks the queue size at regular intervals, as querying queue size requires making an API call. I then would need to compare the queue size value

to the rules I have created, based on the outcome from those rules there would be 1 of 3 outcomes; scale up, scale down or do nothing. Because of the event driven programming architecture of node while the requests to scale up are taking place it is possible to execute storing the current queue size value and send the data to graphite so that a graph of the number of nodes can be plotted. When both of these have finished executing it then continues to wait until the wait period is expired and the next polling of the queue size is required.

3.5.2 Updated Class Diagram

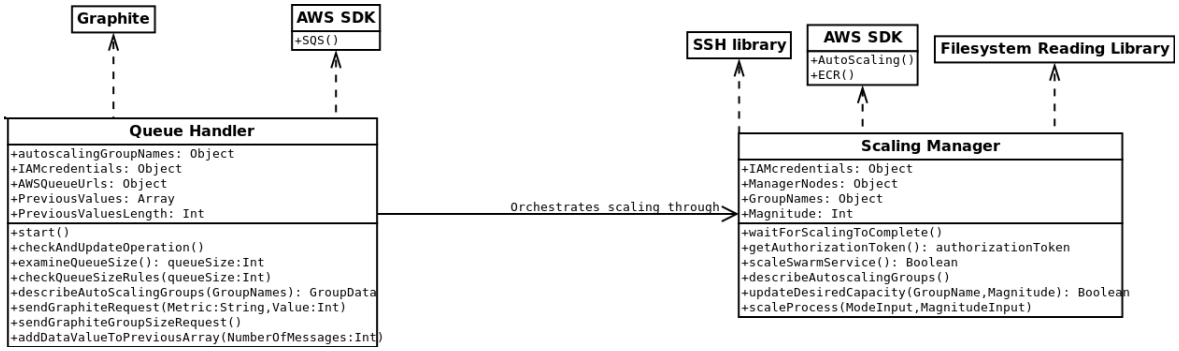


Figure 5: Class Diagram of the Updated Layout of the Monitoring and Scaling System

The main changes between the original class diagram and the final design is the movement of all of the scaling operations into the Scaling Manager file and the addition of the new file the Queue Handler. The scaling manager is able to scale the autoscaling group, wait for it to complete then scale up the swarm. This queue manager contains a start method which creates a timer, this timer at regular intervals then checks the queue size and sends scaling requests. I also chose to implement a graphite library so I could plot the performance of my code in terms of queue size and node count. The queue handler orchestrates scaling through calling the scaleProcess function directly from the checkQueueSizeRules function.

3.5.3 Updated Component Diagram

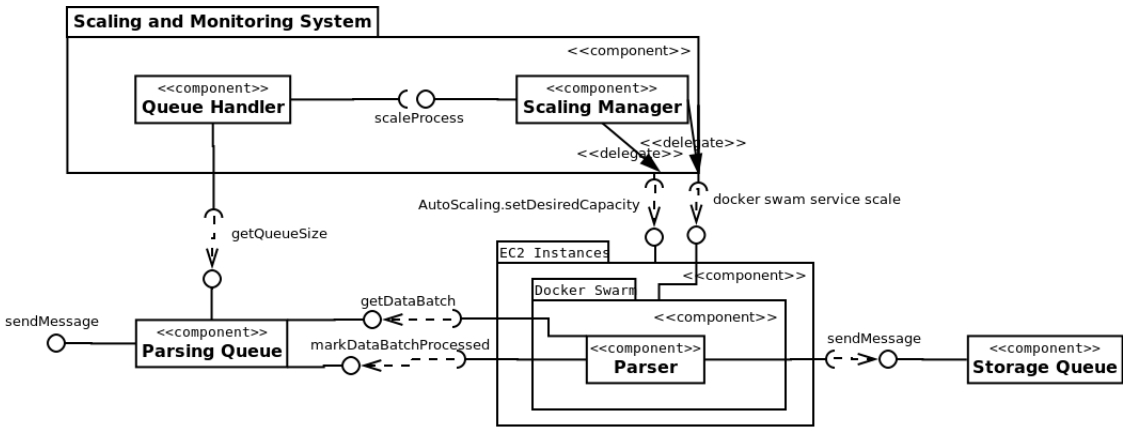


Figure 6: Component Diagram of the Updated Layout of the Monitoring and Scaling System

The were only a few changes in the component diagram from the original to updated version. The most significant changes were the removal of the service scaling function, as it was rolled into the scaling manager along with the functionality of the scaling lambda function and the removal of AWS as the trigger for the scaling which was replaced by the queue handler. This together formed a new component which was a combination of the queue handler and scaling manager which acts as one to form the scaling and monitoring system. This updated design is much more streamlined and easier to understand, making it easier to extend and develop further while also being less difficult for other potential developers to understand.

4 The Implementation

4.1 Implementing Docker Swarm

4.1.1 Overview of Docker Swarm Implementation

Using the template for Docker Community Edition (CE) for AWS (stable) [4] (which at the time of implementation was merely called Docker for AWS but has since been rebranded as Docker has brought in Docker Enterprise edition). I created up a Docker swarm with 1 manager node and 2 worker nodes, with t2.small as the the instance type. I set the SSH key on the cluster as a new RSA key I created without out any password protection, I left the rest of the settings as default. This gave me a working swarm cluster onto which I could then use the key I created to programmatically access the manager nodes from the scaling manager later.

While the choice of t2.small is good for proof of concept and very tuned scaling levels it might not be the best way to implement the system in production as t2.small instances are limited from consistent usage by their CPU credits model, which means by default it has performance equal to 20% of a CPU core and allows for burst above that through spending credits it accrues every hour. These credits allow for a minute of burst usage, in the case of the small it accrues 12 credits an hour meaning 12 minutes of burst usage, any extra work CPU is done at the 20% rate baseline rate. As such in production it might be better to implement this with another instance such as c4.large, their compute instance, or m4.large their general purpose instance.

4.1.2 How the Docker Swarm Was Implemented

At first in order to get myself accustomed to using Docker, before I had access to an AWS account at the start of the project (I had to get it approved by my manager at Blurrt and the accounting department), I familiarised myself by setting up simple docker swarm with 1 manager node and 2 worker nodes on Rackspace. This was relatively simple to do following the guide from the docker website I simply set up 3 Ubuntu virtual servers on Rackspace and then installed docker and configured each of them for swarm mode with 1 manager and 3 parsers.

After AWS access was approved and I was given the credentials to log in I was able to set up a docker for AWS swarm through the following steps:

1. Using the template for Docker for AWS I specified 1 manager node and 2 worker nodes and the instance type as t2.small the rest of the settings as default. I created a new RSA key, separate to my personal RSA without any password protection and enabled that as the key to log in to the manager nodes. This meant I was able to ssh into the server and also use it for programmatic access via the scaling manager later.

2. I then went through the process in section 4.3.2 and got the docker images on the manager node, at that point I attempted to create a Docker service with 3 replicas one for each node.

3. In order to do this I ran the command "docker service create –with-registry-auth –replicas 3 –name parsingprocess [repository url]/[repository name]:[repository image] php parse_blurrts.php gp_tweets" into the shell on the manager node. Where "php parse_blurrts.php gp_tweets" are the commands to execute a GNIP parser.

This demonstrated that I had a working system to create orchestrated containers across the Docker Swarm from my prebuilt images.

I also realised that the IP addresses of the manager nodes were not set and would change if a node exited and then automatically started again, preventing the scaling system from working. I researched the way to set a static IP on AWS and found the best way to do it would be with AWS elastic IP, which I created and assigned it to the manager node to help prevent unforeseen circumstances occurring.

I did encounter a few problems during this implementation, firstly when I tried to run the docker service in step 3 I found that while the containers on the manager node had been created the containers on the worker nodes had not. Upon further research about the error I discovered that you need to pass the with-registry-auth flag in order to pass the login authorisation to the other nodes as well as just the node you execute the command on.

Another problem I found during this implementation was when I specified t2.micro as my instance type containers were regularly exiting every few seconds with the error code 137, which

corresponds to out of memory exception [11] which suggested to me I needed to increase the size of the the node memory. Upon inspection the default size of instance was a t2.micro which only has 1 gb of available memory, I ran a parsing process on the development server I created and found it uses 700mb on average and I expect if that was to spike it could potentially exceed the memory limit when combined with the overhead of running the operating system as well. I then went into the set up of the Docker for AWS template and changed the instance size for all nodes from t2.micro to t2.small, which changed the memory limit to 2gb rather than just one, from my testing after that I was able to consistently maintain 2 parsers on it at all times. I chose t2.small as it allowed for the smallest granular scaling as it has the lowest resources available (with 1 CPU and 2GB of RAM) which was also capable of supporting a parser.

4.2 Implementation of Containerised Parsers

4.2.1 Overview of Containerised Parser Implementation

In order to take existing code and turn it into a Docker container I created a Dockerfile, which is Docker's configuration for building an image, this Dockerfile contains the bash install instructions required to set up the bare minimum resources required to run my code. It installs PHP, all necessary PHP modules and then transfers the existing parser codebase on the server I am building the image on, into the container. This codebase had already had composer (a dependency management system for PHP) pull the dependencies into it so it contained all the necessary files. The base of the container image was Ubuntu, this is because it is what the production servers use, if I wished to get the size of the image down significantly I could use a smaller distribution like Alpine but I did not implement this during my project. I created Dockerfiles for both PHP 5 and PHP 7 in order to test the performance of both.

4.2.2 How the Parser Containerisation Was Implemented

The first step was to work out what was required to get the parsers working natively on an Ubuntu virtual machine with the minimum amount of required packages, as unnecessary packages would increase the size of the container image. Setting up an Ubuntu virtual machine in this way would give me the same steps to set up a container image based on ubuntu with these packages.

