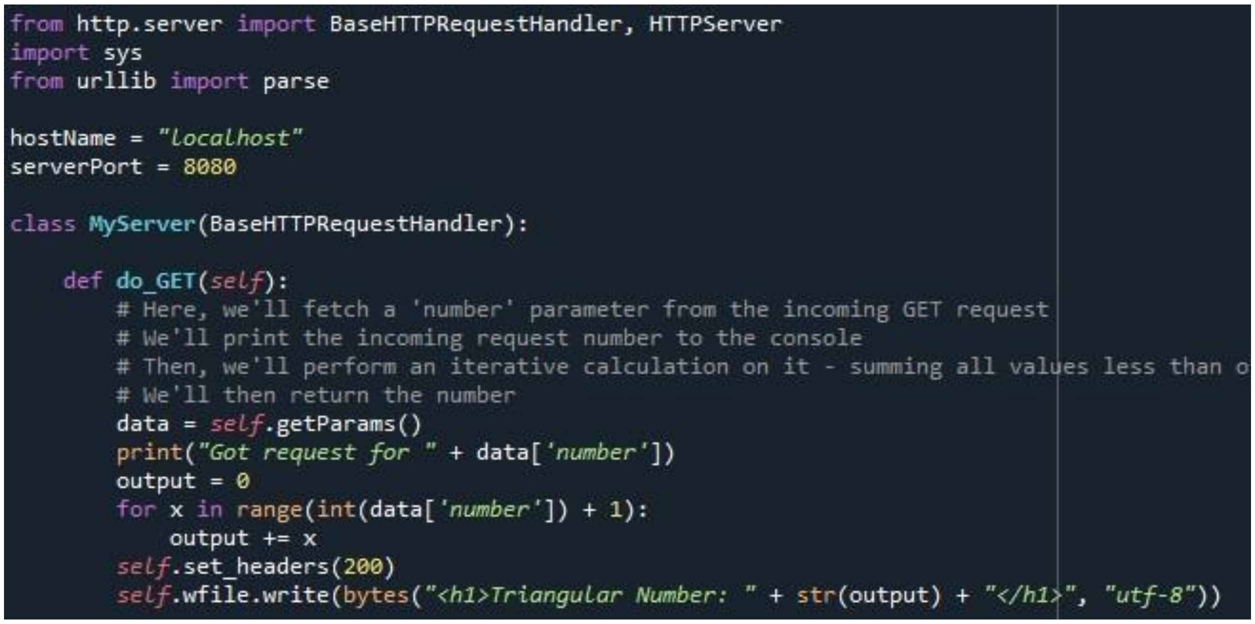
COSC 331
Lab 7: Deploying Docker with Google Kubernetes Engine (GKE)

We've already looked at building a ***Docker*** image using ***Docker Desktop***, but how can we host those containers in a scalable manner? The answer is using a ***cluster management software***, such as ***Kubernetes***.

In today's lab, we're going to use our previous lab knowledge to build a Docker image version of our Lab 6 triangle number service. We'll need to modify the underlying ***python*** application, then create a new ***Dockerfile*** to build it into an image. Then, we'll host that image using a ***Google Kubernetes Engine*** cluster, and we'll actually be able to see the dynamic cluster sizing and load balancing in real time. Note – this lab includes some material we haven't covered in the lectures yet, so you may want to wait to get started (although you should be able to follow the instructions).

Setting Up the Triangle Number Service

To start, open up your Lab 6 ***mathservice.py*** file. It should look something like this:

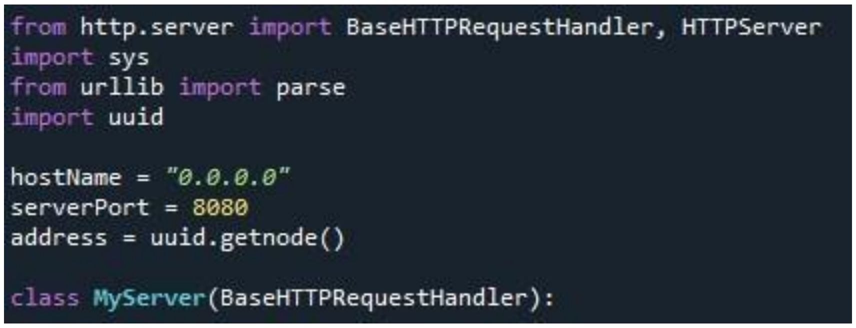


```python
from http.server import BaseHTTPRequestHandler, HTTPServer
import sys
from urllib import parse

hostName = "localhost"
serverPort = 8080

class MyServer(BaseHTTPRequestHandler):

    def do_GET(self):
        # Here, we'll fetch a 'number' parameter from the incoming GET request
        # We'll print the incoming request number to the console
        # Then, we'll perform an iterative calculation on it - summing all values less than o
        # We'll then return the number
        data = self.getParams()
        print("Got request for " + data['number'])
        output = 0
        for x in range(int(data['number']) + 1):
            output += x
        self.set_headers(200)
        self.wfile.write(bytes("<h1>Triangular Number: " + str(output) + "</h1>", "utf-8"))
```

In order to get this working with an actual online host, we'll need to make some minor modifications to this file. We'll also want to include a modification which will allow us to log which ***particular instance we've connected to***, since we're going to be using GKE's built-in load balancer.

