UNIVERSITÉ Grenoble Alpes

Grenoble INP

Master of Science in Informatics at Grenoble

M2 MOSIG

Cloud Computing: Lab Assignment

DEMIR Aysegul

January 2025

Content

Base steps

Advanced steps

Base steps

1) Deploying the original application in GKE

For this step, I initially used Autopilot Mode (create-auto) to test the deployment.
Then, I deployed the application using standard mode specifying:

- The number of nodes: 4
- The machine type: e2-standard-2
- And the region: europe-west6-a.

In GKE, Autopilot mode simplifies application deployment by automatically managing most of the cluster operation such as resources allocation, scaling, and node configuration. This mean that we do not need to manually specify the resources needed for our application.

However, Autopilot Mode hides problem because it automatically scales resources to meet the application's needs. As a result, problem such as resource consumption or inefficiencies are harder to detect, since the scaling happens automatically.

2) Analyzing the provided configuration

Checkoutservice :
Deployment:

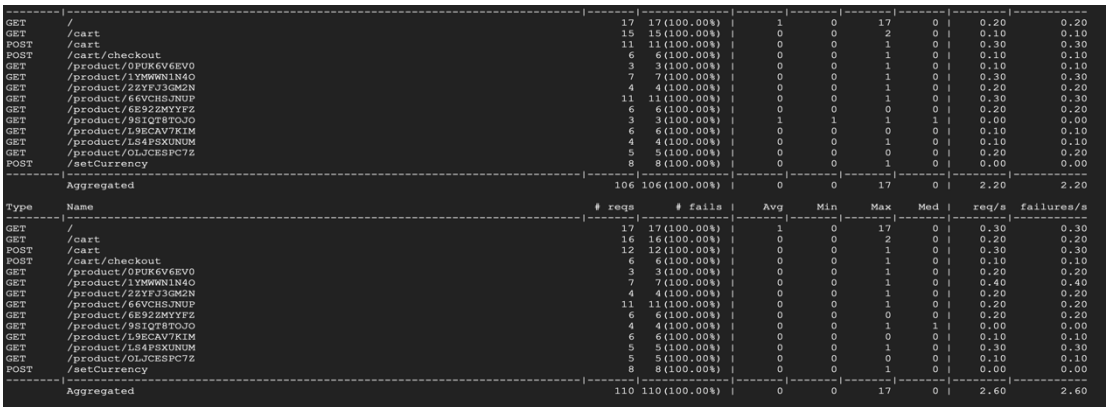
- Deploys version v0.10.2 of checkoutservices using a secure, resource-efficient container
- Exposes gRCP on port 5050 and connect to other microservices using environment variable.
- Defines resource requests and limits.

Service:

- Type: ClusterIP, which exposes the service only within the Kubernetes cluster
- Port Mapping: The service listens for incoming traffic on port 5050. The targetPort is also port 5050, which means the service forwards traffic it receives on port 5050 to port 5050 on the target pods.

3) Deploying the load generator on a local machine

To manually deploy the load generator, I used Google cloud shell and ran it as a Docker container. Initially, I tried to run the container without specifying the FRONTEND_ADDR, but all requests failed, as shown below:



Type	Name	# reqs	# fails	Avg	Min	Max	Med	req/s	failures/s
GET	/	17	17(100.00%)	1	0	17	0	0.30	0.30
GET	/cart	15	15(100.00%)	0	0	2	0	0.10	0.10
POST	/cart	11	11(100.00%)	0	0	1	0	0.30	0.30
POST	/cart/checkout	6	6(100.00%)	0	0	1	0	0.10	0.10
GET	/product/0PUK6V6EV0	3	3(100.00%)	0	0	1	0	0.10	0.10
GET	/product/1YMWWN1N4O	7	7(100.00%)	0	0	1	0	0.30	0.30
GET	/product/2ZYFJ3GM2N	4	4(100.00%)	0	0	1	0	0.20	0.20
GET	/product/66VCHSJNUP	11	11(100.00%)	0	0	1	0	0.30	0.30
GET	/product/6E92ZMYYFZ	6	6(100.00%)	0	0	0	0	0.20	0.20
GET	/product/9SIQT8TOJO	3	3(100.00%)	1	1	1	1	0.00	0.00
GET	/product/L9ECAV7KIM	6	6(100.00%)	0	0	0	0	0.10	0.10
GET	/product/LS4PSXUNUM	4	4(100.00%)	0	0	1	0	0.10	0.10
GET	/product/OLJCESPC7Z	5	5(100.00%)	0	0	0	0	0.20	0.20
POST	/setCurrency	8	8(100.00%)	0	0	1	0	0.00	0.00
	Aggregated	106	106(100.00%)	0	0	17	0	2.20	2.20

