Cloud Computing Coursework Report

Table of Contents

Introduction

This report documents the process of the development of a cloud application stack in a local host virtual machine and Google Cloud using Docker containers that were grouped as a stack. This report specifically documents the tools and techniques that were used to deploy the stack, the structure and deployment process of the stack and lastly how the stack utilised resources and performed in the cloud and locally.

Key Tools and Techniques

Docker

Docker is a platform used for development and deployment of applications in what is called containers. Containers are Linux based instances of images (program executable packages) that can run on a Docker Engine without the need of a guest OS (reduces overhead).[1]

In this project, the application will be deployed as a stack. A Docker stack is a deployed set of services that can share dependencies and scaling, and in this project the services will be deployed using a swarm as well (cluster of machines running docker).[2] Communication with the Docker Engine in this project will be carried either through command line or using the Docker Python SDK.

Google Cloud Compute Engine

The google compute engine is a cloud service offered by Google which provides highly scalable servers based on fiber networked Google data centres[3]. In this project, google cloud's compute engine will be used to host to the project in Linux based servers that are optimised for containers. A two core CPU with 2GBs of RAM compute engine was selected for this project.

[1] Docker Documentation. (2018). *Get Started, Part 1: Orientation and setup*. [online] Available at: https://docs.docker.com/get-started/ [Accessed 8 Dec. 2018].
[2] Docker Documentation. (2018). *Get Started, Part 5: Stacks*. [online] Available at: https://docs.docker.com/get-started/part5/#introduction [Accessed 8 Dec. 2018].
[3] Google Cloud. (2018). *Compute Engine - IaaS | Compute Engine | Google Cloud*. [online] Available at: https://cloud.google.com/compute/ [Accessed 8 Dec. 2018].

Mongo and cAdvisor

Mongo is a NOSQL JSON based database with emphasis on flexible data structures and data iteration without a predefined schema [4]. Mongo will be used in this project to store the results benchmarks of stack's performance monitored by cAdvisor (a tool that runs as a container to record other containers usage statistics[5]). Communication with Mongo would be handled by Pymongo, Mongo's official python library[6].

GitHub and DockerHub

GitHub is a platform that can be used to host coding projects using git syntax[7], DockerHub is similar, but is used to host Docker images instead[8]. For the scope of this project, Github will host all project files, while DockerHub will be used to load the images (e.g. Mongo DB that will be used in the yaml/python SDK scripts.

[4] MongoDB. (2018). *NoSQL Databases Explained*. [online] Available at: https://www.mongodb.com/nosql-explained [Accessed 8 Dec. 2018].
[5] GitHub. (2018). *google/cadvisor*. [online] Available at: https://github.com/google/cadvisor [Accessed 8 Dec. 2018].
[6] Api.mongodb.com. (2018). *PyMongo 3.7.2 Documentation — PyMongo 3.7.2 documentation*. [online] Available at: https://api.mongodb.com/python/current/ [Accessed 8 Dec. 2018].
[7] Guides.github.com. (2018). *Hello World · GitHub Guides*. [online] Available at: https://guides.github.com/activities/hello-world/#what [Accessed 8 Dec. 2018].
[8] Docker.com. (2018). *Docker Hub*. [online] Available at: https://www.docker.com/products/docker-hub [Accessed 8 Dec. 2018].

Solution Structure and Tasks

This section of the report covers how the solution described in the introduction is implemented step by step using the key tools described in the previous section.

File Hierarchy

- docker-compose.yml: docker compose file used to deploy stack
- sh-scripts: directory used to store shell scripts
    - pull_run.sh: used to pull and run the docker prime check docker image using Linux command line
    - start_swarm.sh: Used to setup swarm stack using Linux shell
    - remove-swarm.sh: Used to remove the stack and swarm when it is initiated with start-swarm using Linux shell
- py-scripts: directory used to store python scripts
    - pull_run.py: used to pull and run the docker prime check docker image using Docker's Python SDK
    - start_swarm.py: Used to setup swarm stack using Docker's Python SDK
    - distr.py: used to carry analytical load on the server using poisson /normal distribution and stores the results to the mongo database
    - Plots.ipyn: ipython notebook file used to plot the benchmarks stored in the mongo database

Tasks and Design/Implementation Diagrams

Task 1: Pulling the Image

Files: pull_run.py (Python SDK) and pull-run.sh (shell)

To complete this task, the "nclcloudcomputing/javabenchmarkapp" image was pulled from docker hub and ran on the Docker Engine using both the Python SDK and the Docker commands available in the Linux shell. In the Python SDK, the image is pulled and ran using client commands (images.pull() and images.run()) after getting the client from the environment with the appropriate options (port 8080 -> 8080 and named prime). In the shell script, the pull and run commands are simply used with the same options (i.e. ports and name (prime)).

