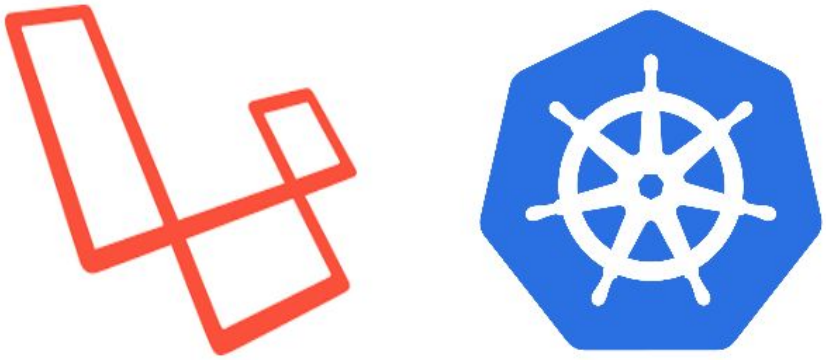
INFINITE-SCALE

Deploying and Scaling a Laravel application on Kubernetes



Cloud Computing

2019-2020

Theodoros Mandilaras

Cs2.190018

INTRODUCTION

In the today's world, the applications on the web are getting a huge amount of attention from people from all over the world. Some of those applications are providing services for more than a million uses every day. Most of the time, those applications are host over the cloud cause it provides the required infrastructure and minimizes the deployment effort with a very low cost, and also, in a global scale.

In order an application to be successful, it should fulfil some basic standards. Not only, it should be constantly working, but also, it should work smoothly with the accepted latencies. Those basic standards require a DevOps team that will constantly check the health status of the application. But a lot of things can go wrong very fast. What if the application get a sudden amount of attention (e.g. by getting viral). It will cannot handle that and it will fail, disappointing many users. In order to avoid that, more instances of the application should be deployed in the cloud before the application crash. However, when the attention goes low, some of those new instances will be useless and it will produce an unnecessary cost. So there is no reason to maintain them.

In order to solve all those problems we need a Container[1] Orchestrator. Most famous Container Orchestrators right now in the market are:

- Docker Swarm
- Kubernetes
- MESOS

In this project I worked to solve the above mentioned problem using **Kubernetes**.

KUBERNETES (K8s[2])

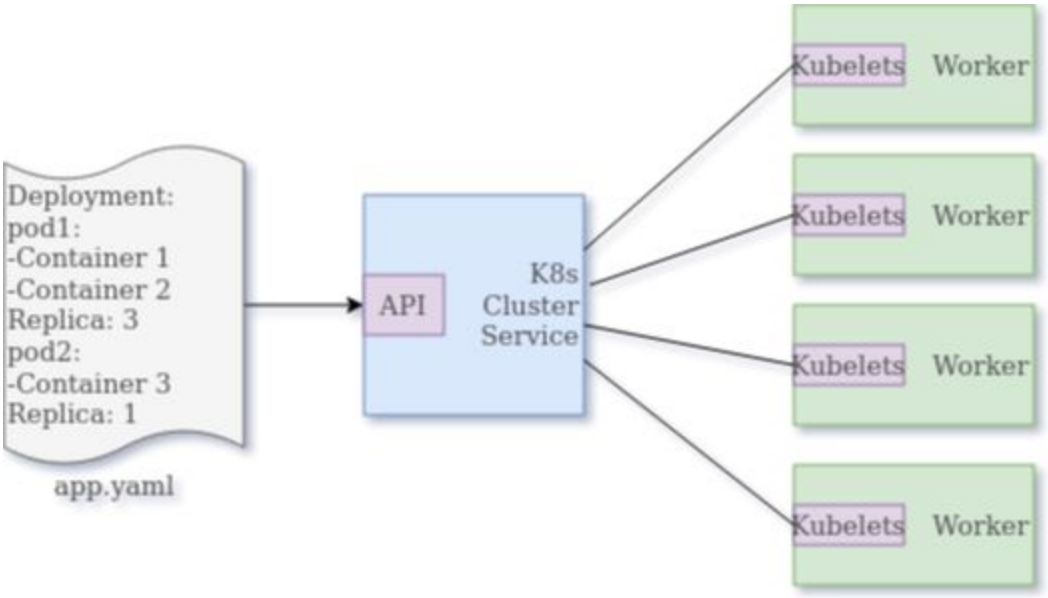
Kubernetes is an open-source tool that was initially born from Google to facilitate numerous deployments across an infrastructure. It also works just like a dedicated 24/7 DevOps team, that makes sure the application runs smoothly, and when things go wrong it will make sure to handle it smoothly. Kubernetes keeps track of how many copies of the application are running. If a copy fail, it will recreate it immediately and guarantee the application will still be accessible. Moreover, if it is required, it can scale-up the application if the load increased and scale-down when there is no more need.
Furthermore, Kubernetes can provide rolling-updates for the copies of an application, one instance at a time, and if the things go bad can roll-back to the previous state. Also, it can help us test new feature for our application using A/B testing methods, by only upgrading a percentage of our instances.