I followed the below steps to get working parsers on the virtual machine:

- I cloned the git repo for the parsers into a directory on the virtual server, for simplicity I chose ~/Parsers.
- Installed php5.6, the version used in production for the parsers.
- I then installed composer via their recommended install method the steps of which are here [8]
- I then ran composer install on the root directory of the parser's directory, I encountered a problem whereby I was unable to pull from the private github repository with composer (changing over to github was relatively recent). I fixed this issue by creating a github authorisation token on my account and adding it to the git URLs to pull from within the composer.json configuration file.
- I then attempted to run the parsers and found that I was missing certain PHP modules which have to be installed separately, after installing the bcmath, mbstring, xml, curl and mysql PHP modules the parsers would run correctly.

After that I created a Dockerfile to set up an image based on this install process I had created, it starts with "FROM ubuntu" which specifies to use ubuntu as the base image for the container. The next command in the Dockerfile was to install php and all of the required modules, I did this all on one line as it prevents adding extra layers to the container image. I then used the ADD command to add the codebase I had pulled the dependencies into with composer into the container. I chose to put it into the "/var/blurrt/miles/blurrt-phirehose/" folder as that is the folder used for the parsers currently. I then set that folder as the working directory so when commands are executed on the virtual server they are executed from that folder by default.

During this installation I made the mistake of trying to install the tools to clone the repository with git inside the container rather than specifying the directory to copy over from the server you

are building the image on via the add command. This goes against the underlying ethos of Docker containers doing one thing only. After realising this seemed like a poor way of doing things I found out about how you transfer the contents of a chosen directory into Docker with add.

After testing the container was working and I was confident I had a working docker parser, I created a docker container with PHP 7, which was configured almost identically but I had to change the install instructions in the dockerfile to refer to the PHP 7 versions of the modules and had to rerun composer on the build server changing the composer configuration slightly so that it pulled the PHP 7 version of a module rather than the PHP 5 one.

4.3 Implementation of the Container Repository

4.3.1 Overview of Container Repository Implementation

I created an EC2 container registry, this allows for the pushing and storage of containers into a private repository. Setting this up was very simple. I had to name it and clicking OK however the process leading up to it was very time consuming and I shall outline this in section 4.3.2 below. The ordinary way to implement a container repository would be to simply upload it to the Docker hub [5] however after discussing this with my manager he did not want me to do this for secrecy reasons as it would involve placing the entire Blurrt parsing codebase on another remote server which we have little control over compared to our own virtual instances.

4.3.2 How the Container Repository Was Implemented

In section 4.1.2, after I had created my swarm and was trying to test the swarm with my container image, I realised I had no way to get the container image onto the manager node. My initial thought was I could FTP across and build the container but I then realised this was not possible as the manager does not support FTP. Also doing so would not be desirable for multiple reasons:

- Firstly, if I was able to implement FTP on the manager node in order to get the codebase across manager nodes can scale and if they were to scale up and down the manager node containing FTP configuration could potentially be destroyed.
- If you were to FTP across the codebase you would have to do so on every single node, including the worker nodes (which in Docker for AWS do not have a documented way to login) and build the image on each node this is because the manager node itself cannot transfer the images across through "docker service create" command when using local images.

This made me realise there must be a better way of doing this, I researched if there was a private way to store and download container images. I looked into if it was possible to host a private docker repository on a virtual server we had administration of and discovered it was possible [12]. When trying to implement this I discovered to do so you have to have https enabled, which we did not have a valid certificate for, so I looked into the potential of self signing a certificate and placing that on the docker manager which would work provided the manager instance remains up. However this would not be suitable if we wished to potentially scale the manager instances or the manager instance went down. After further research found amazon has their own docker repository service ECR, which is an AWS Docker container repository, I set up this docker container repository and an IAM policy user (the AWS method of controlling access to services) with programmatic access to ECR services.

I then tried to push to the ECR repository with the docker push commands given in the AWS web portal, but was unable to so without first logging in to the repository. When I looked up how to do this I discovered I needed to get a special login command from AWS from the AWS command line interface, I installed the AWS command line interface on the server I wish to create the container images on and ran AWS configure which asked for an access key, secret key and the region I provided the user ID and secret key for the IAM user I created before and the region as the same region with the docker cluster on. I then ran the command on the container creation server "aws ecr get-login" which gave me a login in command to use in the format "docker login -u AWS -p [password] -e none [repository url]" which when executed granted me login rights for the ECR for 12 hours, which is a security measure.

Then I was able to push container images by first ensuring the image is built, tagging the image so it has an easily descriptive name from which to pull on the other side, then push the image to the repository.

From there I was then able to pull down on the Docker for AWS manager node side by running "aws ecr get-login", I then encountered a problem where the password token it gave me was too large to copy paste into the shell, which was not bash but a minimal shell which had a buffer length less than the password length, so I entered a proper bash shell and entered the command there before exiting the bash session. After that I was authenticated and able to pull with a Docker pull command in the same way I used a docker push command before.

4.4 Implementation of the SQSObjectCache

4.4.1 Overview of the SQSObjectCache

I implemented an extension to the PostedObjectCache interface, the code for which is section B of the appendix, which is part of the parser codebase and was created as a way to allow for the implementation of a new queuing system instead of RabbitMQ. Provided it followed this interface any queuing system could be used. Implementing the SQSObjectCache allowed me to make use of Simple Queue Service (SQS) with the parsers, enabling me to use the queue size triggers I believed I would need to use to implement the lambda function. In the end I used it as a simple queue which would be isolated for the rest of the Blurrt system.

It has 4 important methods which needed to be implemented from the interface, and one less so; getDataBatch, save, markDataBatchProcessed, markDataBatchUnprocessed and addToQueue. Every time a data batch is taken the message tags are added to the currentMessageTags array which was responsible for holding the IDs that need to be sent in the markDataBatchProcessed method in order to have them deleted off the queue. MarkDataBatchUnprocessed was interesting with SQS because it handles that for you, you don't need to send a request to the queue for this it simply re-adds it if it hasn't been marked as processed, this meant all I had to do was clear out the currentMessageTags array in order to ensure this worked correctly when something is unprocessed otherwise it would be deleted with the next getDataBatch.

4.4.2 How the SQSObjectCache Was Implemented

The implementation of SQSObject cache was straight forward I simply implemented the methods required in the interface and referred to the RabbitObjectCache if I was unsure what was needed for a certain method.

The most notable problem I encountered during the implementation of this was that you don't mark data batches as unprocessed, anything that isn't marked as processed, which is done through deletion off the queue, is re-added after a set period of time (which defaults to 3 minutes) after it has first been taken off the queue, that meant initially I believed I could just leave the mark data batch unprocessed method blank but I was finding I was accidentally marking too large batches processed because I forgot to clear my record of what has been processed Once I had remedied this by ensuring markDataBatchUnprocessed cleared this array of previously processed values SQS worked as expected.

4.5 Implementation of the Scaling System

4.5.1 Overview of the Scaling System

The scaling system which is implemented in the file scaling-manager.js is the system responsible for performing scaling operations, it has 5 important methods scaleSwarmService, getAuthorizationToken, waitForScalingToComplete, updateDesiredCapacity and describeAutoScalingGroups. There is also a method _scaleProcess which takes as arguments whether we want to scale up or down and the magnitude of that scaling, it then triggers off those 5 methods in the correct order, this method forms the way to trigger the scaling from the monitoring system. Essentially this scaling system performs the following steps:

1. Ssh into the server.
2. Perform a docker login command to ensure that the token hasn't expired from the previous time it was scaled.
3. If scaling up check whether the new node has been added to the swarm yet if it's scaling down skip to 5.

4. If it hadn't been added wait a time period and then check again.

5. Scale the size of the swarm to the size requested using a docker swarm scale command.

6. Disconnect from the server.

One of the most complicated methods for this is waitForScalingToComplete. This is a loop which, if the system is scaling up, performs an ssh operation onto the manager node and then checks the number of nodes on the system, if it matches up with the number that would have been expected from the updateDesiredCapacity operation it then stops looping and calls the scaleSwarmService function. It only performs this waiting if it is scaling up as if it did this when scaling down the swarm service would automatically recreate the parsing containers (which were destroyed from the destruction of the instance) on the remaining instances to maintain the service size, this would then cause the memory on the instance to be exceeded. This meant it would be preferable to scale down the size of the swarm instantly, I placed a condition into the code which said if it was scaling down to not wait for the swarm to scale before scaling the service.

The other complicated method is scaleSwarmService which is the function directly responsible for altering the amount of parsers. It performs an ssh operation into the manager node and then scales the swarm size by performing a "docker service scale parsingprocess=[desired service size]" command, where the desired service size is passed as a method to the function, it then disconnects from the server.

4.5.2 How the Scaling System Was Implemented

After node was set up I had to install the AWS sdk, this is done through node package manager (npm) and the configuration file package.json, which I created a small one of for my project, I then performed npm install within the folder installing the sdk. I also needed some way to get onto the server to perform the scaling operation via ssh, this was done via another package from npm called ssh2.

When I first started writing the scaling code I was unaware of the proper coding conventions for node, I created my code as many nested callbacks which quickly became unreadable, after further research I found it was possible to use promises. This did a few things I was finding very difficult to do with nested callbacks and made my code much more readable. This required me rewriting a fair chunk of the scaling system so it all used them however I believe this paid off in the end as my code is now more readable and extensible for anyone who wishes to alter or work on my code in the future.

Another problem I encountered when I initially began implementing this was when I believed it would be in the form of a lambda function which triggered via the queue size, my plan being to have the built in Amazon SQS triggers available for lambda to fire off my lambda function when the queue size got above or below the monitoring queue size thresholds I set via the AWS web portal and it would then perform the scaling on the docker swarm for me. During the implementation of the scaling function I realised the methods I was using to try and change the size of the queue were simply AWS methods available in the SDK and I could simply implement it by using npm to install the SDK without needing to use the lambda function as an extra step.