Task 2: Deploying the Services in the Stack

Files: start_swarm.py (Python SDK). start-swarm.sh and remove-swarm.sh (shell). docker-compose.yml (docker compose yaml, used with shell).

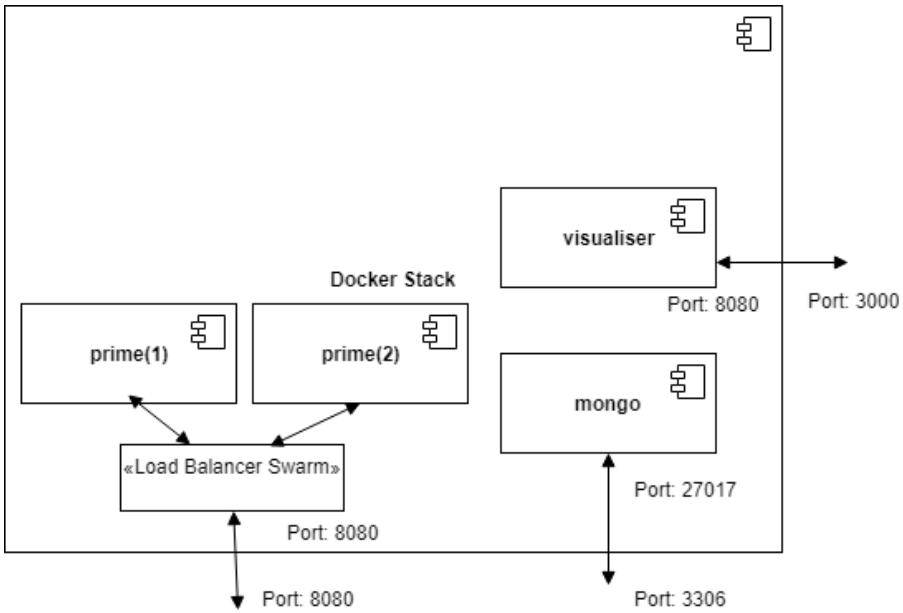
In the task, several applications (dockersample/visualiser, nclcloudcomputing/javabenchmarkapp and mongo) are ran in a single docker stack as services using their images in DockerHub. This done again using both the Python SDK and the Linux Bash Shell.

In the python SDK the swarm is initialised from the client using client.swarm.init() and the services are created using client.service.create() and joined to a network created using client.networks.create() with swarm mode. In shell script, a swarm is initialised using swarm init and the stack is deployed using a docker compose yaml with the images deployed as services on the same network using the "network" yaml property.

Nonetheless, both implementations share the following configuration settings for these containers:

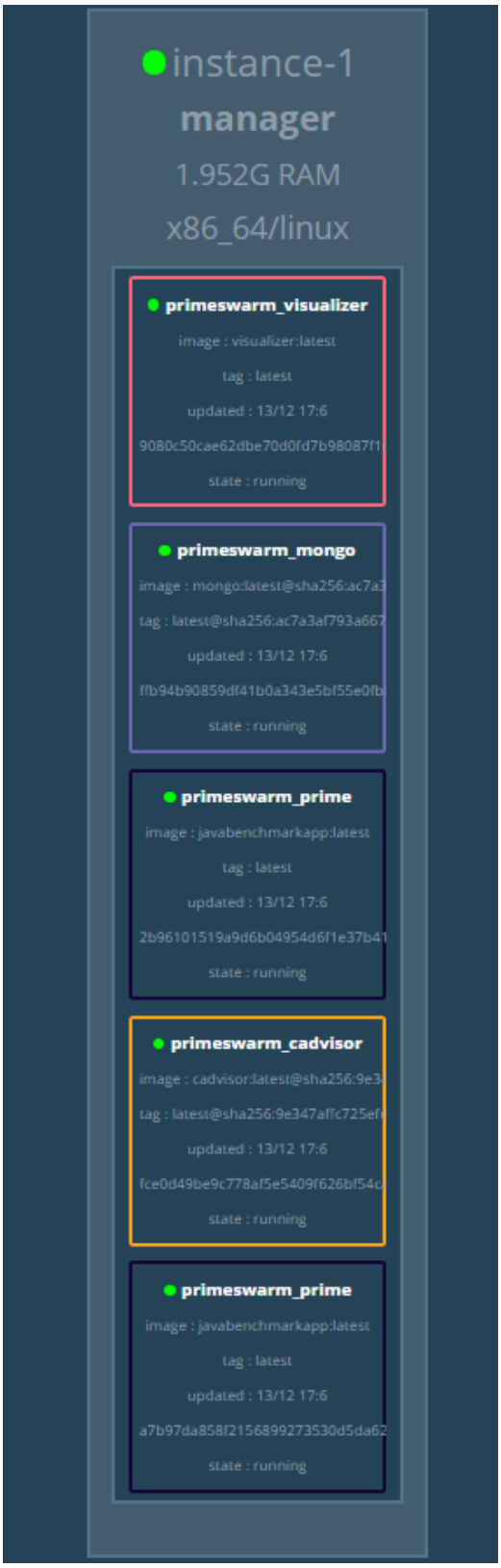
- nclcloudcomputing/javabenchmarkapp (name prime):
  - Port 8080: 8080
  - 2 replicas that are load balanced with the help of the swarm
- dockersample/visualiser (named visualizer):
  - Port 4000: 8080
  - Constraint to a node role of manager
  - Volumes according to[9]
- mongo (named mongo):
  - Port 3306: 27017
  - Volumes according to[10]

This results in the following setup:



[9] GitHub. (2018). *dockersamples/docker-swarm-visualizer*. [online] Available at: https://github.com/dockersamples/docker-swarm-visualizer [Accessed 14 Dec. 2018].
[10] Docker Documentation. (2018). *mongo*. [online] Available at: https://docs.docker.com/samples/library/mongo [Accessed 14 Dec. 2018].

And using the swarm visualiser application the setup looks like:



Task 3: Load Generation Scripts

Files: distr.py

The load generation scripts were created using python scripts which used key libraries such as Numpy and Request. Using Numpy the script generates a random values using a normal distribution (numpy.random.normal()) and poission (1 / numpy.random.exponential()) that is used to wait till the next request to the website using the request library.

To use the program, the following parameters need to be passed (parsed by argparse library):

- Method: 0 for normal or 1 for poisson
- URL: URL to load
- Mean: lambda/mean for normal/poisson distribution
- Sigma: Standard deviation for normal (ignored when using poisson)

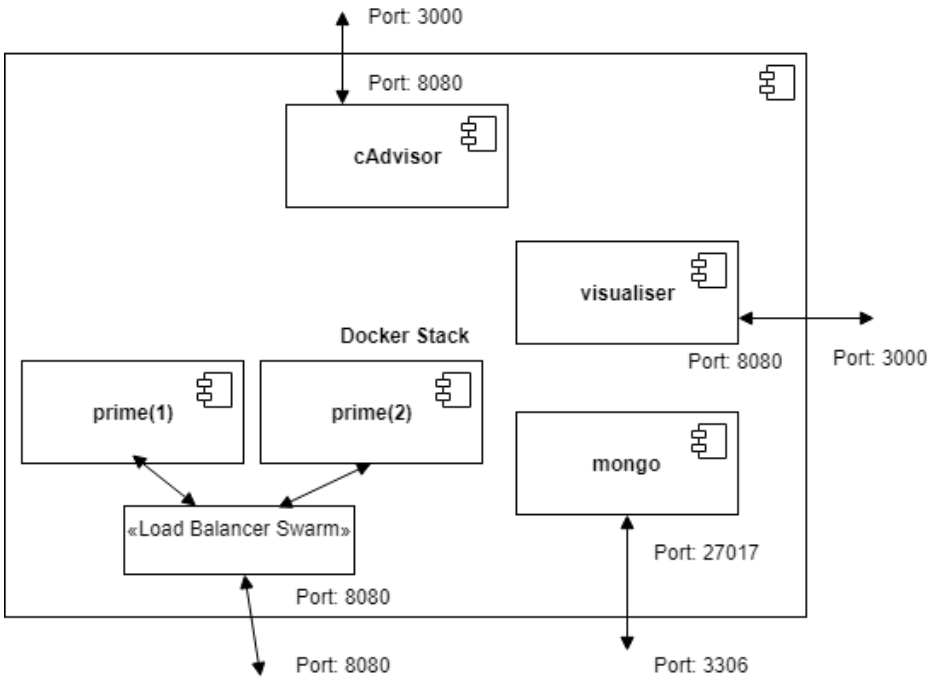
Task 4: Docker Monitoring (cAdvisor)

