**Intelligent real-time fleet management platform**

Abdelouahed Gherbi, Ahmed Bali, Varun Shiri

École de technologie supérieure – Montréal

Author Note

This project was supported by the Mitacs Globalink Research Internship program

21 August, 2021

**Content**

Abstract 3

1. Introduction 3

1.1 Project Statement 3

1.2 Purpose 3

2. Working and architecture 4

2.1 Web Application 4

2.1.1 Dashboard 4

2.1.2 Server 4

2.2 Deployment in cloud 5

2.2.1 Compute Engine 5

2.2.2 Google Kubernetes Engine 6

3. Design 7

3.1 Previous work 7

3.2 Explanation of code 8

3.2.1 Client side 8

3.2.2 Server side 10

4. Configuration in cloud 11

4.1 Managed Instance Group (MIG) 11

4.2 Google Kubernetes Engine (GKE) 11

5. Load testing results 12

5.1 CPU utilization 12

5.2 Response time 13

5.3 Scalability 17

6. Conclusion 19

[References 19](#_Toc80451405)

## 

## Abstract

The objective of this project is to contribute to the design, implementation and testing a platform software (Web), which is international (languages and standards), scalable, secure and extensible for a variety of business sizes. This platform allows the management of the rolling stock of such companies in real time. It allows in particular to follow, monitor and remotely control this equipment in real time. The advantages of such a platform would be improving the efficiency of fleet use and reducing the risk of driving.

Keywords: Fleet Management System, scalability

## 1. Introduction

#### 1.1 Project Statement

The project aims to develop a fleet management system (FMS) software. We have developed a Web application that allows a fleet manager to perform relevant actions and monitor his fleet. For the purpose of testing, we have simulated parameters such as the location, speed, gas level of the vehicle. The dashboard also contains useful visualizations to represent the data. Our project also focuses on testing the scalability and performance of the application under load. We have employed the most common and basic features of a fleet management system for the purpose of this prototype.

#### 1.2 Purpose

The objective of this project is to contribute to the design, implementation and testing of a platform software (Web) that helps companies follow, monitor and remotely control their fleet in real time. This should help provide managers better insight on their fleet, enabling them to make calculated decisions. Monitoring statistics like the speed of the vehicle can help them keep a tab on drivers’ behavior and promote a safer environment. Similarly, monitoring fuel consumption can help predict the needs of their vehicles. Finally, our aim is to evaluate and compare the performance of the application in the cloud using two popular virtualization technologies – Virtual Machines (VMs) and containers.

## 2. Working and architecture

#### 2.1 Web Application

The web application allows fleet managers to register and log in and view information pertaining to their fleet. The other main components are:

##### 2.1.1 Dashboard

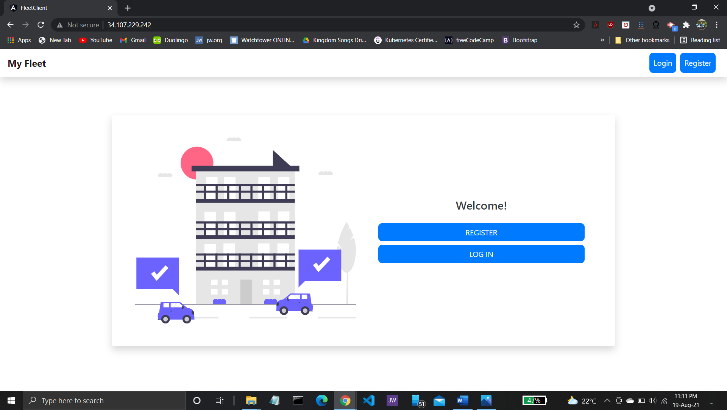
The dashboard provides visualizations for the data collected from the fleet. It consists of a live map that tracks and displays the location of all the vehicles in real time. The open source mapping tool OpenStreetMap[[1]](#footnote-1) was used for this purpose. The page also contains two plots – a bar chart representing the current gas level in the respective vehicles and a line chart displaying the speed of the selected vehicle in real time. In production, this data would be collected through the CAN BUS interface of the vehicle, but we are simulating this data for the project. There is also a page that allows managers to add vehicles to and delete them from the fleet. The dashboard was built using Angular and deployed using Nginx.[[2]](#footnote-2)

##### 2.1.2 Server

The server is created using Node.js[[3]](#footnote-3), with MongoDB Atlas[[4]](#footnote-4) as the database. It processes

requests from the client and communicates with the database. The server uses RESTful APIs to communicate with the client. On the client side, requests are directed to the backend server through Nginx reverse proxy.