4.6 Implementation of the Monitoring System

4.6.1 Overview of the Monitoring System

The final part of the system which was implemented was the queue handling system, this was a separate file called queue-handler.js which required the scaling manager file. This file ran as a continuous process with a loop every 30 seconds to examine the queue size and apply my scaling rules in order to trigger the scaling process through the scaling manager's _scaleProcess method. It contains 5 main functions, examineQueueSize responsible for returning the queue size, checkQueueSizeRules responsible for triggering the scaling operations based on the queue size, addDataValueToPreviousArray responsible for documenting the previous values which is used by one of the rules, sendGraphiteRequest which sends a request to the graphite monitoring server and sendGraphiteGroupSizeRequest which sends a graphite request specifically with the group size. There is also the start method which sets up a timer every 30 seconds to trigger another method, checkAndUpdateOperation, which pulls together all of the aforementioned methods together and executes them in the correct order.

The most important method is checkQueueSizeRules, this method either scales up by 2 instances if the queue size is above 4000 or scales up by 1 instance if it is above 2000, this is important as it is able to account for if a very dramatic increase in queue size of over 2000 was able to occur since the previous check which would require dramatic scaling to keep up with. If the queue size is less than 500 it scales down by 1 instance, this doesn't scale down as dramatically as it scales up deliberately in order to prevent downscaling too quickly and having to then scale up again. The last rule in this function is a rule which looks at the result from the previous 5 queue size checks and if each one is greater than the previous scales up, this rule is in place to prevent a creeping effect where the queue could be consistently going up, leading to a constant backlog of a minute in parsing time or so, but not by enough to scale for a considerable amount of time. The number of previous queue size checks to look at are handled by a variable hard-coded at the top of the file previousValuesLength which can be altered as you wish but needs to be small enough to have a tangible chance of triggering with the number of messages difference between the scaling up and scaling down thresholds. This array containing the previous queue size values is handled by the addDataValueToPreviousArray method which creates a rolling queue of the size specified at the top, filling it up until it reaches the desired length and then repeatedly deleting the oldest value to add the newest value at the end of the array.

Another notable method is sendGraphiteRequest this allows for sending to the server monitoring system the value of the number of nodes in swarm and the size of the queue. In the implementation of sending the number of nodes in the swarm I hardcoded the manager node count to be 1 for the sake of simplicity but this could implemented dynamically by querying the size of the manager scaling group as well as the worker scaling group.

4.6.2 How the Monitoring System Was Implemented

This file makes use of the AWS sdk also used by the scaling manager as well as a new npm package graphite, this allows for easy communication with the graphite server and prevented me having to implement the UDP request manually in javascript, which would have been time consuming and less extensible.

The only significant issue I had during implementing the queue-handler was this graphite module, which was simple to use once I worked out the correct way to formulate requests however I found very little documentation for it online and there was little in the way of a readme on the github page which meant that I had to work out its functionality mostly on my own.

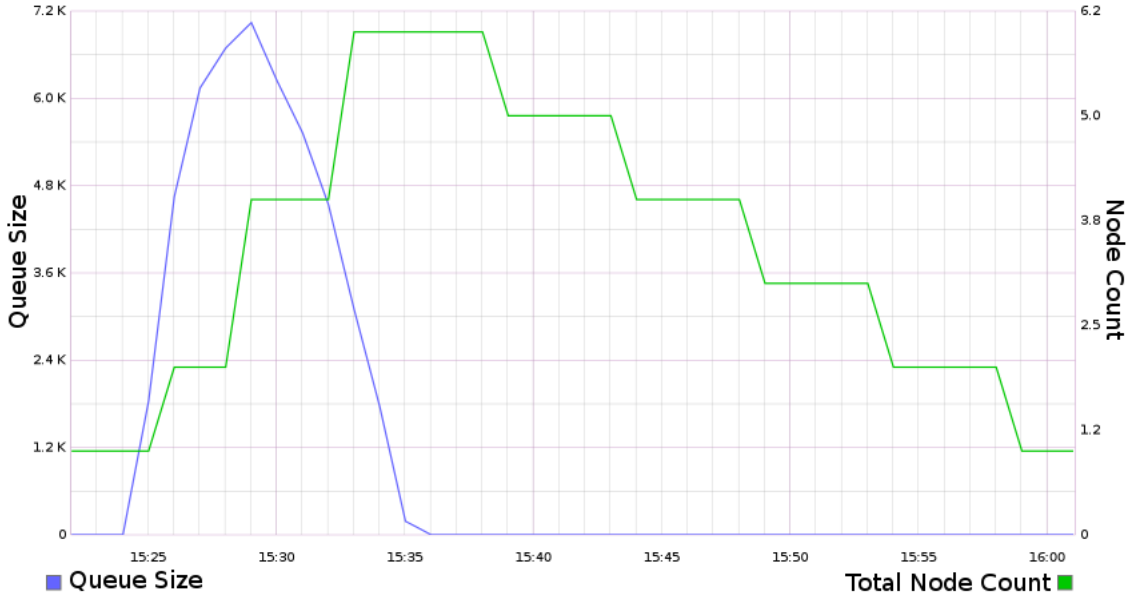
5 Results and Evaluation

5.1 System Demonstration

In figure 7 you can see the graphite graph of the data sent from the monitoring script over a 40 minute period, it starts with an empty queue and a one node cluster running 2 parsing operations and I then add 9000 tweet json objects to the queue in 1000 tweet chunks. I chose to use the PHP 5 iteration of the parser to demonstrate this as it parses slightly slower and as such would give a larger peak on the queue size graph and take longer to go down.

The system responded to increased demand as soon as the queue reached above the 2000 size threshold and as expected scaled up by 1 node, the 3 minute cooldown was then in place to prevent many scaling operations from occurring before the system has finished scaling. It appears from the chart that this operation took less than the 3 minutes I specified, this is on Amazon's end as they are in control of when the cooldown period actually gets acted upon. After this period was over the system looked again at queue size and found it was still above threshold this time over the 4000 size threshold and so scaled up by 2 nodes rather than just one, after this operation was completed there was a total cluster size of 4 nodes running 8 parsing processes. Up until the peak at 15:29 you can see the increase in size of the queue is starting to be brought down by the increased number of parsing processes, at 15:29 the 9000 messages have been completely added and the queue only starts to go down. When the cooldown is completed at 15:32 the queue is still above the 4000 tweet threshold and so scales up again despite the fact no more messages are actually being added. The cooldown for this operation seems to have taken 5 minutes at Amazon's end as the system refused to scale down before 15:38 even though several requests to scale down had been made during this period. After 15:36 the queue had been completely emptied but the

Figure 7: Graph Showing Increase in Deployment Speed Between Docker and Rackspace



system still needed to scale down, all of these subsequent scaling operations took 4 minutes to be allowed to cooldown on Amazon's end.

5.2 Performance Analysis

5.3 Parser Deployment Speed Testing

5.3.1 Justification of Parser Deployment Speed Testing

The first and main aim I wished to show through my performance analysis was an increase in the speed of the deployment of the parsers. In order to test the speed of the existing parser deployment I created a small script by editing the existing deployment script in appendix A so that it always scaled when called and display the exact time when executed, I then edited the existing parsing image and added a print statement inside the parsing method to display the date every time it goes through the parsing loop, which is logged in the log files. I then created a new image from this edited code and created a new autoscaling group and assigned the new image I had created to it. I am then able ssh into the server and view the log to see when the earliest print statement is in the log files and from that I can work out quite accurately when the parsing process began by subtracting the time taken between the time shown from the altered scaling script which creates the server to the time shown in the log files, accounting for any difference in the system time on the machines after running the date command at the same time on both.

In order to test the speed of the Docker development server I created a script by editing the scaling-manager code so that it would execute on demand and ran it through the command line prepending "date &&" to the command this gave me the time it started and appending "&& date" as it would exit when the scaling is completed. I then noted the the 2 times down and calculated the difference.

5.3.2 Results of Parser Deployment Speed Testing

From figure 8 you can see a reduction in parser creation speed of approximately 200 seconds, which is a dramatic improvement to the original deployment speed, part of this is likely due to improvements in architecture compared from AWS to Rackspace. I also think a substantial proportion of the deployment speed improvement is due to the instances themselves needing a very minimal base image to set up docker and the inherent speed of deployment of docker containers on top of that.

Figure 8: Graph Showing Increase in Deployment Speed Between Docker and Rackspace

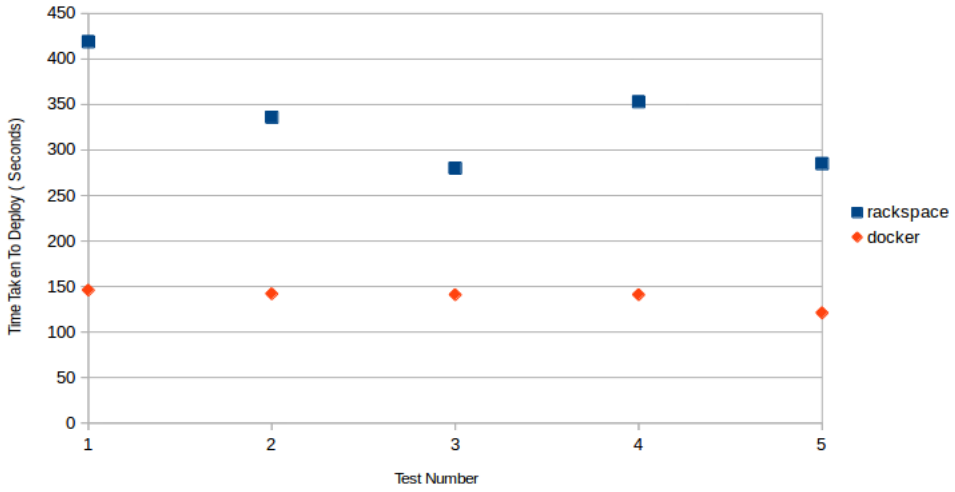


Table 1: Parser Deployment Speed Results (Seconds)

Test Number	Rackspace	Docker
1	419	146
2	336	142
3	280	141
4	353	141
5	285	121
Average	334.25	137.25

5.4 Throughput Testing

5.4.1 Justification for Throughput Testing

The next test I did was to see if there was a increase of reduction in the throughput of the parser, both in terms of switching from Rackspace in order to do this I measured the rate at which 1 parser parses 1000 tweets when comparing Rackspace hosting to Docker on AWS because whilst the parser deployment speed looks promising it is not useful if that then leads to a significant drop in the actual speed of the parsing itself. On the flip side if the change from switching to a Docker or a different hosting service can actually lead to performance gains it has the potential to save money as less parsing processes are required to parse the same throughput of tweets.