How it works?

In the K8s environment the main building block is called **K8s Cluster Service.** We enforce a "Desired State Management" by feeding the **K8s Cluster Service** a specific configuration (called Deployment) and it is up to that service to run that configuration on the infrastructure. In this configuration we specify the **pods**[3] of our application and which images will run and how many replicas of each pod we want to have.

---

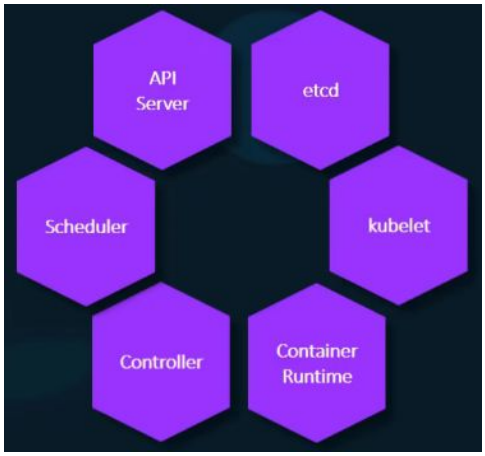
[1] Containers are called the running instances of an application in the host.
[2]Aabbreviation for Kubernetes. Is created by the first letter "K", the last letter "s" and the 8 letters between them.
[3] Pods are the smallest unit of a deployment, in which docker containers are running.



In the k8s-environment there are **workers**. The workers are simple container hosts with a kubelet process running inside them. This process is responsible to communicate with the **K8s Cluster Service**.

When, the **K8s Cluster Service** gets a deployment file with the configuration from its API, it's up to him to figure out how to schedule the required pods in the environment and make sure that it has the right number of pods running in the workers. In a scenario where a worker fails, the **K8s Cluster Service** has to make sure the deployment is imposed. So the scheduler will pick one of the available workers to fill that gap.

Main Components



The main component of the **Kubernetes** are:

- An **API server** which acts as a front-end in order to interact with the Kubernetes cluster
- An **etcd** server which a distributed reliable key-value store, in order the Kubernetes manage all the data for multiple workers and multiple masters.
- A **kubelet** service that runs in the workers.
- A **container runtime** engine like Docker.
- Many **Controllers** which are the brain behind orchestration.
- Many **Schedulers** which are responsible to distribute the work to the workers.

Kubectl tool

Is the Kubernetes CLI which is used to deploy and manage applications on a Kubernetes cluster, and to get related information about the states of the nodes and other things.

