Virtual Machine Orchestration and Autoscaling using a real-world application

PROJECT REPORT

Submitted by:

Hrishikesh P Kaulwar

(11640460)

Guided by:

Dr. RATAN K. GHOSH

Dr. SUBHAJIT SIDHANTA

INDIAN INSTITUTE OF TECHNOLOGY BHILAI

CS525: Distributed Systems

MAY 2020

Table of Contents

Declaration

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Hrishikesh Kaulwar
11640460

OVERVIEW

The aim of the project is to deploy a real-world application to a Kubernetes cluster to demonstrate orchestration and autoscaling of the application using distributed systems. The real-world application developed and used here is “**djangostore”** which is a simplified implementation of a shopping web app. The load on “**djangostore”** will get distributed and served by pods present on slave nodes in the Kubernetes cluster. The novelty of the project lies in types of autoscaling that are present and results of benchmarking tests on the webserver by using various benchmarking tools like ApacheBench as well as on database by using YCSB (Yahoo! Cloud Serving Benchmark).

1. INTRODUCTION

Kubernetes

Kubernetes (k8s) is an open-source system used for automating deployment, scaling and management of containerised applications.

Kubernetes Cluster

Kubernetes cluster has at least one master and multiple worker machines which are known as nodes. These master nodes and worker nodes together form kubernetes cluster orchestration system. I have used Google Kubernetes Engine (GKE) to make a kubernetes cluster, which provides a managed environment for deploying, managing, and scaling your containerized applications using Google infrastructure.
Some common terminologies:

Node

A node is the smallest unit of computing hardware in Kubernetes. It represents a single machine in the cluster. A node is either a physical machine in a datacenter, or virtual machine hosted on a cloud provider.

Pods

Pods are the unit of replication in Kubernetes. Kubernetes can deploy new replicas of the pod to the cluster depending on the situation. There are multiple copies of pod running at any given time to avoid failure in application and to allow load balancing.

Containers

A container is a standard unit of software that packages code and all its dependencies so the application runs quickly and reliably from one computing environment to another environment.

Google Kubernetes Engine

Some of the basic terminologies that we must know are described as follows.

Kubernetes Cluster

In GKE, a cluster consists of at least one cluster master and multiple worker machines called nodes. These master and node machines run the kubernetes cluster orchestration system.

Cluster Autoscaling

Cluster autoscaler works on a per-node pool basis. When you configure a node pool with cluster autoscaler, you specify a minimum and maximum size for the node pool.
Cluster autoscaler increases or decreases the size of the node pool automatically, based on the resource requests of Pods running on that node pool's nodes.

Horizontal Pod Autoscaling

The Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment, replica set or stateful set based on observed CPU utilization.

Node Auto Provisioning

Node auto-provisioning automatically manages a set of node pools on the user's behalf. Without Node auto-provisioning, GKE considers starting new nodes only from the set of user created node pools. With node auto-provisioning, new node pools can be created and deleted automatically.

Kubernetes Controllers

Kubernetes controllers act in a similar manner to that of a thermostat. The thermostat acts in a way to bring the current state to the desired state by turning on or off. In the same way, Controllers are the control loops that watch the current state of the cluster and try to make it closer to desired state by making or requesting for changes needed.
Some of the controllers are described below.

1. **ReplicaSet**
   ReplicaSet is the controller for maintaining a stable set of replica Pods running at a given point in time.
   ReplicaSet is defined with fields:

a) Selector
   To specify how to identify Pods it can acquire.

b) Number of Replicas
   Number of pods it should be maintaining to meet the criteria

c) Pod Template
   The template to specify new pods information

2. **Deployments**
   Deployment is where desired state is defined. Deployment Controller changes the present state to desired state at a specific rate. Some of the use cases where Deployment is used:
   - To rollout a Replicaset
   - Declare the new state of the pods
   - For doing a rollback to the previous Deployment version
   - Scaling up the Deployment to facilitate more load

3. **StatefulSets**
   StatefulSet is a workload API object which manages deployment and scaling of a set of Pods. In StatefulSets, ordering and uniqueness of Pods is maintained. Use cases of StatefulSets is for the applications where:
   - Unique Network identifiers are required
   - Persistent Storage is necessary
   - Ordered Deployment and scaling required.

Service

Service is an abstract way of exposing an application on a set of Pods as a Network service. Kubernetes ServiceTypes is to specify the kind of service wanted. The default ServiceTypes is ClusterIP.

Various service types are as follows:

1. **ClusterIP**

   It exposes the service on cluster-internal IP. So, the service is reachable from inside the cluster.

2. **Nodeport**

   It exposes the service on each Node’s static port. (Used in this project)

3. **LoadBalancer**

   It exposes the service externally using the cloud provider's load balancer. ((Used in this project))

Pods in YAML

The manifest YAML can be broken down into four parts:

APIVersion : Version of the Kubernetes API being used
Kind : The type of object that should be created
Metadata : This has information which will uniquely identify the object
Spec : Name, Container Name, Volumes, etc configurations for the pods.

Resource Quota for pods :
requests.cpu is the maximum combined CPU requests in millicores for all the containers in the Namespace.

requests.memory is the maximum combined Memory requests for all the containers in the Namespace.

limits.cpu is the maximum combined CPU limits for all the containers in the Namespace.

limits.memory is the maximum combined Memory limits for all containers in the Namespace.

Ingress

Ingress is not a type of service. Ingress sits in front of multiple services.
It acts as a smart router. The default GKE ingress controller is HTTP Load Balancer. Ingress enables us to do path based and subdomain based routing to backend services.

2. PROJECT WORKFLOW

The entire project is divided into 2 parts ie. running a MySQL server and running a Django web app on the Kubernetes cluster. The Django app communicates with the MySQL server for querying into the database, and both run as separate pods in the cluster. Both of these have horizontal autoscaling enabled so that they can scale up and down horizontally (ie. increase or decrease the number of pods.) depending upon the number of requests or load fed on them. A brief description of the MySQL server and the Django app is described below.

2.1 Django App

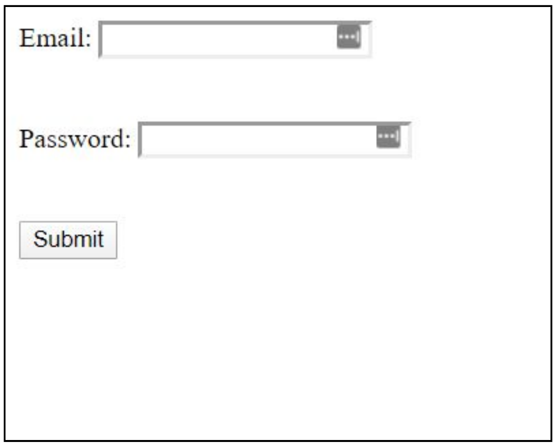
The Django app I have created is named **djangostore**. It is a simplified implementation of an online store app. This app provides a basic interface for purchasing some items from a given list of items that can be updated by the database administrator. On opening the website the user lands on a login page where he can enter his email and password. On successful login, the user is provided with a list of items that he can purchase by selecting them using the checkboxes provided.

Even if the user is not logged then also he can view the available items by going to the “allitems” page. To add the items to the database the administrator needs to go to the “additem” page where he needs to provide the item name, price, and image of the item to be added.

The implemented interface is pretty basic as currently, I am not using any external libraries for the UI purpose. All the APIs defined in the app are defined below.

