Project Report
SRE-TEST

Abdul Hannan Ghafoor

Background

In response to the challenges faced by our colleague Osama in manually resizing and watermarking a high volume of DSLR images for web use, our team developed an automated solution. This initiative aimed to address the inefficiencies and inconsistencies noted by our Site Reliability Engineering (SRE) team, such as missing image formats due to time constraints and the tedious nature of the task. The solution, designed to be scalable and user-friendly, significantly reduces the manual effort required, ensuring consistent image processing and enhancing operational efficiency across our digital platforms.

Tools:

1. Kubernetes/GKE
2. GCP
3. NextJS
4. Flask
5. Prometheus
6. Grafana.
7. K6

Architecture.

The solution is based on NextJS application for the demo and the Flask application which does the resizing part of the images. API routes are developed to match the format given in the project statement. Osama now just have to mention the route along with the desired paramters to get the image resized and watermarked.

For the demo application, the main page just shows three resized versions of the same source image in different sizes and quality. This application takes the source image URL from the environment variables set in the server and renders the images on the server side. In order to make this application more modular in nature, IP address of the flask application, which is serving as the backend, is also provided in terms of environment variable.

For the deployment of the whole solution, GKE standard and the autopilot modes were considered. Each mode had their own advantages and disadvantages, but in the end autopilot was chosen because of its more flexibility and lower cost.

The architecture of the whole solution is given in the diagram below.

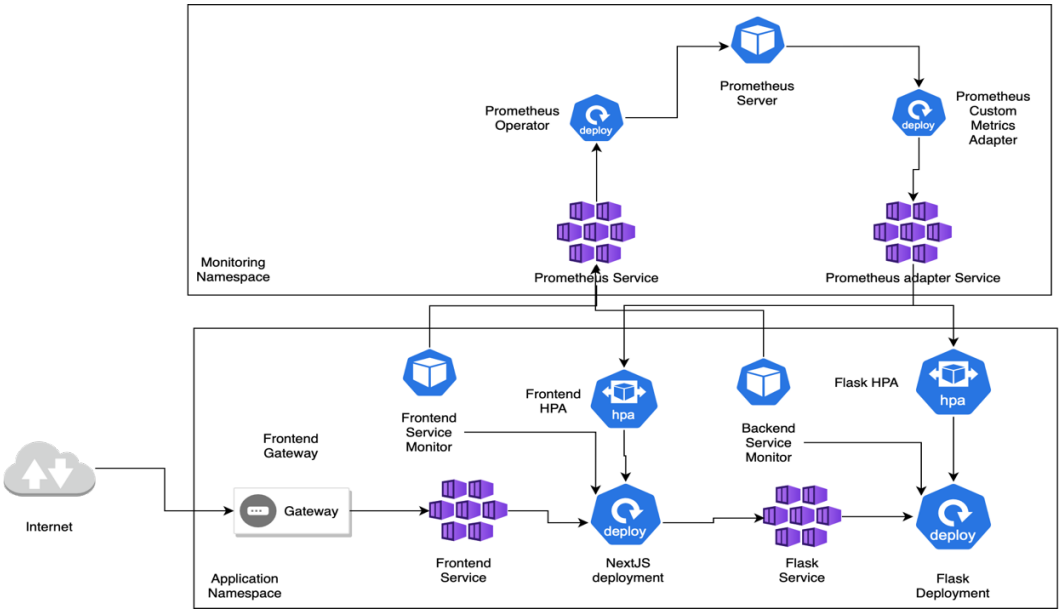

Fig 1. Architectural Diagram of Kubernetes Deployment

The solution is divided into two separation of concerns. One of them is monitoring and the other one is application. This separation of concerns is reflected in the form of different namespaces, application and monitoring.

The Nextjs application, serving as the frontend application, is deployed on the Kubernetes deployment resource within application namespace. It is getting the address of the backend service through the environment variables stored in the configmap. The source image url is also stored in the configmap and passed to the frontend application through the environment variables. This makes the application more flexible and modular in nature. The frontend application is serving on the port 3000. In front of the frontend deployment, is the frontend service. A gateway is deployed in front of this service to provide the connected with the internet. There are two ways through which a connection to the internet can be provided here. One way is through gateway resource, and the other way is through the LoadBalancer service which deploys the external LoadBalancer. The reason of choosing the gateway resource is because of the experimentation of different types of Autoscaling configuration deployments. Gateway resource provides integration with the google managed Prometheus service, which is deployed by GCP in the GMP-system namespace, through which we can use different types of metrics to provide autoscaling features. LoadBalancer does not provide those metrics. In the end, we were able to deploy our own Prometheus server and collect metrics through our own service monitors. The collected metrics are application level metrics which provides more control than the gateway level metrics. These application level metrics are integrated within both nextjs applications and the flask application and are available at the /metrics API. There is a HTTP route between the gateway and the frontend service to provide the connectivity of the internet to the frontend application.

Frontend application is connected with the Backend application through the Backend service. Flask application is also deployed on the Kubernetes deployment resource. There are two Service Monitors that are deployed within the application namespace. One is

gathering the metrics from the frontend application through /metrics API while the other one is getting the metrics from the backend application through /metrics API implemented within these applications. The polling interval is set to 2 seconds so that latest changes are reflected in the Prometheus server as soon as they occur.

In the monitoring namespace, Prometheus operator, server and the adapters are deployed. Prometheus server collects the metrics through the service monitor which are then available to the HorizontalPodAutoscaler to trigger the autoscaling actions. The HorizontalPodAutocaler or HPA are deployed in the application namespace gets the metrics from the Prometheus adapter. The metric used for scaling actions are requests per second more specifically, python_requests_total for flask application and the http_requests_total for the nextjs application. Thus, the pod scaling actions are taken on the number of requests per seconds handled by the pods.

Multicluster deployment.

For the multicluster deployment, we can use different techniques and tools to make sure that all our clusters are coherent in terms of deployment and the traffic is evenly distributed among different clusters.