My Project

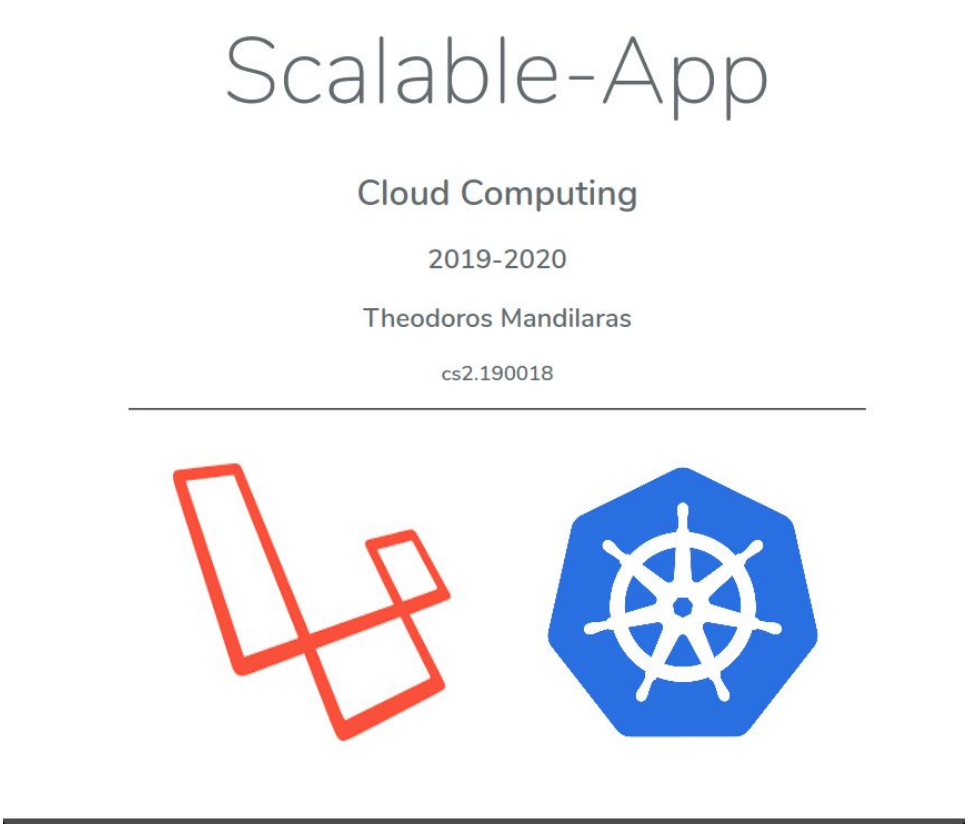
My project purpose is to create a web application and to deploy safely it in a cluster using Kubernetes. Also, I want my application to be cable to scale horizontally when some metric's thresholds are overcome. Those metrics are the *average percentage of the CPU* utilization and it is set to 50%, and the *average request time* to be less than a second.

My Web Application

For my web application I used the **Laravel** framework which is a free, open-source PHP web framework. It intends for the development of web applications following the model view controller (MVC) architectural pattern and it is based on Symfony.

I created a very simple application which does only two basic things.

1. It responds in a GET request with the index view page which is the following:



2. It responds in a GET request on the route: /do-work by doing an exhaustive work. The purpose of the task is for the application evaluation.

```
Route::get('/do-work', function(){
   $x = 0.0001;
   for ($i = 0; $i <= 1000000; $i++) {
       $x += sqrt($x);
   }
   return response()->json(["OK!"=>$x], 200);
});
```

My Cluster

Because of the absence of a cloud cluster, for deploying my application I used the **Minikube.** Minikube is a tool that makes it easy to run Kubernetes locally. It runs a single-node Kubernetes cluster inside a Virtual Machine(VM) on a personal computer. It also, is delivered with a monitoring dashboard which makes the application checking very easy.

In order to set up minikube, we need a virtualization driver for the computer. In my case I installed **Virtual Box** which s a free and open-source hosted hypervisor for x86 virtualization created by Oracle.

Application Deployment

Setting up the environment

After installing the minikube, we have to start it, check that it worked with success and open the minikube dashboard.

```
minikube start --driver=virtualbox
minikube status
minikube dashboard
kubectl config use-context minikube
```

In order the application to be executable we have to install some dependencies. In order to do that we have to install the composer. Composer is an application-level package manager for the PHP. It can be easily installed from here. Then create the dependencies with the following commands:

```
cd in/the/project/scalable-app
composer install
```

Then we have to create the docker image. The image downloads the php-fpm image, which contains all the import software to run a heavy load PHP application like a Laravel one. Then it copies our application inside the image. Also, in order the image to be viewable in the minikube we have to allow it to have access.

```
eval $(minikube docker-env)
cd in/the/project  # or cd ..
docker build . -t  teo/scalable-app
```

