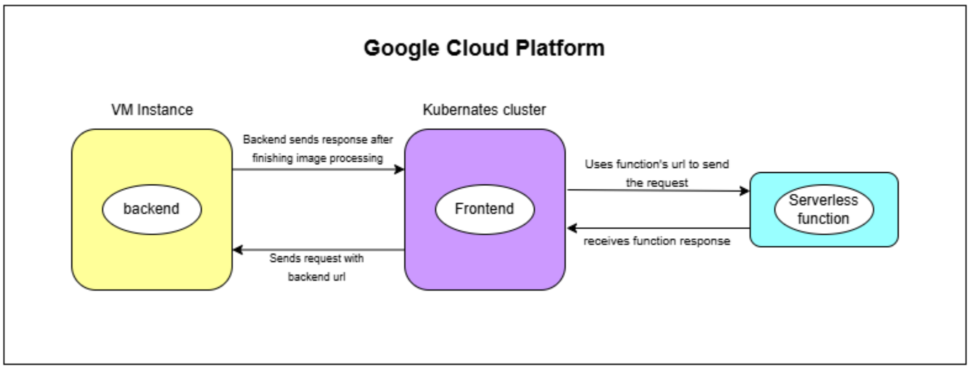
SWE 590 – Cloud Computing Architectures

Esra Nur Özüm – 2024719084

Project Demo Video Link: https://drive.google.com/file/d/147LI3sPeK0ErrdJs2zmW-v9qEdXLtHvV/view?usp=sharing

1. Cloud architecture structure

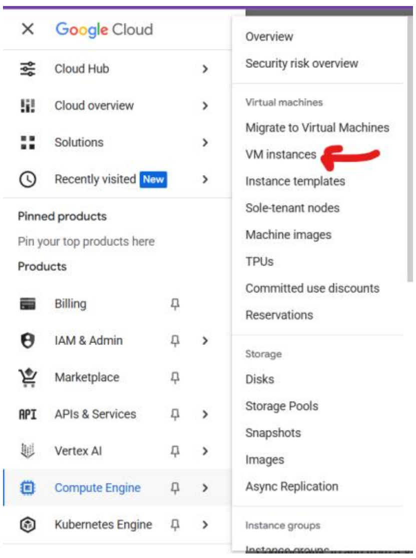


In the architecture that I established I have three components that work together. Vm Instance carries the backend application and runs the server inside. Kubernates Cluster carries the frontend image and uses external IP of the VM Instance to send a request to the backend and receive the related response. It also uses a public URL generated by Google cloud platform for a serverless function that simply prints a text message. When the cluster calls it, the response will be a text that carries the message. The system can be reachable from the URL http://34.41.239.214:30036/

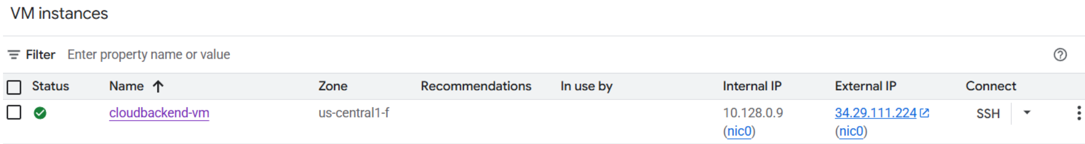
2. Deployment Process

Backend Deployment Process:

- First, I created a new Vm in my google cloud platform



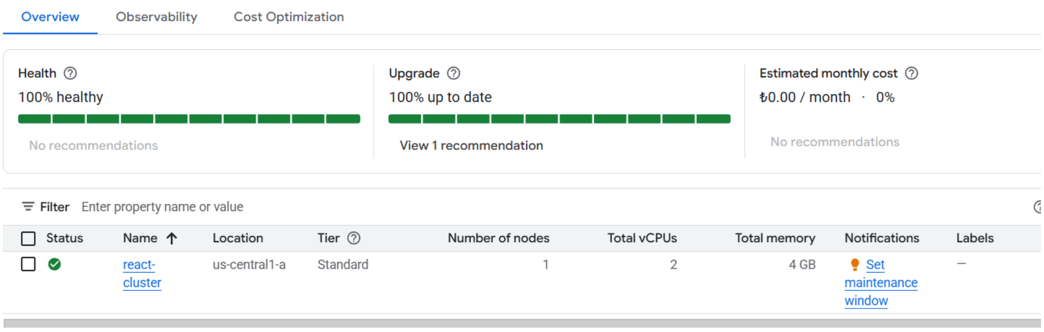
-

- Then, I connected to VM and installed the needed libraries with the following commands
    - Sudo apt update
    - sudo apt install python3-pip python3-venv -y
    - python3 -m venv venv
    - source venv/bin/activate
    - pip install fastapi uvicorn
    - mkdir backend && cd backend
    - nano main.py
    - I put all the code to the main.py file
    - Then, wrote the following command to run it, uvicorn main:app --host 0.0.0.0 --port 5000
    - I created a firewall rule to make it reachable from outside of the VM
        - Name: `allow-backend-port`
        - Targets: All instances in the network
        - Protocols: TCP `5000`
        - Source IP: `0.0.0.0/0`
    - 

    - After these steps when I go to http://34.29.111.224:5000/process I see the following result:
    - 

    - This proves that the backend that I put into the VM is reachable from outside

Frontend Deployment Process:

- First I created a cluster by running the following comments
    - gcloud services enable container.googleapis.com
    - gcloud container clusters create react-cluster --num-nodes=1 --zone=us-central1-a
- These commands enabled me to create a kubernates cluster with 1 node and named it as react-cluster.
- 

- Then, I go to my project's frontend folder and run "npm run build" to create a build file for production

- I created a Dockerfile inside the same folder directory and add the following configurations:

```
FROM nginx:alpine

COPY build/ /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

  - After this point I was ready to create a docker image
  - First, I added some artifacts to the google Artifact Registry by running the following command
    - gcloud artifacts repositories create react-ui --repository-format=docker --location=us-central1
  - Then I build and push the docker image for the frontend:
    - docker build -t us-central1-docker.pkg.dev/swe590-project-458808/react-ui/react-ui:latest .
    - docker push us-central1-docker.pkg.dev/swe590-project-458808/react-ui/react-ui:latest
  - After these steps I moved on to the deploying UI to the google kubernates engine
  - I run the following commends to achieve this
    - gcloud services enable container.googleapis.com
    - gcloud container clusters create react-cluster --num-nodes=1 --zone=us-central1-a
    - gcloud container clusters get-credentials react-cluster --zone=us-central1-a
  - I prepared the following deployment yaml file for configurations

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: react-ui
spec:
  replicas: 2
  selector:
    matchLabels:
      app: react-ui
  template:
    metadata:
      labels:
        app: react-ui
    spec:
      containers:
      - name: react-ui
        image: us-central1-docker.pkg.dev/swe590-project-458808/react-ui/react-ui:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: react-ui-service
spec:
  type: NodePort
  selector:
    app: react-ui
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30036
```