These tests were done on t2.small instances which is relevant to this test as t2 instances have a CPU credit system meaning that they have an amount of "tokens" allocated to them allowing for burst CPU usage but after that has been used up their CPU usage is throttled, this is an issue for parsing because it is something which tends to run continuously and it could potentially lead to a situation where the server runs out of credits and is not able to get rid of the queue sizes as fast as this testing shows. If that was the case I could look into changing the instance type from a t2 small to one of the compute instances c4.large or m4.large. This test should therefore be seen as a view of the throughput of these instances when running at their highest capacity possible.

In order to get fair timings of the parsing speed I had to ensure the docker images required to do this testing were already in place on the nodes before executing the command to start the parser, if I did not do this it would take around 5 to 10 seconds to download the image which would adversely affect the validity of the results. I did this by creating a docker service across all nodes to ensure each has the image on it already meaning when I execute the test none of them would need to download the image. In order to ensure I had a fixed testing benchmark I made use of a modified version of the SQSObjectCache.php with the following additions which are available for view in appendix C.5.

While this was not a perfect test as I could not easily replicate 1000 different tweet objects that also happen to be realistic, I could however replicate 1000 very similar tweet objects, which is what

I did here. This ensured that while I might not necessarily have a completely accurate picture of what the parsing speed of the parser would be in reality, as the very similar objects might by chance be very quick or very slow compared to other objects, I could perform a good comparison between the different hosting systems on the same objects which is what I did here. In an ideal test I would have 1000 different real tweet objects and have each parser go through them and time that for speed.

Another important thing to note about my results is that the Dockerised parser on Rackspace and non-dockerised parsers were on different servers, one of which has 8GB of RAM and 8CPUs and the other 2GB of RAM and 2 CPUs, while the process itself does not use 2GB of RAM and is only single threaded this may still impact the results if other processes are running and need to use the CPU while the process is running or the individual CPUs are better on either virtual machine.

5.4.2 Results of Throughput Testing

	PHP 5			PHP 7	
	Rackspace Normal Parser	Rackspace Dockerised Parser	AWS Dockerised Parser	Rackspace Dockerised Parser	AWS Dockerised Parser
Start	14:34:11	15:28:50	15:11:10	16:12:44	16:24:07
End	14:38:09	15:33:08	15:14:02	16:14:30	16:25:46
Total	238	258	172	106	99
Start	14:59:29	15:37:11	15:16:03	16:19:29	16:29:25
End	15:03:27	15:41:50	15:19:05	16:21:00	16:31:03
Total	238	279	182	91	98
Start	15:05:57	15:43:40	15:20:14	16:35:54	16:56:32
End	15:09:57	15:48:16	15:23:08	16:37:34	16:58:15
Total	240	276	174	100	103
Average	238.67	271	176	99	100
Standard Dev	1.15	11.36	5.29	7.55	2.65

Figure 9: Speed of Parsing 1000 Tweets (Seconds)

From the results you can see in Figure 9 it shows some interesting conclusions such as running a Dockerised parser is slightly slower than a non dockerised parser on Rackspace it is only slower by around 30 seconds which may partly be down to the less powerful server but you most likely would expect a slight overhead from running it on Docker. Docker on AWS is significantly faster by around 60 seconds which suggests that the infrastructure on Amazon is using significantly more powerful CPUs than Rackspace are. Another reason for the increase may be that the more minimal instances running the docker cluster on AWS are causing it take up less CPU cycles when it is not executing the parsing process compared to Ubuntu on Rackspace. In terms of PHP 7 compared to PHP 5 it is significantly faster, with the time taken to run being over a minute less than PHP 5 on AWS and nearly 3 minutes on Rackspace. The most interesting thing about the results from this being that both Rackspace and AWS have very similar execution times on for Dockerised PHP 7 even though when comparing the exact same machines the only difference being PHP 5 the performance on AWS is several magnitudes better. If I had more time I would wish to do further tests to see if these trends continue or if there just happened to be some anomalous work on the server which was causing the slow down.

5.5 Cost Analysis

5.5.1 Justification of Cost Analysis

The final test I did was an analysis of the costs of running the servers over a month long period compared to the existing Rackspace system. In order to do this I calculated the costs of the AWS and Rackspace servers per month and made use of the throughput testing results to work out what the relative cost would be to get the same parsing processing power, I then projected these results onto 2 other potential instance types, c4 and m4 in order to see what the potential costs could be for running this system if the t2 instance regularly exceeds all the allotted credits. In order to simplify the numbers for the equivalent number of parsers I chose to use 0.4 as the parser speed ratio rather than the more slightly accurate value of 0.42, simply as this makes it much more easy to understand how one Rackspace virtual machine on Rackspace equates to one instance on AWS. This value is calculated through the percentage of time taken to parse 1000 tweets on AWS compared to Rackspace using the values in figure 9:

$$\frac{100}{238} \times 100 = 42.02 \tag{1}$$

Another important thing to note is that while the Rackspace server has 8 CPUs for 8 parsing processes on AWS I placed twice the number of processes compared to the number of CPUs, this was due to AWS instances tend to have RAM GB values twice the number of CPUs, this could lead to a reduction in parsing speed on all 3 of the instances. However at the worst case scenario the performance would be twice as slow as the best case, increasing the costs by two times, in reality it is more likely to be between the two as while 2 processes competing for 1 CPU would over-utilise the resources the time taken to wait on the response of requests to the queuing system would be able to be utilised by the greater number of parsing processes.

5.5.2 Results of Cost Analysis

	Rackspace	AWS			
	General Purpose 8gb	t2.small	c4.large	m4.large	
Cost Per Hour	£0.24	$0.026	$0.119	$0.125	1 USD = 0.776698 GBP
Cost Per Hour in £	0.24	0.02	0.0924	0.0971	
Cost Per Month	172.8	14.4	66.528	69.912	
Number of Parsers per server	8	2	4	4	
Single Parser Speed Ratio	1	0.4	0.4	0.4	
Equivalent Number of Parsers	8	5	10	10	
Number of Parsers needed to equal 1 Rackspace Parser	1	1.6	0.8	0.8	
Relative Cost of Running per Month £	172.8	23.04	53.2224	55.9296	
Relative cost if all parsing credits are exceeded £		115.2			
Reduction in running costs worst case scenario £	172.8	230.4	106.4448	111.8592	
			= projected server parser ratio		

Figure 10: Chart showing the relative costs of running servers on AWS and rackspace

In figure 10 you can see the calculation of the relative cost of running the system on AWS as opposed to rackspace. All servers show an improvement over the the costs on rackspace however the very low cost of running the t2 instance is deceptive as if it exceeds the number of instance credits the CPU power will drop to 20% of what is shown here, which is reflected in the value of relative cost if all parsing credits are exceeded. Even in the very worst case scenario I am still able to demonstrate a reduction in costs with c4 and m4 instances. At best case scenario with a t2 instance not utilising all the credits the cost reduction would be around 87% however on worst case this could be an increase of 33% which would not be ideal. The other instances are always superior however, with c4 and m4 at worst showing a reduction of 39% and 35% respectively.

5.6 Comparison to Initial Design

In section 3.5 I include my final component and class diagrams and discuss some of the improvements I made to my design. In this section I wish to expand on more detail some things which aren't apparent from those finished system diagrams as they were things which came up during the implementation of the new design. The most notable change was that after the conversion to using a scaling manager to scale both the swarm and the service I initially only had one method which was responsible for scaling both the swarm and afterwards the service, this quickly became unmaintainable as the complexity of it increased. By splitting these up into the functions you can see in the class diagram in figure 5 it makes it much simpler and more object oriented.

5.7 Meeting the Aims

I have been able to demonstrate meeting my aims in terms of improving the speed of creation of parsers by reducing time taken to create parsers by up to 60%, the system I have created is also easier to implement patches and updates for as it is very conceivable that the creation of a docker image can be easily automated although I didn't have time to do this in this project. In regards to increasing the granularity of deployment I can definitely say that has been achieved as I have moved from scaling up in 8 parser instances on Rackspace to 2 parser instances on AWS, while the "sawtoothing" is still present and unlikely to be removed completely until the implementation of throughput measuring and predictive algorithm has been achieved the problem has been improved substantially. In terms of costs I cannot conclusively say that the costs have been reduced without running the server for a sustained period of time and documenting how much it cost. In terms of cost analysis however the current system could potentially at the very worst cost slightly more to run than the original system or at the best significantly less, however from the analysis I have done

I am able to demonstrate that if I haven't managed to reduce the costs with the current instance type I have the potential to through changing the instance type to a larger one which handles consistent CPU throughput better such as c4 or m4.