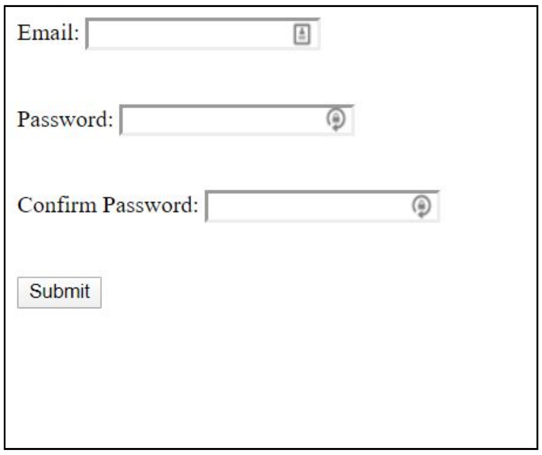
1. login()
   provides functionality for user sign in by verifying email and password from the database.
2. signup()
   provides functionality for user signup by adding the user details into the database.

3. additem()
   provides functionality for adding items (name, price, image) to the database.
4. allitem()
   provides functionality for listing all the items present in the database.
5. buy()
   provides functionality for buying items and generating a simple receipt with the total amount.


Email:

Password:

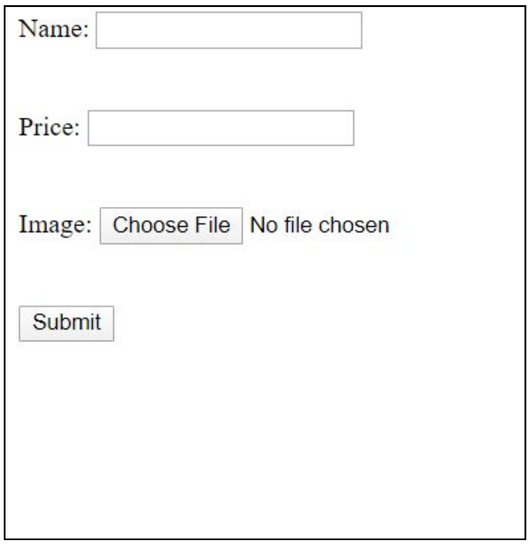
login page


Email:

Password:

Confirm Password:

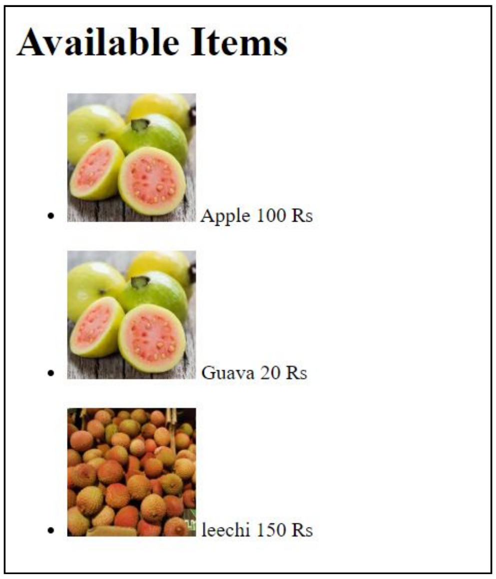
signup page


Name:

Price:

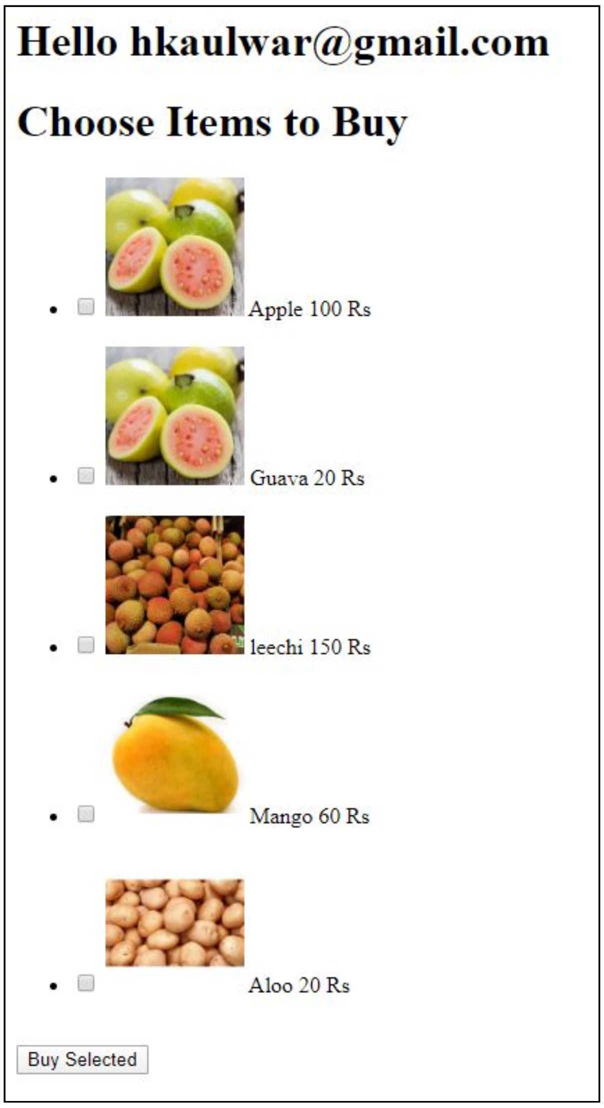
Image: No file chosen

add item page


Available Items

- Apple 100 Rs
- Guava 20 Rs
- leechi 150 Rs

all item page

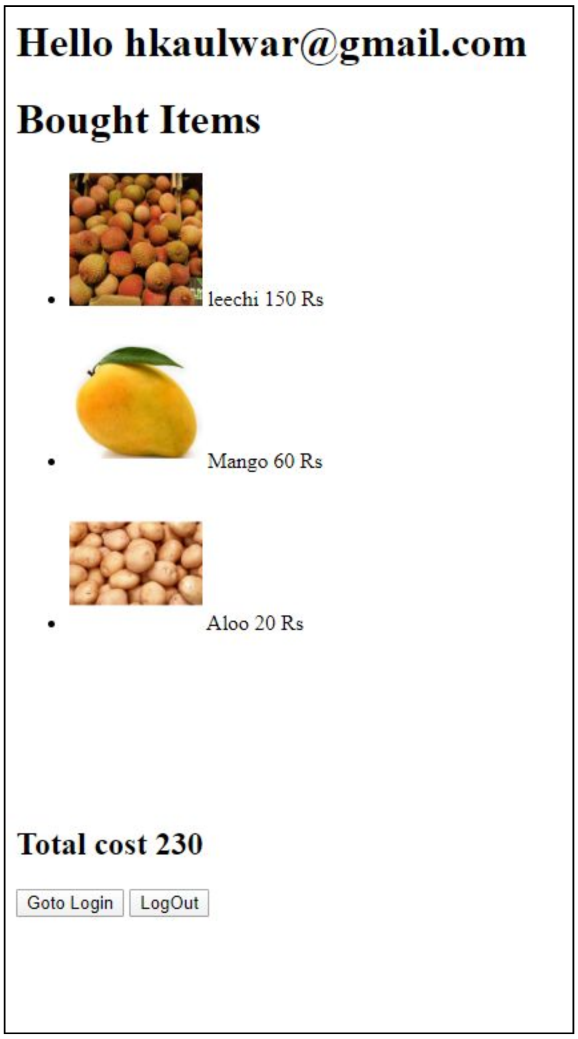


Hello hkaulwar@gmail.com

Choose Items to Buy

- ☐ Apple 100 Rs
- ☐ Guava 20 Rs
- ☐ leechi 150 Rs
- ☐ Mango 60 Rs
- ☐ Aloo 20 Rs

buy item page



Hello hkaulwar@gmail.com

Bought Items

- leechi 150 Rs
- Mango 60 Rs
- Aloo 20 Rs

Total cost 230

receipt page

The "buy item page" is displayed once the user is logged in. Here the user can select the items to buy and then click on the "Buy Selected " button. This will redirect the user to the "receipt page" showing the items purchased items along with the total item.