Type	Name	# reqs	# fails	Avg	Min	Max	Med	req/s	failures/s
GET	/	17	17(100.00%)	1	0	17	0	0.30	0.30
GET	/cart	16	16(100.00%)	0	0	2	0	0.20	0.20
POST	/cart	12	12(100.00%)	0	0	1	0	0.30	0.30
POST	/cart/checkout	6	6(100.00%)	0	0	1	0	0.10	0.10
GET	/product/0PUK6V6EV0	3	3(100.00%)	0	0	1	0	0.20	0.20
GET	/product/1YMWWN1N4O	7	7(100.00%)	0	0	1	0	0.40	0.40
GET	/product/2ZYFJ3GM2N	4	4(100.00%)	0	0	1	0	0.20	0.20
GET	/product/66VCHSJNUP	11	11(100.00%)	0	0	1	0	0.20	0.20
GET	/product/6E92ZMYYFZ	6	6(100.00%)	0	0	0	0	0.20	0.20
GET	/product/9SIQT8TOJO	4	4(100.00%)	0	0	1	1	0.00	0.00
GET	/product/L9ECAV7KIM	6	6(100.00%)	0	0	0	0	0.10	0.10
GET	/product/LS4PSXUNUM	5	5(100.00%)	0	0	1	0	0.30	0.30
GET	/product/OLJCESPC7Z	5	5(100.00%)	0	0	0	0	0.10	0.10
POST	/setCurrency	8	8(100.00%)	0	0	1	0	0.00	0.00
	Aggregated	110	110(100.00%)	0	0	17	0	2.60	2.60

To resolve this issue, I provided the FRONTEND_ADDR environment variable with the external IP of the frontend-external services. I found this external IP by running the following command: kubectl get services.

```
NAME                    TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)          AGE
adservice               ClusterIP      34.118.225.181   <none>          9555/TCP         17m
cartservice             ClusterIP      34.118.229.55    <none>          7070/TCP         17m
checkoutservice         ClusterIP      34.118.231.41    <none>          5050/TCP         17m
currencyservice         ClusterIP      34.118.227.216   <none>          7000/TCP         17m
emailservice            ClusterIP      34.118.237.69    <none>          5000/TCP         17m
frontend                ClusterIP      34.118.229.188   <none>          80/TCP           17m
frontend-external       LoadBalancer   34.118.228.228   34.65.248.128   80:31499/TCP     17m
kubernetes              ClusterIP      34.118.224.1     <none>          443/TCP          19m
paymentservice          ClusterIP      34.118.226.40    <none>          50051/TCP        17m
productcatalogservice   ClusterIP      34.118.226.59    <none>          3550/TCP         17m
recommendationservice   ClusterIP      34.118.238.46    <none>          8080/TCP         17m
redis-cart              ClusterIP      34.118.228.163   <none>          6379/TCP         17m
shippingservice         ClusterIP      34.118.235.52    <none>          50051/TCP        17m
```