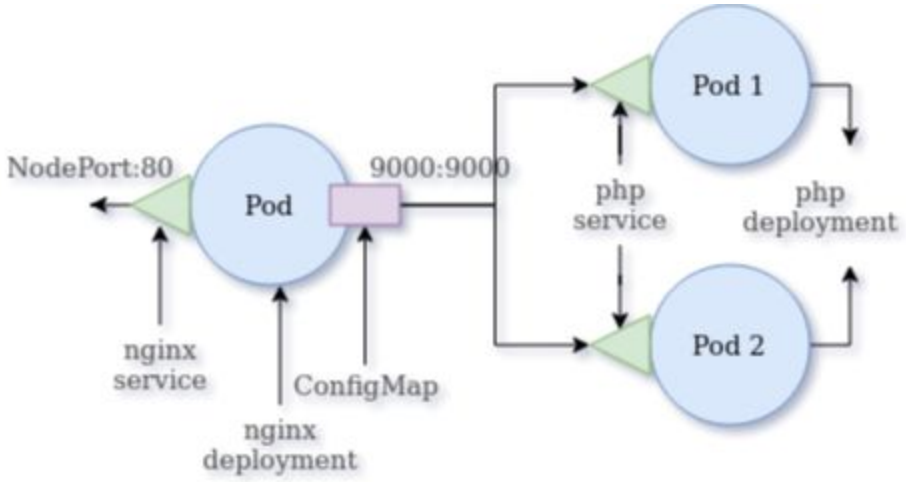
Setting up the application

Now that the application's image is built and available, we can go ahead and deploy it on the Kubernetes cluster.

```
kubectl apply -f php_deployment.yaml
kubectl apply -f php_service.yaml
kubectl apply -f nginx_deployment.yaml
kubectl apply -f nginx_service.yaml
kubectl apply -f nginx_configMap.yaml
```

Let's review the commands:

- First we deploy the `php_deployment` file which it sets the name of our application: “scalable-app”, it says to the **K8s Cluster Service** to create one replica for that pod, it mentions our image that we created and to look first locally to use it. Moreover, it set some other parameters just like paths, tiers, commands to run when it is created and some mounted volumes.
- In order to expose our application to the cluster we create a service with the `php_service` which exposes our deployment on 9000 port.
- With the `nginx_deployment` we create a deployment for our nginx server, using the online image `nginx:1.7.9` and setting again some parameters like the replicas number which is one, the mount for the config file of the nginx and the port.
- And again we have to expose our nginx deployment by creating a service with the `nginx_service` file in which we expose it in a node_port.
- Last, we create a config-map in the cluster which it just contains the basic configurations for a php application using nginx.



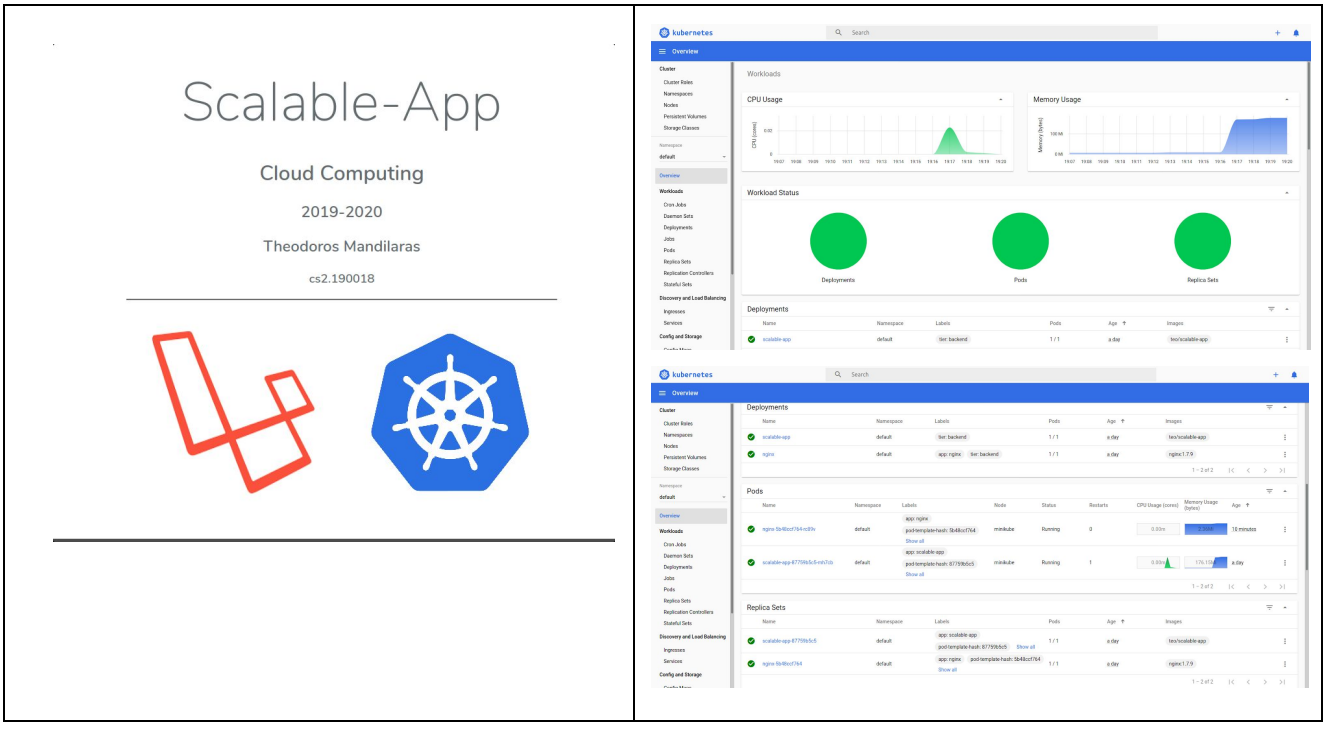
Now we have deployed the application with success.