2.2 MySQL Server

The MySQL server provides the database for the Django web app. For all the data that is displayed on the web app, a query is made to the database.

The structure of the database tables in the form of Django models is defined below.

1. user

```
class user(models.Model):
    email = models.EmailField(unique=True,primary_key=True)
    passwd = models.CharField(blank=False,max_length=30)
    Token = models.IntegerField(null=True)
    issueTime = models.DateTimeField(null=True)
```

2. item

```
class item(models.Model):
    name = models.CharField(max_length=30)
    price = models.IntegerField()
    iid=models.IntegerField(unique=True,primary_key=True)
    img = models.ImageField(upload_to='img/',null=True)
```

These Django models get translated to SQL tables once the migrations are done in the Django app (more details explained in the implementation section).

3. IMPLEMENTATION DETAILS

Locally running the Django web app

After defining the APIs and the database structure and configuring the URLs (urls.py included in the project files) our app is complete (full source code is provided in the project files). Before deploying the app on the Kubernetes cluster we need to first test the app locally.
To run the app locally we first need to make the tables in the database whose structure we defined as Django models.

To make the tables run the following command,

```
python manage.py migrate
```

After running the command, tables will be created in our database, now our app is ready to run, to run the app on the Django development server run the following command,

```
python manage.py runserver
```

This will run a development server on the localhost, we can then go to http://127.0.0.1:8000 to start using the app.

Setting up the MySQL server and database

Before deploying our app to the Kubernetes cluster we first need to create a database on the cluster. The first step is to run MySQL server on the localhost, for now, to connect YCSB with the database and check how the benchmarking can be done locally and look at the output data.
The command used for loading the workload of YCSB is :

```
./bin/ycsb load jdbc -P workloads/workloada -p
db.driver=com.mysql.jdbc.Driver -p
db.url=jdbc:mysql://localhost:3306/database_name -p db.user=bill
-p db.passwd=passpass -s -threads 5 -p db.batchsize=1000 -p
jdbc.fetchsize=1000 -p jdbc.autocommit=false -p
jdbc.batchsize=1000
```

```
The command to run the workload :

./bin/ycsb run jdbc -P workloads/workloada -p
db.driver=com.mysql.jdbc.Driver -p
db.url=jdbc:mysql://localhost:3306/database_name -p db.user=bill
-p db.passwd=passpass -s -threads 5 -p db.batchsize=1000 -p
jdbc.fetchsize=1000 -p jdbc.autocommit=false -p
jdbc.batchsize=1000


Arguments:
db.batchsize=1000   the no. of rows batched before commit
jdbc.batchsize=1000    batch size for batched insert
jdbc.fetchsize=1000    fetch size hinted to the driver
db.user=bill          username for database
db.passwd=passpass    password for database
db.driver=com.mysql.jdbc.Driver --the jdbc driver class
db.url=jdbc:mysql://localhost:3306/database_name : url of database
connection
workloads/workloada -- type of workload
```

Before running the load command, I created the usertable in the database named database_name. So that YCSB client will load the values in here.

```
CREATE TABLE usertable (
      YCSB_KEY VARCHAR(255) PRIMARY KEY,
      FIELD0 TEXT, FIELD1 TEXT,
      FIELD2 TEXT, FIELD3 TEXT,
      FIELD4 TEXT, FIELD5 TEXT,
      FIELD6 TEXT, FIELD7 TEXT,
      FIELD8 TEXT, FIELD9 TEXT
);
```

The MySQL is deployed by using replicated stateful applications. StatefulSets Controller is used for this. The MySQL has a master-slave structure with Single master and multiple slaves running. The master has write access to the database and slaves have read-only access to the database.

MySQL working :

For Deploying ReplicaSet of MySQL Server on kubernetes cluster there are 3 yaml files:

1. mysql-configmap.yaml
   Configmap provides the mechanism through which master will be able to serve replication logs to slaves and slaves will be able to reject any writes that don't come via replication.

2. mysql-services.yaml
   This creates two services.
   1. Mysql Service for stable DNS members of StatefulSet
   2. Client service for reads through mysql instances

3. mysql-statefulset.yaml
   The StatefulSet Controller makes one pod at a time and waits for the pod to become ready before creating the next pod.
   Each pod has unique stable name of the form
   <statefulset-name>-<index>
   here, example mysql-0, mysql-1.

```
$ kubectl create -f mysql-configmap.yaml

$ kubectl create -f mysql-services.yaml

$ kubectl create -f mysql-statefulset.yaml
```

So, now before creating any container in the pod spec, init containers are run first. The init container init-mysql runs and creates a server-id.cnf file which is used to provide a stable identity.
The script in the init-mysql also applies config master.cnf or slave.cnf from configmap. For this to be a single master multiple script topology, script assigns 0 to be the master and rest all to be the slaves.

The second init container clone-mysql performs operation of cloning on slave who will have empty persistent volume. MySQL itself does not provide a mechanism to do this cloning, so a popular open-source tool called Percona XtraBackup is used.

The MySQL Pods consist of a mysql container that runs the actual MySQL server, and **xtrabackup** container that acts as a sidecar.

Command for exposing mysql read for the outside world and for exposing pod mysql-0 for benchmarking purposes.

```
$ kubectl expose service mysql-read --name mysql-service-read
--type LoadBalancer --port 3306 --protocol TCP

$ kubectl expose pod mysql-0  --name mysql-service-write  --type
LoadBalancer --port 3306 --protocol TCP
```

Now we can connect to this mysql read and master by giving the ip address as input url in YCSB command and this way benchmarking is done.

Installing Google Cloud SDK and create Kubernetes cluster

For easy management of our cluster and ease in deployment process we first need to install the Google Cloud SDK, to install the SDK follow these steps,

1. Download the appropriate compatible version of the SDK from the following link,
   https://cloud.google.com/sdk/docs/downloads-versioned-archives

2. Extract the contents of the file to any location on your file system. Preferably, this is your Home folder.
3. To install the SDK and add Cloud SDK tools to your path, run the install script. This should be run from the root of the folder you extracted above. Running this script will also generate instructions to enable command completion in your bash shell (Linux and macOS only) and enable usage reporting.

```
$ ./google-cloud-sdk/install.sh
```

4. Run gcloud init to initialize the SDK:

```
$ gcloud init
```

If the Cloud SDK is added to your path then this command will run successfully without any errors.

Now install **kubectl** to easily deploy the files directly from the terminal.

```
$ gcloud components install kubectl
```

After installing the prerequisites we can create our cluster, the command to create a Kubernetes cluster with cluster autoscaling enabled is,

```
$ gcloud container clusters create cluster-name --num-nodes 2 \
    --enable-autoscaling --min-nodes 1 --max-nodes 8 --zone
us-central1-c
```

The cluster formed by this will have a default node pool having 2 nodes and is capable of autoscaling to become a cluster having 8 nodes at max.