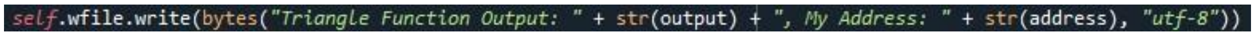
The first modification required is changing the ***hostName*** that is defined at the top of the file. We don't want to use ***localhost*** as our host name, because this application will actually be served on the World Wide Web. So instead, we're going to change our hostname to **"0.0.0.0"** instead.

To keep track of which instance we're connecting to, we'll use the ***MAC address*** of the instance. This is the actual physical hardware address for the instance. We can get the MAC address in Python easily using the ***uuid*** module and the ***getnode()*** function. This function will return the 48-bit hardware address of the machine, ***or*** a random 48-bit number. Both of these will work, we just need it as a way of differentiating between instances. Go ahead and ***import*** the uuid module, and add an extra line at the top of the file setting a variable ***address*** to the output of the ***getNode()*** function. It should look like this:



```python
from http.server import BaseHTTPRequestHandler, HTTPServer
import sys
from urllib import parse
import uuid

hostName = "0.0.0.0"
serverPort = 8080
address = uuid.getnode()

class MyServer(BaseHTTPRequestHandler):
```

As a final change, we'll have the server return the ***address*** variable, along with the calculated triangle number. This way, when we make a request, we'll be able to see the ***address*** in the response. This will let us know which instance we're connecting to, as each instance will have either a different MAC address, or a different randomly generated identifier. All we have to do is add it to the ***wfile.write()*** function:



```python
self.wfile.write(bytes("Triangle Function Output: " + str(output) + ", My Address: " + str(address), "utf-8"))
```

Once you're done this, ***create a new folder somewhere easy to access from the command prompt, and save your edited mathservice.py file there.***

Making Our Docker Image

Now, we're ready to create our Docker image. To do this, create a new ***Dockerfile*** inside the folder you just made. Remember that the file shouldn't have any file extension, and it needs to be named ***Dockerfile*** with a capital ***D***. Then, open up your Dockerfile using your text editor of choice.

For a base image, we'll use the latest ***Python 3*** base image that's available. You can do so by using the following ***FROM*** line:

FROM python:3

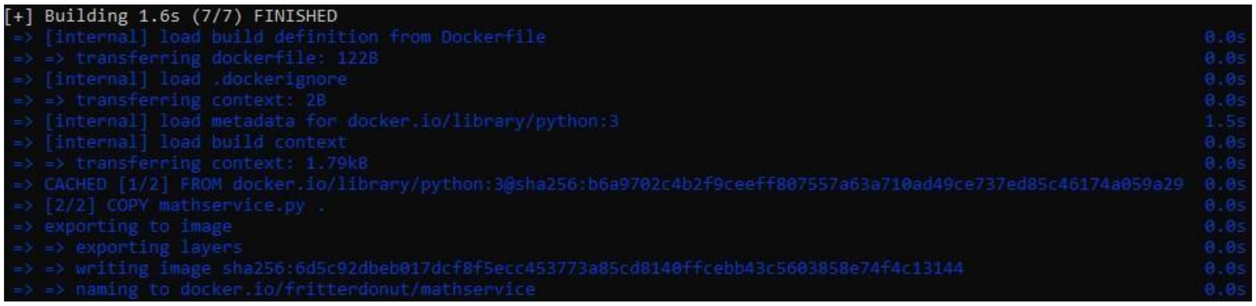
Then, using the same syntax as last lab, copy your ***mathservice.py*** file into your image. The file doesn't have to go anywhere in particular, so just place it in the ***current working directory***. You can reference the current working directory using the ***period*** (**.**) character.

Once you have copied the mathservice.py file into your image, you'll need to expose the port that the service is expecting to communicate on. This is the same port as the ***serverPort*** that we have defined in our Python file. Use the ***EXPOSE*** command to do this.

Finally, we'll need to actually start our mathservice.py file when we start up our container. This is done using the ***CMD*** command in your Dockerfile. The following syntax should work, assuming that you've placed the mathservice.py file in the current working directory:

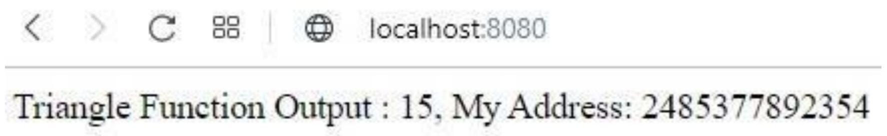
CMD ["python", "./mathservice.py"]

