POLYTECHNIQUE
MONTRÉAL

UNIVERSITÉ
D'INGÉNIERIE

Department of Software Engineering

LOG8415E

Advanced Concepts of Cloud Computing

Lab 1 Section 2

Submitted to Majdinasab, Vahid

Aymeric Labrie, Corentin Glaus, Maximiliano Falicoff, William Trépanier

Submitted on October 11th 2023

Flask Application Deployment Procedure

To simplify the deployment of our application, we have opted to utilize Terraform. Terraform is an Infrastructure as Code (IaC) tool that enables us to define cloud resources through configuration files. For this specific task, our chosen approach involves deploying the following components using Terraform: AWS EC2 Instances, Security Groups, Application Load Balancers for our clusters. By leveraging Terraform, we can efficiently manage and automate the provisioning of these essential resources in our AWS environment.

In order to deploy our Flask application, we provision our EC2 instances through a setup script that is specified in our Terraform configuration. This script does the following actions:

- Installs dependencies including python
- Creates a directory where our server will reside in /app and echoes the flask server code into our app.py file, we then do the same for our requirements file.
- We retrieve the instance id through a HTTP request to the metadata service that we export as an environment variable.
- We create a run script specifying the Gunicorn server to run the Flask application. The number of Gunicorn worker processes is set to twice the number of CPU cores ($(($(nproc) * 2))).
- We create a systemd service to run our application.
- We install our python dependencies.
- We enable and start our service.

Cluster setup using Application Load Balancer

Our Load Balancer is set up to listen on port 80 and has a fixed response on the / endpoint to return a 404. We then look at the path pattern of the request in order to forward it to the correct cluster defined in the EC2 configuration.

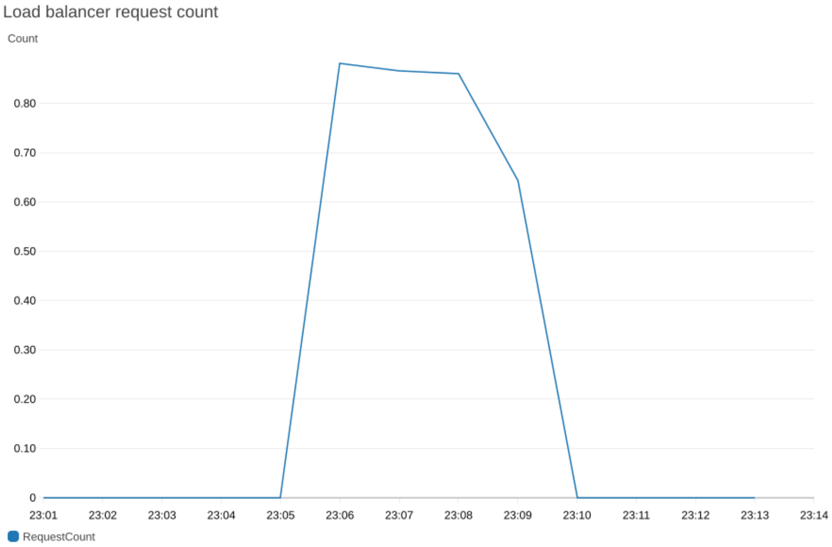
We have a setup with two clusters, one light of type t2.large with 4 instances and one heavy with 5 instances of m4.large, configured with 2 endpoints cluster1 and cluster2.