The code for the application has been adopted from an existing project, and can be found here [1]. All rights belong to the original authors. The frontend was implemented in AngularJS[[5]](#footnote-5), which is an older framework whose Long Term Support (LTS) will end on December 31, 2021. As this could introduce potential vulnerabilities in the application, we decided to rewrite the client side in the popular successor framework – Angular[[6]](#footnote-6).



#### Figure 1: Home Page

#### 2.2 Deployment in cloud

The entire application has been hosted on the Google Cloud Platform. We have deployed the application on two different GCP services – Compute Engine[[7]](#footnote-7) and Google Kubernetes Engine.[[8]](#footnote-8)

i.) Compute Engine

To evaluate the performance of hardware level virtualization through virtual machines (VM), the

application was deployed as a Compute Engine Managed Instance Group (MIG). An instance group is a collection of VM instances that you can manage as a single entity. This helps to make the application scalable. [2] We start with a single VM instance, and use the MIG’s autoscaling feature to scale the application up to 4 VMs, when the CPU utilization of an instance crosses 60%. The MIG has been configured with a load balancer to serve the application through a single endpoint.

ii.) Google Kubernetes Engine

To test operating system level virtualization through containers, we also packaged the application into two Docker images – for the client and server each. The application was then deployed on Google Kubernetes Engine, a fully managed Kubernetes service for deploying, managing, and scaling containerized applications using Google infrastructure [3]. Two deployments were created, one for the client and one for the server, with 2 pod replicas in each. In addition, a Horizontal Pod Autoscaler[[9]](#footnote-9) was configured to scale the deployment to a maximum of 5 replicas, with a target pod CPU utilization of 60%. There is an additional pod for the MongoDB database. The client is exposed to the internet via a load balancer service,[[10]](#footnote-10) while the client communicates with the server internally.

## 3. Design

#### 3.1 Previous work

We examined different open source and private fleet management solutions existing online. Accordingly, we decided to implement the most common features for our prototype version.

Most applications have a fleet admin with particular capabilities like adding or managing vehicles in a fleet, tracking their location in real time on a map-based interface, and obtaining statistical data such as the speed of the vehicles, level of gas, engine temperature, distance travelled etc. This data is usually obtained through the vehicle’s On-Board Diagnostics II (OBD-II) port. However, since our team communicated remotely, we decided to simulate the data for the vehicles.

We noticed that there was little previous literature that discussed the deployment architecture and performance of such applications. Thus, for our project, we determined to evaluate and compare the performance and scalability of such a real-time application in the cloud using two different deployment architectures – Virtual Machines (VMs) and containers. We use Google Cloud Platform[[11]](#footnote-11) as our cloud provider. To focus more closely on the testing, we decided to modify an existing project, that implemented all the basic functionalities, according to our needs. The project we chose to adapt was developed by a senior design team at Iowa State University (ISU).[[12]](#footnote-12) We obtained the source code through their GitHub repository.[[13]](#footnote-13)

#### 3.2 Explanation of code

The code has been hosted in a repository on GitHub.[[14]](#footnote-14)

##### 3.2.1 Client side

The ISU project designed the client side using the framework AngularJS. However, this framework is currently in Long Term Support (LTS) mode, and is scheduled to reach the end of its life on December 31, 2021. After this date, Google will no longer make patches or updates for the AngularJS framework. Thus, the application could become less secure over time as there will be no further updates [4]. Therefore, we completely reconfigured the frontend in Angular, the re-engineered successor of AngularJS.

The client has the following structure:

1. Auth module

It handles authentication on the client, and forwards the appropriate requests to the server.