Save your Dockerfile, and open up a ***Windows Command Prompt***. In the command prompt, navigate to your new folder, and then use the ***docker build*** command like we did in the last lab to build your new docker image. Call your image ***mathservice***:



```
[+] Building 1.6s (7/7) FINISHED
 => [internal] load build definition from Dockerfile                                                       0.0s
 => => transferring dockerfile: 122B                                                                       0.0s
 => [internal] load .dockerignore                                                                          0.0s
 => => transferring context: 2B                                                                            0.0s
 => [internal] load metadata for docker.io/library/python:3                                                1.5s
 => [internal] load build context                                                                          0.0s
 => => transferring context: 1.79kB                                                                        0.0s
 => CACHED [1/2] FROM docker.io/library/python:3@sha256:b6a9702c4b2f9ceeff807557a63a710ad49ce737ed85c46174a059a29  0.0s
 => [2/2] COPY mathservice.py .                                                                            0.0s
 => exporting to image                                                                                     0.0s
 => => exporting layers                                                                                    0.0s
 => => writing image sha256:6d5c92dbeb017dcf8f5ecc453773a85cd8140ffcebb43c5603858e74f4c13144               0.0s
 => => naming to docker.io/fritterdonut/mathservice                                                        0.0s
```

Once you've built your image, try running it in a locally hosted container as we did last lab to verify that it works. Don't forget that you'll need to map the input to port 8080, rather than port 80 as we did in the previous lab with NGINX, since our mathservice expects to communicate on port 8080. Once the container is running, you should be able to visit ***localhost:8080/?number=5*** and see some output like the following:



Triangle Function Output : 15, My Address: 2485377892354

Take a screenshot of this output with your Docker command prompt in the screenshot as well. If you see output like this, that means your image is working properly and is ready to go. Finish up by pushing your new Docker image to ***Docker Hub***:

fritterdonut / mathservice
Updated a few seconds ago
Not Scanned
0
0
Public

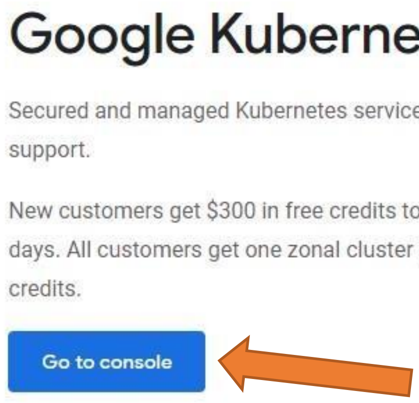
At this point, we've tested our Docker image locally, and pushed the completed image to our Docker Hub. We're now ready to actually deploy our image using GKE.

Making a Google Kubernetes Engine Cluster

First, head over to the Google Kubernetes Engine web page, which you can get to with the following link:

https://cloud.google.com/kubernetes-engine

You may need to register for the Google Cloud service and/or create a Google account. If so, do that now. Once you're registered, you should see a link to take you to the GKE console:

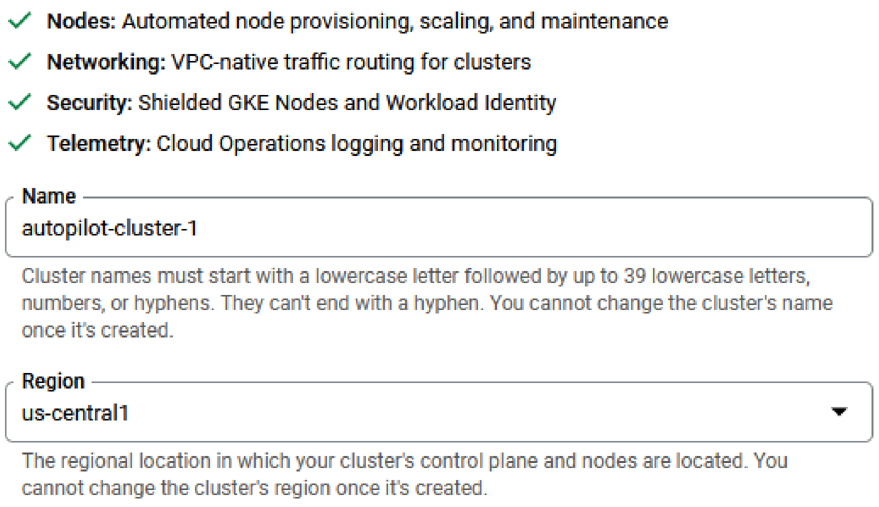

Google Kubernetes Engine

Secured and managed Kubernetes service with four-way auto scaling and multi-cluster support.

New customers get $300 in free credits to spend on Google Cloud during the first 90 days. All customers get one zonal cluster per month for free, not charged against your credits.

Once you're in the GKE console, you may need to create a new project. You can do this from the project menu at the top, next to the Google Cloud Platform banner. Once you've created the project, you will need to hit the ***Enable*** button that appears to enable the Kubernetes API. Once you have a new project with Kubernetes enabled, click on the ***Clusters*** menu option in the sidebar menu, and click ***Create Cluster***.

You may name your cluster whatever you wish. You should be able to skip through the remaining cluster options like networking. The ***autopilot*** cluster will take care of most of the scaling and availability concerns for you.