Overall I believe from this that my project was on the whole successful. While there are still a lot of ways I would like to expand this project further, especially in terms of predicting throughput and performing proper analysis on what the most cost effective instances to run would be, it achieved the main goal of creating a more responsive autoscaling system. In terms of the strengths of my project I believe the system I have created takes advantages of the best aspects of my chosen architecture such as fast container creation while helping alleviate the poorer parts of the architecture such as the disconnect between the instances themselves and the containers running on the instances making the system quite resilient. In terms of weaknesses I would liked to have been able to show a more substantial improvement in terms of the costs of running the system and I would also have liked to be able to implement scaling rules based off of throughput rather than just the existing threshold based ones. I would also have liked to have implemented some form of predictive algorithm to calculate trends in demand and scale up in advance but I did not have time, I have done the performance analysis required in order to potentially implement this in the future however.

I believe the choice of technologies was mostly appropriate, I feel if I had used Kubernetes instead of Docker Swarm there would have been the potential for some of the work I had to do in terms of scaling up in groups being done for me via pods rather than them have to be implemented by hard-coding in the number of processes I wished to have on each node myself. I believe node, my choice of programming language, was not the necessarily best choice for this project as it made setting timed loops rather difficult and the the event driven programming could be complicated to work with at times. With that in mind I think python could have been a better overall language to use for this project. Node did have a lot of benefits however, it fits in well with the software stack at Blurrt, where it is being used already and python is not. It also has a large number of modules available and easy to include in the project which made interacting with other systems very simple through modules such as the ssh and graphite ones I used.

6 Future Work

I did not have time to achieve everything I would have liked to have done during my project and there are several areas I would liked to have improved on in order of decreasing importance:

- Implement proper throughput calculation within the queue handler and implement scaling rules based off of tested parser throughput values, this would require doing more testing on throughput of parsers and potentially future work to update this rate each patch based on alterations to parser code.
- Find or create a way to perform throughput tests with an accurate number of messages per minute input in order to better demonstrate the workings of the system over time.
- Test other instance sizes to work out which would be the most cost effective for running the parsers and the correct number of parsers to number of CPUs ratio to run on them.
- Implement the cooldown periods myself within the queue size handler to avoid the inconsistent cooldown periods provided by Amazon and allow for longer downscaling cooldowns to implement a "fast upscale slow downscale" architecture.
- Implement a system which instead of immediately scaling down nodes from the swarm waits until the end of the hour long billing period is about to come to an end to do so and in the meantime scales only the service instead. Then implement a system which checks if there is a greater number of nodes available than there are containers to utilise the resources when it comes to scaling up and if so only scales up the service not the swarm itself. This will mean that if a the size of the swarm were to go down and then go up within an hour the scaling up will be essentially free for the remaining time until the hour billing period as the node is already paid for as opposed to creating a new one, this will lead to almost instantaneous scaling of the size of the service as the majority of scaling time is creating the instance and adding it to the swarm whereas creating containers on existing nodes takes less than 10 seconds.

- Research which AWS instance would be the most suitable to use as the nodes of the swarm. This would be based on analysis of how heavily the parsers utilise the CPU and the cost of each instance relative to the number of parsers that could run on it, accounting for the fact that we wish to keep the number of parsers per instance relatively low to avoid over scaling.
- Allow for altering the size of the manager nodes and remove the hard coded values which refer to there being 1 manager node from the code which was done for simplicity.
- Create a system to automate the patching process through use of a continuous integration system or some other method.
- Clear up some of the command line output display from running the queue-handler which can sometimes show poorly formatted.

7 Conclusions

This project's aim was to see if it was possible to implement a more responsive autoscaling system which was able to scale more rapidly to meet demand while also improving the granularity of the scaling to prevent sawtoothing as well as aiming to reduce the running costs. After selecting Docker for AWS as the underlying infrastructure I then designed and implemented a comprehensive scaling strategy capable of gracefully scaling the underlying node count and the size of the service running on it. Through my performance analysis I have managed to demonstrate significant improvements in parser deployment speed with 59% improvements compared to the original system. Further analysis of the processing performance yielded 58% improvement in terms of time taken to perform a parsing task. While I was not able to conclusively prove the system in its current state has cost benefits I have demonstrated the potential for significant cost savings by altering the choice of instance the swarm uses for its nodes with at the bare minimum a 39% reduction in running costs. This project makes a respectable contribution into the area of end user autoscaling deployment responsiveness an area which there has been little to no other research on so far and provides a reasonable starting point for anyone wishing to look further into the matter. Further continuation of this research could lead to improvements in scaling rules and the potential for more complex scaling management systems which take into account aspects such as the hosting platform's billing model to gain cost reductions and scaling speed improvements.

8 Reflections

Although during this project I was working on a system I already some understanding of, having worked at the company during my placement year last year, when I started working on this project I did not have a deep understanding of the intricacies of the existing system or the technologies I would be using to implement this project. As such may have overestimated the time I had to implement the system to reduce sawtoothing through some form of predictive algorithm. From looking at my initial plan and gantt chart I exceeded the amount of time required to implement the scaling system by 2 weeks and as such had very little time to implement new rules for the scaling system, this backs up my belief that I overestimated my ability to implement the system when there were so many different parts I needed to get working I had no prior knowledge of.

Another aspect I found challenging in the project was implementing meaningful performance analyses, were I to go back and do the project again I would plan out my performance analyses earlier on in the project instead of waiting until the end of the project as I believe it would make it easier to demonstrate significant improvements in the software. Had I planned out in advance the cost analysis I would have realised that while the t2 instance is great for simple development its properties make it very difficult to demonstrate a significant cost improvement.

In terms of the implementation of the software I was very satisfied with the eventual end product. While during my initial designs using lambda I was concerned that I wouldn't be able to get everything working as smoothly as I would have liked when I realised that I could implement a system without using lambda the design became much more streamlined. Aspects of the system which were previously quite complicated to orchestrate became much simpler when everything was directly controlled from one queue handling system.

In terms of the report I believe I would have benefited greatly from starting my write up aspect earlier, while I documented the work I did as I went through in notes it would have been much easier to write it up formally when it was fresh in my mind. This was a prioritisation issue on my part as I prioritised working on the implementation than the report itself as I believed that was the most important aspect. It would also have been beneficial me to get feedback from my supervisor on my report contents much earlier. Through writing the report I have learnt a lot about the process and motivation required to write very long texts, not just a report, which is something I had previously not done during my schooling or university.

Overall I think the project went well especially in regards to the finished product which while not having every feature I would have liked, has the work in place to expand upon and implement further extensions. The skills I have learnt in taking a concept and idea and implementing an entire functioning working system is something I will be able to take with me into future work not just in computer science but any other field I put my mind to.

A

update_blurrt_stats.sh

```
1  #!/bin/sh
2
3  HOSTNAME=$(hostname)
4  CPU_USAGE=$(top -b -n2 | grep "Cpu(s)"|tail -n 1 | awk '{print $2 + $4
       ↪ }')
5  TWEET_COLLECTION_RATE=$(curl -s -u user:password -H "Content-Type:
       ↪ application/json" localhost:15672/api/queues | python -c 'import
       ↪  json,sys;obj=json.load(sys.stdin);print obj[6]["message_stats
       ↪ "]["publish_details"]["rate"]*60')
6  TWEETS_NOT_PARSED=$(curl -s -u user:password -H "Content-Type:
       ↪ application/json" localhost:15672/api/queues | python -c 'import
       ↪  json,sys;obj=json.load(sys.stdin);print obj[6]["messages"]')
7
8  PARSERS=$(ps aux | grep parse_blurrts.php | grep gp_tweets | wc -l)
9  echo "blurrt.gnipparsers.${HOSTNAME} ${PARSERS} `date +%s`" | nc -q0
       ↪ serverstats.blurrt.co.uk 2003
10
11 echo "blurrt.${HOSTNAME}.dbspace_used ${DBSPACE_USED} `date +%s`" | nc
       ↪  -q0 serverstats.blurrt.co.uk 2003
12 echo "blurrt.${HOSTNAME}.cpu_usage ${CPU_USAGE} `date +%s`" | nc -q0
       ↪ serverstats.blurrt.co.uk 2003
13
14
15 echo "blurrt.${HOSTNAME}.tweet_gp_collection_rate ${
       ↪ TWEET_COLLECTION_RATE} `date +%s`" | nc -q0 serverstats.blurrt.
       ↪ co.uk 2003
16 echo "blurrt.${HOSTNAME}.unparsed_gp_tweets ${TWEETS_NOT_PARSED} `date
       ↪  +%s`" | nc -q0 serverstats.blurrt.co.uk 2003
17
18
19 if [ "$TWEETS_NOT_PARSED" -gt "2000" ]; then
20   curl -s https://lon.autoscale.api.rackspacecloud.com/v1.0/execute/1/7
       ↪ a88dc397443f1028e6d462ce629395f353a68ef497007750dc821757b48663a
       ↪ / -X 'POST' -d '{"auth":{"passwordCredentials":{"username":"
       ↪ user", "password":"password"}}}' -H "Content-Type:
       ↪ application/json" ;
21 fi
22
23 if [ "$TWEETS_NOT_PARSED" -lt "500" ]; then
24   curl -s https://lon.autoscale.api.rackspacecloud.com/v1.0/execute
       ↪ /1/314
       ↪ b19ce819173945d4beb4748746fbbed48a55e6d5a534ecdc61c3bed38eb07/
       ↪  -X 'POST' -d '{"auth":{"passwordCredentials":{"username":"
       ↪ user", "password":"password"}}}' -H "Content-Type:
       ↪ application/json" ;
25 fi
```

B

PostedObjectCache.php

```
1  <?php
2  abstract class PostedObjectCache
```

```
{
        var $defaultBatchCount = 100;

        abstract protected function addToQueue($data);
        abstract protected function getDataBatch();
        abstract protected function markDataBatchProcessed();
        abstract protected function markDataBatchUnprocessed();

        public function debugMessage($message)
        {
                print_r($message);
                print_r("\n");
        }
}
?>
```