Once I retrieved the IP, I ran the load generator using the updated command with the FRONTEND_ADDR. After specifying the FRONTEND_ADDR, the load generator appeared to work, and I obtained the following results showing 0% failures:

```
Type     Name                                                                          # reqs      # fails |    Avg     Min     Max    Med |   req/s  failures/s
--------|----------------------------------------------------------------------------|-------|-------------|-------|-------|-------|------|--------|-----------
GET      /                                                                                 10     0(0.00%) |     49      45      66     47 |    0.00        0.00
GET      /cart                                                                             13     0(0.00%) |     29      24      41     26 |    0.50        0.00
POST     /cart                                                                             11     0(0.00%) |     46      43      58     45 |    0.40        0.00
POST     /cart/checkout                                                                     5     0(0.00%) |     41      34      51     39 |    0.40        0.00
GET      /product/0PUK6V6EV0                                                                1     0(0.00%) |     27      27      27     27 |    0.10        0.00
GET      /product/1YMWWN1N4O                                                                6     0(0.00%) |     27      25      31     26 |    0.30        0.00
GET      /product/2ZYFJ3GM2N                                                                3     0(0.00%) |     26      26      27     26 |    0.10        0.00
GET      /product/66VCHSJNUP                                                                2     0(0.00%) |     28      26      30     26 |    0.10        0.00
GET      /product/6E92ZMYYFZ                                                                4     0(0.00%) |     26      25      27     26 |    0.00        0.00
GET      /product/9SIQT8TOJO                                                                8     0(0.00%) |     27      26      31     27 |    0.40        0.00
GET      /product/L9ECAV7KIM                                                                3     0(0.00%) |     27      27      27     27 |    0.20        0.00
GET      /product/LS4PSXUNUM                                                                4     0(0.00%) |     41      26      88     26 |    0.10        0.00
GET      /product/OLJCESPC7Z                                                                5     0(0.00%) |     31      27      40     28 |    0.10        0.00
POST     /setCurrency                                                                       2     0(0.00%) |     53      49      58     49 |    0.00        0.00
--------|----------------------------------------------------------------------------|-------|-------------|-------|-------|-------|------|--------|-----------
         Aggregated                                                                        77     0(0.00%) |     35      24      88     28 |    2.70        0.00

Type     Name                                                                          # reqs      # fails |    Avg     Min     Max    Med |   req/s  failures/s
--------|----------------------------------------------------------------------------|-------|-------------|-------|-------|-------|------|--------|-----------
GET      /                                                                                 10     0(0.00%) |     49      45      66     47 |    0.00        0.00
GET      /cart                                                                             13     0(0.00%) |     29      24      41     26 |    0.70        0.00
POST     /cart                                                                             11     0(0.00%) |     46      43      58     45 |    0.30        0.00
POST     /cart/checkout                                                                     5     0(0.00%) |     41      34      51     39 |    0.30        0.00
GET      /product/0PUK6V6EV0                                                                1     0(0.00%) |     27      27      27     27 |    0.10        0.00
GET      /product/1YMWWN1N4O                                                                6     0(0.00%) |     27      25      31     26 |    0.30        0.00
GET      /product/2ZYFJ3GM2N                                                                4     0(0.00%) |     26      26      27     26 |    0.10        0.00
GET      /product/66VCHSJNUP                                                                4     0(0.00%) |     27      26      30     27 |    0.10        0.00
GET      /product/6E92ZMYYFZ                                                                6     0(0.00%) |     26      25      28     26 |    0.10        0.00
GET      /product/9SIQT8TOJO                                                                8     0(0.00%) |     27      26      31     27 |    0.20        0.00
GET      /product/L9ECAV7KIM                                                                3     0(0.00%) |     27      27      27     27 |    0.10        0.00
GET      /product/LS4PSXUNUM                                                                5     0(0.00%) |     38      26      88     26 |    0.10        0.00
GET      /product/OLJCESPC7Z                                                                5     0(0.00%) |     31      27      40     28 |    0.10        0.00
POST     /setCurrency                                                                       2     0(0.00%) |     53      49      58     49 |    0.00        0.00
--------|----------------------------------------------------------------------------|-------|-------------|-------|-------|-------|------|--------|-----------
         Aggregated                                                                        83     0(0.00%) |     35      24      88     28 |    2.50        0.00
```

To more simplify the deployment, I now explicitly override the entrypoint of the Docker container to directly use the Locust command with the correct parameters.

docker run -d -p 8089:8089 --entrypoint locust locust-loadgen --host="http://34.65.53.172" --headless --users=1000 --run-time=1m --spawn-rate=1

4) Deploying automatically the load generator in Google Cloud

In this step, I used Terraform to automate the deployment of the load generator on a Google Compute Engine virtual machine. Initially, the Terraform file I cloned from the repository was configured for Autopilot mode, which I modified to deploy resources in standard mode instead.
To automate the creation of the VM and the deployment of the load generator I wrote a new Terraform file (load_generator.tf) to create a VM instance with the required specifications (e2-standard-2 machine type and Debian 11 as the OS)
I also included a startup script that:

- Install Docker on the VM
- Clone the Google Cloud microservices-demo repository
- Build the Locust load generator Docker image and run it with predefined parameters.

The problems I encountered:

- One issue I faced was extracting the external IP address of the frontend services automatically. I have to do manually by looking at the kubectl get services command to find it and to add it in the load_geneartor.tf file before applying the terraform.
- I also tried to avoid cloning the repository each time the VM starts, but I did not succeed in doing so.