- ✓ **Nodes:** Automated node provisioning, scaling, and maintenance
- ✓ **Networking:** VPC-native traffic routing for clusters
- ✓ **Security:** Shielded GKE Nodes and Workload Identity
- ✓ **Telemetry:** Cloud Operations logging and monitoring

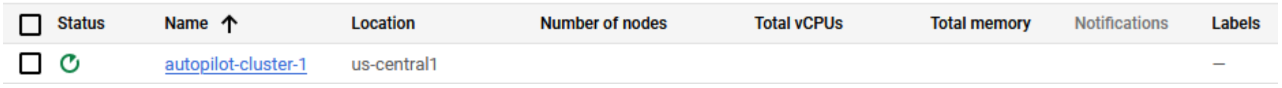
Name: autopilot-cluster-1

Cluster names must start with a lowercase letter followed by up to 39 lowercase letters, numbers, or hyphens. They can't end with a hyphen. You cannot change the cluster's name once it's created.

Region: us-central1

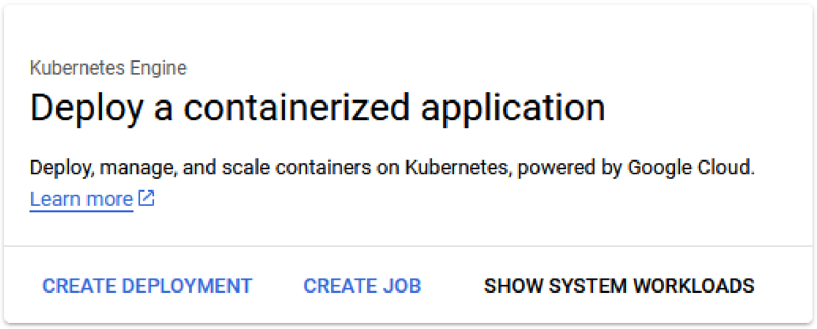
The regional location in which your cluster's control plane and nodes are located. You cannot change the cluster's region once it's created.

After you've set the machine type, click the ***Create*** button at the bottom of the page to create your new cluster. This may take some time to initialize, wait for the cluster creation to complete before continuing.



Status	Name ↑	Location	Number of nodes	Total vCPUs	Total memory	Notifications	Labels
✓	autopilot-cluster-1	us-central1					–

Once your cluster is built, click on the ***Workloads*** tab in the sidebar, and then select the ***Deploy*** button on the prompt it provides:

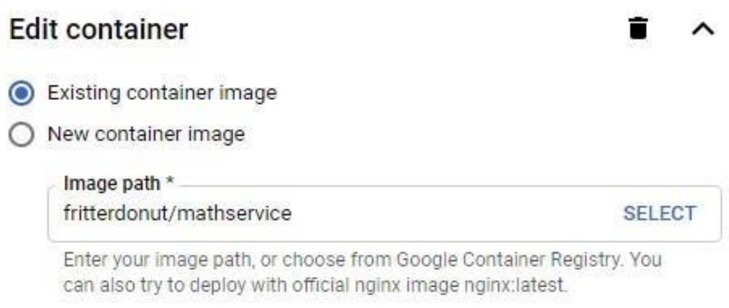


Kubernetes Engine

Deploy a containerized application

Deploy, manage, and scale containers on Kubernetes, powered by Google Cloud.
Learn more

You can name the deployment whatever you want. Make sure it's using your autopilot cluster. For the ***image path***, use your ***Docker Hub*** username followed by ***/mathservice***:



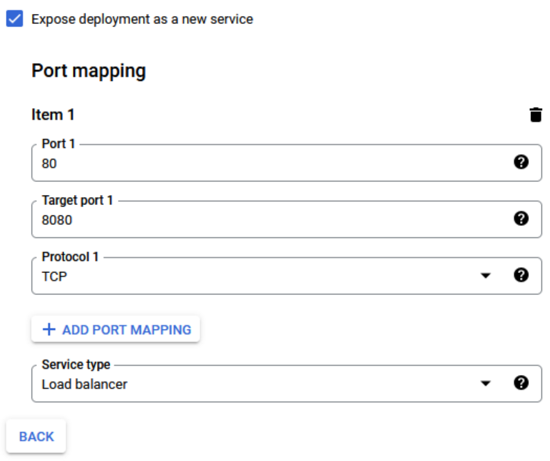
Edit container

(●) Existing container image
( ) New container image

Image path *: fritterdonut/mathservice

Enter your image path, or choose from Google Container Registry. You can also try to deploy with official nginx image nginx:latest.

Then hit the ***Next: Expose*** button. Check the checkbox and fill in the target port with ***8080*** (the same as our Python service), and make sure the ***Service type*** is ***load balancer***. This will automatically create a load balanced entrypoint to our application – the client will connect on the standard ***HTTP port*** (80), and the requests will be routed to port ***8080*** of one of our cluster instances.