* 1. Register component

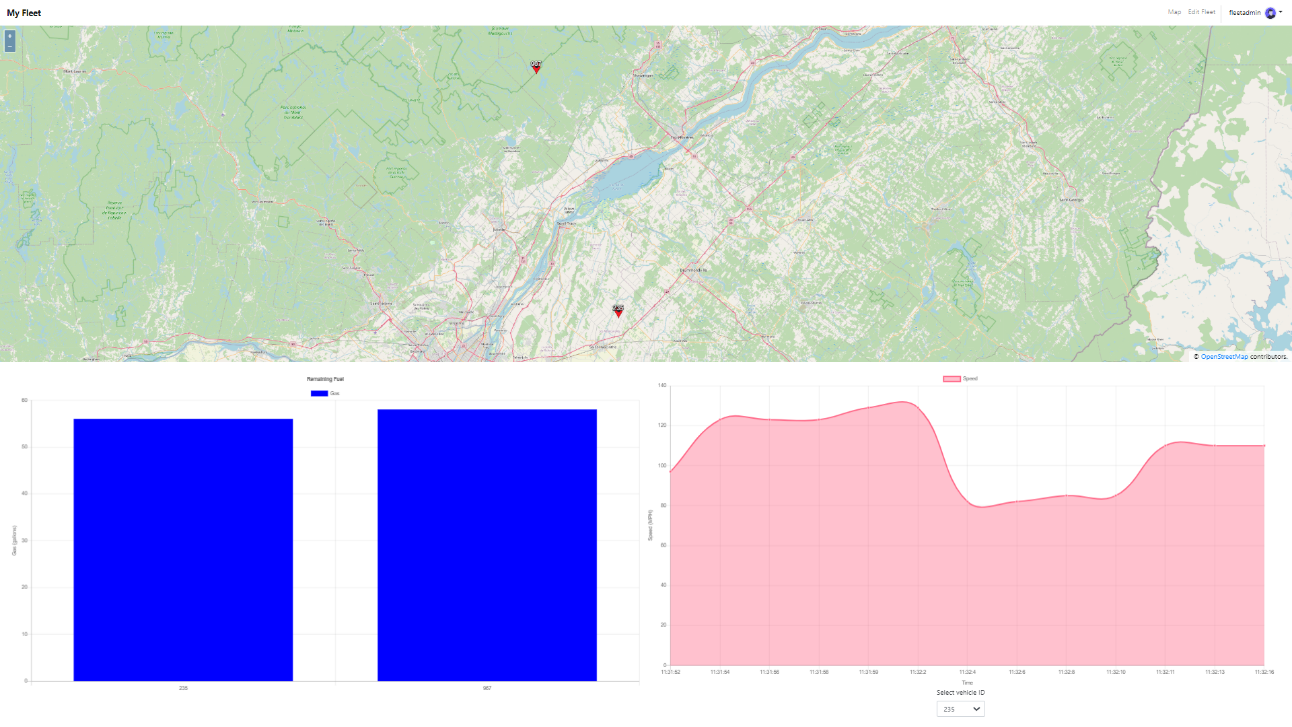
A new fleet admin can be registered through this component

* 1. Login component

Existing users can log in to view their dashboards by providing their credentials here.

1. Dashboard module
   1. Fleet Map component

This is the main dashboard page that contains a map interface with live tracking



*Image 2: Dashboard*

of the respective fleet’s vehicles. For this, we used the mapping library OpenLayers.[[15]](#footnote-15) OpenStreetMap was used as the TileLayer, to provide “pre-rendered, tiled images in grids that are organized by zoom levels for specific resolutions” [5]. The coordinates of all vehicles were randomly set to be close in or around the city of Montréal.

In addition, the dashboard contains two simple charts to visualize the data. A bar chart represents the gas level of the vehicles, and you can view the current speed of a vehicle in a line chart by selecting the appropriate vehicle from a dropdown list. The charts were plotted using ng2-charts,[[16]](#footnote-16) an Angular2 library based on Chart.js.[[17]](#footnote-17)

The location data for the map and the speed and gas level for the charts are queried from the database every two seconds. Currently, we randomly generate these values, which are continuously updated in the database. On the client, we poll the database every 2 seconds to retrieve the latest values.

* 1. Edit Fleet component

This component provides the admin functionalities to manage his fleet. The admin can add vehicles to his fleet and also remove vehicles. Vehicles are identified by their unique ID, which the manager needs to enter. The requests are forwarded to the server which executes the appropriate database queries.

##### 3.2.2 Server side

The server side code was written in Node.js. We have largely retained the code and structure of the original project and modified it as per our requirements. The server side code can be divided into two main groups:

1. Manager controller

This module consists of the APIs pertaining to admin functionalities. Authentication is handled here using the library bcryptjs, which hashes and stores the password in the database. Other APIs include getting a list of the vehicles belonging to the admin’s fleet and updating the list. There is also a Manager schema in Mongoose.

1. Vehicle controller

This module has APIs that control the vehicles in a fleet. It defines APIs to create and delete vehicles in a fleet. It also contains an API to update the details of each vehicle such as the position, gas level etc. This is used to track the vehicle in real time.