For deployment, we can use either ArgoCD or FluxCD to make sure that any deployments across the clusters are consistent with each other. We can also deploy an external multi cluster gateways to route traffic to clusters deployed in one or different regions. We can do weight based traffic splitting and route traffic to appropriate clusters.

Google API metrics vs Application specific metrics.

While google API metrics provide a lot of information, some metrics must be calculated at the application to make better scaling decisions. GCP provides loadbalancer based metrics especially the metric '**autoscaling.googleapis.com|gclb-capacity-utilization'** which basically collects the metric from the Kubernetes Service calculating the network traffic in terms of request per second per pod. This requires us to apply rate limiter at the services. While this works for the load balancer facing services. This does not work for the backend services as load balancer is not connected with them. We can always guess what the backend utilization would be based on the traffic coming at the frontend application. However, in a more complex application architecture, the traffic from the frontend to the backend would be more dynamic in nature and therefore, any metric that will be used on the GCP level would be just an estimate of what that traffic would be instead of based on actual traffic.

Because of the reason mentioned above, we have implemented the application level metrics both on the frontend and the backend and deployed our own Prometheus for coherency of the solution.

Load Test

Different architectural decisions have different performance issues which are evident from the below mentioned load tests. It is impossible to test every possible configuration because the limited resources. However, with few smart tests, we can establish the trends and therefore extrapolate on the basis of the results. To narrow down the scope of the tests, we considered different factors including

1. Number of pods per node
2. Number of nodes.
3. Auto provisioning nodes/node scale up
4. Application level metrics vs GKE level metrics.
5. Autopilot vs Standard GKE clusters.

Number of pods per node is a really important number to figure out as the Performance can change based on the number of pods on one node. In order to understand the number of pods on a node which would give the optimized performance, we have to restrict the hardware machines used as a worker node within the cluster.

Number of nodes is also important because after figuring out the optimized number of pods on a single node, it is also important that if the performance scales appropriately with the increase in number of nodes.

Node scale up time is an indicator how fast the system copes up to the changing load. If there is a surge where the load changes from 500 Request per second to 100,000 request per second in a matter of seconds, then how would the system react to that surge. While that question is really important, the hardware resources required to perform such testing would be immense, hence this type of scalability and surge testing should be conducted later.

Application level metrics and GKE level metrics, might only be useful for the frontend application. For GKE level metrics, like Loadbalancer utilizations, we have to implement the rate limiter on the service which does not seem like a good idea as we are not sure if the pods are fully utilized or not. If there are too many pods, then there might come some routing delays which would decrease the throughput.

Autopilot cluster vs Standard cluster is an important consideration as well. But considering that the autopilot cluster charges more money for the same set of resources as shown in the cost analysis section, it does not make sense to consider it a financially viable option.

So we have performed load tests to determine to the number of pods and number of the nodes that would match our current extimated maximum load, average load and minimum load. The results from these load tests would eventually be useful for cost estimation for running the whole application. The testing tool used for the load test is Grafana's K6. Different parameters were considered for testing. Those parameters are mentioned below.

The script profile is the following.

1. 300 virtual users

2. 30 seconds as load incremental period.
3. 2 mins of the maintained request per second load at 100, 1000 and 10000 request per second

The script is given below.

```
import http from "k6/http";


export const options = {
  scenarios: {
    contacts: {
      executor: 'ramping-arrival-rate',
      preAllocatedVUs: 300,
      timeUnit: '1s',
      startRate: 50,
      stages: [
        { target: 10000, duration: '30s' },
        { target: 10000, duration: '2m' },
      ],
    },
  },
};

export default function () {
  const response = http.get("http://34.128.149.125");
}
```

Based on the parameter above, following two test case scenarios have been explored.

1. Optimum number of pods on a single node having fixed hardware specs against a fixed load (traffic)
2. Once having determined the optimum number of pods on a single node, changing the number of nodes to observe the performance difference by keeping the number of pods per node constant.

In both scenarios, standard cluster is deployed with auto-provisioning nodes disabled. Load test is done 10000 request per second from 300 virtual users. The hardware resources that were used in this scenario were 940m vCPU and 2.95GB RAM per node.

Name	Status	CPU requested	CPU allocatable	Memory requested	Memory allocatable	Storage requested	Storage allocatable
gke-sre-test-default-pool-04398b3c-25lj	Ready	805 mCPU	940 mCPU	1.07 GB	2.95 GB	0 B	0 B

Testing Scenario 1:

Test scenario 1 is about determining the optimum number of pods on a single node having fixed hardware specs against the fixed load. The pods did not have rate limitation and they

were free to serve as many requests as possible. Auto provisioning was disabled and application-level metrics were used for this scenario. In each test, the virtual users were fixed at 300 while the number of requests per second was limited to 10000RPS. In this particular tests, the number of pods were changed to get the maximum performance in terms of requests per second that one node can achieve. While the HPA pod was available and kept on checking on the metrics, the maximum and minimum pods remain fixed to rule out any scaling delays. The test was repeated against 1 pods, 2 pods, 3 pods, 4 pods, 5 pods, 8 pods, 10pods, 15pods.

Following are the results of each test along with their screenshots.