Files: docker-compose.yml

Like the other containers added in task 2, cAdvisor was added to the stack as a service created from a Docker Image in Docker Hub, this time however this was done using Linux Shell and yaml only. To add cAdvisor to the stack, a new entry was added to the yaml file with the following options:

- google/cadvisor (named cadvisor):
    - Port 3000:8080
    - Volumes according to [11]

This results in this updated setup for the stack:



[11] GitHub. (2018). *google/cadvisor*. [online] Available at: https://github.com/google/cadvisor [Accessed 8 Dec. 2018].

Example outputs

cAdvisor produced the following outputs memory and network after a run with a normal with mean 3000 and sigma 100 (60 seconds) for web container prime 1:

Task 5: Recording and Plotting Benchmarks (MongoDB)

Files: distr.py, Plot.ipyn

To record the benchmark results, the script from step 3 (distr.py) was extended to connect to the mongo database hosted in port 3306 using pymongo and connect to the cAdvisor container using its API. The script creates a database and two collections (one for prime calculation time returned by server and one for the cAdvisor stats). The script would then iterate the cAdvisor JSON stats and the recorded server responses from request.text to store both of them.

The plotting is handled by an ipython notebook (Plot.ipyn) which uses matplotlib and pandas libraries to plot various stats loaded from the two collection for the containers.

Task 6: Uploading the Application to Google Cloud

Files: All files specified in File Hierarchy copied to Google Cloud

Google Cloud's compute engine was used to create a virtual instance with 2 CPUs and 2GB of RAM (like the Virtual Machine localhost) using the London servers. The virtual instances also used a container optimised image for the boot disk (most recent stable version). To allow the services to run using the ports specified in task 4 and 2, the VPC firewall was configured using new rules for ports 3000, 3306, 4000 and 8080.

To deploy the stack, GitHub was used to copy the files from the previous tasks to the virtual instance in an SSH terminal. The stack is deployed as usual using the shell script start-swarm.sh which uses the docker compose yaml file.

The locations of the hosts in the script were also modified to include the locations the hosts on the Google Service instead of the localhost for the benchmarks and database communication.

Benchmarks

Requirement Recognition

The basic objective is to compare the performance and utilisation of the Docker stack from task 4 when deployed to the local host of a virtual machine compared to when it is deployed to the Google Cloud Compute Engine. Moreover, the load balancing behaviour used with the 2 web containers will be examined.

Service Feature Identification

The following variables will be used to measure the application utilisation and performance:

1. Network I/O usage
2. CPU usage
3. Memory usage
4. Computation time

Service Features and Metrics/Benchmarks listing and Selection

Service Feature	Metric	Benchmark
Network I/O usage	Network Bytes from cAdvisor (rx bytes)	Poission and Normally distributed http requests (Analytical)
CPU usage	CPU total usage from cAdvisor	Poission and Normally distributed http requests (Analytical)
Memory usage	Memory usage from cAdvisor	Poission and Normally distributed http requests (Analytical)
Computation time	Time to find prime check from the server response	Poission and Normally distributed http requests (Analytical)

Local Host Virtual Machine:

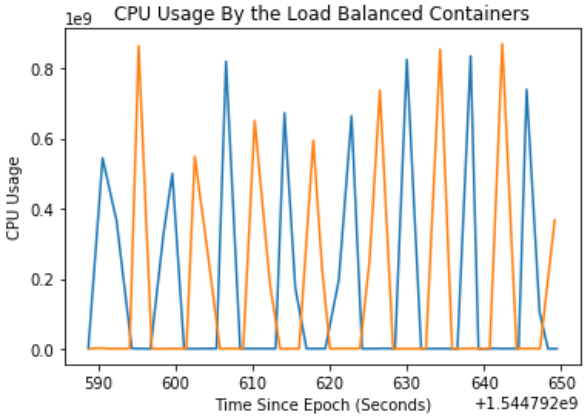
- OS image: Linux Mint 19 Cinnamon
- CPU: 2 cores allocated from a i7-7700HQ (Skylake based, 2.8-3.8GHz)
- RAM: 2GB
- Storage: 19.40GB SSD dynamically allocable to 61.7 GB
- Virtualisation/Cloud Platform: Virtual Box 5.2.22

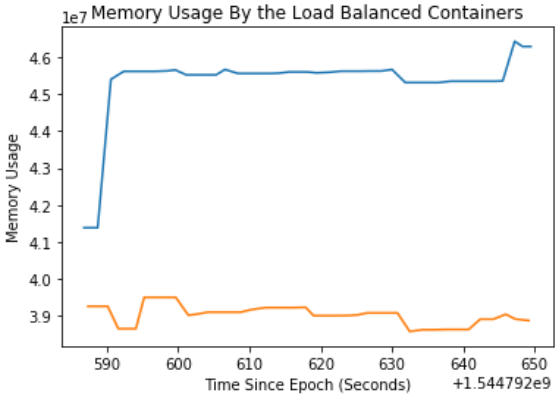
Google Cloud Compute Engine:

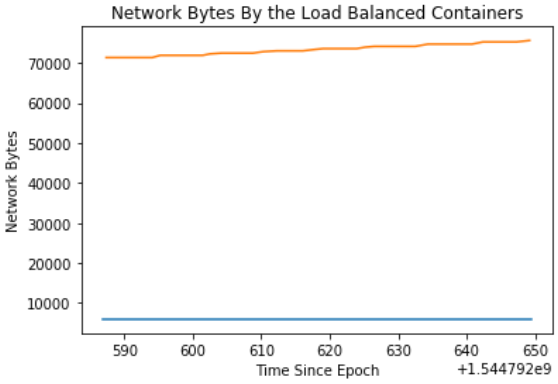
- OS image: Container-Optimized OS 70-11021.99.0 (stable)
- CPU: 2 vCPUs (Broadwell based)
- RAM: 2GB
- Storage: 10GB Standard persistent disk
- Virtualisation/Cloud Platform: Google Cloud Compute

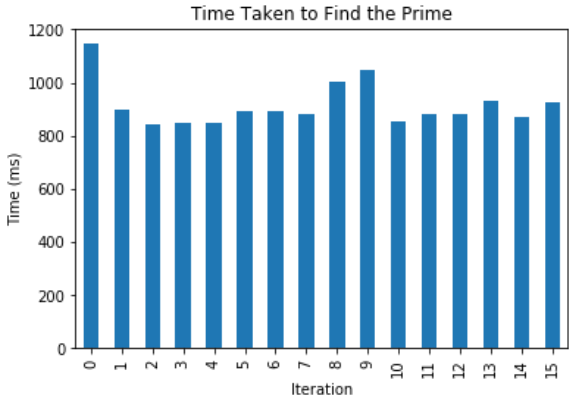
Experimental Design and Outcomes Local Host

Baseline Test (Normal mean 3000 sigma 1000 for 60 seconds)