Results of your benchmark

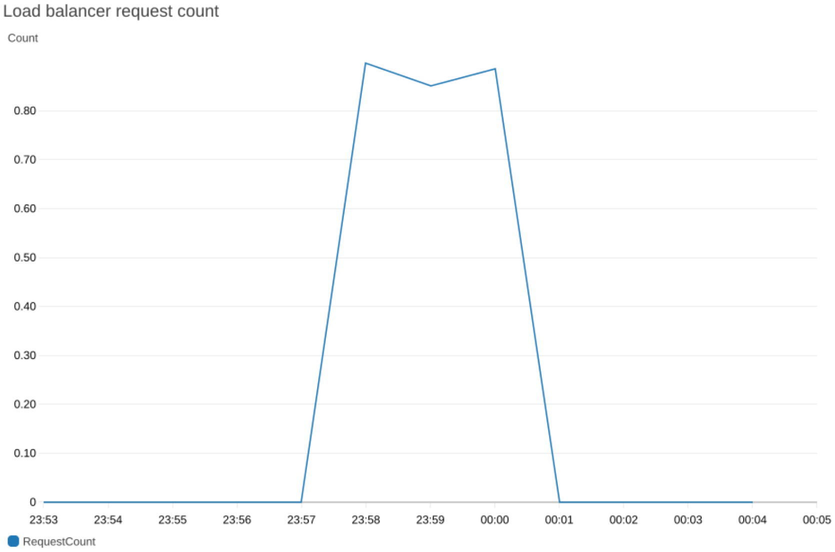
Only relevant metrics were chosen for the benchmarking results. The chosen metrics all show a correlation with the benchmarking. For the EC2 instances, CPU utilization and network output metrics are presented. For the application load balancer, only the request count is included. All these metrics are exported as images using boto3 and the GetMetricWidgetImage API. The following images were generated by running our benchmarking script.

A noteworthy challenge is that the assigned benchmarking procedure runs for about three minutes. However, the metrics granularity is only a minute with monitoring enabled. Hence, the results may show one or two spikes depending on the timing of the benchmark. The expected

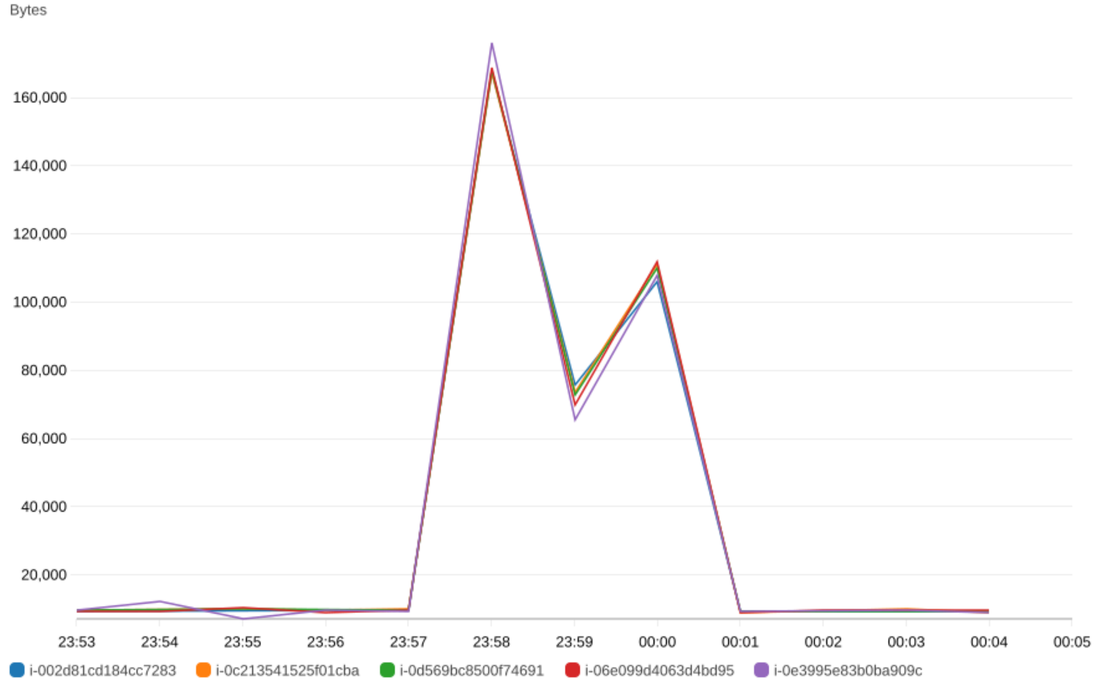
curve for the metrics includes two spikes. The following graphs show the request count of two different benchmark executions. This highlights how outputted results may vary.
Other execution request count:



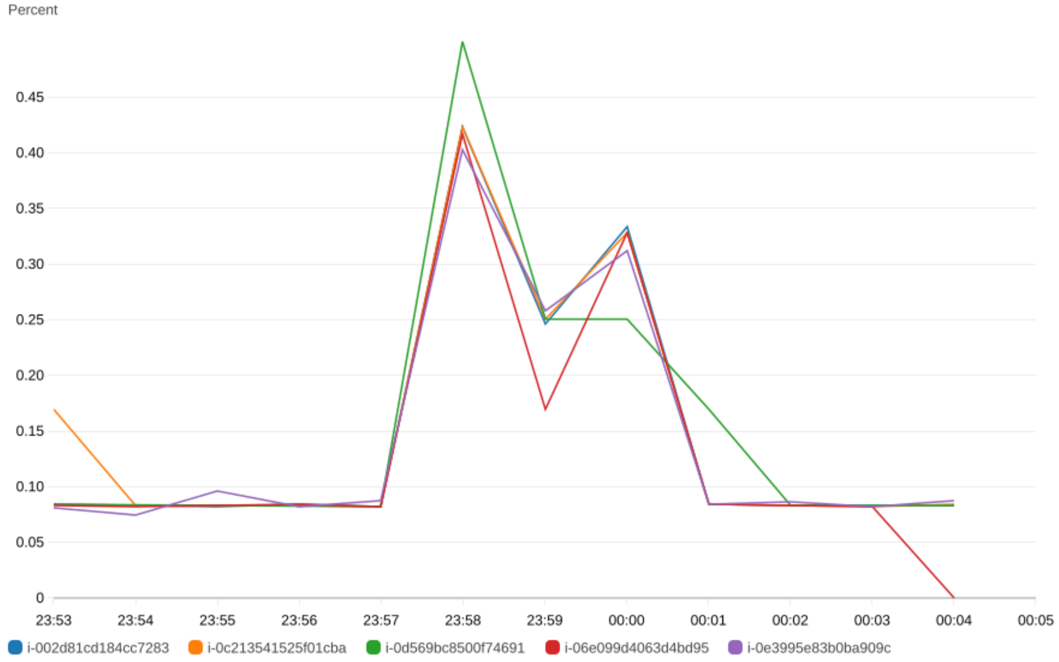
Request count, related to the other results below:



Here are the benchmarking results for the M4 cluster:

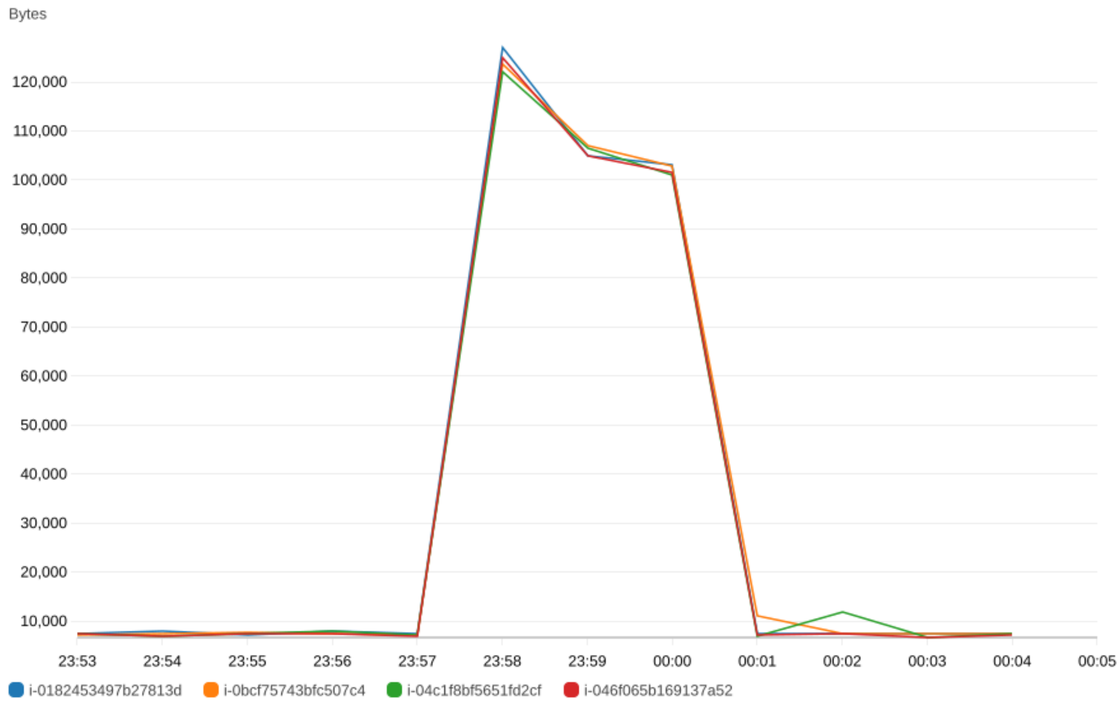


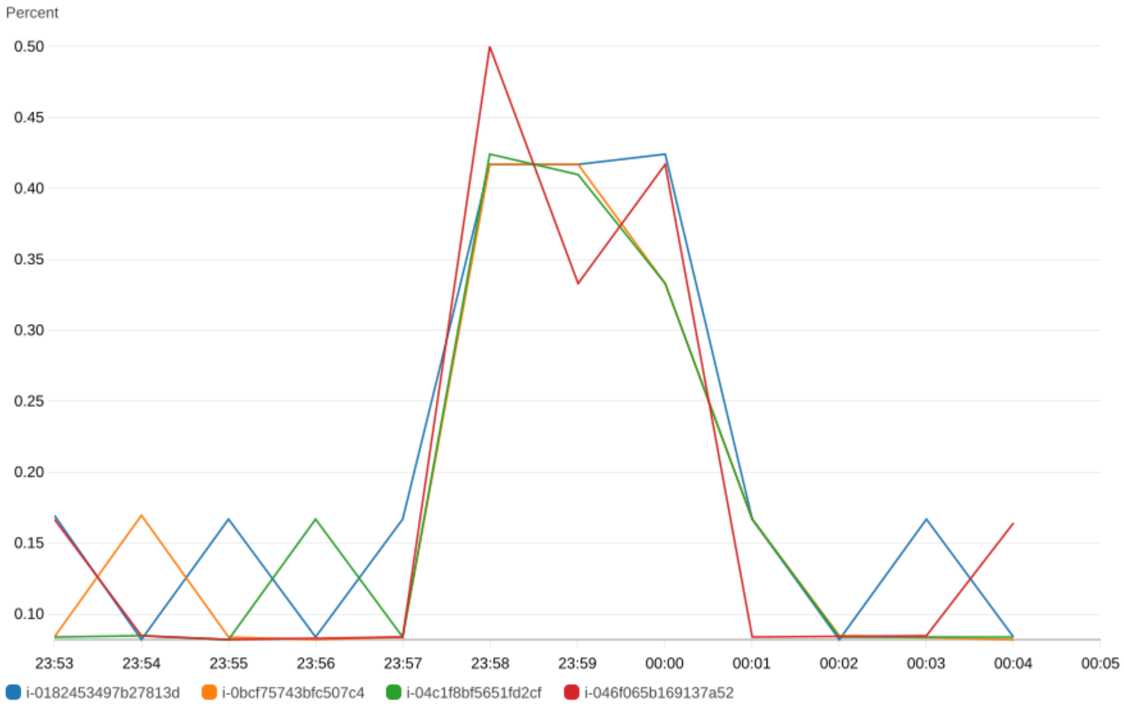
Network bytes out for cluster of 5 m4_large instances