Test case 1: Pod = 1

test score:

```
  scenarios: (100.00%) 1 scenario, 300 max VUs, 3m0s max duration (incl. graceful stop):
           * contacts: Up to 10000.00 iterations/s for 2m30s over 2 stages (maxVUs: 300, gracefulStop: 30s)

WARN[0002] Insufficient VUs, reached 300 active VUs and cannot initialize more  executor=ramping-arrival-rate scenario=contacts

     data_received..................: 175 MB  1.2 MB/s
     data_sent......................: 6.0 MB  40 kB/s
     dropped_iterations.............: 1274717 8443.350381/s
     http_req_blocked...............: avg=64.47µs  min=291ns    med=875ns    max=324.41ms p(90)=1.79µs   p(95)=2.45µs
     http_req_connecting............: avg=62.86µs  min=0s       med=0s       max=324.35ms p(90)=0s       p(95)=0s
     http_req_duration..............: avg=588.21ms min=221.88ms med=368.35ms max=10.41s   p(90)=555.23ms p(95)=2.66s
       { expected_response:true }...: avg=588.21ms min=221.88ms med=368.35ms max=10.41s   p(90)=555.23ms p(95)=2.66s
     http_req_failed................: 0.00%   ✓ 0          ✗ 76004
     http_req_receiving.............: avg=5.6ms    min=5.2µs    med=1.75ms   max=691.96ms p(90)=4.53ms   p(95)=7.15ms
     http_req_sending...............: avg=15.3µs   min=1.62µs   med=4.45µs   max=20.01ms  p(90)=16.54µs  p(95)=49.41µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=582.59ms min=221.06ms med=364.07ms max=10.41s   p(90)=537.96ms p(95)=2.65s
     http_reqs......................: 76004   503.428135/s
     iteration_duration.............: avg=588.31ms min=222.46ms med=368.39ms max=10.41s   p(90)=555.31ms p(95)=2.66s
     iterations.....................: 76004   503.428135/s
     vus............................: 5       min=5        max=300
     vus_max........................: 300     min=300      max=300


running (2m31.0s), 000/300 VUs, 76004 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/300 VUs  2m30s  10000.00 iters/s
```

Node utilization

```
Every 2.0s: kubectl top nodes

NAME                                        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gke-sre-test-default-pool-04398b3c-25lj     1344m        142%   1729Mi          61%
```

HPA Statistics:

```
Every 2.0s: kubectl -n application get hpa

NAME              REFERENCE                 TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
sre-flask-hpa     Deployment/sre-flask      0/500m       1         100       1          4d11h
sre-frontend-hpa  Deployment/sre-frontend   255238m/2    1         1         1          3d10h
```

Test case 2: Pod = 2

Test score

```
  scenarios: (100.00%) 1 scenario, 300 max VUs, 3m0s max duration (incl. graceful stop):
           * contacts: Up to 10000.00 iterations/s for 2m30s over 2 stages (maxVUs: 300, gracefulStop: 30s)

WARN[0002] Insufficient VUs, reached 300 active VUs and cannot initialize more  executor=ramping-arrival-rate scenario=contacts

     data_received..................: 219 MB  1.5 MB/s
     data_sent......................: 7.5 MB  50 kB/s
     dropped_iterations.............: 1255560 8318.013582/s
     http_req_blocked...............: avg=36.78µs  min=291ns    med=833ns    max=29.16ms  p(90)=1.75µs   p(95)=2.37µs
     http_req_connecting............: avg=35.29µs  min=0s       med=0s       max=29.12ms  p(90)=0s       p(95)=0s
     http_req_duration..............: avg=469.61ms min=219.61ms med=331.78ms max=7.51s    p(90)=586.95ms p(95)=1.11s
       { expected_response:true }...: avg=469.61ms min=219.61ms med=331.78ms max=7.51s    p(90)=586.95ms p(95)=1.11s
     http_req_failed................: 0.00%   ✓ 0          ✗ 95160
     http_req_receiving.............: avg=3.65ms   min=5.12µs   med=1.44ms   max=449.89ms p(90)=3.99ms   p(95)=5.12ms
     http_req_sending...............: avg=13.81µs  min=1.58µs   med=4.33µs   max=32.97ms  p(90)=14.7µs   p(95)=39.08µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=465.95ms min=219.01ms med=329.03ms max=7.51s    p(90)=575.75ms p(95)=1.11s
     http_reqs......................: 95160   630.429587/s
     iteration_duration.............: avg=469.68ms min=219.64ms med=331.83ms max=7.51s    p(90)=586.98ms p(95)=1.11s
     iterations.....................: 95160   630.429587/s
     vus............................: 300     min=87       max=300
     vus_max........................: 300     min=300      max=300


running (2m30.9s), 000/300 VUs, 95160 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/300 VUs  2m30s  10000.00 iters/s
```

Node utilization

```
Every 2.0s: kubectl top nodes

NAME                                        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gke-sre-test-default-pool-04398b3c-25lj     1970m        209%   1888Mi          67%
```

HPA statistics

```
Every 2.0s: kubectl -n application get hpa

NAME              REFERENCE                   TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
sre-flask-hpa     Deployment/sre-flask        0/500m      1         100       1          4d11h
sre-frontend-hpa  Deployment/sre-frontend     154188m/2   2         2         2          3d11h
```

Test case 3: Pods =3

Test Score:

```
  scenarios: (100.00%) 1 scenario, 300 max VUs, 3m0s max duration (incl. graceful stop):
           * contacts: Up to 10000.00 iterations/s for 2m30s over 2 stages (maxVUs: 300, gracefulStop: 30s)

WARN[0002] Insufficient VUs, reached 300 active VUs and cannot initialize more  executor=ramping-arrival-rate scenario=contacts

     data_received..................: 223 MB  1.5 MB/s
     data_sent......................: 7.7 MB  51 kB/s
     dropped_iterations.............: 1253845 8323.052987/s
     http_req_blocked...............: avg=38.15µs  min=291ns    med=833ns    max=33.3ms   p(90)=1.7µs    p(95)=2.33µs
     http_req_connecting............: avg=36.62µs  min=0s       med=0s       max=33.28ms  p(90)=0s       p(95)=0s
     http_req_duration..............: avg=461.19ms min=220.03ms med=347.68ms max=6.76s    p(90)=662.2ms  p(95)=1.03s
       { expected_response:true }...: avg=461.19ms min=220.03ms med=347.68ms max=6.76s    p(90)=662.2ms  p(95)=1.03s
     http_req_failed................: 0.00%   ✓ 0          ✗ 96863
     http_req_receiving.............: avg=2.96ms   min=5.16µs   med=1.86ms   max=363.29ms p(90)=4.37ms   p(95)=5.69ms
     http_req_sending...............: avg=11.47µs  min=1.62µs   med=4.29µs   max=11.8ms   p(90)=13.16µs  p(95)=31.41µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=458.21ms min=218.99ms med=345.17ms max=6.76s    p(90)=655.31ms p(95)=1.02s
     http_reqs......................: 96863   642.978902/s
     iteration_duration.............: avg=461.26ms min=220.05ms med=347.74ms max=6.76s    p(90)=662.33ms p(95)=1.03s
     iterations.....................: 96863   642.978902/s
     vus............................: 300     min=89       max=300
     vus_max........................: 300     min=300      max=300


running (2m30.6s), 000/300 VUs, 96863 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/300 VUs  2m30s  10000.00 iters/s
```