C Full Code Listings

C.1 queue-handler.js

```
'use strict';

const AWS = require("aws-sdk");
const _graphite = require( 'graphite' );
const _client = _graphite.createClient( 'plaintext://serverstats.blurrt
    ↪ .co.uk:2003/' );

const scalingProcess = require( './scaling-manager.js' );

const sqs = new AWS.SQS({
  //sqsuser
  accessKeyId:'AKIAJEPTNRHRDNIY2KDA',
  secretAccessKey: 'password',
    region: 'eu-west-2'
});

const aws_queue_urls =
{
  dev: {
    gnip: 'https://sqs.eu-west-2.amazonaws.com/697738154271/
        ↪ GnipParsingQueueTest',
    parsed: 'https://sqs.eu-west-2.amazonaws.com/697738154271/
        ↪ ParsedQueueTest'
  },
};

var previousValues = [];
var previousValuesLength = 5;

function examineQueueSize(){
    return new Promise(function(resolve, reject){
    var params = {
          QueueUrl: aws_queue_urls.dev.gnip,
          AttributeNames:['All']
        };
```

```
33        sqs.getQueueAttributes( params, function(err, data) {
34        if (err){
35          console.log(err, err.stack); // an error occurred
36          reject( err );
37          return;
38        }// an error occurred
39        else{
40          console.log(data);
41          resolve(data);
42        }
43        });
44      });
45  }
46
47  function checkQueueSizeRules(data){
48    if(data.Attributes.ApproximateNumberOfMessages > 4000){
49      console.log("calling to scale with up 2");
50      scalingProcess._scaleProcess('up', 2);
51    }
52    else if(data.Attributes.ApproximateNumberOfMessages > 2000){
53      console.log("calling to scale with up");
54      scalingProcess._scaleProcess('up');
55    }
56    else if(data.Attributes.ApproximateNumberOfMessages < 500){
57      console.log("calling to scale with down");
58      scalingProcess._scaleProcess('down');
59    }
60    else{
61      var scaleUpCount = 0;
62      //for every value in the previous values array
63      for(var i = 1; i < previousValuesLength-1; i++){
64        //if it's larger than the previous one
65        if(previousValues[i]>previousValues[i-1]){
66          //increment the counter
67          scaleUpCount += 1;
68        }
69      }
70      //if they are all larger than the previous one scale up
71      if(scaleUpCount==previousValuesLength-1){
72        console.log("calling to scale with up");
73        scalingProcess._scaleProcess('up');
74      }
75      else{
76        console.log("we aren't doing anything");
77      }
78    }
79
80    addDataValueToPreviousArray(data.Attributes.
        ↪ ApproximateNumberOfMessages);
81    sendGraphiteGroupSizeRequest();
82    sendGraphiteRequest('blurrt.sqs.unparsed_messages.dev', data.
        ↪ Attributes.ApproximateNumberOfMessages);
83    return;
84  }
85
86  function addDataValueToPreviousArray(numberOfMessages){
87    if(previousValues.length>=previousValuesLength){
88      //5 is newest 0 is oldest
```

```
89      var newArray = []
90      for(var i = 1; i < previousValuesLength; i++){
91        newArray[i-1] = previousValues[i];
92      }
93      newArray[previousValuesLength-1] = numberOfMessages;
94      previousValues = newArray;
95      console.log(previousValues);
96    }
97    else{
98      previousValues.push(numberOfMessages);
99      console.log(previousValues);
100   }
101 }
102
103 function sendGraphiteRequest(metric,value){
104   var metrics = {};
105   metrics[metric] = value;
106   console.log(metrics);
107
108   _client.write( metrics, function( err )
109   {
110     if( err )
111       return console.error( 'Error sending stats to graphite:', err );
112
113     console.log( 'Stats sent to Graphite:', metrics );
114   });
115 }
116
117 function sendGraphiteGroupSizeRequest(){
118   //console.log("inside group size request");
119   scalingProcess.describeAutoScalingGroups({
120       AutoScalingGroupNames: [
121           "DockerDevelopment-NodeAsg-10ARXIFYRZIED"
122       ]
123   }).then(function(data){
124     //size of node group plus 1 manager node, shouldn't be hardcoded
125     sendGraphiteRequest('blurrt.docker.total_node_count.dev',data.
          ↪ AutoScalingGroups[0].Instances.length + 1)
126   });
127 }
128
129 function checkAndUpdateOperation(){
130   examineQueueSize()
131         .then(checkQueueSizeRules);
132 }
133
134 function start(){
135   var timer = setInterval(function(){
136     checkAndUpdateOperation();
137   }, 30000);
138 }
139 start();
```

C.2 scaling-manager.js

```
1 'use strict';
2
```

```
3  const AWS = require("aws-sdk");
4  const Client = require('ssh2').Client;
5  const fs = require('fs');
6
7  //the user which has access to lambda and autoscaling permissions
8  //lambda not needed any more
9  const autoscaling = new AWS.AutoScaling({
10   //lambdauser
11   accessKeyId:'AKIAIR6UXIYL3VEG5GOA',
12   secretAccessKey: 'password',
13     region: 'eu-west-2'
14 });
15
16 const ecr = new AWS.ECR({
17   //dockeruser
18   accessKeyId:'AKIAI5W32RTG55UOR2RA',
19   secretAccessKey: 'password',
20     region: 'eu-west-2'
21 });
22
23 const managerNodes = {
24   dev: '52.56.158.203',
25   prod: ''
26
27 };
28
29 var GroupNames = {
30     AutoScalingGroupNames: [
31         "DockerDevelopment-NodeAsg-10ARXIFYRZIED"
32     ]
33 };
34
35 var mode = "";
36 var magnitude = 1;
37
38 function describeAutoScalingGroups(groupNames){
39   return new Promise(function( resolve, reject )
40   {
41     autoscaling.describeAutoScalingGroups(groupNames, function( err,
       ↪ groups )
42     {
43       if( err ){
44         console.log(err, err.stack);
45         reject( err );
46         return;
47       }
48
49       resolve(groups);
50     });
51   });
52 }
53
54 function updateDesiredCapacity(input) {
55
56     var params = {
57         AutoScalingGroupName: "DockerDevelopment-NodeAsg-10ARXIFYRZIED
           ↪ ",
58         HonorCooldown: true
```

```
59      };
60
61    if(mode=='up'){
62      params.DesiredCapacity = input.AutoScalingGroups[0].DesiredCapacity
            ↪  + magnitude;
63    }
64    else{
65      params.DesiredCapacity = input.AutoScalingGroups[0].DesiredCapacity
            ↪  - magnitude;
66    }
67
68    return new Promise(function(resolve, reject){
69      autoscaling.setDesiredCapacity(params,function(err, data) {
70        if (err){
71          if(err.code == 'ScalingActivityInProgress'){
72            console.log("Already scaling");
73          }
74          else if(err.code == 'ValidationError' && err.message == 'New
                ↪ SetDesiredCapacity value -1 is negative.'){
75            //do nothing if it's below desired capacity
76          }
77          else{
78            console.log(err, err.stack); // an error occurred
79          }
80
81          return;
82        }
83        console.log('successfully updated');
84        resolve(params);
85      });
86    });
87
88  }
89
90  function waitForScalingToComplete(params){
91    return new Promise(function(resolve, reject){
92      //if scaling down don't bother waiting scale down in advance and
            ↪ let the nodes be rebalanced by docker
93      if(mode=='up'){
94        var conn = new Client();
95        conn.on('ready', function() {
96          var timer = setInterval(function(params){
97            //view current node status
98            conn.exec('docker node ls', function(err, stream) {
99              if (err) throw err;
100             stream.on('close', function(code, signal) {
101               //ending the connection here was causing issues, moved
                      ↪ into on data
102               //conn.end();
103
104             }).on('data', function(data) {
105               console.log('STDOUT: ' + data);
106               data = data.toString().split("\n");
107               var dataCount = 0
108               //check number of ready nodes
109               for(var i = 0; i < data.length ;i++){
110                 if(data[i].includes(' Ready ')){
111                   dataCount += 1;
```

```
112                 }
113               }
114               console.log(dataCount);
115               if(data.length - 2 == params.managerNodes+params.
                      ↪ workerNodes){
116                 console.log("Swarm has scaled");
117                 conn.end();
118                 clearInterval(timer);
119                 resolve(params);
120               }
121
122             }).stderr.on('data', function(data) {
123               console.log('STDERR: ' + data);
124               reject(params);
125             });
126           });
127
128         }, 20000, params);
129       }).connect({
130               host: managerNodes.dev,
131               port: 22,
132               username: 'docker',
133               privateKey: fs.readFileSync('/home/greg/Parsers2/Node/
                      ↪ keys/id_rsa')
134             });
135     }
136     else{
137       resolve(params);
138     }
139   });
140 }
141
142
143 function getAuthorizationToken(params){
144   //can't pass params as reference have to define new variable here
145   var input = params;
146   return new Promise(function(resolve, reject){
147     ecr.getAuthorizationToken({}, function(err, data){
148       if (err){
149         console.log(err, err.stack); // an error occurred
150         reject( err );
151         return;
152       }
153       else{
154
155         var authToken = Buffer.from(data.authorizationData[0].
                ↪ authorizationToken, 'base64').toString('ascii').
                ↪ substring(4);
156         var connectString ='docker login -u AWS -p '+ authToken +' -e
                ↪ none https://697738154271.dkr.ecr.eu-west-2.amazonaws.
                ↪ com';
157         var params = {
158           authString: connectString,
159           containersPerNode: 2,
160           workerNodes: input.DesiredCapacity,
161           managerNodes: 1
162         };
163
```

```
164           resolve(params);
165         }
166       });
167     });
168 }
169
170 function scaleSwarmService(params)
171 {
172   return new Promise(function(resolve, reject){
173     var swarmSize = params.containersPerNode*(params.managerNodes+
          ↪ params.workerNodes);
174
175     var conn = new Client();
176     conn.on('ready', function() {
177       conn.exec(params.authString, function(err, stream) {
178           if (err) throw err;
179           stream.on('close', function(code, signal) {
180               console.log('Stream :: close :: code: ' + code + ',
                    ↪ signal: ' + signal);
181           conn.exec('docker service scale parsingprocess='+swarmSize,
                ↪ function(err, stream) {
182             if (err) throw err;
183             stream.on('close', function(code, signal) {
184
185               conn.end();
186
187             }).on('data', function(data) {
188               console.log('STDOUT: ' + data);
189               conn.end();
190               resolve(data);
191
192             }).stderr.on('data', function(data) {
193               console.log('STDERR: ' + data);
194               conn.end();
195               reject(data);
196
197             });
198           });
199
200           }).on('data', function(data) {
201               console.log('STDOUT: ' + data);
202
203           }).stderr.on('data', function(data) {
204               console.log('STDERR: ' + data);
205
206           });
207       });
208     }).connect({
209       host: managerNodes.dev,
210       port: 22,
211       username: 'docker',
212       //change this to the location of your rsa key
213       privateKey: fs.readFileSync('/home/greg/Parsers2/Node/keys/id_rsa
            ↪ ')
214     });
215   });
216 }
217
```

```
218 function _scaleProcess(modeInput, magnitudeInput){
219   //by default scale up if no parameters given
220   if(modeInput==null){
221     mode = 'up'
222   }
223   //throw error if bad parameter passed
224   else if(modeInput!='up'&&modeInput!='down'){
225     throw new Error('Unknown parameter');
226   }
227   else{
228     mode = modeInput
229   }
230
231   if(magnitudeInput==null){
232     magnitude = 1;
233   }
234   //throw error if bad parameter passed
235   else if(isNaN(magnitudeInput)){
236     throw new Error('magnitude value is not a number');
237   }
238   else{
239     magnitude = magnitudeInput
240   }
241
242   console.log('Scaling '+mode);
243   describeAutoScalingGroups(GroupNames)
244               .then(updateDesiredCapacity)
245               .then(getAuthorizationToken)
246               .then(waitForScalingToComplete)
247               .then(scaleSwarmService);
248
249 }
250
251 module.exports = {
252   _scaleProcess: _scaleProcess,
253   describeAutoScalingGroups: describeAutoScalingGroups
254 }
```