Installing Docker and building Docker image

Now to start the containerization process we need to install Docker. Docker has various distributions available but the one we will be using is "docker-ce" which is the latest certified release provided directly by the official docker website.

To install docker follow these steps:

1. Update the apt package index and install packages to allow apt to use a repository over HTTPS:

```
$ sudo apt-get update
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common
```

2. Add Docker's official GPG key:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
apt-key add -
```

3. Use the following command to set up the **stable** repository.

```
$ sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

4. Update the apt package index, and install the latest version of ***Docker Engine*** and ***containerd*** :

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

5. Verify that Docker Engine is installed correctly by running the hello-world image.

```
$ sudo docker run hello-world
```

Once the Docker installation is verified we need to create a **Dockerfile** inside our project directory. This **Dockerfile** contains all the commands a user could call on the command line to assemble an image. This will automate the image building process. We also need to have a **requirements** file for our project which contains the list of all the dependencies with their versions needed for our project. To see the dependencies and store them in a file we can use,

```
$ pip freeze >> requirements.txt
```

Dockerfile for our project,

```
FROM python:3.7

ENV PYTHONUNBUFFERED 1

RUN mkdir /app
WORKDIR /app

# Copying & Installing python requirements
COPY requirements.txt /app/
RUN pip install -r requirements.txt

# Syncing the source of the application
COPY . /app/

EXPOSE 8000
```

```
# Run the gunicorn web server
CMD ["gunicorn", "--bind", "0.0.0.0:8000", "module.wsgi"]
```

Now to build the image, run the following command,

```
$ docker build -t gcr.io/project-ds-275505/store-app:v1.0.0 .
```

Before pushing the image to the container registry we first need to create a service account using the Google Cloud Console, and generate an authentication key. We will use this key for authenticating our machine to push the image to the container registry.
(Reference: https://cloud.google.com/container-registry/docs/advanced-authentication#gcloud-helper)

After authenticating we can push the image to the container registry using the following command,

```
# Common format to push an image to google container registry is
gcr.io/$PROJECT_ID/$IMAGE_NAME:$TAG

$ docker push gcr.io/project-ds-275505/store-app:v1.0.0
```

Deploying the app on Kubernetes cluster

After pushing the image to the Kubernetes cluster we need to deploy our app, so that we can access it from a public IP. For this purpose we need to create three YAML files.

1. store-migration.yml
   This will run the database migrations creating a table in the database.
2. storeapi.yml
   This will create deployment and "storeapi" service to start the application.
3. storeapi-ingress.yml
   This will create an ingress to expose the application.

After the ingress is successfully created we can navigate to the ingress address generated in order to access our application.

```
$ kubectl create -f store-migration.yaml
$ kubectl create -f storeapi.yaml

$ kubectl create -f storeapi-ingress.yaml
```

Enabling HPA in Kubernetes

Horizontal pod autoscaling for MySQL, Store-API.
For enabling horizontal pod autoscaler, for mysql,

```
$ kubectl autoscale statefulset mysql --cpu-percent=60 --min=1
--max=10
```

For enabling horizontal pod autoscaler, for store-api,

```
$ kubectl autoscale deployment mysql --cpu-percent=60 --min=1
--max=15
```

4. BENCHMARKING RESULTS

Web Server Benchmarking

For Benchmarking the webserver we have used different benchmarking tools Gunicorn is the webserver used here. Gunicorn is a consistent performer for medium loads.