Node utilization:

```
Every 2.0s: kubectl top nodes

NAME                                        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gke-sre-test-default-pool-04398b3c-25lj     2013m        214%   2044Mi          72%
```

HPA statistics

```
Every 2.0s: kubectl -n application get hpa

NAME              REFERENCE                  TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
sre-flask-hpa     Deployment/sre-flask       0/500m      1         100       1          4d11h
sre-frontend-hpa  Deployment/sre-frontend    105043m/2   3         3         3          3d11h
```

Comments:

Test case 4: Pod = 4

Test Score

```
  scenarios: (100.00%) 1 scenario, 300 max VUs, 3m0s max duration (incl. graceful stop):
           * contacts: Up to 10000.00 iterations/s for 2m30s over 2 stages (maxVUs: 300, gracefulStop: 30s)

WARN[0002] Insufficient VUs, reached 300 active VUs and cannot initialize more  executor=ramping-arrival-rate scenario=contacts

     data_received..................: 208 MB  1.4 MB/s
     data_sent......................: 7.1 MB  47 kB/s
     dropped_iterations.............: 1260718 8359.988658/s
     http_req_blocked...............: avg=40.4µs   min=292ns    med=916ns    max=35.74ms  p(90)=2.04µs   p(95)=2.99µs
     http_req_connecting............: avg=38.5µs   min=0s       med=0s       max=35.72ms  p(90)=0s       p(95)=0s
     http_req_duration..............: avg=496.31ms min=219.92ms med=337.57ms max=6.97s    p(90)=759.14ms p(95)=1.25s
       { expected_response:true }...: avg=496.31ms min=219.92ms med=337.57ms max=6.97s    p(90)=759.14ms p(95)=1.25s
     http_req_failed................: 0.00%   ✓ 0          ✗ 90001
     http_req_receiving.............: avg=1.86ms   min=5.37µs   med=1.23ms   max=362.92ms p(90)=3.46ms   p(95)=4.3ms
     http_req_sending...............: avg=17.2µs   min=1.75µs   med=5.54µs   max=19.28ms  p(90)=17.45µs  p(95)=42.7µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=494.42ms min=219.32ms med=335.88ms max=6.97s    p(90)=756.47ms p(95)=1.25s
     http_reqs......................: 90001   596.808596/s
     iteration_duration.............: avg=496.39ms min=219.95ms med=337.64ms max=6.97s    p(90)=759.24ms p(95)=1.25s
     iterations.....................: 90001   596.808596/s
     vus............................: 300     min=81       max=300
     vus_max........................: 300     min=300      max=300


running (2m30.8s), 000/300 VUs, 90001 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/300 VUs  2m30s  10000.00 iters/s
```

Node utilization

```
Every 2.0s: kubectl top nodes

NAME                                        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gke-sre-test-default-pool-04398b3c-25lj     2018m        214%   2293Mi          81%
```

HPA statistics

```
Every 2.0s: kubectl -n application get hpa

NAME              REFERENCE                  TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
sre-flask-hpa     Deployment/sre-flask       0/500m      1         100       1          4d13h
sre-frontend-hpa  Deployment/sre-frontend    75651m/2    4         4         4          3d12h
```

Comments

Test case 5: Pod = 5

Test Score

```
  scenarios: (100.00%) 1 scenario, 300 max VUs, 3m0s max duration (incl. graceful stop):
           * contacts: Up to 10000.00 iterations/s for 2m30s over 2 stages (maxVUs: 300, gracefulStop: 30s)

WARN[0002] Insufficient VUs, reached 300 active VUs and cannot initialize more  executor=ramping-arrival-rate scenario=contacts

     data_received..................: 199 MB  1.3 MB/s
     data_sent......................: 6.8 MB  45 kB/s
     dropped_iterations.............: 1264537 8377.504448/s
     http_req_blocked...............: avg=109.76µs min=333ns    med=917ns    max=177.9ms  p(90)=2.33µs   p(95)=3.54µs
     http_req_connecting............: avg=97.56µs  min=0s       med=0s       max=124.89ms p(90)=0s       p(95)=0s
     http_req_duration..............: avg=519.24ms min=220.59ms med=352.76ms max=5.95s    p(90)=901.46ms p(95)=1.41s
       { expected_response:true }...: avg=519.23ms min=220.59ms med=352.76ms max=5.95s    p(90)=901.38ms p(95)=1.41s
     http_req_failed................: 0.00%   ✓ 1          ✗ 86175
     http_req_receiving.............: avg=1.85ms   min=5.5µs    med=1.41ms   max=267.34ms p(90)=3.93ms   p(95)=5.13ms
     http_req_sending...............: avg=35.84µs  min=1.75µs   med=5.95µs   max=131.23ms p(90)=20.12µs  p(95)=50.33µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=517.35ms min=219.68ms med=350.83ms max=5.95s    p(90)=899.07ms p(95)=1.41s
     http_reqs......................: 86176   570.912376/s
     iteration_duration.............: avg=519.42ms min=220.61ms med=352.92ms max=5.95s    p(90)=901.85ms p(95)=1.41s
     iterations.....................: 86176   570.912376/s
     vus............................: 20      min=20       max=300
     vus_max........................: 300     min=300      max=300


running (2m30.9s), 000/300 VUs, 86176 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/300 VUs  2m30s  10000.00 iters/s
```