## 4. Configuration in cloud

#### 4.1 Managed Instance Group (MIG)

In order to have the same configuration for all VMs in the instance group, we created an instance template. Instance templates define the machine type, boot disk image or container image, labels, and other instance properties. [6]

Each VM instance had the following configuration:

1. Image type: Ubuntu
2. Machine type: e2-medium (2 vCPUs, 4 GB memory)
3. Boot disk type: Balanced persistent disk
4. Boot disk size: 25 GB
5. Zone: us-east-4c (Ashburn, Virginia, North America)

#### 4.2 Google Kubernetes Engine (GKE)

Our application was deployed in a single cluster. The cluster autoscaler was enabled with a minimum of 1 node and a maximum of 4 nodes. Each node in the cluster had the following configuration:

1. Image type: Ubuntu with Containerd (ubuntu\_containerd)
2. Machine type: e2-medium
3. Boot disk type: Standard persistent disk
4. Boot disk size (per node): 10 GB
5. Zone: northamerica-northeast1-a (Montréal, Québec, North America)

### 5. Load testing results

One of our main aims was to compare the performance of our application when deployed with VMs as opposed to containers. We used an open-source load testing tool, Locust to collect the performance statistics.[[18]](#footnote-18) One advantage of Locust is that all the test code is written in Python, making it simpler to implement. We tested our application for a load of 50, 100 and 500 concurrent users.

We will compare 3 metrics from our results – CPU utilization, Response time and Scalability.

#### 5.1 CPU utilization

The charts below show the CPU utilization for each VM instance/node. We can observe that this metric remains fairly consistent across different loads. As the respective autoscalers create new instances or nodes, the pressure on the initial instance reduces and CPU utilization is subsequently distributed. In general, nodes in the GKE cluster consume more CPU than the VM instances in the MIG.

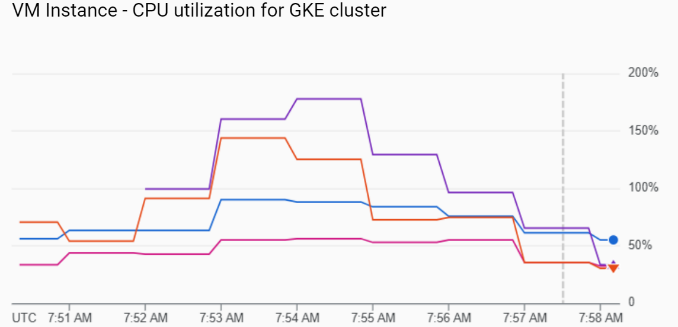
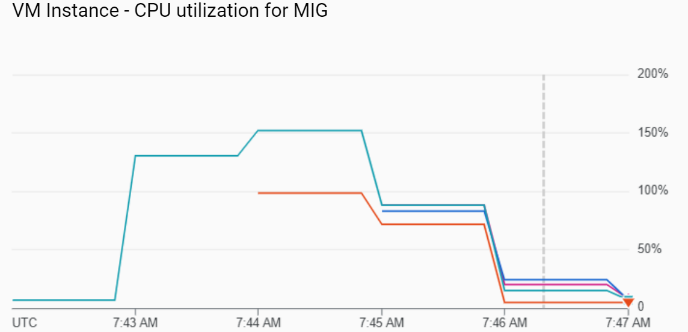


Figure 3 & 4: CPU utilization for load of 50 users

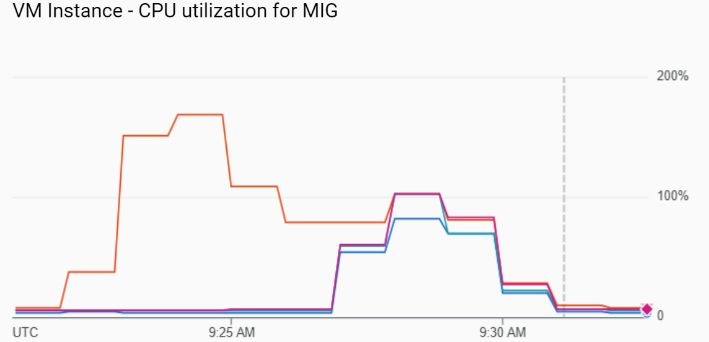
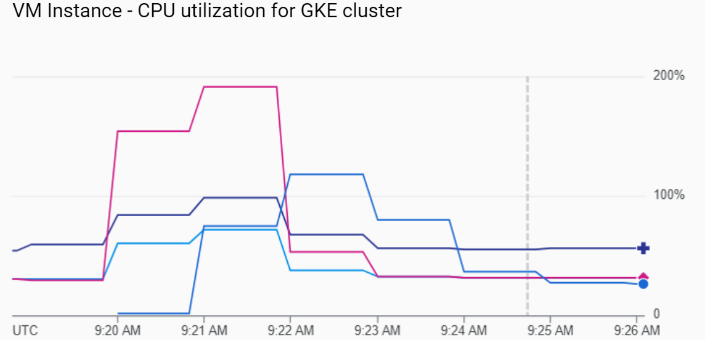
 

Figure 5 & 6: CPU utilization for load of 100 users

#### 

Figure: 7 & 8: CPU utilization for load of 500 users

#### 5.2 Response time

We used the load testing tool Locust to measure metrics like response time, requests per second (RPS) and average size of each request. 6 API endpoints have been used for the testing. The two URLs */* and */login* are endpoints for the client side of the application, while the URLs prefixed with */api* forward their requests to the server.