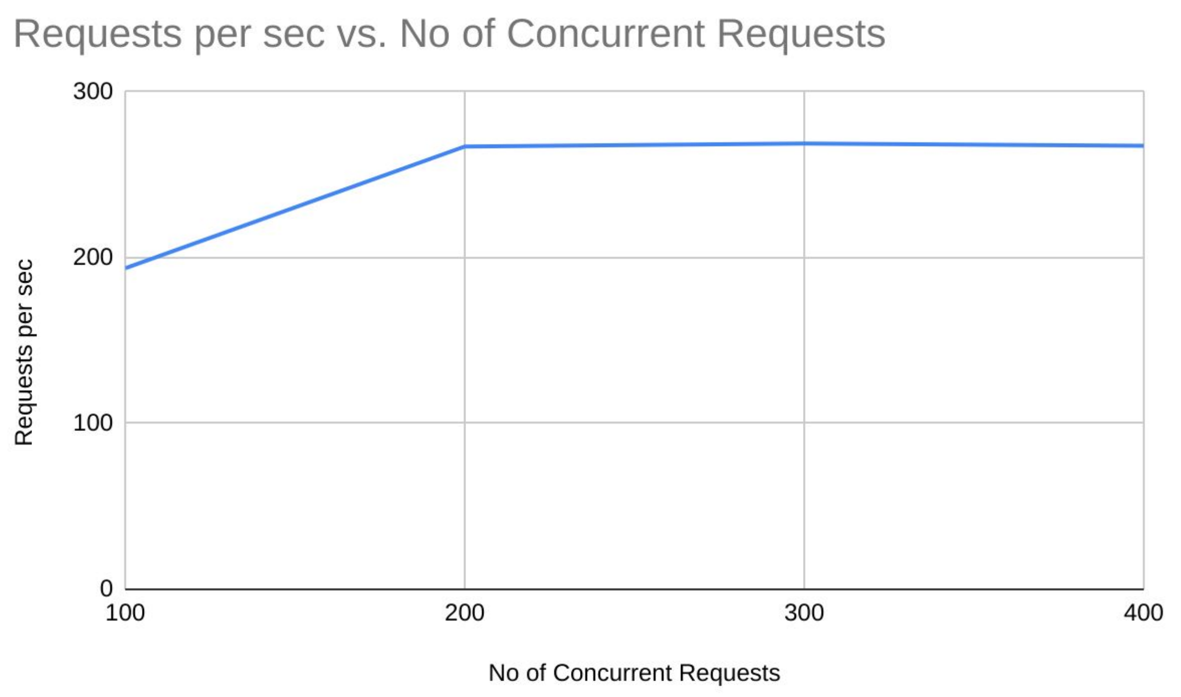
Apache Bench

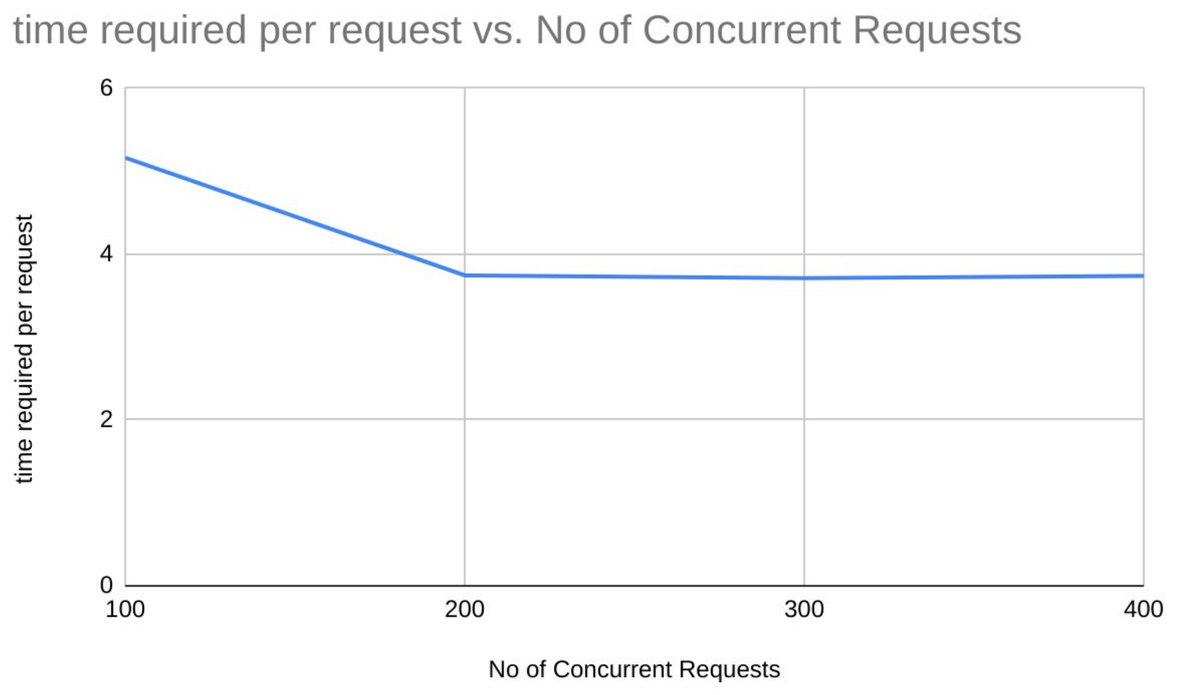
The command to run the apache bench tool is
ab (OPTIONS) (WEB_SERVER_ADDRESS)/(PATH)
Options:
-n: The number of requests to send
-c: The number of concurrent requests to make
The command used for benchmarking here for example is

```
$ ab -n 10000 -c 1000 http://34.98.101.117/allitem/
```

Experiment 1:

Here while testing the webserver, the settings were,
No of total requests fixed at 10000.
The autoscaling was also there with maximum limit of 10 pods.





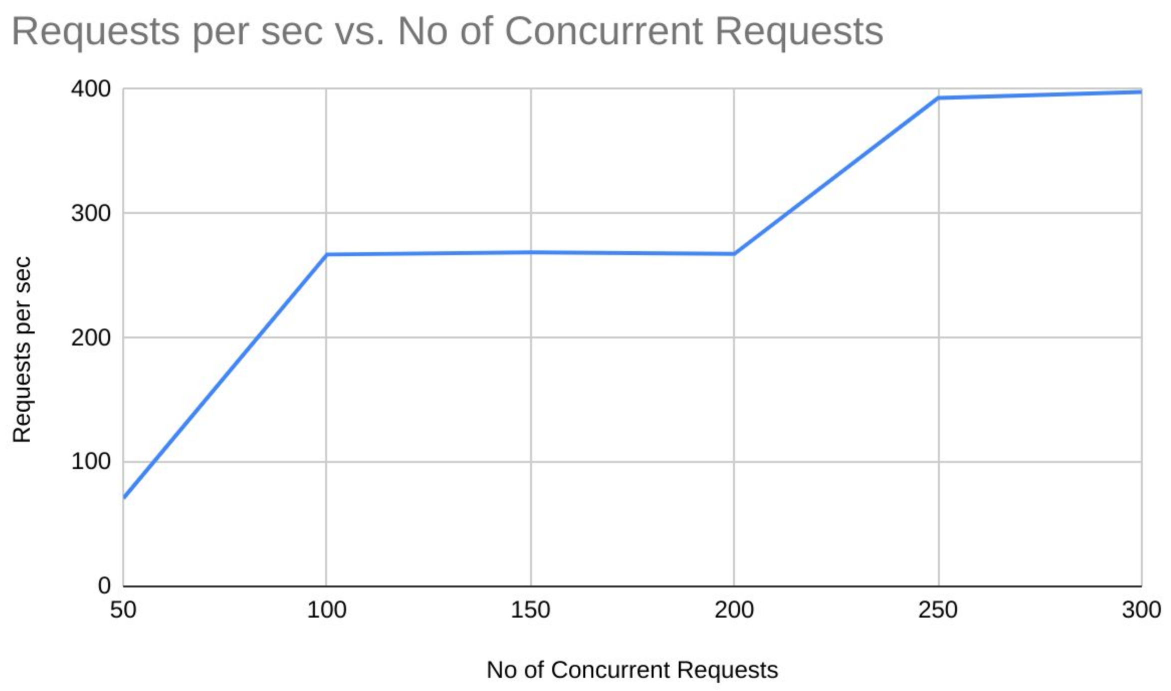
From this experiment, it is observed that 267 requests per sec is the capacity of the webserver when the number of pods are set to 10. This is verified by the observation that the measurements tend to this value whatever may be the value for no of concurrent requests. Similar is the case for the time required per request.

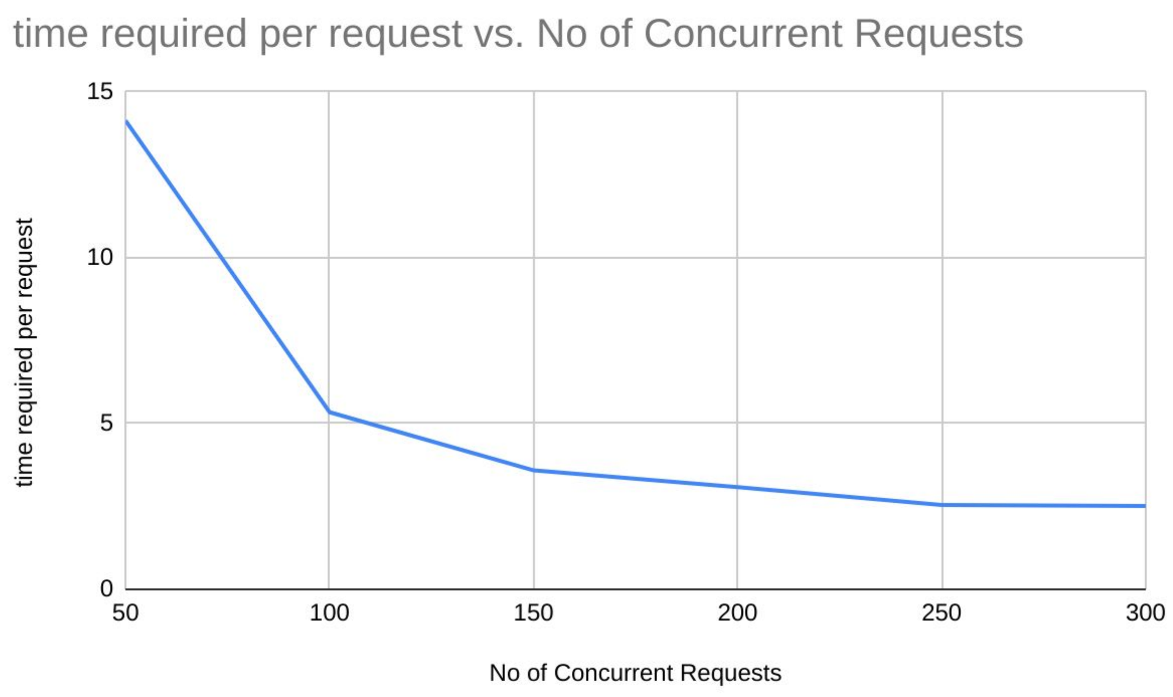
For each pod configs are :

limits:
cpu: 200m
memory: 200Mi
requests:
cpu: 125m
memory: 100Mi

Experiment 2:

Here while testing the webserver the settings were,
No of total requests = 5000.
The autoscaling was observed here. At the start 1 pod was running and at the end, 15 pods were running. (In the autoscaling, 15 was set as maximum pods for this experiment)





From this experiment it is observed that 397.75 requests per sec is the capacity of the webserver when the number of pods are set to 15 . This is verified by the observation that the measurements tend to this value whatever may be the value for no of concurrent requests. Similar is the case for the time required per request.