Node utilization

```
Every 2.0s: kubectl top nodes

NAME                                        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gke-sre-test-default-pool-04398b3c-25lj     2021m        215%   2381Mi          84%
```

HPA statistics

```
Every 2.0s: kubectl -n application get hpa

NAME              REFERENCE                 TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
sre-flask-hpa     Deployment/sre-flask      0/500m     1         100       1          4d13h
sre-frontend-hpa  Deployment/sre-frontend   58009m/2   5         5         5          3d12h
```

Comments

Test case 6: Pod = 8

Test Score

```
  scenarios: (100.00%) 1 scenario, 300 max VUs, 3m0s max duration (incl. graceful stop):
           * contacts: Up to 10000.00 iterations/s for 2m30s over 2 stages (maxVUs: 300, gracefulStop: 30s)

WARN[0002] Insufficient VUs, reached 300 active VUs and cannot initialize more  executor=ramping-arrival-rate scenario=contacts

     data_received..................: 165 MB  1.1 MB/s
     data_sent......................: 5.7 MB  37 kB/s
     dropped_iterations.............: 1279168 8463.95413/s
     http_req_blocked...............: avg=49.2µs   min=333ns    med=958ns    max=29.6ms  p(90)=2.16µs  p(95)=3.16µs
     http_req_connecting............: avg=47.23µs  min=0s       med=0s       max=29.57ms p(90)=0s      p(95)=0s
     http_req_duration..............: avg=626.51ms min=220.18ms med=373.34ms max=7.66s   p(90)=1.3s    p(95)=2s
       { expected_response:true }...: avg=626.52ms min=220.18ms med=373.33ms max=7.66s   p(90)=1.3s    p(95)=2s
     http_req_failed................: 0.00%   ✓ 1          ✗ 71540
     http_req_receiving.............: avg=1.68ms   min=5.33µs   med=1.43ms   max=72.1ms  p(90)=3.66ms  p(95)=4.78ms
     http_req_sending...............: avg=16.25µs  min=1.83µs   med=6.16µs   max=19.05ms p(90)=16.33µs p(95)=30.95µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s      p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=624.81ms min=219.96ms med=371.61ms max=7.66s   p(90)=1.3s    p(95)=2s
     http_reqs......................: 71541   473.369989/s
     iteration_duration.............: avg=626.61ms min=220.2ms  med=373.42ms max=7.66s   p(90)=1.3s    p(95)=2s
     iterations.....................: 71541   473.369989/s
     vus............................: 30      min=30       max=300
     vus_max........................: 300     min=300      max=300


running (2m31.1s), 000/300 VUs, 71541 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/300 VUs  2m30s  10000.00 iters/s
```

Node utilization

```
Every 2.0s: kubectl top nodes

NAME                                          CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gke-sre-test-default-pool-04398b3c-25lj   2017m        214%   2832Mi          100%
```

HPA statistics

```
NAME              REFERENCE                   TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
sre-flask-hpa     Deployment/sre-flask        0/500m     1         100       1          4d13h
sre-frontend-hpa  Deployment/sre-frontend     30193m/2   8         8         8          3d13h
```

Test case 7: Pod = 10

Test Score

```
  scenarios: (100.00%) 1 scenario, 300 max VUs, 3m0s max duration (incl. graceful stop):
           * contacts: Up to 10000.00 iterations/s for 2m30s over 2 stages (maxVUs: 300, gracefulStop: 30s)

WARN[0001] Insufficient VUs, reached 300 active VUs and cannot initialize more  executor=ramping-arrival-rate scenario=contacts

     data_received..................: 150 MB  992 kB/s
     data_sent......................: 5.1 MB  34 kB/s
     dropped_iterations.............: 1285702 8526.042876/s
     http_req_blocked...............: avg=56.48µs  min=292ns    med=1µs      max=28.64ms p(90)=2.04µs  p(95)=3µs
     http_req_connecting............: avg=54.54µs  min=0s       med=0s       max=28.56ms p(90)=0s      p(95)=0s
     http_req_duration..............: avg=689.97ms min=220.15ms med=394.7ms  max=5.99s   p(90)=1.53s   p(95)=2.29s
       { expected_response:true }...: avg=689.99ms min=220.15ms med=394.69ms max=5.99s   p(90)=1.53s   p(95)=2.29s
     http_req_failed................: 0.00%   ✓ 4          ✗ 64889
     http_req_receiving.............: avg=3.87ms   min=5.41µs   med=1.33ms   max=3.38s   p(90)=3.32ms  p(95)=4.19ms
     http_req_sending...............: avg=10.97µs  min=1.87µs   med=6.04µs   max=9.02ms  p(90)=14.91µs p(95)=24.95µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s      p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=686.08ms min=220.12ms med=391.89ms max=5.99s   p(90)=1.53s   p(95)=2.28s
     http_reqs......................: 64893   430.33339/s
     iteration_duration.............: avg=690.07ms min=220.19ms med=394.8ms  max=5.99s   p(90)=1.53s   p(95)=2.29s
     iterations.....................: 64893   430.33339/s
     vus............................: 300     min=114      max=300
     vus_max........................: 300     min=300      max=300


running (2m30.8s), 000/300 VUs, 64893 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/300 VUs  2m30s  10000.00 iters/s
```

Node utilization

```
Every 2.0s: kubectl top nodes

NAME                                        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gke-sre-test-default-pool-04398b3c-25lj     2024m        215%   3068Mi          109%
```

HPA statistics

```
Every 2.0s: kubectl -n application get hpa

NAME              REFERENCE                 TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
sre-flask-hpa     Deployment/sre-flask      0/500m     1         100       1          4d13h
sre-frontend-hpa  Deployment/sre-frontend   21950m/2   10        10        10         3d13h
```