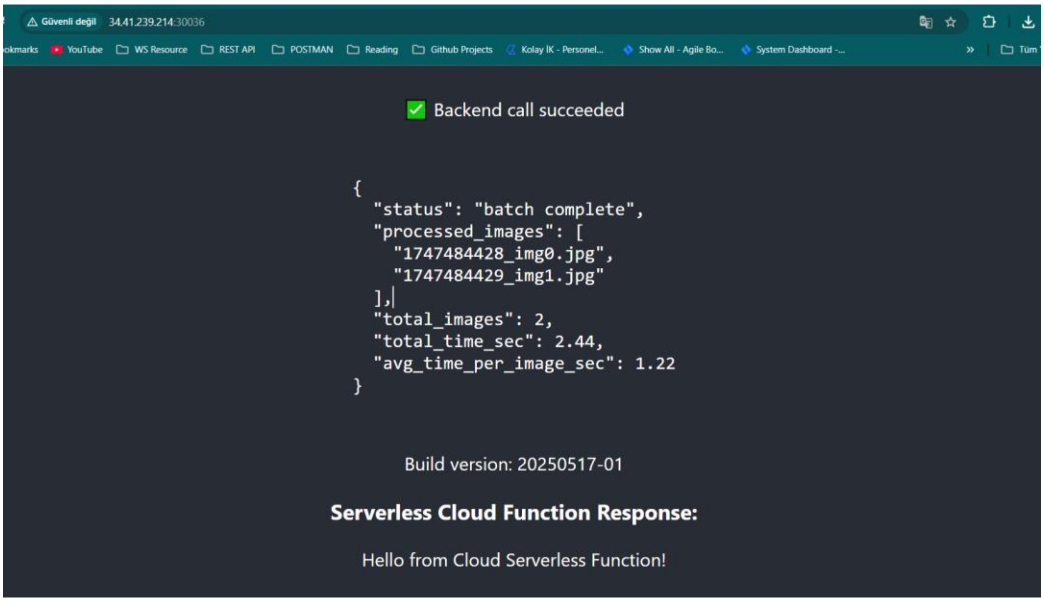
- After this step, I just run the following commands to apply all changes
  - kubectl apply -f react-deployment.yaml
  - kubectl get services => for checking if it runs

```
service/react-ui-service unchanged
PS C:\Users\esra.ozum\WebAppProjects\cloudProject\frontend> kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
react-ui-7fdd87c47d-hr6n8   1/1     Running   0          9s
react-ui-7fdd87c47d-wbmpc   1/1     Running   0          4s
```

```
PS C:\Users\esra.ozum\WebAppProjects\cloudProject\frontend> kubectl get service react-ui-service
NAME               TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
react-ui-service   NodePort   34.118.231.142   <none>        80:30036/TCP   14m
NAME               TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
react-ui-service   NodePort   34.118.231.142   <none>        80:30036/TCP   14m
PS C:\Users\esra.ozum\WebAppProjects\cloudProject\frontend> kubectl get nodes -o wide
```

```
react-ui-service   NodePort   34.118.231.142   <none>   80:30036/TCP   14m
PS C:\Users\esra.ozum\WebAppProjects\cloudProject\frontend> kubectl get nodes -o wide
NAME                                          STATUS   ROLES    AGE   VERSION               INTERNAL-IP   EXTERNAL-IP     OS
NAME                                          STATUS   ROLES    AGE   VERSION               INTERNAL-IP   EXTERNAL-IP     OS
-IMAGE                        KERNEL-VERSION   CONTAINER-RUNTIME
gke-react-cluster-default-pool-d4784351-4srv  Ready    <none>   19m   v1.32.2-gke.1297002   10.128.0.11   34.41.239.214   Co
gke-react-cluster-default-pool-d4784351-4srv  Ready    <none>   19m   v1.32.2-gke.1297002   10.128.0.11   34.41.239.214   Co
ntainer-Optimized OS from Google   6.6.72+       containerd://1.7.24
PS C:\Users\esra.ozum\WebAppProjects\cloudProject\frontend>
```

- After this point when I used the external IP and tried to reach the whole project I got the following result:



Serverless Function Implementation

- For the serverless function I created a file under the project folder and named it as serverless
- I created a main.py file and write a dummy function that will serve as a serverless function

```python
# serverless > main.py > ...
def hello_esra(request):
    headers = {
        'Access-Control-Allow-Origin': '*',
        'Access-Control-Allow-Headers': 'Content-Type',
    }

    if request.method == 'OPTIONS':
        return ('', 204, headers)

    return ('Hello from Cloud Serverless Function!', 200, headers)
```

- Then I run the following commands to put it inside the google cloud platform

- functions deploy hello-esra --runtime python311 –trigger region us-central1

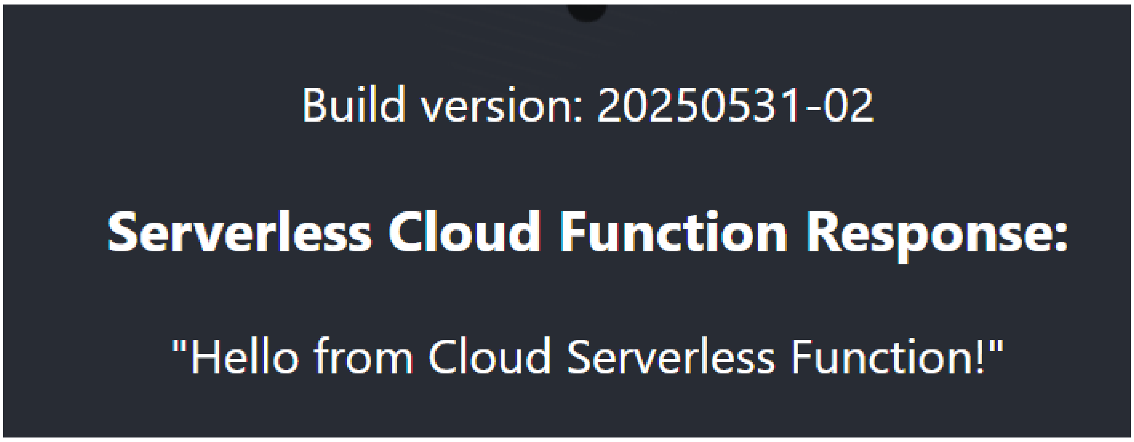
```
  uri: https://hello-esra-3tcjl5qq2a-uc.a.run.app
state: ACTIVE
updateTime: '2025-05-17T11:51:46.677207166Z'
updateTime: '2025-05-17T11:51:46.677207166Z'
url: https://us-central1-swe590-project-458808.cloudfunctions.net/hello-esra
PS C:\Users\esra.ozum\WebAppProjects\cloudProject\serverless>
```