**50 users**

i) Compute Engine MIG



Figure 9: Compute Engine statistics for load test with 50 users

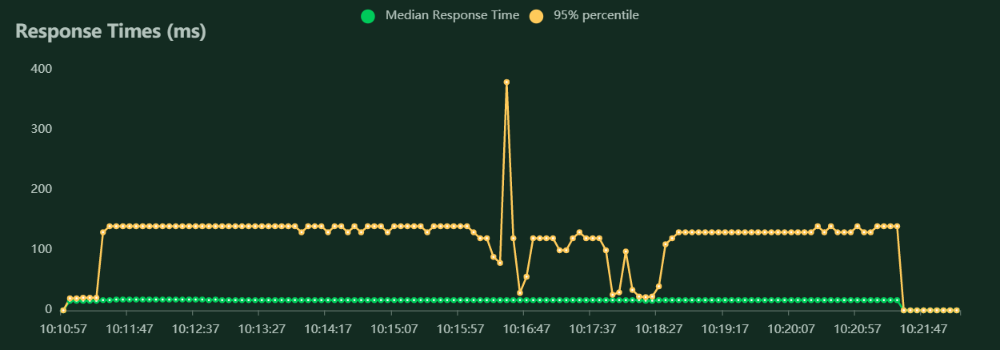


Figure 10: Compute Engine response time graph for 50 users

ii) Google Kubernetes Engine



Figure 11: GKE statistics for load test with 50 users

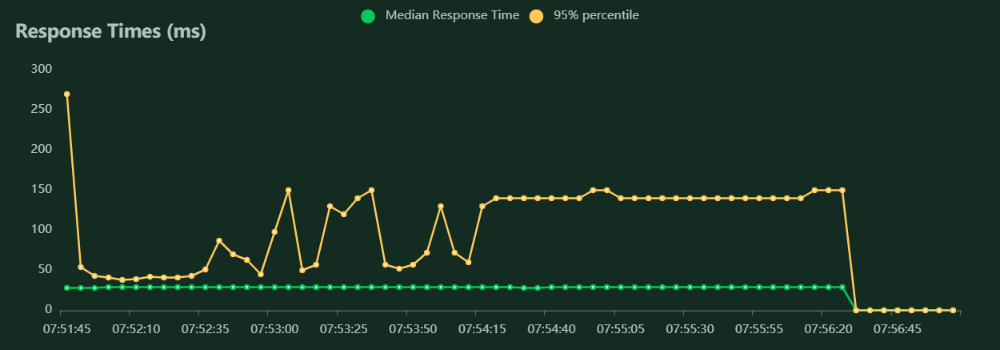


Figure 12: GKE response time graph for 50 users

As can be observed from the graphs above, the aggregate 95% percentile response time of all the APIs is similar for a load of 50 users (~ 150ms), however an inspection of the individual response time for each API indicates the application hosted on GKE was faster in each case except a negligible difference in the client requests. A notable difference can be identified in the API to add a vehicle to the database (POST /api/vehicle), which sends a payload containing data like vehicle number, gas tank size etc in the request. The Compute Engine application experienced 235 failures while the GKE application saw 2. In addition, the response time of the GKE application is almost 2,800 ms lesser than the Compute Engine application.

**500 users**

For a load of 500 users, GKE is comparatively faster with a greater success rate than Compute Engine. In addition, the graph of the reponse times shows that the GKE application was relatively more stable than the MIG which saw some spikes in the response time.