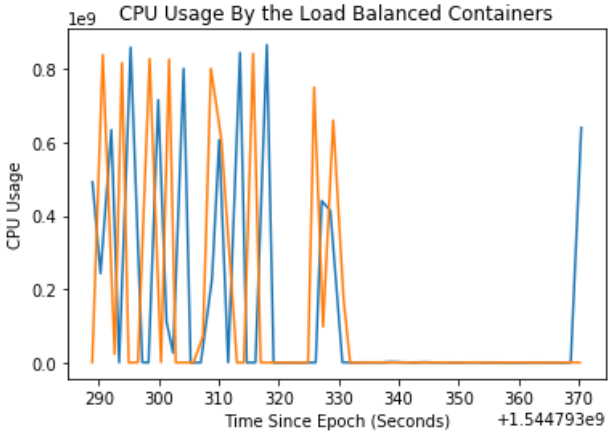


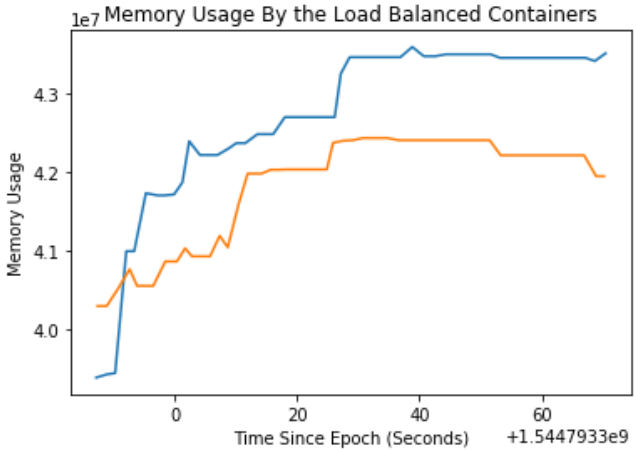


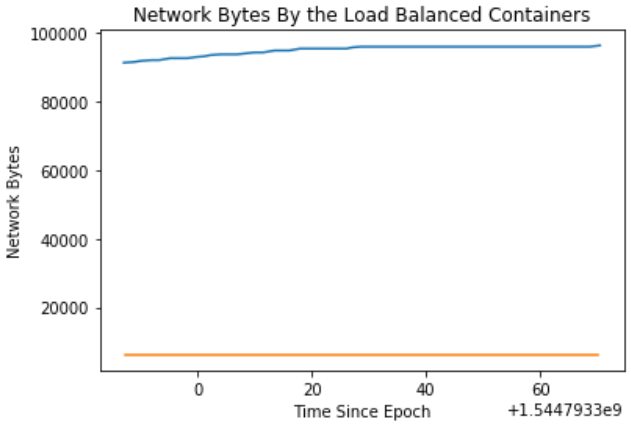


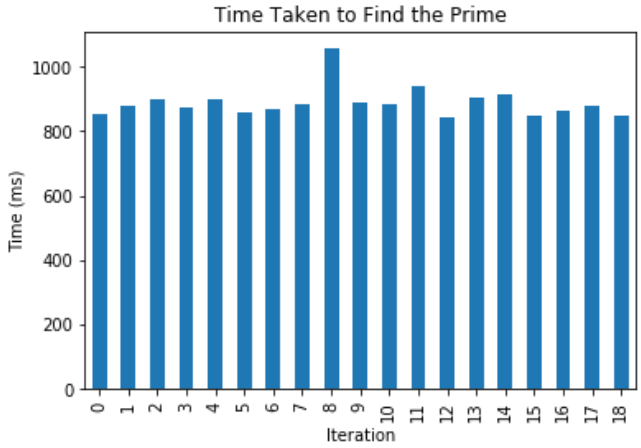


Load Test (Poission lambda 0.002 for 60 seconds)