- To keep the serverless function secured, I added an API endpoint to the backend and used that in frontend to reach the serverless function's response and show it in the UI.

```python
@app.get("/api/hello")
async def proxy_hello():
    try:
        request = google.auth.transport.requests.Request()
        credentials.refresh(request)
        id_token = credentials.token

        response = requests.get(
            CLOUD_FUNCTION_URL,
            headers={'Authorization': f'Bearer {id_token}'}
        )
        response.raise_for_status()
        return response.text
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

- And showed the result on the page



3. Horizontal scaling (HPA)

To ensure high availability and performance of the frontend React UI service under varying loads, I implemented Horizontal Pod Autoscaling (HPA) in the Kubernetes cluster deployed on Google Kubernetes Engine (GKE).

Purpose of HPA:

- HPA automatically adjusts the number of running pods based on observed CPU utilization. This helps maintain responsiveness during high traffic and optimize resource usage during low traffic.

HPA Configuration:

- I configured the HPA for my cluster using a YAML manifest. The configuration scales the number of pods based on CPU usage:

```yaml
frontend > ! react-ui-hpa.yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: react-ui
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: react-ui-deployment
  minReplicas: 1
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

- To make autoscaling effective, I configured CPU resource requests and limits in the deployment:

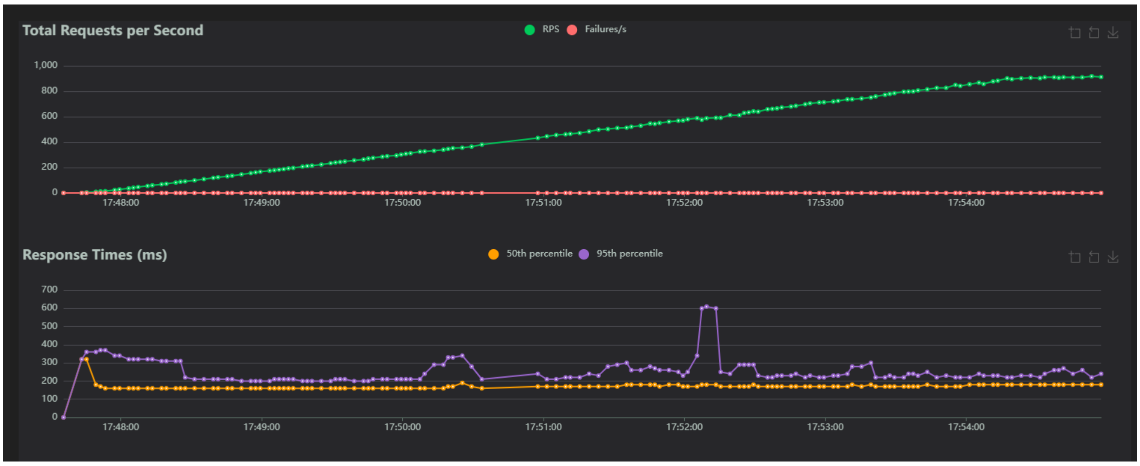
```yaml
resources:
  requests:
    cpu: "100m"
  limits:
    cpu: "400m"
```

4. Locust Experiments

Test Cases:

1-Effect of number of users

- Number of users = 2000, ramp up = 5, number of pods = 2



- Number of users = 1000, ramp up = 5, number of pods = 2

- Observations:
  - When there are 1000 users in the test load the 95th percentile graph is more stable and has the value around 220 ms.
  - Whereas, when the number of users is set to 2000, there are important fluctuations in the 95th percentile graph.
  - This shows that some users in the first test are waiting too long to get the response compared to the second test users.
  - Stability level of the first run is lower than the second run.
  - This shows us that increasing the load amount with the same amount of ramp up effects the smoothness of the program in terms of usability.
  - The possible reasons for the fluctuations in experiment 2 is the difference between the processed image in each run.