i.) Compute Engine MIG

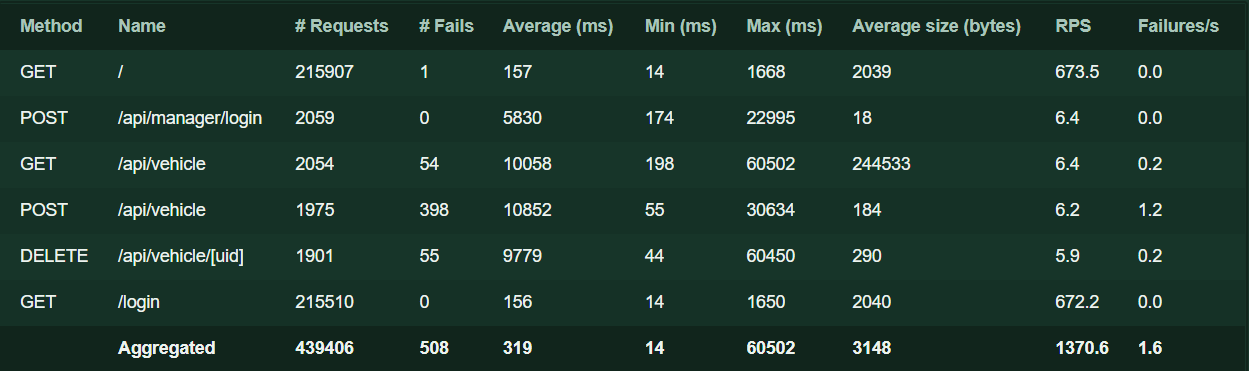


Figure 13: Compute Engine statistics for load test with 500 users

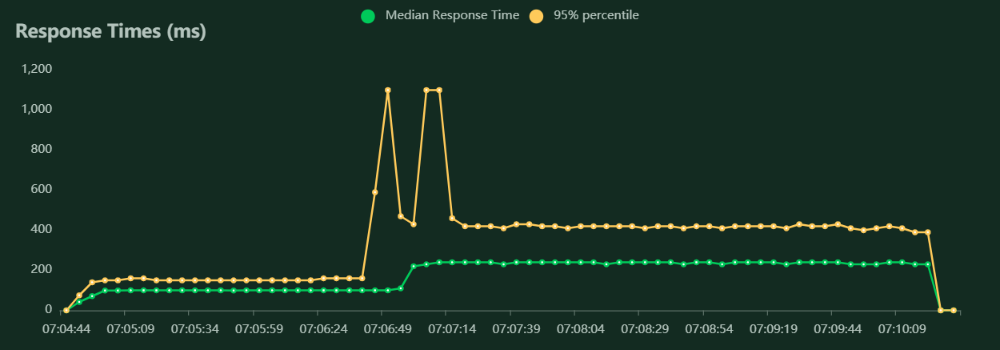


Figure 14: Compute Engine response time graph for 500 users

ii.) Google Kubernetes Engine

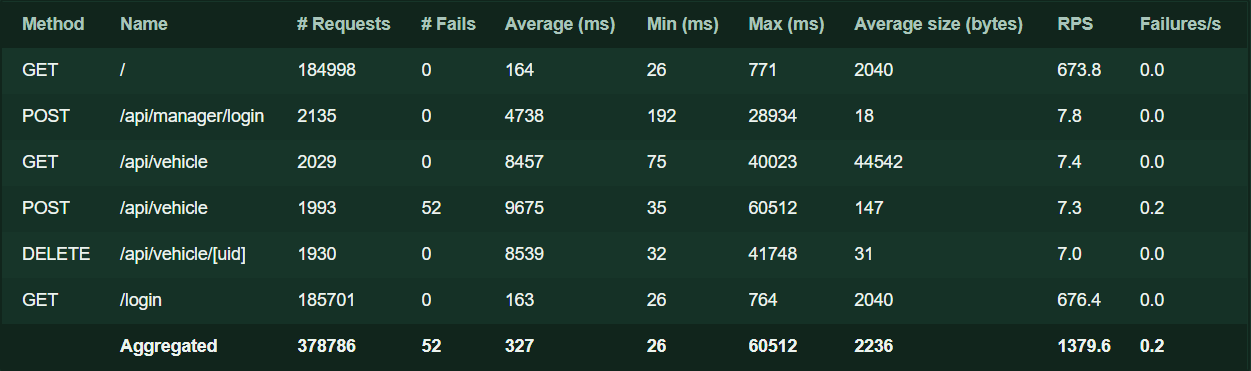


Figure 15: GKE statistics for load test with 500 users

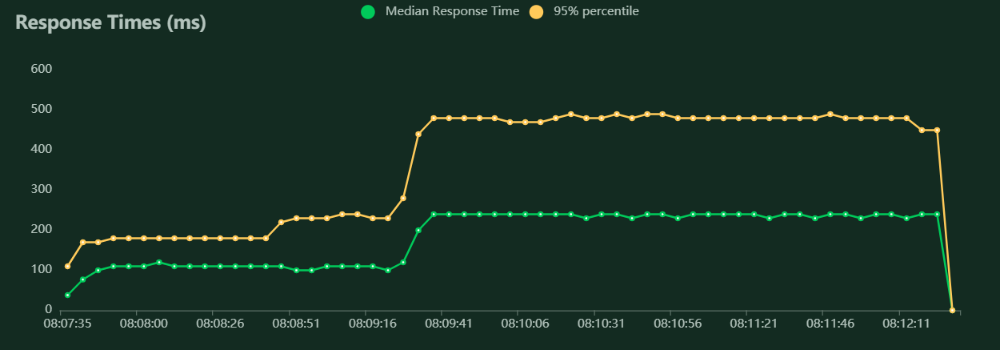
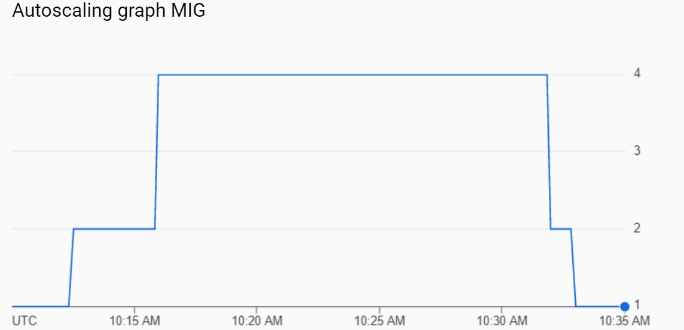
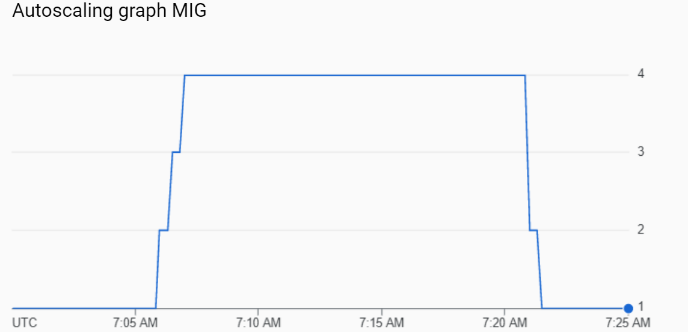


Figure 16: GKE response time graph for 50 users

#### 5.3 Scalability

i.) Compute Engine MIG

The Managed Instance Group (MIG) was configured with autoscaling capabilities. The metric used for autoscaling was average CPU utilization of the MIG. If the average utilization of the total vCPUs exceeds or goes below the target utilization of 60%, new VM instances are added or removed accordingly [7]. The charts below illustrate the working of the autoscaler.