C.3 package.json

```
1  {
2    "name": "DockerScalingManager",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "author": "Greg Nichols",
7    "license": "ISC",
8    "dependencies": {
9      "aws-sdk": "^2.36.0",
10     "ssh2": "^0.5.4",
11     "graphite": "^0.0.7"
12   }
13 }
```

C.4 SqsObjectCache.php

```php
<?php

require_once('db_lib.php');
require_once('PostedObjectCache.php');
require_once('../vendor/autoload.php');

use Aws\Sqs\SqsClient;

class SqsObjectCache extends PostedObjectCache
{
    private $environment;
    public $sqsClient;
    private $outputQueue;
    private $inputQueue;


    var $currentMessageTags = [];

    private static $AWS_CREDENTIALS =
    [
        'dev' =>
        [
            'region' => 'eu-west-2',
            'version' => 'latest',
            'credentials' =>
            [
                //SQSuser credentials, a user specifically set up only for SQS
                'key' => 'AKIAJEPTNRHRDNIY2KDA',
                'secret' => 'password'
            ]
        ],
        'prod' =>
        [
        ]
    ];
    //these aren't complete, space in place to put production queues
    private static $AWS_QUEUE_URLS =
    [
        'dev' =>
        [
            'gnip' => 'https://sqs.eu-west-2.amazonaws.com/697738154271/GnipParsingQueueTest',
            'parsed' => 'https://sqs.eu-west-2.amazonaws.com/697738154271/ParsedQueueTest'
        ],
        'prod' =>
        [
        ]
    ];

    function __construct($postType="tweets",$targetDb="elastic", $dbWrapper=null,$environment=null,$ErrorLogger=null){
        $this->environment = $environment;
        $credentials = self::$AWS_CREDENTIALS[ $environment ];
        //this ideally shouldn't be hardcoded here
        $this->inputQueue = self::$AWS_QUEUE_URLS[$this->environment]['gnip'];
```

```
54          $this->outputQueue = self::$AWS_QUEUE_URLS[$this->environment
                ↪ ]['parsed'];
55
56          $this->sqsClient = SqsClient::factory($credentials);
57
58      }
59      //currently unused in blurrt parser, useful to put raw data onto
            ↪ queue directly
60      //currently hardcoded to place onto the output queue
61      public function addToQueue($data){
62          $this->sqsClient->sendMessage(array(
63              'QueueUrl' => $this->outputQueue,
64              //'QueueUrl' => $this->inputQueue,
65              'MessageBody' => $data
66          ));
67      }
68
69      public function save($Blurrt){
70          $data = json_encode( $Blurrt->getPayload() );
71
72          $this->sqsClient->sendMessage(array(
73              'QueueUrl' => $this->outputQueue,
74              'MessageBody' => $data
75          ));
76      }
77
78      public function getDataBatch($batchSize=10, $strictBatchSize=false)
            ↪ {
79
80          $output = $this->sqsClient->receiveMessage(
81              array(
82                  'QueueUrl' => $this->inputQueue,
83                  'MaxNumberOfMessages' => $batchSize
84              ));
85
86          $outputArray = array();
87          foreach($output['Messages'] as $messages){
88              $this->currentMessageTags[] = ['Id'=> $messages['MessageId
                    ↪ '] ,'ReceiptHandle'=>$messages['ReceiptHandle']];
89              $outputArray[] = ['json' => $messages['Body']];
90          }
91          //print_r($this->currentMessageTags);
92
93          return $outputArray;
94      }
95      //Amazon SQS has a timeout system, if a message isn't deleted it
            ↪ can be pulled in again after the timeout
96      //this method therefore deletes the returned message from getting
            ↪ the data batch
97      //input = array of arrays contain id, key pairs
98      public function markDataBatchProcessed(){
99          if(count($this->currentMessageTags)>0){
100             $output = $this->sqsClient->deleteMessageBatch([
101                 'Entries' => $this->currentMessageTags,
102                 'QueueUrl' => $this->inputQueue
103             ]);
104         }
105         $this->currentMessageTags = [];
```

```
106     }
107     //SQS doesn't need marking as unprocessed because it does that
            ↪ automatically
108     //If the databatch isn't marked at processed it's requeued
109     public function markDataBatchUnProcessed(){
110         $this->currentMessageTags = [];
111     }
112
113     public function getAttribute($queue){
114         return $this->sqsClient->getQueueAttributes([
115             'QueueUrl' => self::$AWS_QUEUE_URLS[$this->environment][
                    ↪ $queue],
116             'AttributeNames' => ['All']
117         ])['Attributes'];
118     }
119
120 }
```