☑ Expose deployment as a new service

Port mapping

Item 1

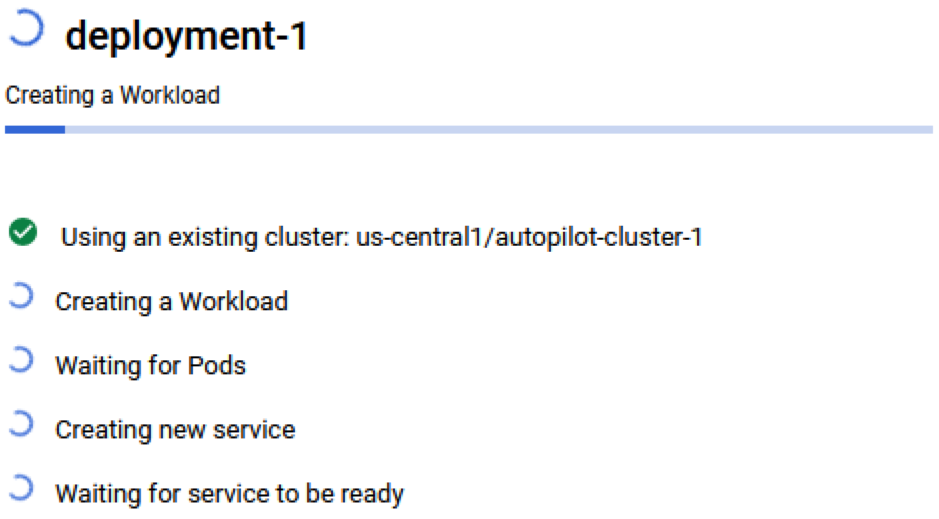
Port 1: 80

Target port 1: 8080

Protocol 1: TCP

Service type: Load balancer

Then click the ***Deploy*** button at the bottom. GKE should begin deploying your new workload. This may take a while. If you check the ***Workloads*** tab again, you may see an error or warning status – wait for these to disappear.

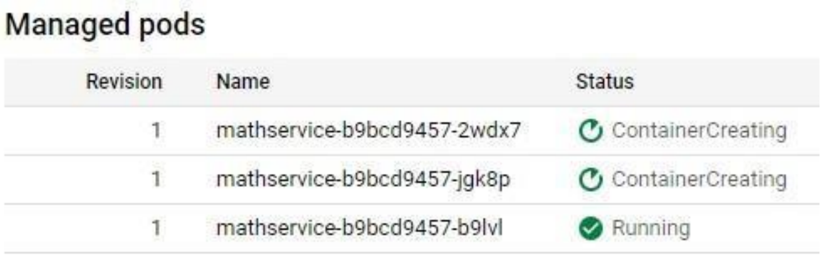


deployment-1

Creating a Workload

- ✓ Using an existing cluster: us-central1/autopilot-cluster-1
- Creating a Workload
- Waiting for Pods
- Creating new service
- Waiting for service to be ready

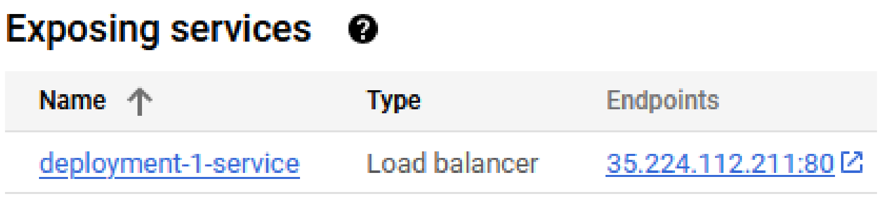
It may take a while for all pods in your deployment to finish creating. You can see the status of each individual pod by clicking the name of your workload, then scrolling down the workload page to the ***Managed Pods*** section:



Managed pods

Revision	Name	Status
1	mathservice-b9bcd9457-2wdx7	ContainerCreating
1	mathservice-b9bcd9457-jgk8p	ContainerCreating
1	mathservice-b9bcd9457-b9lvl	✓ Running

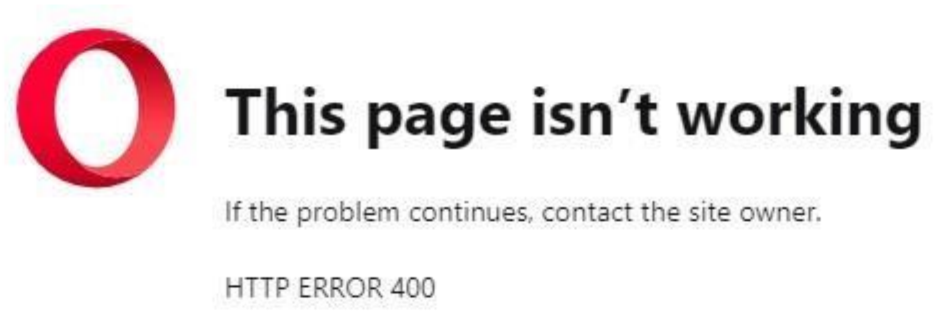
At this point, your cluster is running, your containers have been deployed, and your load balancer should be online. This means your service should now be accessible via the World Wide Web. Look under the ***Exposing services*** header at the bottom of the workloads page and find the Load Balancer ***endpoint***. Click on the link to visit your service.