*Figure 17: MIG autoscaler for 50 users Figure 18: MIG autoscaler for 500 users*

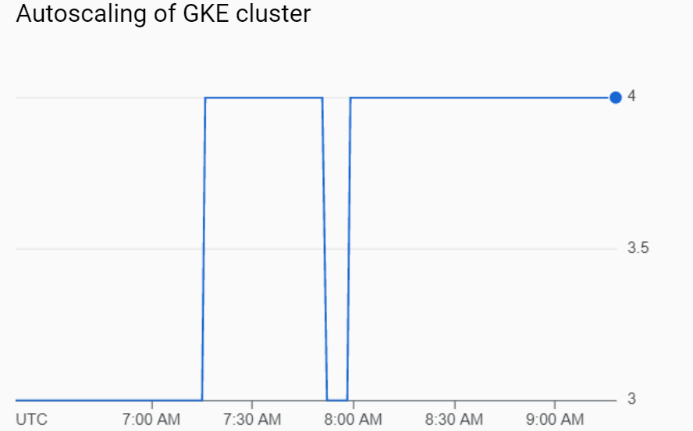
*ii.) Google Kubernetes Engine*

Autoscaling was configured at two levels in GKE.

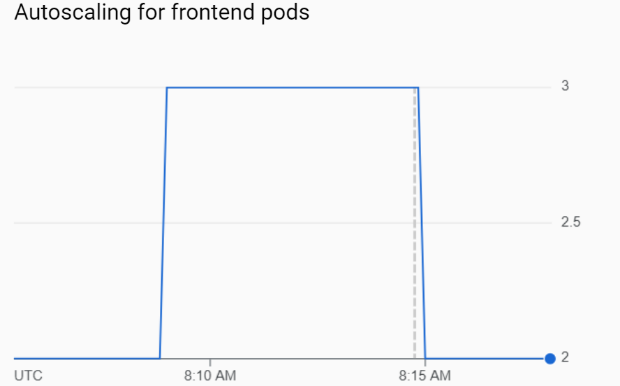
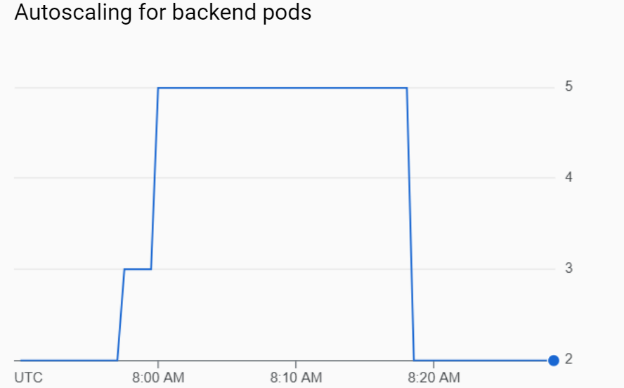
At the first level, cluster autoscaling was enabled with a minimum of 1 node and a maximum of 4 nodes. “Cluster autoscaler increases or decreases the size of the node pool automatically by adding or removing virtual machine (VM) instances in the underlying Compute Engine Managed Instance Group (MIG) for the node pool. Cluster autoscaler makes these scaling decisions based on the resource requests (rather than actual resource utilization) of Pods running on that node pool's nodes.” [8]