Comments

Test case 8: Pod = 15

Test Score

```
  scenarios: (100.00%) 1 scenario, 300 max VUs, 3m0s max duration (incl. graceful stop):
           * contacts: Up to 10000.00 iterations/s for 2m30s over 2 stages (maxVUs: 300, gracefulStop: 30s)

WARN[0002] Insufficient VUs, reached 300 active VUs and cannot initialize more  executor=ramping-arrival-rate scenario=contacts

     data_received..................: 122 MB  804 kB/s
     data_sent......................: 4.2 MB  28 kB/s
     dropped_iterations.............: 1297872 8566.865323/s
     http_req_blocked...............: avg=65.28µs  min=333ns    med=1.04µs   max=23.59ms p(90)=2.33µs  p(95)=3.41µs
     http_req_connecting............: avg=63.19µs  min=0s       med=0s       max=23.56ms p(90)=0s      p(95)=0s
     http_req_duration..............: avg=848.83ms min=221.15ms med=462ms    max=14.32s  p(90)=1.91s   p(95)=2.73s
       { expected_response:true }...: avg=848.84ms min=221.15ms med=462ms    max=14.32s  p(90)=1.91s   p(95)=2.73s
     http_req_failed................: 0.00%   ✓ 3          ✗ 52846
     http_req_receiving.............: avg=4.03ms   min=5.66µs   med=1.35ms   max=3.38s   p(90)=3.71ms  p(95)=5.66ms
     http_req_sending...............: avg=13.11µs  min=1.83µs   med=6.83µs   max=26.14ms p(90)=16.17µs p(95)=25.54µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s      p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=844.78ms min=219.96ms med=457.52ms max=14.32s  p(90)=1.9s    p(95)=2.72s
     http_reqs......................: 52849   348.84046/s
     iteration_duration.............: avg=848.94ms min=221.2ms  med=462.1ms  max=14.32s  p(90)=1.91s   p(95)=2.73s
     iterations.....................: 52849   348.84046/s
     vus............................: 78      min=78       max=300
     vus_max........................: 300     min=300      max=300


running (2m31.5s), 000/300 VUs, 52849 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/300 VUs  2m30s  10000.00 iters/s
```

Node utilization

```
Every 2.0s: kubectl top nodes

NAME                                        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gke-sre-test-default-pool-04398b3c-25lj     2026m        215%   3520Mi          125%
```

HPA statistics

```
Every 2.0s: kubectl -n application get hpa

NAME              REFERENCE                 TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
sre-flask-hpa     Deployment/sre-flask      0/500m     1         100       1          4d13h
sre-frontend-hpa  Deployment/sre-frontend   11994m/2   15        15        15         3d13h
```

Comments

In all the tests performed above, the http_request_failed was zero when the pods were 1,2,3 and 4 respectively. However, with 5 and onwards pods, we can see that requests started to get timed out. Basically http_request_failed is a metric that show how many requests are there against which the responses are not received. This clearly shows that 4 might be the limit of the pods if we don't want to risk the possibility of http request getting timed out given the hardware resources.

Dropped Iteration is another metric in the above test. Initially 10,000 iterations were requested. However, dropped iterations indicate the number of iterations that were not even initialized or executed. This could be because of multiple reasons. One of those reasons is the limitation of the hardware resources of the testing machine. The other reason would be the limitation of the hardware resources of the system under test. The reason for high number of dropped iterations is because of the limitation of the testing machine. This can be seen from the above-mentioned tests that dropped iteration per second remained almost constant through the testing.

We can also see that the average http_request_duration also peaked at 3pods on a single node. The same is true for maximum http_request_duration. This indicates that 3 pods on a single node might be the performance that this solution can achieve.

In terms of resource utilizations, we have seen that the CPU resources achieved its maximum limit at 3 pods with 215 percent. After that it remains at this limit. This defines CPU utilization that each pod would need to perform. We have also seen that the pods performed best at the 3 pods within the node, with 642 requests per second. The performance deteriorates exponentially when the pods were increased.

Testing Scenario 2:

Keeping every parameter fixed, but checking the effects of nodes on the application.

Test 1: node = 2, pod = 3 per node (6 total)

Test Score:

```
  scenarios: (100.00%) 1 scenario, 300 max VUs, 3m0s max duration (incl. graceful stop):
           * contacts: Up to 10000.00 iterations/s for 2m30s over 2 stages (maxVUs: 300, gracefulStop: 30s)

WARN[0003] Insufficient VUs, reached 300 active VUs and cannot initialize more  executor=ramping-arrival-rate scenario=contacts

     data_received..................: 405 MB  2.7 MB/s
     data_sent......................: 14 MB   92 kB/s
     dropped_iterations.............: 1174961 7814.570645/s
     http_req_blocked...............: avg=20.56µs  min=292ns    med=750ns    max=27.05ms  p(90)=1.41µs   p(95)=2.12µs
     http_req_connecting............: avg=19.16µs  min=0s       med=0s       max=27.02ms  p(90)=0s       p(95)=0s
     http_req_duration..............: avg=252.69ms min=218.93ms med=231.76ms max=1.63s    p(90)=293.54ms p(95)=345.46ms
       { expected_response:true }...: avg=252.69ms min=218.93ms med=231.76ms max=1.63s    p(90)=293.54ms p(95)=345.46ms
     http_req_failed................: 0.00%   ✓ 0          ✗ 175771
     http_req_receiving.............: avg=4.23ms   min=4.99µs   med=1.33ms   max=810.21ms p(90)=4.05ms   p(95)=5.25ms
     http_req_sending...............: avg=13.4µs   min=1.5µs    med=3.91µs   max=28.78ms  p(90)=12.33µs  p(95)=31.33µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=248.45ms min=218.43ms med=229.97ms max=1.63s    p(90)=283.98ms p(95)=329.18ms
     http_reqs......................: 175771  1169.038714/s
     iteration_duration.............: avg=252.75ms min=218.96ms med=231.81ms max=1.63s    p(90)=293.6ms  p(95)=345.49ms
     iterations.....................: 175771  1169.038714/s
     vus............................: 300     min=80       max=300
     vus_max........................: 300     min=300      max=300


running (2m30.4s), 000/300 VUs, 175771 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/300 VUs  2m30s  10000.00 iters/s
```