Getting the application URL

In order to get the dynamic generated application URL we have to run **one** of the below commands:

```
minikube service nginx  # auto load
```

```
minikube service nginx --url  # for manual copy
```



We can also check the state in the minikube dashboard or with the bellow command.

```
kubectl get pods
```

We can try manually to delete one of our pods using:
`kubectl delete pods <one of the pods name>`

We will see that the **K8s Cluster Service** will instantly recreate the deleted pod. That is because we have asked the **K8s cluster service** to always have one replica of every pod running.

Error Handling

Sometimes because we are using busybox image in initContainer lifecycle event, this will not copy code from our image to volume because PHP container or pod is not yet created. So we might get forbidden or NOT FOUND error when we open our nginx service in browser, so we have to copy code manually to the /code/scale-app/ directory.

```
kubectl get pods
# login into the pod
kubectl exec -it scalable-app-<one random pod name> -- /bin/bash
# copy the app in the correct destination
cp -r /var/www/. /code/scalable-app
# fix some permissions
chown -R $USER:www-data /code/scalable-app/storage/
chown -R $USER:www-data /code/scalable-app/bootstrap/cache/
chmod -R 775 /code/scalable-app/storage/
chmod -R 775 /code/scalable-app/bootstrap/cache/
```

Ingress

Now, we will see how to access the application through an assigned URL as we would do when deploying to the cloud. In order to use a URL in Kubernetes, we need an Ingress. An Ingress is a set of rules to allow inbound connections to reach a Kubernetes cluster. Normally, we have used Nginx or Apache as a reverse proxy. The Ingress is the equivalent of a reverse proxy in Kubernetes. It will also handle the inbound traffic to the most available pod, according to a set of rules that we define in Kubernetes resource file.

In order to use Ingress we need an Ingress controller. Minikube comes with the Nginx as Ingress controller, and you can enable it with:

```
minikube addons enable ingress
```

Once it is enabled, we can create the ingress like:

```
kubectl apply -f ingress.yaml
```

We can verify and obtain the Ingress information by running:

```
kubectl describe ingress
```

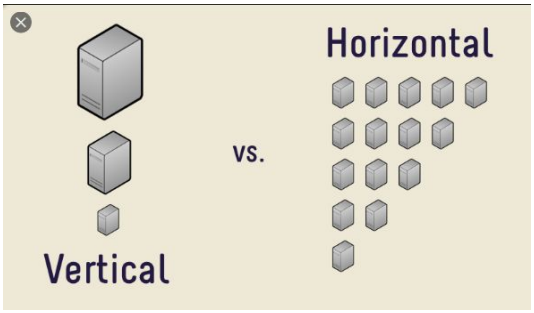
Which returns:

```
Name:             scalable-app
Namespace:        default
Address:          192.168.99.100
Default backend:  default-http-backend:80 (<error: endpoints "default-http-backend" not found>)
Rules:
  Host        Path  Backends
  ----        ----  --------
  *
              /   scalable-app:80 (172.17.0.7:9000)
Annotations:  Events:
  Type        Reason  Age   From                      Message
  ----        ------  ----  ----                      -------
  Normal      CREATE  46m   nginx-ingress-controller  Ingress default/scalable-app
```

In the **Address** part of the result in our URL.

Auto Scaling

To enable the auto-scaling feature we have to apply an **`HorizontalPodAutoscaler`** kind Kubernetes file.



Generally **Horizontal scaling** means that you **scale** by adding more machines into your pool of resources, whereas Vertical **scaling** means that you **scale** by adding more power (CPU, RAM) to an existing machine.

In our case when the application needs to scale up, it will create more replicas of itself into new pods. In the autoscaling.yaml file I set it to scale at max to 10 replicas and the lowest to be just 1. Here are also set the criteria for scaling. As I have already mentioned those are the:

- Average CPU utilization: 50%
- Average Request time: 1 second

However, in order the minikube to have access to those metrics we have to enable them by running

```
minikube addons enable metrics-server
```

After that we can set our auto-scaling by

```
kubectl apply -f autoscaling.yaml
```

Now, the auto-scaling is active.