Exposing services

Name	Type	Endpoints
deployment-1-service	Load balancer	35.224.112.211:80

If everything is working correctly, you should see something like the following:

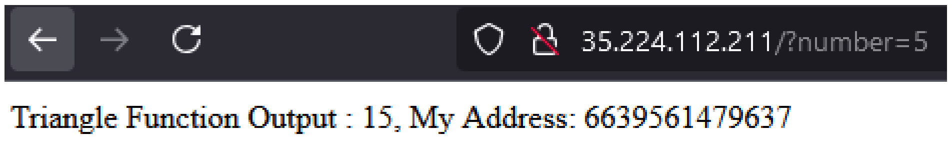


This page isn't working

If the problem continues, contact the site owner.

HTTP ERROR 400

D'oh! This is because our mathservice.py just returns a 400 error when no input has been given to the service. Try adding ***?number=5*** to the end of the link, and you should see that your service is actually working:

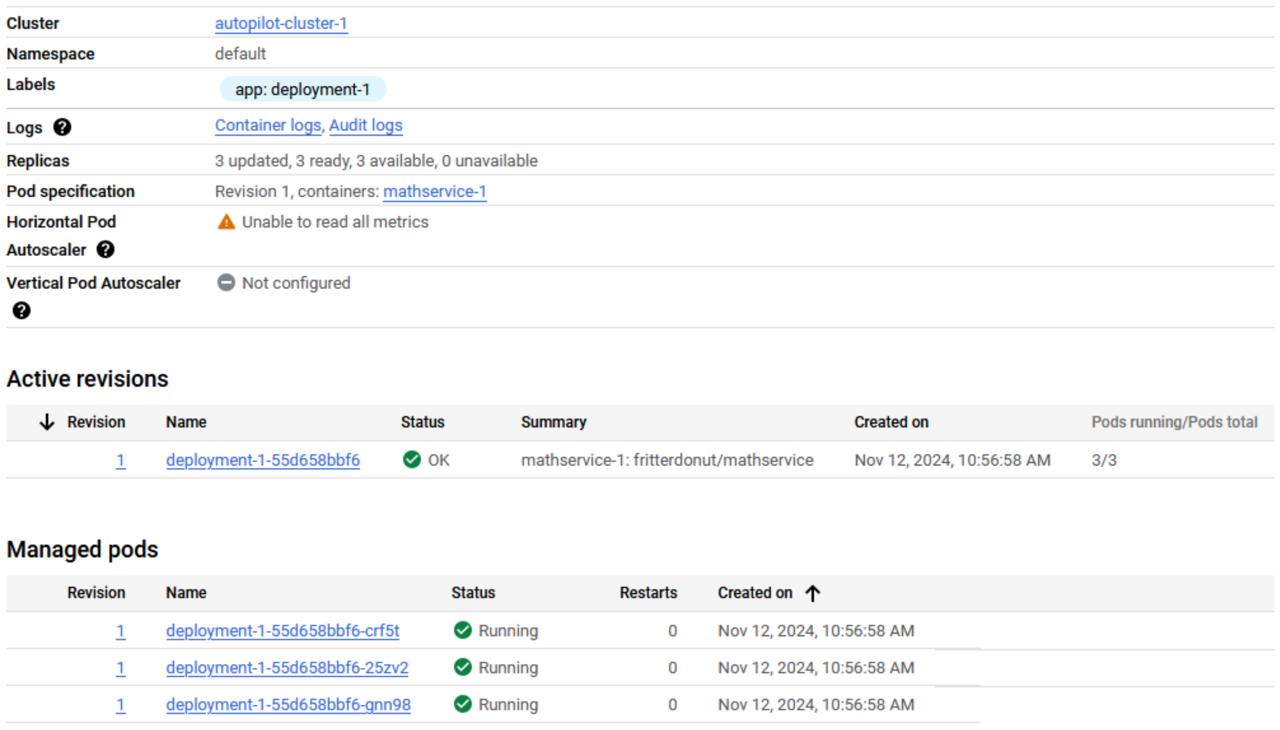


```
35.224.112.211/?number=5
```

```
Triangle Function Output : 15, My Address: 6639561479637
```

Congratulations, you now have now successfully deployed your new Docker image into a clustered, load-balanced, online microservice!

Go back to the workload page for your service, and ***take a screenshot that looks like the screenshot below:***



Cluster	autopilot-cluster-1
Namespace	default
Labels	app: deployment-1
Logs	Container logs, Audit logs
Replicas	3 updated, 3 ready, 3 available, 0 unavailable
Pod specification	Revision 1, containers: mathservice-1
Horizontal Pod Autoscaler	Unable to read all metrics
Vertical Pod Autoscaler	Not configured

Active revisions

Revision	Name	Status	Summary	Created on	Pods running/Pods total
1	deployment-1-55d658bbf6	✓ OK	mathservice-1: fritterdonut/mathservice	Nov 12, 2024, 10:56:58 AM	3/3

Managed pods

Revision	Name	Status	Restarts	Created on
1	deployment-1-55d658bbf6-crf5t	✓ Running	0	Nov 12, 2024, 10:56:58 AM
1	deployment-1-55d658bbf6-25zv2	✓ Running	0	Nov 12, 2024, 10:56:58 AM
1	deployment-1-55d658bbf6-gnn98	✓ Running	0	Nov 12, 2024, 10:56:58 AM

Your screenshot should include your general cluster information (at the top), as well as your ***Managed pods*** section and your ***Exposing services*** section, showing that you have a working load balancer with an external endpoint and that your pods are running properly.