Node utilization:

```
Every 2.0s: kubectl top nodes

NAME                                        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gke-sre-test-default-pool-04398b3c-25lj     1599m        170%   2418Mi          86%
gke-sre-test-default-pool-04398b3c-xtkr     1664m        177%   1891Mi          67%
```

HPA statistics:

```
Every 2.0s: kubectl -n application get hpa

NAME              REFERENCE                 TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
sre-flask-hpa     Deployment/sre-flask      0/500m      1         100       1          4d16h
sre-frontend-hpa  Deployment/sre-frontend   95478m/2    6         6         6          3d16h
```

Test 2: node = 2, pod = 4 per node (8 total)

Test Score:

```
  scenarios: (100.00%) 1 scenario, 300 max VUs, 3m0s max duration (incl. graceful stop):
           * contacts: Up to 10000.00 iterations/s for 2m30s over 2 stages (maxVUs: 300, gracefulStop: 30s)

WARN[0002] Insufficient VUs, reached 300 active VUs and cannot initialize more  executor=ramping-arrival-rate scenario=contacts

     data_received..................: 361 MB  2.4 MB/s
     data_sent......................: 12 MB   82 kB/s
     dropped_iterations.............: 1194153 7927.63637/s
     http_req_blocked...............: avg=23.71µs  min=291ns    med=791ns    max=82.32ms  p(90)=1.45µs   p(95)=2.12µs
     http_req_connecting............: avg=21.77µs  min=0s       med=0s       max=28.18ms  p(90)=0s       p(95)=0s
     http_req_duration..............: avg=284.65ms min=219.15ms med=235.67ms max=5.07s    p(90)=353.38ms p(95)=470.99ms
       { expected_response:true }...: avg=284.65ms min=219.15ms med=235.67ms max=5.07s    p(90)=353.38ms p(95)=470.99ms
     http_req_failed................: 0.00%   ✓ 0          ✗ 156574
     http_req_receiving.............: avg=3.06ms   min=5.08µs   med=1.4ms    max=595.25ms p(90)=4.06ms   p(95)=5.26ms
     http_req_sending...............: avg=11.17µs  min=1.62µs   med=3.83µs   max=23ms     p(90)=11.41µs  p(95)=26.89µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=281.57ms min=218.4ms  med=233.63ms max=5.07s    p(90)=345.5ms  p(95)=461.15ms
     http_reqs......................: 156574  1039.449499/s
     iteration_duration.............: avg=284.71ms min=219.18ms med=235.76ms max=5.07s    p(90)=353.44ms p(95)=471.04ms
     iterations.....................: 156574  1039.449499/s
     vus............................: 300     min=86       max=300
     vus_max........................: 300     min=300      max=300


running (2m30.6s), 000/300 VUs, 156574 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/300 VUs  2m30s  10000.00 iters/s
```

Node utilization:

```
Every 2.0s: kubectl top nodes

NAME                                        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gke-sre-test-default-pool-04398b3c-25lj     1731m        184%   2589Mi          92%
gke-sre-test-default-pool-04398b3c-xtkr     1799m        191%   1821Mi          64%
```

HPA statistics:

```
Every 2.0s: kubectl -n application get hpa

NAME              REFERENCE                   TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
sre-flask-hpa     Deployment/sre-flask        0/500m      1         100       1          4d17h
sre-frontend-hpa  Deployment/sre-frontend     65055m/2    8         8         8          3d16h
```

Test 3: node = 3, pod = 3 per node (9 total)

Test Score:

```
WARN[0002] Insufficient VUs, reached 300 active VUs and cannot initialize more  executor=ramping-arrival-rate scenario=contacts

     data_received..................: 380 MB  2.5 MB/s
     data_sent......................: 13 MB   86 kB/s
     dropped_iterations.............: 1185951 7840.896856/s
     http_req_blocked...............: avg=22.8µs   min=291ns    med=792ns    max=25.12ms p(90)=1.79µs   p(95)=2.58µs
     http_req_connecting............: avg=21.05µs  min=0s       med=0s       max=25.11ms p(90)=0s       p(95)=0s
     http_req_duration..............: avg=270.38ms min=219.11ms med=231.08ms max=3.61s   p(90)=320.38ms p(95)=437.57ms
       { expected_response:true }...: avg=270.38ms min=219.11ms med=231.08ms max=3.61s   p(90)=320.38ms p(95)=437.57ms
     http_req_failed................: 0.00%   ✓ 0          ✗ 164761
     http_req_receiving.............: avg=4.05ms   min=4.99µs   med=1.53ms   max=3.33s   p(90)=4.2ms    p(95)=5.49ms
     http_req_sending...............: avg=20.08µs  min=1.62µs   med=4.12µs   max=38.55ms p(90)=14.2µs   p(95)=37.11µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s      p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=266.3ms  min=218.64ms med=229.1ms  max=3.61s   p(90)=310.99ms p(95)=411.99ms
     http_reqs......................: 204761  1689.314826/s
     iteration_duration.............: avg=270.44ms min=219.13ms med=231.15ms max=3.61s   p(90)=320.53ms p(95)=437.62ms
     iterations.....................: 204761  1689.314826/s
     vus............................: 25      min=25       max=300
     vus_max........................: 300     min=300      max=300


running (2m31.3s), 000/300 VUs, 164761 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/300 VUs  2m30s  10000.00 iters/s
```

Node utilization:

```
Every 2.0s: kubectl top nodes

NAME                                        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gke-sre-test-default-pool-04398b3c-25lj     866m         92%    2395Mi          85%
gke-sre-test-default-pool-04398b3c-j0g3     1960m        208%   1935Mi          68%
gke-sre-test-default-pool-04398b3c-xtkr     1220m        129%   1498Mi          53%
```

HPA statistics:

```
Every 2.0s: kubectl -n application get hpa

NAME              REFERENCE                   TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
sre-flask-hpa     Deployment/sre-flask        0/500m      1         100       1          4d17h
sre-frontend-hpa  Deployment/sre-frontend     43852m/2    9         9         9          3d17h
```

Comments:

From the above tests, we can see that the performance increment was almost linear when the number of nodes were increased. Now using this information, we can define the hardware requirements for the whole solution.