At the second level, a Horizontal Pod Autoscaler (HPA) was also configured for both the frontend and the backend deployments. The HPA automatically increases or decreases the number of pods in response to the specified metrics. The target CPU utilization per pod was set to 60% and the number of pods was set between 2 to 5. [9]

The charts below demonstrate the functioning of the cluster autoscaler and HPA for a load of 500 users.



*Figure 19: GKE cluster autoscaler for 500 users*

 *Figure 20: HPA for frontend pods (500 users) Figure 21: HPA for backend pods (500 users)*

## 6. Conclusion

Our prototype of a fleet management system sought to implement the basic functionalities found in such an application and to compare its performance in the cloud using two different architectures. The results from the tests show that the containerized application’s performance is generally better than the one deployed directly on a VM. The project can be further expanded to evaluate the performance with more technologies like edge computing or serverless technology.

References

[1] Fleet Management, <https://github.com/lbenothmane/FleetManagement>.

[2] Instance groups | Compute Engine Documentation | Google Cloud, <https://cloud.google.com/compute/docs/instance-groups>

[3] GKE Overview | Compute Engine Documentation | Google Cloud, <https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview>

[4] What the AngularJS End-Of-Life Means For You? <https://www.inmotionhosting.com/support/edu/openstack/what-the-angularjs-end-of-life-means-for-you/>

[5] OpenLayers v6.6.1 API - Class: TileLayer, <https://openlayers.org/en/latest/apidoc/module-ol_layer_Tile-TileLayer.html>

[6] Instance templates | Compute Engine Documentation | Google Cloud, <https://cloud.google.com/compute/docs/instance-templates>

[7] Scaling based on CPU utilization | Compute Engine Documentation, <https://cloud.google.com/compute/docs/autoscaler/scaling-cpu>

[8] Cluster autoscaler | Kubernetes Engine Documentation | Google Cloud, <https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler#how_cluster_autoscaler_works>

[9] Horizontal Pod autoscaling | Kubernetes Engine Documentation, <https://cloud.google.com/kubernetes-engine/docs/concepts/horizontalpodautoscaler#how_horizontal_pod_autoscaling_works>

1. <https://www.openstreetmap.org/>

   [↑](#footnote-ref-1)
2. <https://www.nginx.com/> [↑](#footnote-ref-2)
3. <https://nodejs.org/> [↑](#footnote-ref-3)
4. <https://www.mongodb.com/cloud/atlas> [↑](#footnote-ref-4)
5. <https://angularjs.org> [↑](#footnote-ref-5)
6. <https://angular.io> [↑](#footnote-ref-6)
7. <https://cloud.google.com/compute> [↑](#footnote-ref-7)
8. <https://cloud.google.com/kubernetes-engine> [↑](#footnote-ref-8)
9. <https://cloud.google.com/kubernetes-engine/docs/concepts/horizontalpodautoscaler> [↑](#footnote-ref-9)
10. <https://cloud.google.com/load-balancing/docs/load-balancing-overview> [↑](#footnote-ref-10)
11. <https://cloud.google.com> [↑](#footnote-ref-11)
12. <https://sdmay18-18.sd.ece.iastate.edu/> [↑](#footnote-ref-12)
13. <https://github.com/lbenothmane/FleetManagement> [↑](#footnote-ref-13)
14. <https://github.com/varunks99/fleet-management-system> [↑](#footnote-ref-14)
15. [https://openlayers.org](https://openlayers.org/) [↑](#footnote-ref-15)
16. [https://valor-software.com/ng2-charts](https://valor-software.com/ng2-charts/) [↑](#footnote-ref-16)
17. [https://www.chartjs.org](https://www.chartjs.org/) [↑](#footnote-ref-17)
18. [https://locust.io](https://locust.io/) [↑](#footnote-ref-18)