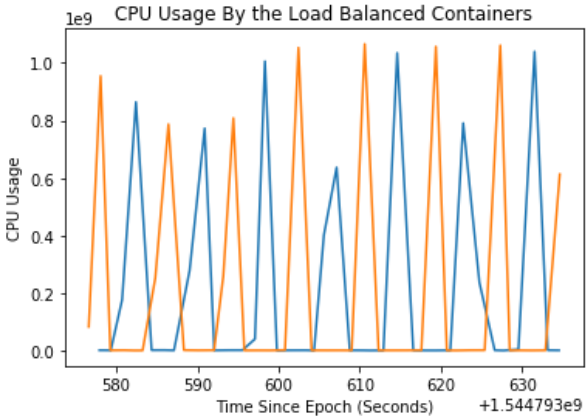


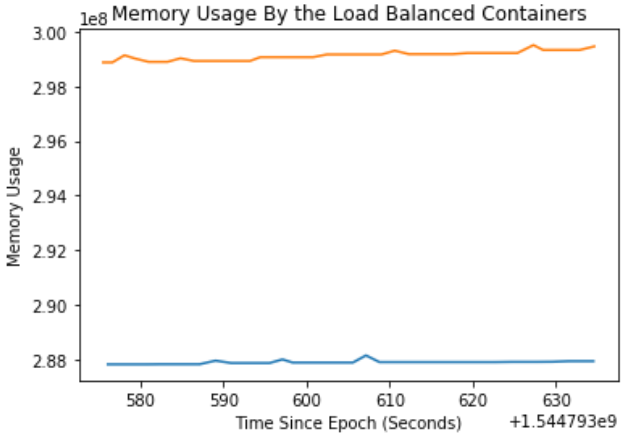


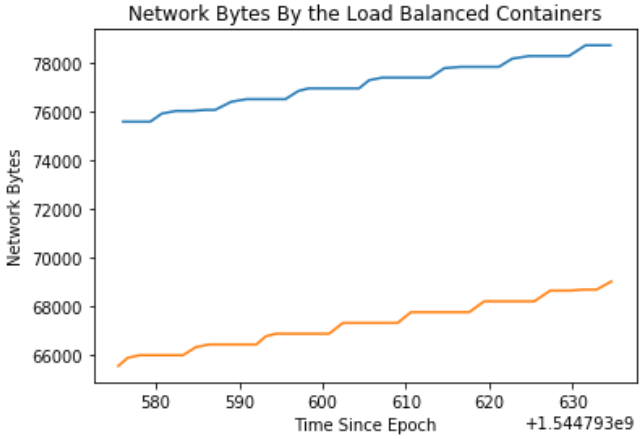


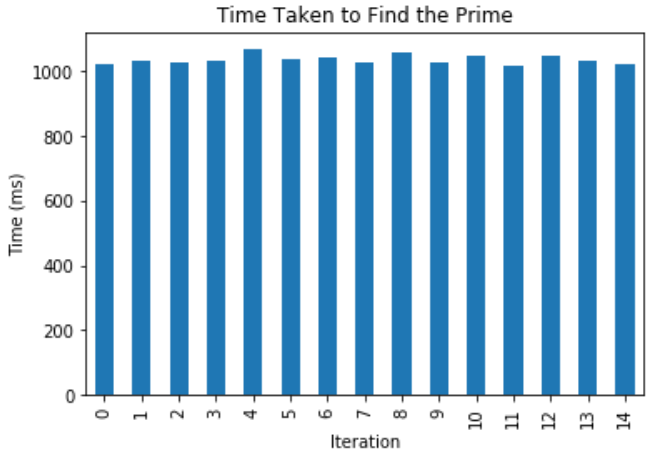
Experimental Design and Outcomes Google Cloud

Baseline Test (Normal mean 3000 sigma 1000 for 60 seconds)

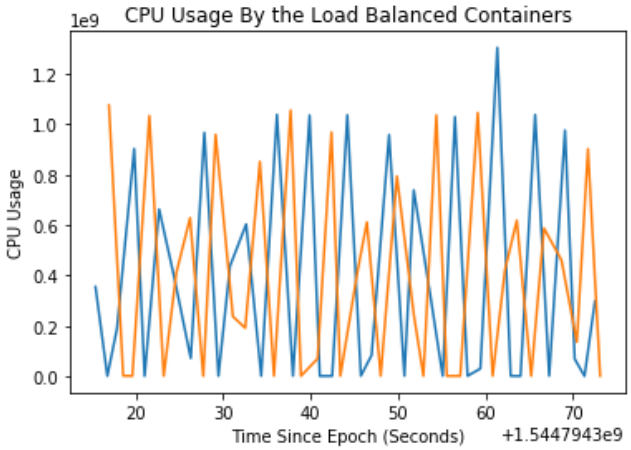


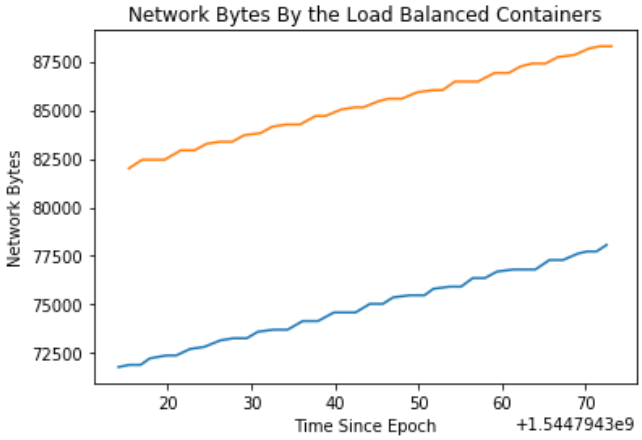

Memory Usage By the Load Balanced Containers
1e8
3.00
2.98
2.96
2.94
2.92
2.90
2.88
Memory Usage
580
590
600
610
620
630
Time Since Epoch (Seconds)
+1.544793e9