Quick set-up

We can set-up all the **kubectl** commands easily at once by running

```
kubectl apply -f .
```

in the correct directory,

Clean-up

We can delete everything that we created by removing them all like:

```
kubectl delete deployment scalable-app
kubectl delete service scalable-app
kubectl delete deployment nginx
kubectl delete service nginx
kubectl delete configmap nginx-config
kubectl delete ingress scalable-app
kubectl delete horizontalPodAutoscaler scalable-app
```

EVALUATION

We already have seen that Kubernetes keeps track of how many replicas of our application exists and how many we have asked it to maintain. When, a replica fails, the Kubernetes cluster service will handle it by recreating the missing replica.

For testing the auto-scaling feature, I have used a command line software called SIEGE, which is used to test the limits of a URL and it can be easily installed with the commands:

```
sudo apt-get update -y
sudo apt-get install -y siege
```

I evaluated the application by requesting for multiple users for one minute using the siege command like:

```
siege -c<number-of-users> -t1M <URL-application>
```

I used it for 1, 10, 20, 30, 40, 50 and 100 different users for the application with and without the auto-scale feature. Firstly, I evaluated the performance of the response for the simple lightweight index page, and then for the exhausted work of the /do-work route. I evaluated the performance by comparing the shortest and the longest transaction, and also the average response time. The results are shown below:

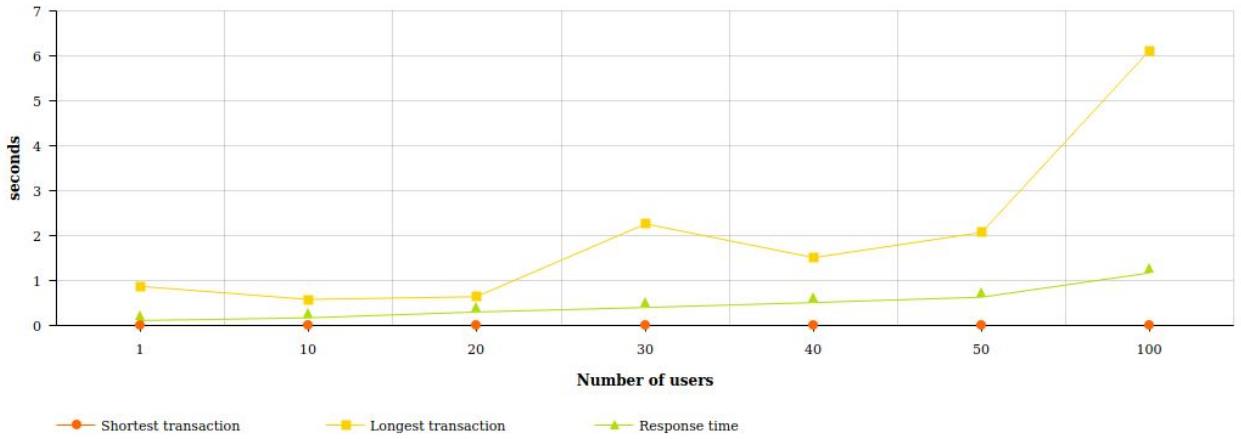


Figure 1: GET request for the index page **without** auto-scaling.