At the maximum load, the maximum traffic that would be expected is 100,000 requests per second. For 3 nodes, 4vCPU and 5.8GB RAM would result in almost 1700 Requests per second. There are 9 application pods running on these nodes. This means that 500m VCPU and 650MiB per pod which will handle almost 188 requests per second. For 100,000 request per second, we would almost need **532 pods or 216vCPUs and around 250GB of memory.**

Cost Analysis:

There are many considerations that can go into determining the costs. The list of factors that can contribute to the total expenditure is as follows.

1. Fully managed (AutoPilot) or Self Managed (Standard GKE deployment).
2. Per pod cost or Per node cost.
3. Hardware resources requirements

There is a cluster management fees of $0.1 per hour that applies to all clusters irrespective of size and mode of operation. That is the fixed price.

AutoPilot clusters offer fully managed cluster especially in terms of node management and node auto provisioning and scaling. Autopilot clusters are billed according to the hardware resources requested and consumed by the individual pods. CPU, Memory and Storage resources are considered in the billing. While it provide more autonomy, it is more expensive than the other options

Standard GKE deployment usually follow the Compute Engine pricing. In this mode of operation, the node management is partially done by the Users especially in terms of nodes autoscaling. While it provides more control over the nodes, it also increases the work required by the user to manage the cluster.

According to the load test, the hardware requirements for the cluster according to the maximum load come about to be 218vCPU and 250GB Memory if there is a load of 100,000 requests per second. Considering that there are three scenarios which should be considered while considering the cost.

1. Cost at the minimum load, i.e. at 10 requests per second
2. Cost at the maximum load i.e. at 100,000 requests per second.
3. Cost at the average load.

In the cost analysis, we are assuming that the region is US-EAST1 as the prices vary according to different regions. We are also assuming that the cluster is Zonal instead of Regional.

Cost at the minimum load:

The minimum load is 10 requests per seconds. According to the load test, the pods would function optimally with the resource limitation of 500m vCPU and 650 MiB of memory. Provided given resources, one pod can handle about 188 requests per second at the peak load. This means that one application pod is required to handle the minimum load of the application.

In the case of autopilot clusters where the cost is calculated at the pod level, the cost would be

Cost of cluster per hour at the minimum load = $0.1 (fixed fee) + 0.0445 * 0.75 vCPU + 0.0049225 * 1GB memory = 0.1 + 0.033375 + 0.0049225 = **$0.13825 per hour or $100.9225 per month.**

In the above equation, the resources were increased because of the fact that there are Prometheus pods and other monitoring pods in the cluster that would have their own resource requirements. But those resource requirements would be very minimal.

The standard GKE configuration would calculate cost at the node level. It would also use compute engine storage classes. We recommend E2-standard-4 with custom configuration, which would give 4 vCPUs and 6GB RAM per VM. According to the following pricing calculator (https://cloud.google.com/products/calculator?hl=en&dl=CiRkZjAyNTM1MC1iMDFjLTRiMTktYTU5Yi0wNDUyYmYwMWU3MmMQDxokOTFDQ0JEQjEtNTI4Qy00NEU0LTgyNDQtOTdCQ0ZBMjIzMUVG), **the total cost per month comes out to be $149.49 or $0.20478 per hour.**

Cost at the maximum load:

The maximum load is 100,000 Requests per second. According to the load tests, around 216vCPU and 250GB resources are required to handle this kind of workload.

The cost of the autopilot clusters would come out to be.

Cost of autopilot cluster per hour at the maximum load = $0.1 (fixed fee) + 0.0445 * 216 vCPU + 0.0049225 * 250GB memory = 0.1 + 9.612 + 1.230625 = **$10.9426 per hour or $7988.11625 per month.**

The standard GKE configuration would calculate the cost at the node level. Our suggestion is to the E2-Standard-4 node class with custom configuration having the resources of 4 vCPUs and 6GB RAM per VM. With this specific configuration, we would need around 54 VMs to carry the load. According to the following price calculator, (https://cloud.google.com/products/calculator?hl=en&dl=CiQwZWYxNTQ3YS00MmJjLTQ2ZjAtYThlZC00ZmU0MjVhMjExNTYQDxokOTFDQ0JEQjEtNTI4Qy00NEU0LTgyNDQtOTdCQ0ZBMjIzMUVG) **the total cost per month comes out to be $4245.72 per month or $5.81605 per hour** which is considerably less than Autopilot clusters.

Cost at the average load:

We can safely assume that the maximum load would only be due to a surge case and hence happen rarely. The average load is the most important factor here as this would be the load almost 90 percent of the time. Since we are not sure how much the average load would be, we are assuming that it would be around **60 percent of the peak load/maximum load**. That would come out to be **130vCPU and 150GB RAM.**

The cost of the autopilot clusters would come out to be.

Cost of autopilot cluster per hour at the average load = $0.1 (fixed fee) + 0.0445 * 130 vCPU + 0.0049225 * 150GB memory = 0.1 + 5.785 + 0.7383 = **$6.6233 per hour or $4835.009 per month.**

According to the pricing calculator (https://cloud.google.com/products/calculator?hl=en&dl=CiQzOTRiMTg5OC1mNTE5LTQwMDktYTU0ZS00OTkyYjYxMDhjYzgQDxokOTFDQ0JEQjEtNTI4Qy00NEU0LTgyNDQtOTdCQ0ZBMjIzMUVG), the cost of running a E2-standard-4 node class with custom configuration of 4vCPU and 6GB RAM per VM at the average load would come out to be **$2622/month or $3.591 per hour**