Looking at Cluster Auto-Scaling

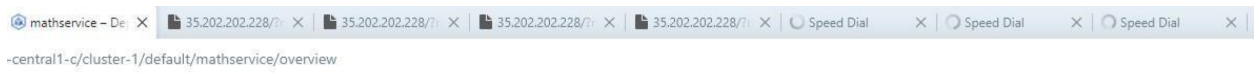
You may notice that your deployment originally started with three pods, but if no requests are made to your service for a while, the ***number of pods will eventually drop down to one***, as you can see in the previous screenshot.

This is due to the automatic horizontal scaling that is provided by the cluster. When the workload is low, it will wind-down unneeded pods to save resources. When load is high, it will do the opposite.

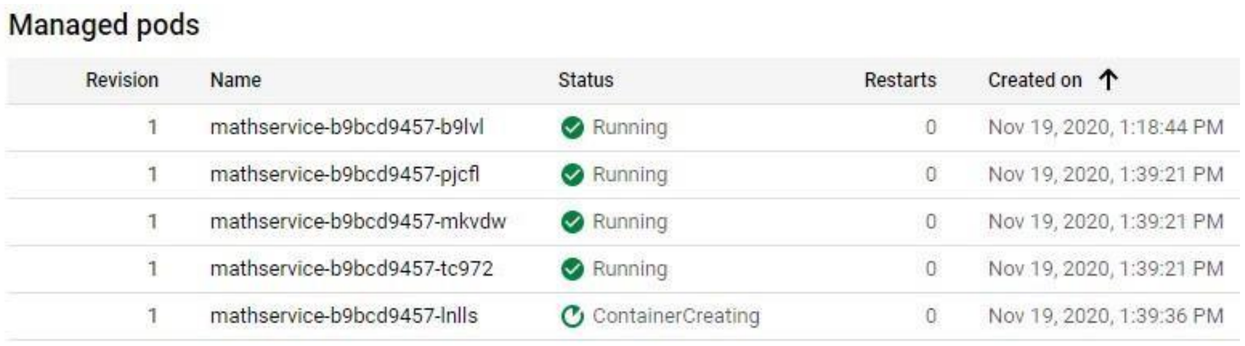
In order to generate a load, use a ***number*** that is in the range of around 100000000. Your link should look something like this:

http://<my load balancer IP>/?number=100000000

Try running this once – it should take a couple of seconds to respond. Then try copying this link, and rapidly opening tabs and pasting it into the tab before hitting ***enter***. This should send a bunch of requests to your load balancer:



If you go back to your workloads page for your cluster, you should notice that your cluster is ***creating new pods*** to help deal with the load:



Managed pods

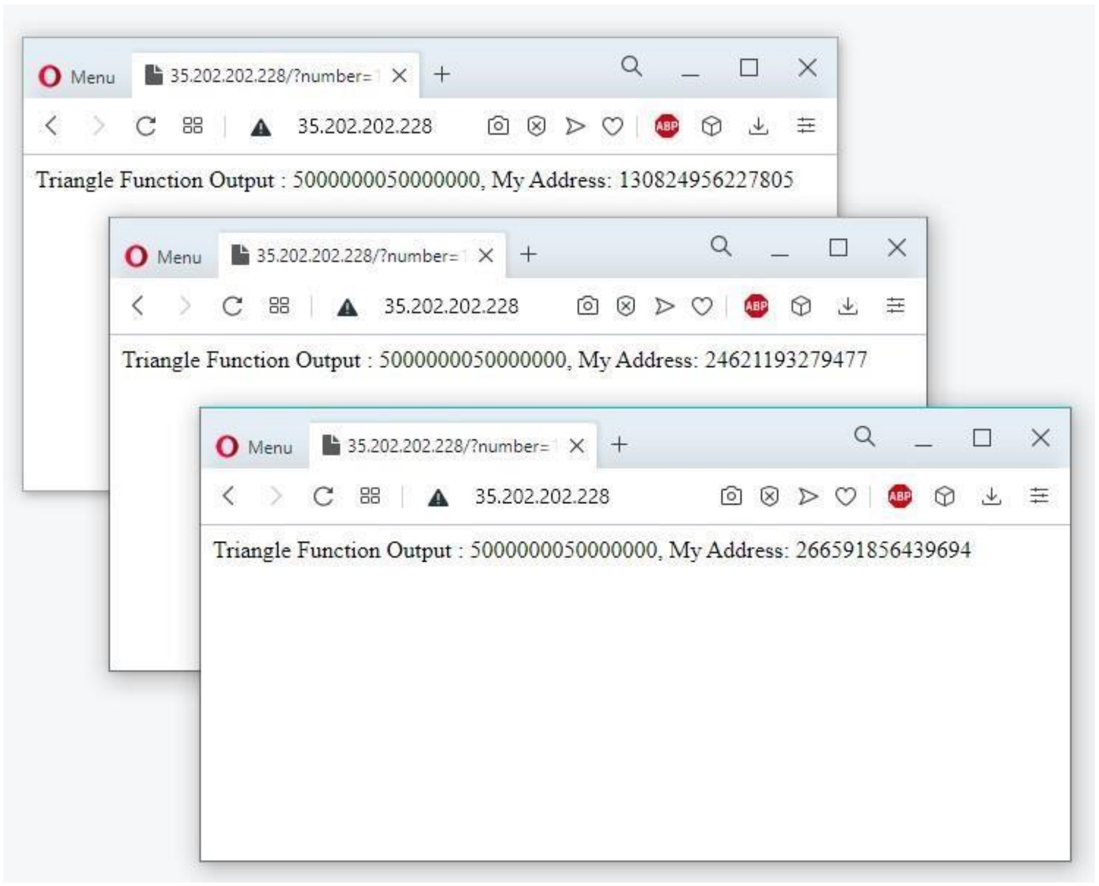
Revision	Name	Status	Restarts	Created on ↑
1	mathservice-b9bcd9457-b9lvl	✓ Running	0	Nov 19, 2020, 1:18:44 PM
1	mathservice-b9bcd9457-pjcfl	✓ Running	0	Nov 19, 2020, 1:39:21 PM
1	mathservice-b9bcd9457-mkvdw	✓ Running	0	Nov 19, 2020, 1:39:21 PM
1	mathservice-b9bcd9457-tc972	✓ Running	0	Nov 19, 2020, 1:39:21 PM
1	mathservice-b9bcd9457-lnlls	ContainerCreating	0	Nov 19, 2020, 1:39:36 PM

Take a screenshot showing that your cluster is resizing itself, similar to the screenshot above.

If you go back and take a look at some of the returned information, you should notice that once the new pods have been added to the pool, you'll start to be able to see them responding to requests: they'll return ***a different address***.

This is because once the new pods are fully started, the load balancer will begin to redirect incoming traffic to the new pods, while the already-created pods are currently busy. Since the cluster by default has a maximum number of 5 pods, you should eventually see up to five different addresses returned by your service.

Find at least three responses that display different addresses, and take a screenshot of them together like the following:

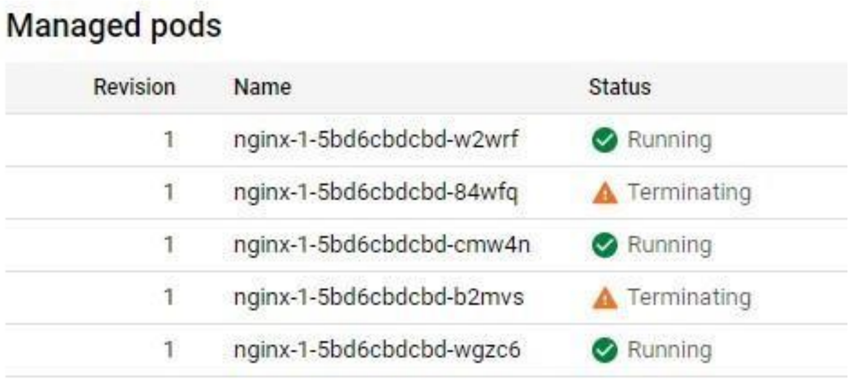


Triangle Function Output : 5000000050000000, My Address: 130824956227805

Triangle Function Output : 5000000050000000, My Address: 24621193279477

Triangle Function Output : 5000000050000000, My Address: 266591856439694

Congratulations! Your new microservice really is scalable now. You may also notice that after this exercise, if you allow your cluster to sit for a time without any additional load, that the cluster will begin terminating the extra pods, as they are no longer needed now that the load has dropped:



Managed pods

Revision	Name	Status
1	nginx-1-5bd6cbdcbd-w2wrf	✓ Running
1	nginx-1-5bd6cbdcbd-84wfq	⚠ Terminating
1	nginx-1-5bd6cbdcbd-cmw4n	✓ Running
1	nginx-1-5bd6cbdcbd-b2mvs	⚠ Terminating
1	nginx-1-5bd6cbdcbd-wgzc6	✓ Running

Submission

Go ahead and submit your screenshots from this lab on Moodle. ***DON'T FORGET TO SHUT DOWN YOUR KUBERENETES CLUSTER AFTER YOU ARE DONE. You must remove the existing workloads and then delete the cluster.***

Marks	Item
2 Marks Each	Four screenshots: One showing your locally hosted docker container, one showing your completed cluster and balancer setup, one showing your cluster resizing itself, and one showing the different addresses that your service responds with.