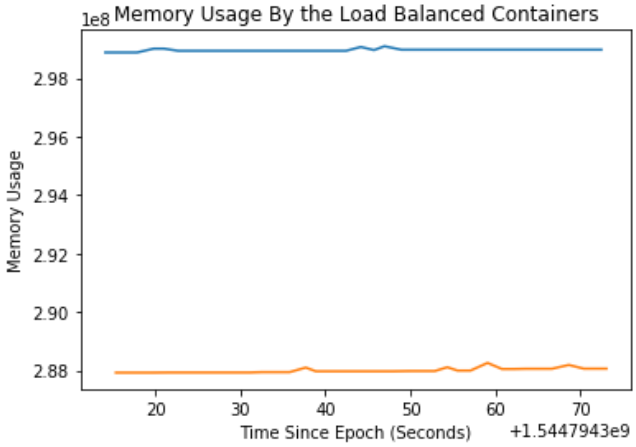

Network Bytes By the Load Balanced Containers
78000
76000
74000
72000
70000
68000
66000
Network Bytes
580
590
600
610
620
630
Time Since Epoch
+1.544793e9

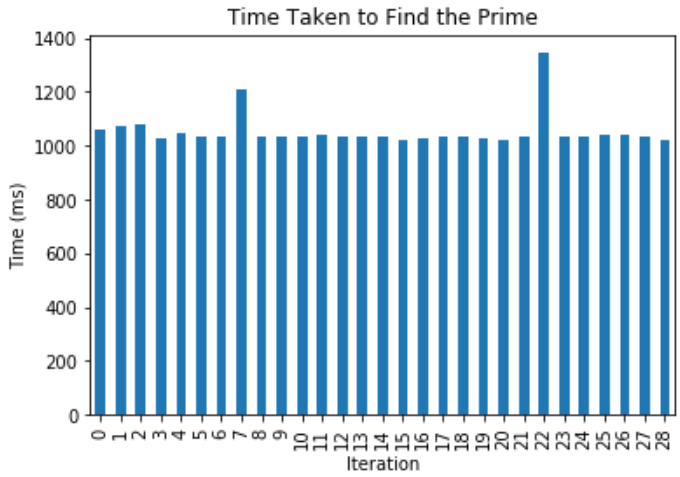

Time Taken to Find the Prime
1000
800
600
400
200
0
Time (ms)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Iteration

Load Test (Poission lambda 0.002 for 60 seconds)









Results Evaluation, Analysis and Conclusions

Looking at the results the following interesting key observations can be made:

- Generally, the swarm shares the processing load nearly equally across the two web containers according to their CPU usage. However, it seems most of the memory and networking functions are handed to one container.
- The cloud compute unit is interestingly slower than the local machine VM. This could be perhaps because the compute unit is using an older generation of processors (Broadwell vs Skylake), or perhaps due to load caused by other application running in the cloud.
- The cloud compute unit seems to be significantly more consistent at its results compared to the local machine (especially during the baseline test). This could be perhaps because the local machine has an operating system (Windows 10) running another virtual machine (Mint 19) with different background tasks.
- The network utilisation in the cloud seems to grow as time goes, perhaps this is because the google cloud platform is allocating more network resources due to the load.

However, some issues with the methodology might need addressing in the future:

- The load is randomly generated, hence the tests between the cloud and the local machine are not identical (especially with the poisson distribution)
    - This perhaps needs to be resolved by generating the random numbers before hand
        - However, the server responses might still vary effecting the next response since the program will wait (using threads might resolve this issue by making a new request regardless).
- The specification of the Broadwell based cloud virtual CPU is not clear compared to the local machine CPU (where the clocks and turbo are well known).
- The benchmark only ran for 60 seconds and might have not collect enough data and patterns.

Regardless, to sum up, it seems that the service does perform best in pure performance terms in the local machine, however, it is nowhere near stable. Moreover, the option to extend the number cores, change server locale, increase ram and other flexibility options are worth that comprise if the business using the application needs flexibility and room for growth.