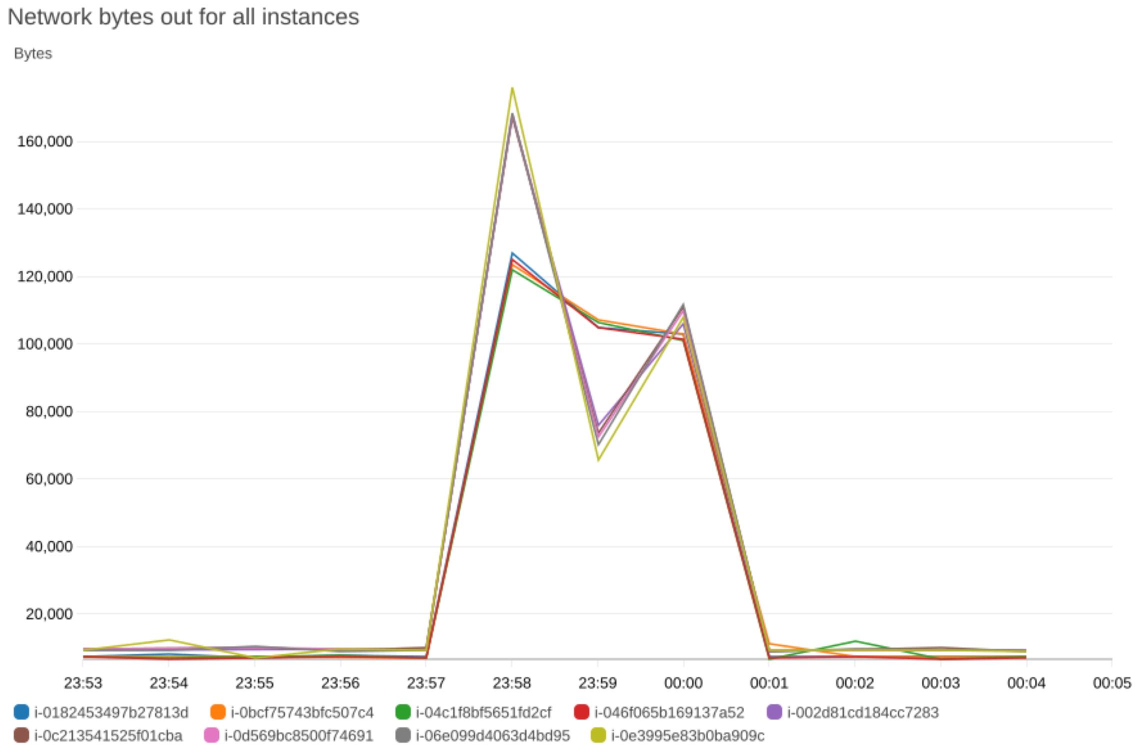
CPU utilization from 5 m4_large instances