C.5 SqsObjectCacheAdd.php

Used to add tweet objects to the queue to be parsed for testing, the only difference between this and SqsObjectCache being lines 63, 64 and 124 onwards.

```
1  <?php
2
3  require_once('db_lib.php');
4  require_once('PostedObjectCache.php');
5  require_once('../vendor/autoload.php');
6
7  use Aws\Sqs\SqsClient;
8
9  class SqsObjectCache extends PostedObjectCache
10 {
11     private $environment;
12     public $sqsClient;
13     private $outputQueue;
14     private $inputQueue;
15
16
17     var $currentMessageTags = [];
18
19     private static $AWS_CREDENTIALS =
20     [
21         'dev' =>
22         [
23             'region' => 'eu-west-2',
24             'version' => 'latest',
25             'credentials' =>
26             [
27                 //SQSuser credentials, a user specifically set up only
                        ↪ for SQS
28                 'key' => 'AKIAJEPTNRHRDNIY2KDA',
29                 'secret' => 'password'
30             ]
31         ],
32         'prod' =>
33         [
34         ]
```

```
35      ];
36      //these aren't complete, space in place to put production queues
37      private static $AWS_QUEUE_URLS =
38      [
39          'dev' =>
40          [
41              'gnip' => 'https://sqs.eu-west-2.amazonaws.com
                    ↪ /697738154271/GnipParsingQueueTest',
42              'parsed' => 'https://sqs.eu-west-2.amazonaws.com
                    ↪ /697738154271/ParsedQueueTest'
43          ],
44          'prod' =>
45          [
46          ]
47      ];
48
49      function __construct($postType="tweets",$targetDb="elastic",
            ↪ $dbWrapper=null,$environment=null,$ErrorLogger=null){
50          $this->environment = $environment;
51          $credentials = self::$AWS_CREDENTIALS[ $environment ];
52          //this ideally shouldn't be hardcoded here
53          $this->inputQueue = self::$AWS_QUEUE_URLS[$this->environment]['
                ↪ gnip'];
54          $this->outputQueue = self::$AWS_QUEUE_URLS[$this->environment
                ↪ ]['parsed'];
55
56          $this->sqsClient = SqsClient::factory($credentials);
57
58      }
59      //currently unused in blurrt parser, useful to put raw data onto
            ↪ queue directly
60      //currently hardcoded to place onto the output queue
61      public function addToQueue($data){
62          $this->sqsClient->sendMessage(array(
63              //'QueueUrl' => $this->outputQueue,
64              'QueueUrl' => $this->inputQueue,
65              'MessageBody' => $data
66          ));
67      }
68
69      public function save($Blurrt){
70          $data = json_encode( $Blurrt->getPayload() );
71
72          $this->sqsClient->sendMessage(array(
73              'QueueUrl' => $this->outputQueue,
74              'MessageBody' => $data
75          ));
76      }
77
78      public function getDataBatch($batchSize=10, $strictBatchSize=false)
            ↪ {
79
80          $output = $this->sqsClient->receiveMessage(
81              array(
82                  'QueueUrl' => $this->inputQueue,
83                  'MaxNumberOfMessages' => $batchSize
84              ));
85
```

```
86          $outputArray = array();
87          foreach($output['Messages'] as $messages){
88              $this->currentMessageTags[] = ['Id'=> $messages['MessageId
                    ↪ '] ,'ReceiptHandle'=>$messages['ReceiptHandle']];
89              $outputArray[] = ['json' => $messages['Body']];
90          }
91          //print_r($this->currentMessageTags);
92
93          return $outputArray;
94      }
95      //Amazon SQS has a timeout system, if a message isn't deleted it
            ↪ can be pulled in again after the timeout
96      //this method therefore deletes the returned message from getting
            ↪ the data batch
97      //input = array of arrays contain id, key pairs
98      public function markDataBatchProcessed(){
99          if(count($this->currentMessageTags)>0){
100             $output = $this->sqsClient->deleteMessageBatch([
101                 'Entries' => $this->currentMessageTags,
102                 'QueueUrl' => $this->inputQueue
103             ]);
104         }
105         $this->currentMessageTags = [];
106     }
107     //SQS doesn't need marking as unprocessed because it does that
            ↪ automatically
108     //If the databatch isn't marked at processed it's requeued
109     public function markDataBatchUnProcessed(){
110         $this->currentMessageTags = [];
111     }
112
113     public function getAttribute($queue){
114         return $this->sqsClient->getQueueAttributes([
115             'QueueUrl' => self::$AWS_QUEUE_URLS[$this->environment][
                    ↪ $queue],
116             'AttributeNames' => ['All']
117         ])['Attributes'];
118     }
119
120 }
121
122
123
124 $asdf = new SqsObjectCache(null, null, null, 'dev');
125
126 for($i=0;$i<1000;$i++){
127     $asdf->addToQueue('{"created_at":"Thu Mar 23 17:08:09 +0000 2017","
            ↪ id":'.(String)(844958961069547500+$i).',"id_str
            ↪ ":"844958961069547521","text":"#MotoGP followed by #F1
            ↪ .....from now til '.date('c'). ' Sunday.....#DoNotDisturb
            ↪ ","source":"<a href=\"http://twitter.com\" rel=\"nofollow\">
            ↪ Twitter Web Client</a>","truncated":false,"
            ↪ in_reply_to_status_id":null,"in_reply_to_status_id_str":null
            ↪ ,"in_reply_to_user_id":null,"in_reply_to_user_id_str":null,"
            ↪ in_reply_to_screen_name":null,"user":{"id":16704454,"id_str
            ↪ ":"16704454","name":"bimblelass/Karen","screen_name":"
            ↪ bimblelass","location":"Leigh NW England","url":null,"
            ↪ description":"1 life lots of loves - my family and pet
```

```
        ↪ pooches; @MCFC (ST block137); #F1- #LH;#McLaren;#MotoGP-#VR/
        ↪ CC/NH/BS. Spiritual.","translator_type":"none","protected":
        ↪ false,"verified":false,"followers_count":1402,"friends_count
        ↪ ":1090,"listed_count":63,"favourites_count":13540,"
        ↪ statuses_count":19092,"created_at":"Sun Oct 12 04:23:24
        ↪ +0000 2008","utc_offset":0,"time_zone":"London","geo_enabled
        ↪ ":true,"lang":"en","contributors_enabled":false,"
        ↪ is_translator":false,"profile_background_color":"642D8B","
        ↪ profile_background_image_url":"http://pbs.twimg.com/
        ↪ profile_background_images/434651370835030016/RqQQO6Cv.jpeg
        ↪ ","profile_background_image_url_https":"https://pbs.twimg.
        ↪ com/profile_background_images/434651370835030016/RqQQO6Cv.
        ↪ jpeg","profile_background_tile":true,"profile_link_color
        ↪ ":"89C9FA","profile_sidebar_border_color":"FFFFFF","
        ↪ profile_sidebar_fill_color":"7AC3EE","profile_text_color":"3
        ↪ D1957","profile_use_background_image":true,"
        ↪ profile_image_url":"http://pbs.twimg.com/profile_images
        ↪ /802900218220003328/pamMwZi_normal.jpg","
        ↪ profile_image_url_https":"https://pbs.twimg.com/
        ↪ profile_images/802900218220003328/pamMwZi_normal.jpg","
        ↪ profile_banner_url":"https://pbs.twimg.com/profile_banners
        ↪ /16704454/1398280620","default_profile":false,"
        ↪ default_profile_image":false,"following":null,"
        ↪ follow_request_sent":null,"notifications":null},"geo":null,"
        ↪ coordinates":null,"place":null,"contributors":null,"
        ↪ is_quote_status":false,"quote_count":0,"reply_count":0,"
        ↪ retweet_count":0,"favorite_count":0,"entities":{"hashtags
        ↪ ":[{"text":"MotoGP","indices":[0,7]},{"text":"F1","indices
        ↪ ":[20,23]},{"text":"DoNotDisturb","indices":[55,68]}],"urls
        ↪ ":[],"user_mentions":[],"symbols":[]},"favorited":false,"
        ↪ retweeted":false,"filter_level":"low","lang":"en","
        ↪ timestamp_ms":"1490288889554","matching_rules":[{"tag
        ↪ ":"3333673","id":843762953522499600}]}');
}
print_r("done\n");
```

C.6 Dockerfile (PHP 5)

```
FROM ubuntu
LABEL Description="This image is used for the GNIP parsers for php5"
    ↪ Vendor="Blurrt" Version="0.1"

RUN apt-get update && apt-get install -y software-properties-common &&
    ↪ add-apt-repository ppa:ondrej/php && apt-get update && apt-get
    ↪ install -y --allow-unauthenticated php5.6 php5.6-mysql php5.6-
    ↪ bcmath php5.6-mbstring php5.6-curl php5.6-xml && mkdir -p /var/
    ↪ blurrt/miles/blurrt-phirehose/

ADD Codebase/blurrt-phirehose /var/blurrt/miles/blurrt-phirehose
WORKDIR /var/blurrt/miles/blurrt-phirehose/src
```

C.7 Dockerfile (PHP 7)

```
FROM ubuntu
LABEL Description="This image is used for the GNIP parsers for php7"
    ↪ Vendor="Blurrt" Version="0.1"
```

```
3
4 RUN apt-get update && apt-get install -y software-properties-common &&
      ↪ add-apt-repository ppa:ondrej/php && apt-get update && apt-get
      ↪ install -y --allow-unauthenticated php php-mysql php-bcmath php-
      ↪ mbstring php-curl php-xml && mkdir -p /var/blurrt/miles/blurrt-
      ↪ phirehose/
5
6 ADD Codebase/blurrt-phirehose /var/blurrt/miles/blurrt-phirehose
7 WORKDIR /var/blurrt/miles/blurrt-phirehose/src
```

References

[1] Alpine linux. URL https://www.alpinelinux.org/.

[2] Google app engine. URL https://cloud.google.com/appengine/.

[3] Container engine. URL https://cloud.google.com/container-engine/.

[4] Docker for aws setup & prerequisites, . URL https://docs.docker.com/docker-for-aws/#docker-community-edition-ce-for-aws.

[5] Docker hub, . URL https://hub.docker.com.

[6] Aws elastic beanstalk. URL https://aws.amazon.com/elasticbeanstalk/.

[7] Carina by rackspace, . URL https://app.getcarina.com/.

[8] Download composer, . URL https://getcomposer.org/download/.

[9] Heroku. URL https://www.heroku.com/.

[10] Swarm v. fleet v. kubernetes v. mesos, comparing different orchestration tools. URL https://www.oreilly.com/ideas/swarm-v-fleet-v-kubernetes-v-mesos.

[11] Container crashes with code 137 when given high load. URL https://github.com/moby/moby/issues/22211.

[12] Docker. Deploying a registry server, . URL https://docs.docker.com/registry/deploying/.

[13] Docker. What is a container, . URL https://www.docker.com/what-container.

[14] Docker. Swarm mode key concepts, . URL https://docs.docker.com/engine/swarm/key-concepts/.

[15] G. X. et al. Rapid virtual machine deployment approach on cloud platform. *Journal of Computational Information Systems*, 9(18):7381–7388, 2013. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.412.5103&rep=rep1&type=pdf.

[16] M. N. A. H. K. et al. Modeling the autoscaling operations in cloud with time series data. *Reliable Distributed Systems Workshop*, 34, 2015. URL http://ieeexplore.ieee.org/document/7371434/.

[17] S. I. et al. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155–162, 2011. URL http://ieeexplore.ieee.org/document/5935022/.

[18] Y. S. et al. Cloudflex: Seamless scaling of enterprise applications into the cloud. *INFOCOM, 2011 Proceedings IEEE*, 2011. URL http://ieeexplore.ieee.org/document/5935022/.

[19] Z. H. et al. Early observations on the performance of windows azure. *Department of Computer Science, University of Virginia*, 2010. URL https://www.cs.virginia.edu/~humphrey/papers/EarlyObservationsPerformanceWindowsAzure.pdf.

[20] R. Laurikainen. Improving the efficiency of deploying virtual machines in a cloud environment. 2012. URL http://lib.tkk.fi/Dipl/2012/urn100558.pdf.

[21] E. Mills. Improve speed and reduce overhead with containers in windows server 2016. URL https://channel9.msdn.com/Shows/OEMTV/OEM1754.