2-Effect of ramp up value

- Number of users = 1000, ramp up = 10, number of pods = 2

- Number of users = 1000, ramp up = 2, number of pods = 2

- Observations:
    - Changing the ramp up value from 10 to 2 does not has a significant impact on the statistics
    - Both test results has similar RPS, min (ms), max (ms), and median (ms) values
    - The only difference that we see is the small fluctuations in the 95th percentile values which is normal since we decreased the ramp up value

3-Effect of number of pods

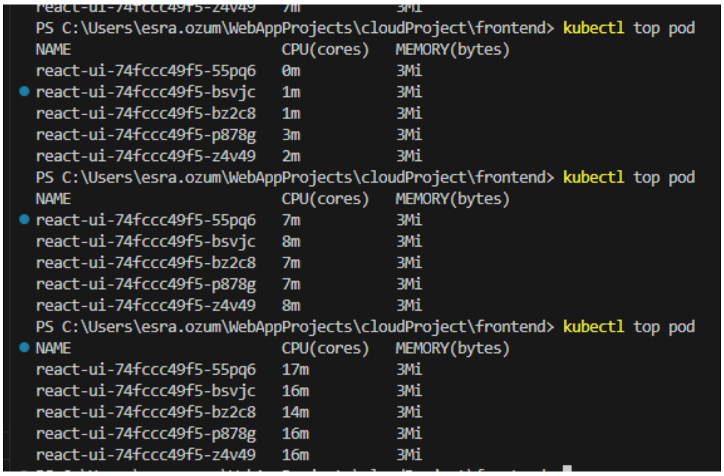
- Number of users = 1000, ramp up = 2, number of pods = 2

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	//	117075	0	170	350	1000	211.51	151	4463	644	453.3	0
	Aggregated	117075	0	170	350	1000	211.51	151	4463	644	453.3	0

```
react-ui-85d48595fd-pnhd8   15m          3Mi
PS C:\Users\esra.ozum\WebAppProjects\cloudProject\frontend> kubectl top pod
NAME                        CPU(cores)   MEMORY(bytes)
react-ui-85d48595fd-5kvtr   35m          3Mi
react-ui-85d48595fd-pnhd8   33m          3Mi
PS C:\Users\esra.ozum\WebAppProjects\cloudProject\frontend> kubectl top pod
NAME                        CPU(cores)   MEMORY(bytes)
react-ui-85d48595fd-5kvtr   8m           3Mi
react-ui-85d48595fd-pnhd8   6m           3Mi
PS C:\Users\esra.ozum\WebAppProjects\cloudProject\frontend>
```

- Number of users = 1000, ramp up = 2, number of pods = 5

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	//	121464	0	160	210	320	174.36	151	1890	644	459.5	0
	Aggregated	121464	0	160	210	320	174.36	151	1890	644	459.5	0

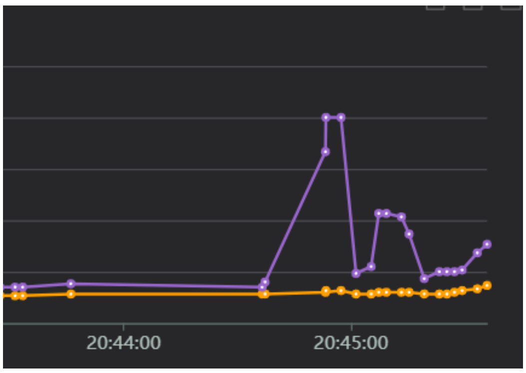


```
PS C:\Users\esra.ozum\WebAppProjects\cloudProject\frontend> kubectl top pod
NAME                        CPU(cores)   MEMORY(bytes)
react-ui-74fccc49f5-55pq6   0m           3Mi
react-ui-74fccc49f5-bsvjc   1m           3Mi
react-ui-74fccc49f5-bz2c8   1m           3Mi
react-ui-74fccc49f5-p878g   3m           3Mi
react-ui-74fccc49f5-z4v49   2m           3Mi
PS C:\Users\esra.ozum\WebAppProjects\cloudProject\frontend> kubectl top pod
NAME                        CPU(cores)   MEMORY(bytes)
react-ui-74fccc49f5-55pq6   7m           3Mi
react-ui-74fccc49f5-bsvjc   8m           3Mi
react-ui-74fccc49f5-bz2c8   7m           3Mi
react-ui-74fccc49f5-p878g   7m           3Mi
react-ui-74fccc49f5-z4v49   8m           3Mi
PS C:\Users\esra.ozum\WebAppProjects\cloudProject\frontend> kubectl top pod
NAME                        CPU(cores)   MEMORY(bytes)
react-ui-74fccc49f5-55pq6   17m          3Mi
react-ui-74fccc49f5-bsvjc   16m          3Mi
react-ui-74fccc49f5-bz2c8   14m          3Mi
react-ui-74fccc49f5-p878g   16m          3Mi
react-ui-74fccc49f5-z4v49   16m          3Mi
```