**Resource constraints configs for pods are the same as the previous experiment.*

For running the experiments, there were a lot of constraints like:

1. The GCP has a limit on No. of IN USE ADDRESSES because of which there is a limit on no. of virtual machines being added to the cluster.
2. There are also various limits on No. of virtual machines per project of GCP, No. of virtual machines per region, and per zone.

Database Benchmarking

YCSB Yahoo Cloud Serving Benchmark is used for these experiments on the MySQL database.
I have used this tool for benchmarking the MySQL database by using 3 types of workloads a,b,c

1. **Workload A**: This workload has a mix of 50/50 reads and writes. An application example is a session store recording recent actions.
2. **Workload B**: This workload has a 95/5 reads/write mix. Application example: photo tagging; add a tag is an update, but most operations are to read tags.
3. **Workload C**: This workload is 100% read. Application example: user profile cache, where profiles are constructed elsewhere.

These are the commands to run YCSB benchmark on MySQL database

This is the command to load the workload into the database.

```
./bin/ycsb load jdbc -P workloads/workloadc  -threads 2 -p
db.driver=com.mysql.jdbc.Driver -p
db.url=jdbc:mysql://35.188.80.11:3306/database_2 -p db.user=root
-s  -p db.batchsize=1000 -p jdbc.fetchsize=1000 -p
jdbc.autocommit=false -p jdbc.batchsize=1000
```

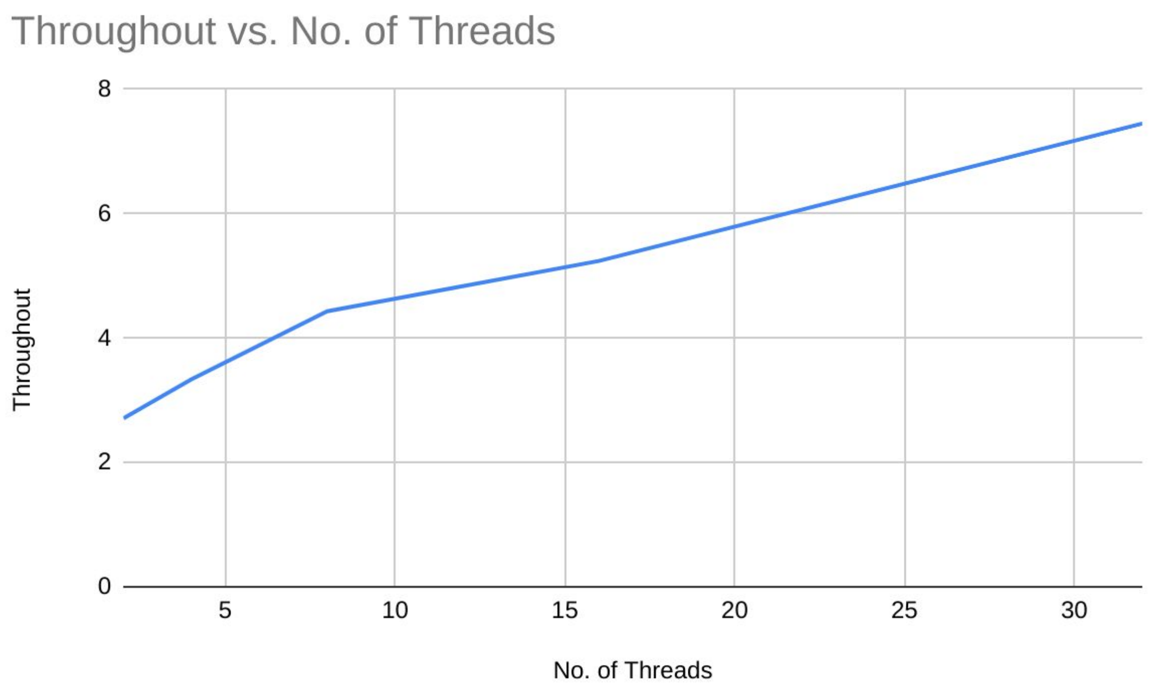
where
workload c = is a workload which has 100/0 read/update ratio
threads = no of threads
db.driver= The JDBC class driver to use
db.url= The database connection url
db.user= Username

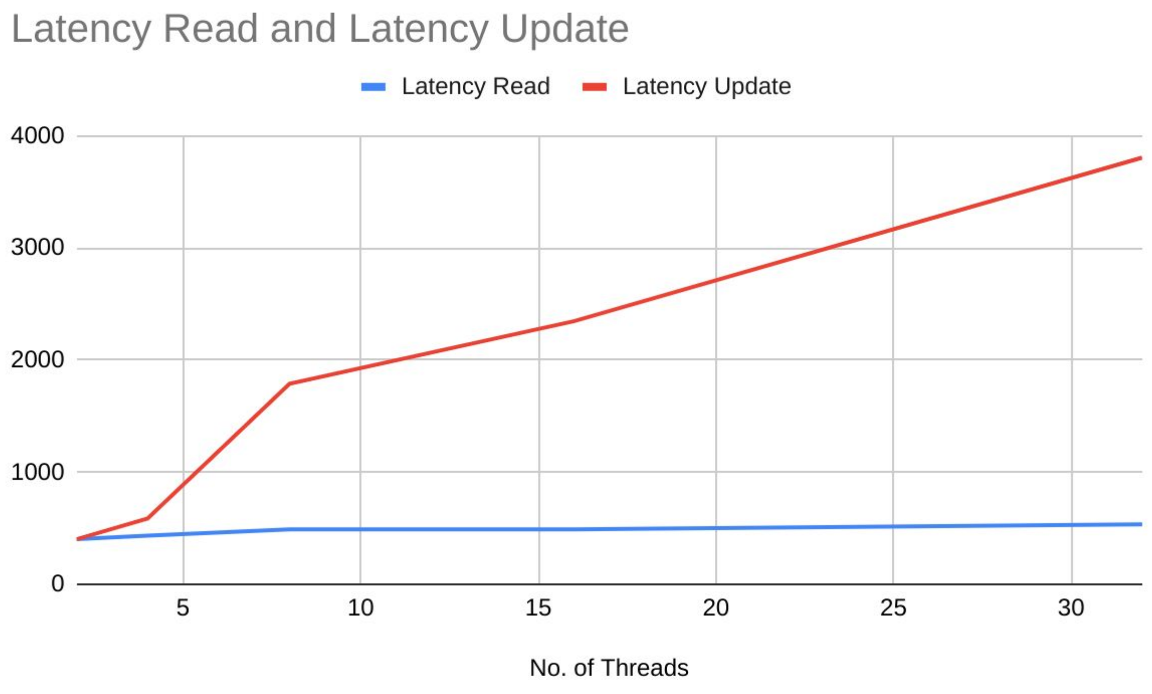
This is the command to run the workload on slave pod of MySQL.

```
./bin/ycsb run jdbc -P workloads/workloadc  -threads 2 -p
db.driver=com.mysql.jdbc.Driver -p
db.url=jdbc:mysql://35.239.24.62:3306/database_2 -p db.user=root
-s -p db.batchsize=1000 -p jdbc.fetchsize=1000 -p
jdbc.autocommit=false -p jdbc.batchsize=1000
```