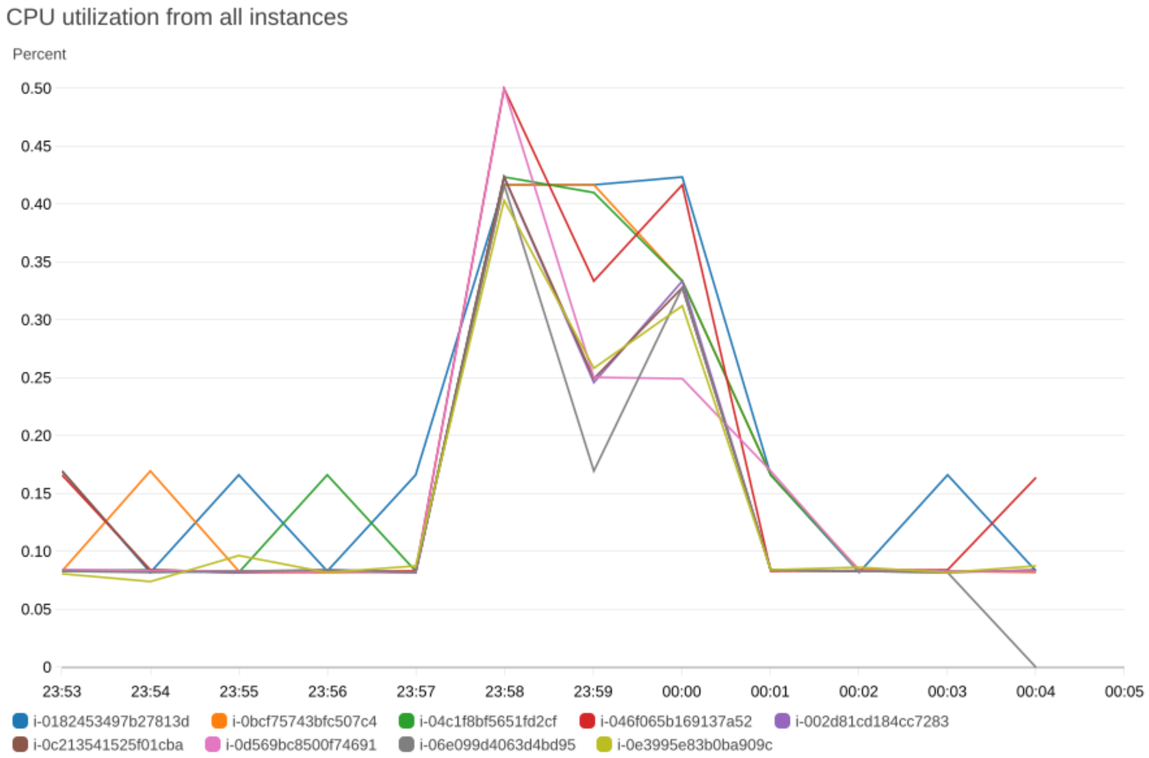
Here are the benchmarking results for the T2 cluster:


Network bytes out for cluster of 4 t2_large instances


CPU utilization from 4 t2_large instances

Here are the results of both clusters combined:


Network bytes out for all instances

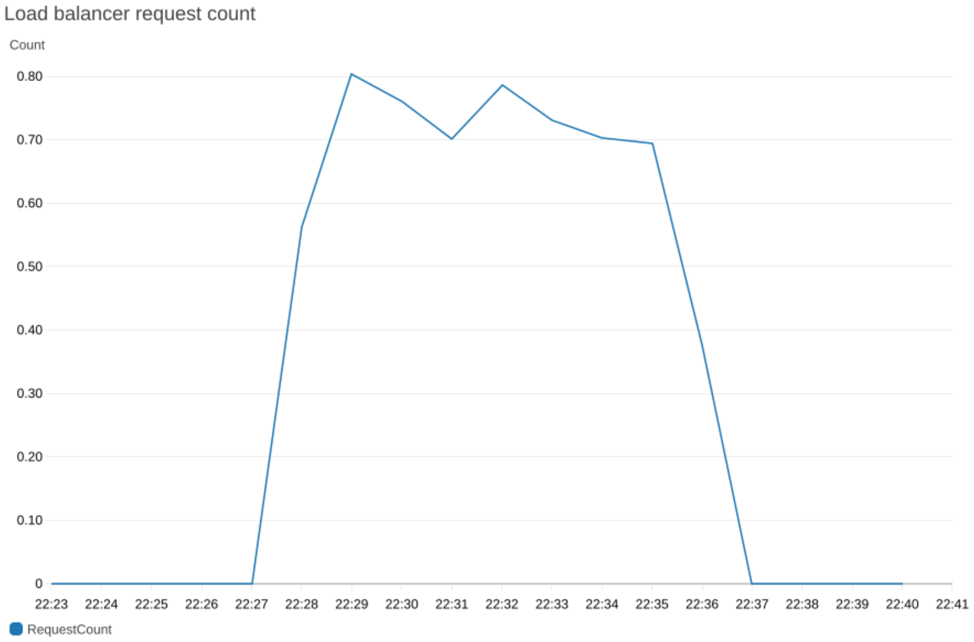

CPU utilization from all instances

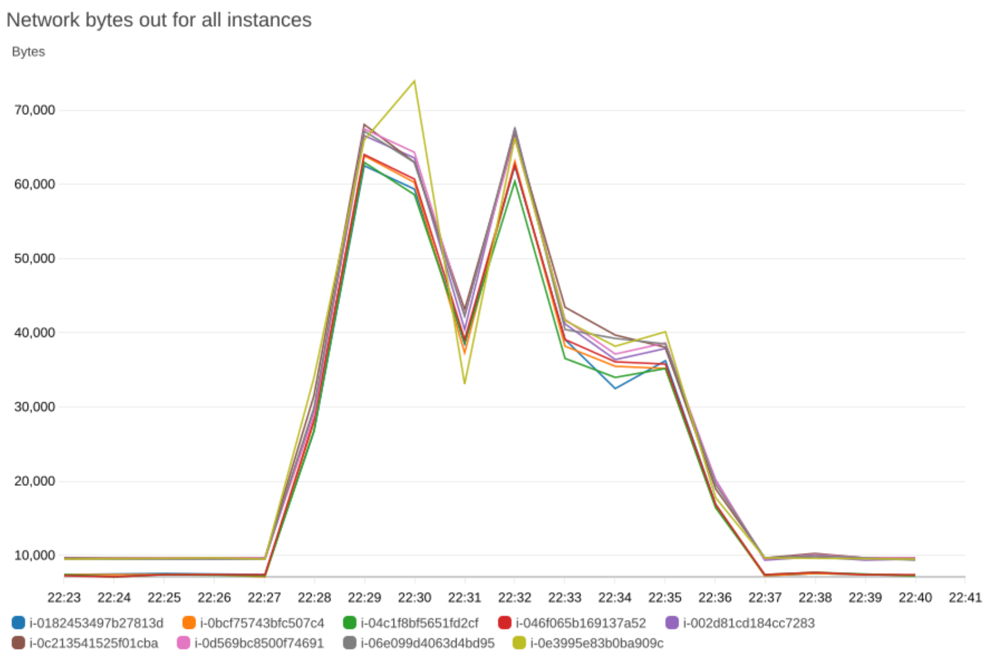
The first observation from the results is that the load balancer request count matches both the CPU utilization and network out metrics of the EC2 instances. All the instances also have the same curve for these metrics. The takeaway is that the load balancing is happening and is even across all instances within their cluster.

The second observation regards the differences between the M4 large and T2 large clusters. Although both clusters had similar curve shapes for their CPU utilization, the value of CPU utilization was different. To some extent, this may be related to each cluster not having the same number of balanced instances with the M4 cluster having 5 instances compared to 4 for the T2 cluster. Other than cost, no major differences were noted between the two clusters.
A third less worthy observation regards the network out being similar for both clusters. This is to be expected since the same benchmark was operated on both clusters and the benchmark had a fixed number of requests that didn't vary on the response size.

Additional experiment and results

In order to limit the differences between benchmarking executions due to a short execution time and poor metric sampling granularity. We added a 200ms delay to each request. This makes it easier to observe the true shape of the benchmark. It also proves that the shape is the same for the load balancer request count and the network out all instances for both clusters.





Instructions to run your code

1. Ensure the following packages are installed on the system:
    - Docker (and the service is enabled and running)
    - aws-cli
    - Terraform
    - pv (https://man7.org/linux/man-pages/man1/pv.1.html)

2. Copy your aws credentials to ~/.aws/credentials, they should be in the following format:
   [default]
   aws_access_key_id=_
   aws_secret_access_key=_
   aws_session_token=_
3. Make the script executable: chmod +x scripts.sh
4. Run the script ./scripts.sh

Note: since the metrics ingest process can take from 5 to 20 minutes (as we observed) we have taken this delay into account in our script and can take a while to run.

Repo Link

https://github.com/mfalicoff/LOG8415