- Observations:
  - With 5 pods, latency is consistently lower across all percentiles**.**
  - The 99th percentile latency jumped from 310ms → 1000ms in the 2-pod test.
  - Max latency increased significantly (4463 ms vs. 1353 ms), which suggests occasional request overloads in the 2-pod test.
  - Both setups achieved similar RPS (~453)**.**
  - The 5-pod setup delivered more predictable and stable latency**.**
  - The 2-pod setup showed more performance variability, especially for tail latencies (95th/99th percentile).
  - The CPU usage among the pods also decreased to 15m when there are 5 pods. It is increased to 35m when there are 2 pods
  - Doing this experiment with Manuel scaling resulted the observed values. Instead of autoscaling the pods, which creates new pods when the load increased, I put constant number of pods sharing the load during the tests.

4- The effect of autoscaling in performance:

- Number of users: 500
- Ramp up: 1
- Started pod count 1
- Ended pod count 3



Here I was able to observe the pod allocation with respect to the load on CPU. When it goes over the threshold, the system created a new pod and this allowed me to spread the overall load to newly created pods. As a result, the response time decreased and became close to the normal line again.

5. Cost Breakdown

	Service	Cost	Discounts	Promotions & others	Subtotal	% Change
■	Cloud Run Functions	₺0.02	-₺0.02	₺0.00	₺0.00	–
◆	Cloud Monitoring	₺144.75	₺0.00	-₺144.75	₺0.00	–
▼	VM Manager	₺37.01	-₺37.01	₺0.00	₺0.00	–
▲	Compute Engine	₺2,313.41	₺0.00	-₺2,313.41	₺0.00	–
●	Kubernetes Engine	₺2,156.55	-₺2,156.55	₺0.00	₺0.00	–
✚	Networking	₺373.75	-₺373.75	₺0.00	₺0.00	–
●	Artifact Registry	₺0.00	₺0.00	₺0.00	₺0.00	–

6. Conclusion:

The Locust load testing experiments provided valuable insights into how different system parameters affect performance under load. Increasing the number of concurrent users from 1000 to 2000, while keeping the ramp-up rate and pod count constant, led to significant instability in response times, especially in the 95th percentile. This suggests that the system becomes less predictable and more prone to latency spikes under higher concurrency, indicating a scalability limitation with the current configuration. Varying the ramp-up rate (from 10 to 2 users/sec) had minimal impact on key performance metrics such as RPS, median latency, and max latency. This shows that the system can handle different ramp-up speeds relatively well, with only minor effects on the 95th percentile, which is expected due to smoother load introduction. Lastly, increasing the number of pods from 2 to 5 had a clear positive impact. Tail latencies (95th and 99th percentiles) were significantly lower and more stable with 5 pods. The maximum latency also decreased, and CPU usage per pod was reduced, suggesting improved load distribution and resource efficiency.

Importantly, enabling Horizontal Pod Autoscaling (HPA) in the cluster allowed the system to dynamically adjust the number of pods based on CPU usage during load tests. As traffic increased, autoscaling automatically spread the load across newly created pods, which improved overall performance and reduced latency. This behavior, demonstrated in Test Case 4, confirms that autoscaling effectively mitigates bottlenecks by distributing workload efficiently, leading to better responsiveness and system stability under varying loads.

Overall, the results indicate that while the system handles moderate traffic well, it benefits greatly from increased pod count and autoscaling to ensure consistent performance at higher loads.