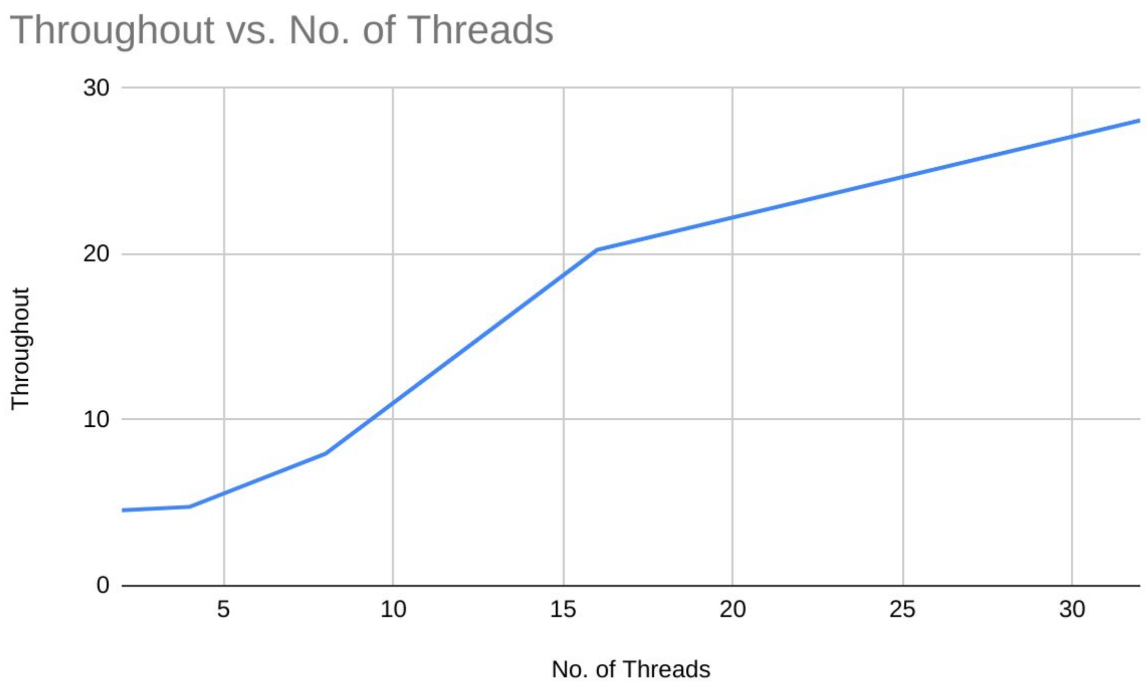
Readings used are given in the submission.
For all these benchmarking results below, record counts were constant at 1000.
The number of client threads. The YCSB Client uses a single worker thread.