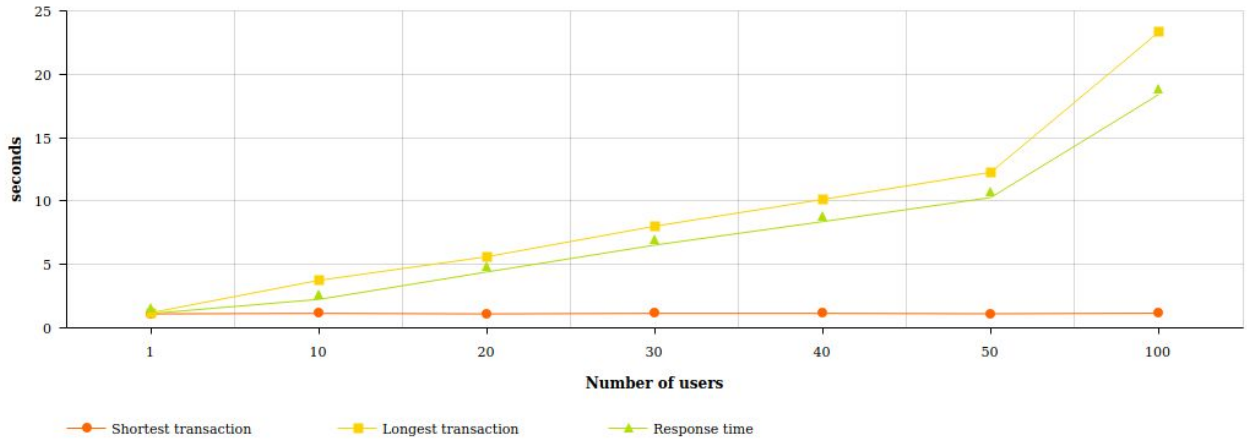


Figure 2: GET request for the /do-work route **without** auto-scaling

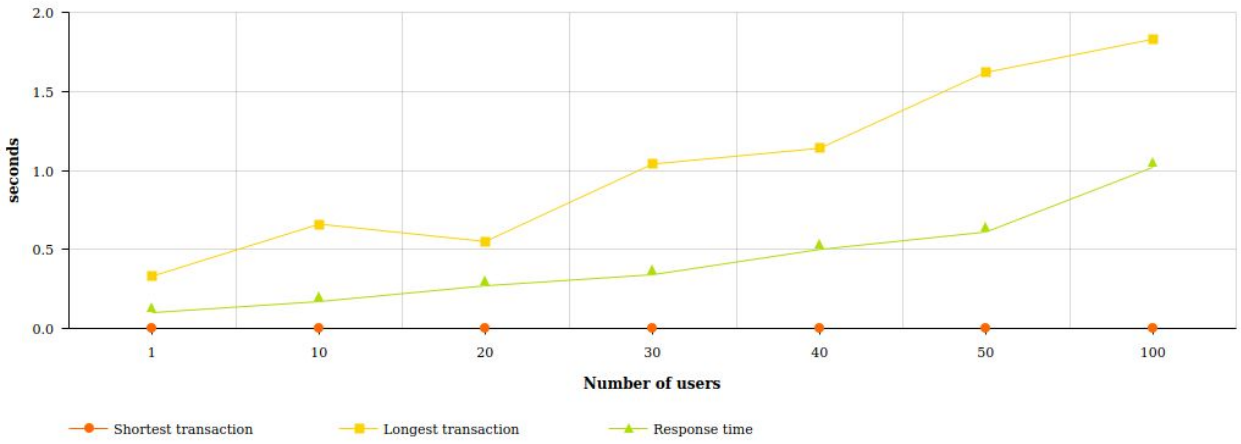


Figure 3: GET request for the index page **with** auto-scaling

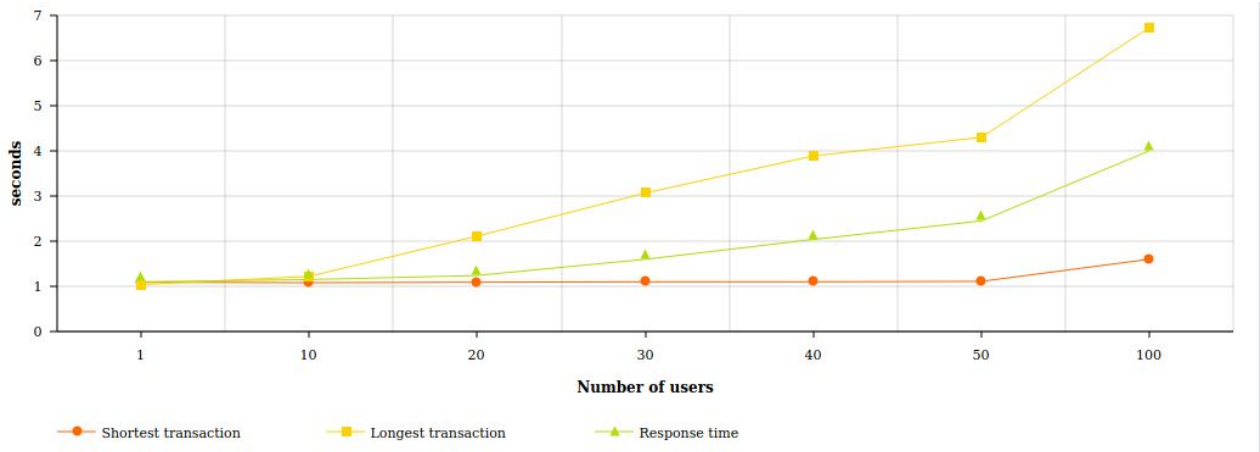
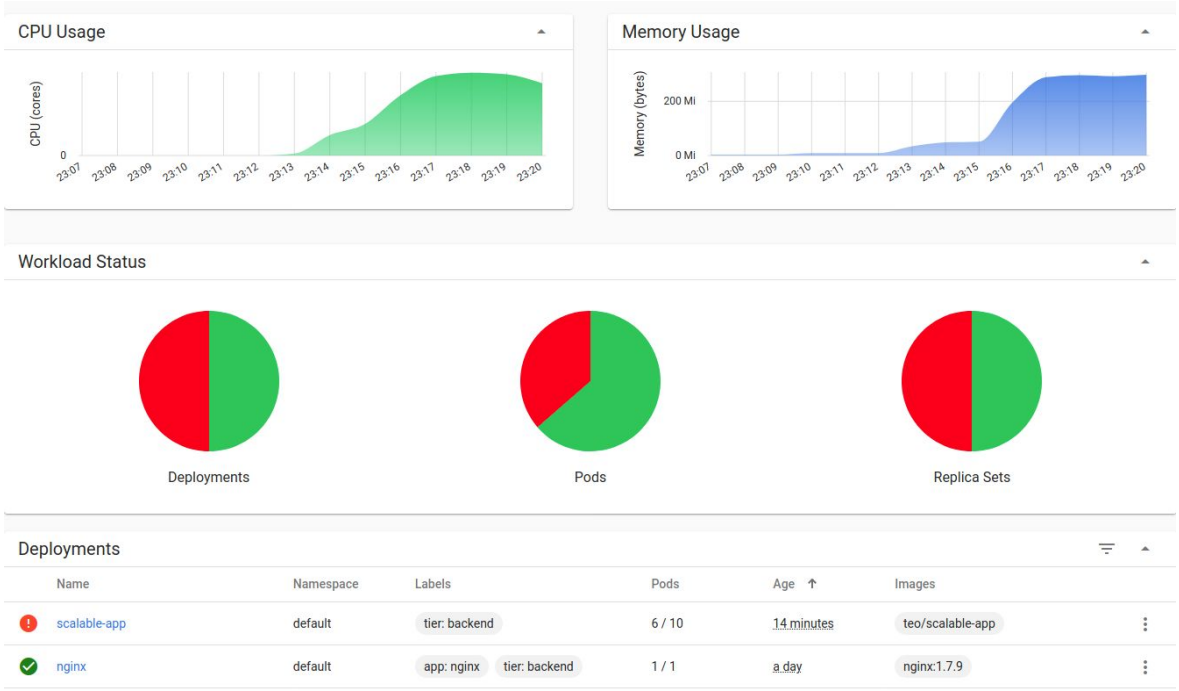


Figure 4: GET request for the /do-work route **with** auto-scaling

COMMENTS

- As we can see, the performance was very better with the auto-scaling feature.
    - In the simple work the longest transaction required more than 1.5 second to handle 100 users requests for one minute, unlike without the auto-scale feature it required more than 6 second.
    - Similar results came up on the /do-work test. Without auto-scale, the longest transaction took over 23 seconds for 100 users test. On the other hand, using the auto scale the longest transaction for the same amount of users only took about 6 seconds.
- Moreover, as we can see the application broke the rule of average response time to be 1 second. Maybe, that happened because the load of the test was too big to be handled, or maybe one minute was not enough for the Kubernetes to stabilize it.

- In the image below we can see the state of minikube from the monitoring dashboard, while the auto-scaling test was taking part. As we can see some pods of the application could not run smoothly and failed. That, may has happened from the inability of my computer to handle all that virtualization.
- Also, some unexpected performances may have been produced caused by the usage I did on the computer over the testing time (e.g. Figure 1: at 30 users test).
- Lastly, after some minutes from the end of the tests, the application successfully scale-down back to 1 replica.



CONCLUSION

To sum up, we saw how to deploy a web application using Kubernetes on a cluster-like environment, and how to enable its horizontal scale feature. Also, we proved that it works as a dedicated DevOps team that protects our application to go down, or to underperform.

Moreover, in this project I learned a lot of useful things about Container Orchestrators and how they help us to handle the complex requirements of the today's applications.

FUTURE WORK

- Deploy similar application over a real cluster on the Cloud.

- Connect the application with a state-full MySQL/Postgress Database deployed on the Cloud using a Cloud providers just like AWS or DigitalOcean.

Κ Α Λ Ο Κ Α Λ Ο Κ Α Ι Ρ Ι