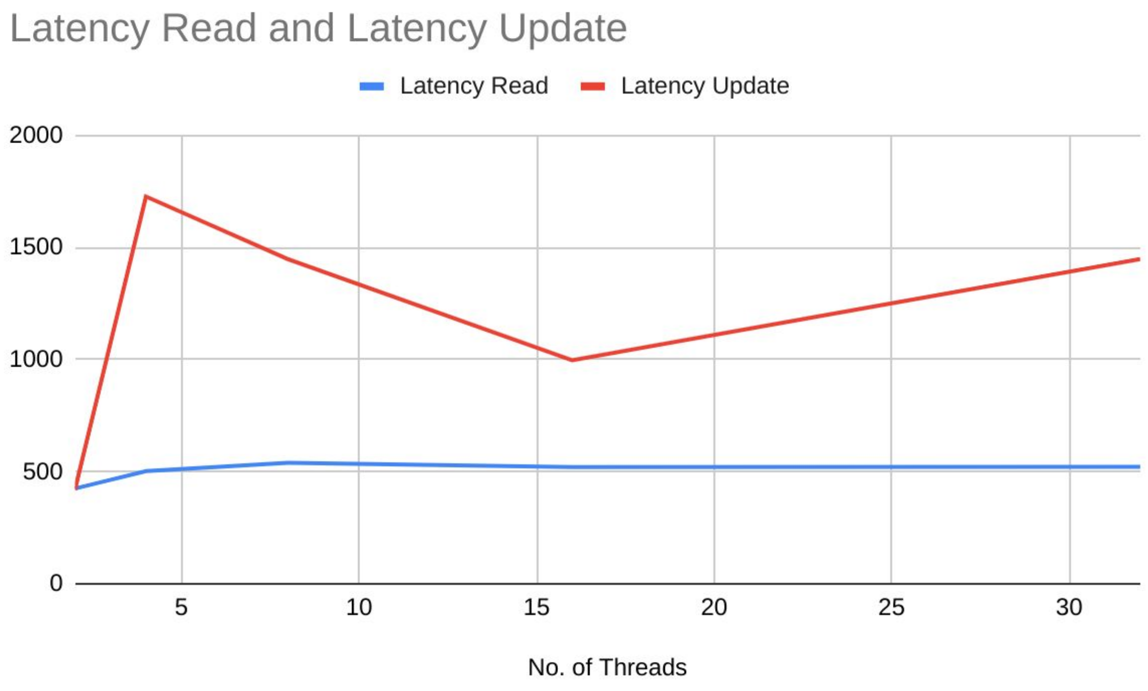
Graphs For Workload A



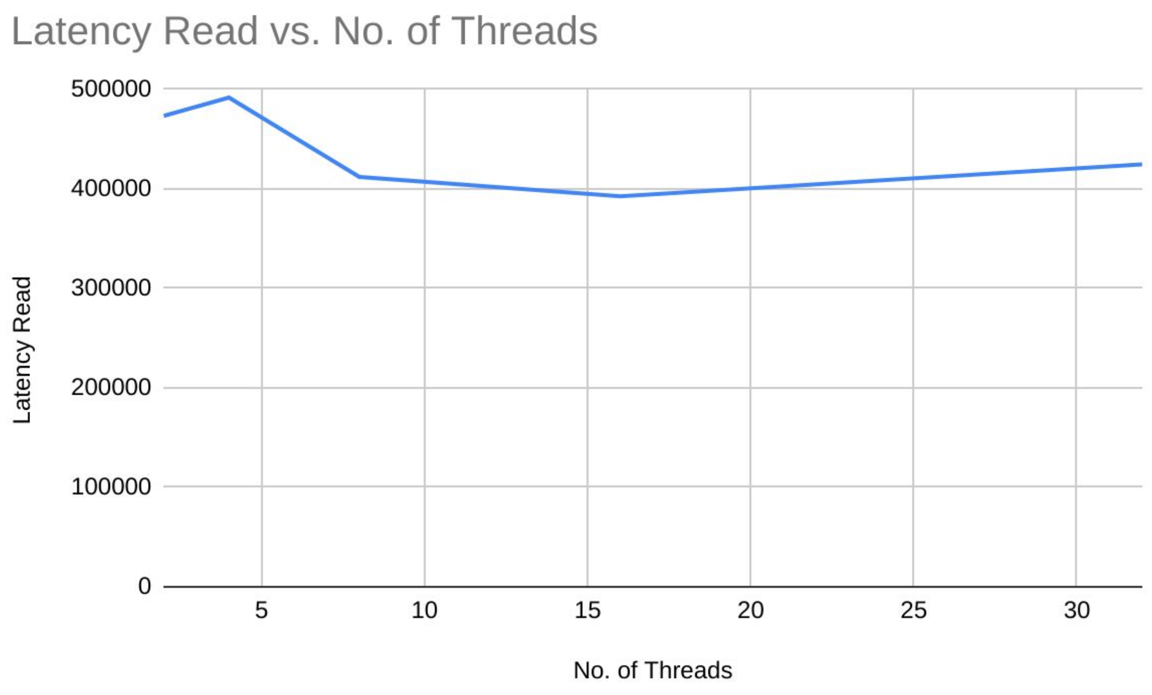


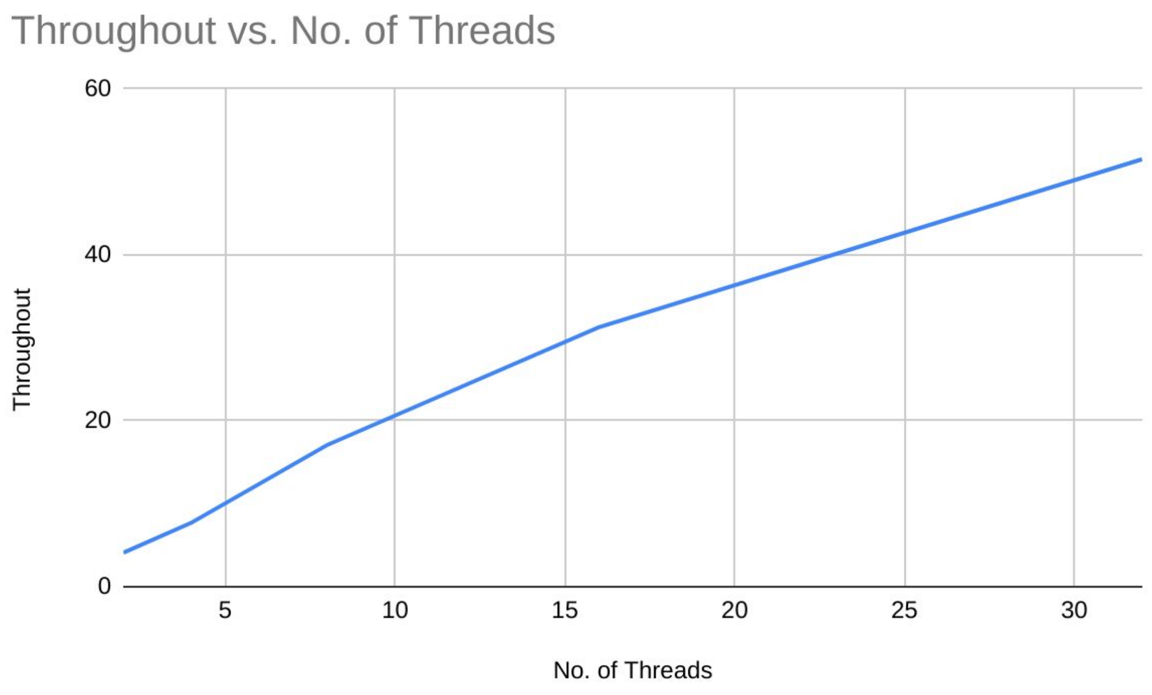
Graphs For Workload B





Graphs For Workload C





WHAT GRAPHS SAY:

For all the workloads, throughput increases as no of worker threads in YCSB client increases. The read latency is nearly constant throughout the variation of no. of worker threads for YCSB client. The update latency increases as the threads increase in most cases.
Above observations are keeping all the parameters in YCSB constant.

5. CONCLUSIONS

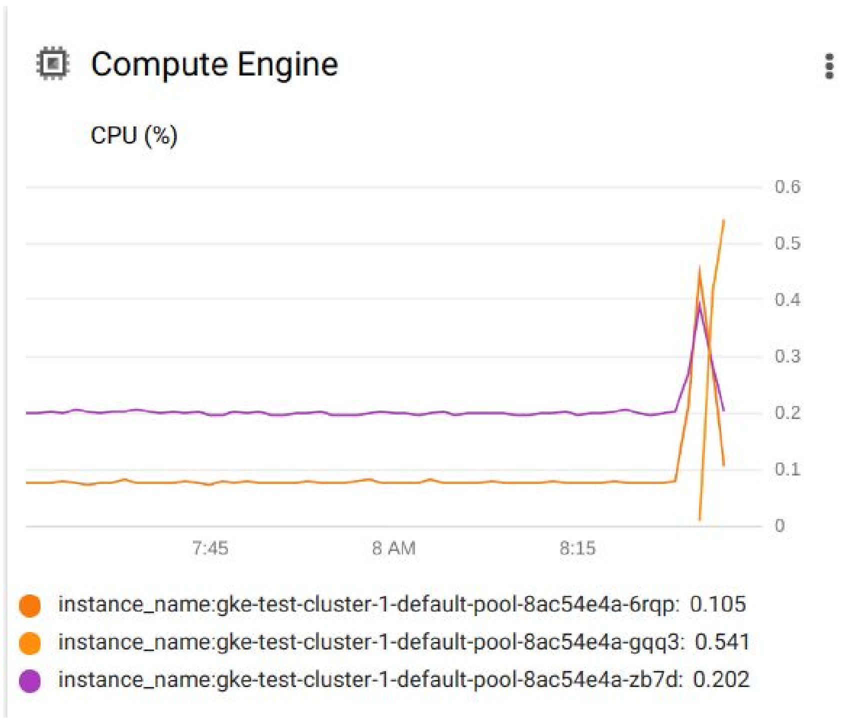
In this report, I have presented work done in a Project for the course CS525 : Distributed Systems. For the Project, I have implemented a sample shopping web app in django which uses MySQL database for storing its data. The app was then containerised with docker and the containers were deployed on kubernetes cluster. The kubernetes cluster on which the application is shows various types of autoscaling including horizontal pod autoscaling and cluster autoscaling. Other types of auto scaling such as node pool provisioning and vertical pod autoscaling can also be demonstrated here. Finally, I tested the web server and MySQL database against various benchmarking tools to gather some interesting results.

6. REFERENCES

1. Gcloud components install | Cloud SDK Documentation
2. Running Django on Google Kubernetes Engine | Python
3. Dockerfile reference
4. Understanding Kubernetes Objects
5. Introduction to YAML: Creating a Kubernetes deployment
6. How to use ApacheBench for web server performance testing
7. https://github.com/brianfrankcooper/YCSB/tree/master/jdbc
8. Running a Workload · brianfrankcooper/YCSB Wiki · GitHub
9. Service
10. Run a Replicated Stateful Application
11. Horizontal Pod Autoscaler Walkthrough

7. APPENDIX

CPU Usage of Nodes in the cluster at a point when



● instance_name:gke-test-cluster-1-default-pool-8ac54e4a-6rqp: 0.105
● instance_name:gke-test-cluster-1-default-pool-8ac54e4a-gqq3: 0.541
● instance_name:gke-test-cluster-1-default-pool-8ac54e4a-zb7d: 0.202