Advanced steps

1) Monitoring the application and the infrastructure

For this step, I use Prometheus and Grafana as specified in the assignment.
I deployed both tools using Helm charts from the Prometheus-community and Grafana repositories.
Challenges faced: I encountered an issue with Grafana displaying the error “origin not allowed” when I tried to add a data source. I spend a lot of time troubleshooting this problem and explore multiple solution, including running Grafana on my local machine, but the issue persisted. Eventually, I found a solution in a YouTube video where the service type for Grafana was changed to LoadBalancer. This resolved the error and I was able to access Grafana successfully and add the data source Prometheus.
To access Prometheus, I used port forwarding and to access Grafana I retrieve the external IP of the Grafana service using:

```
kubectl get services -n monitoring
```

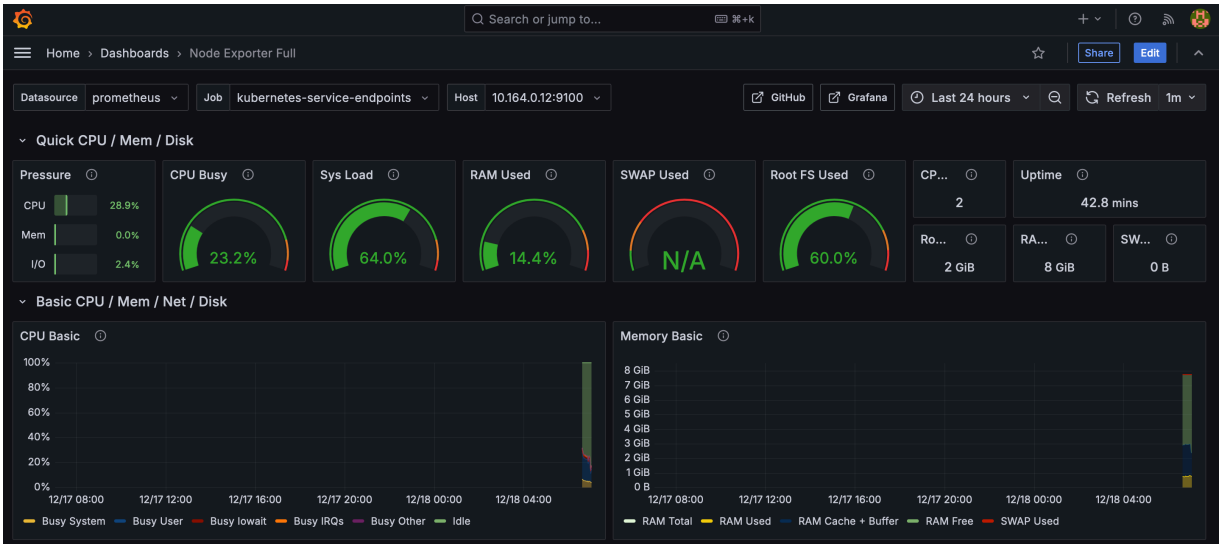
Then access Grafana at `http://<external-ip>`.

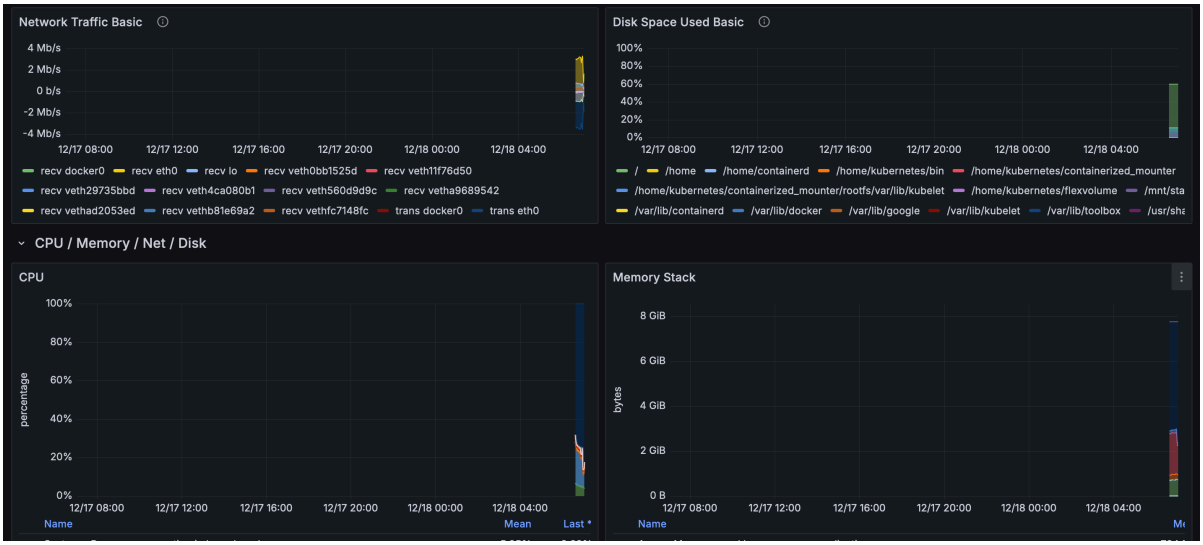
When deploying Prometheus through the Prometheus-community Helm chart, the following components were included: Node exporter and cadvisor.

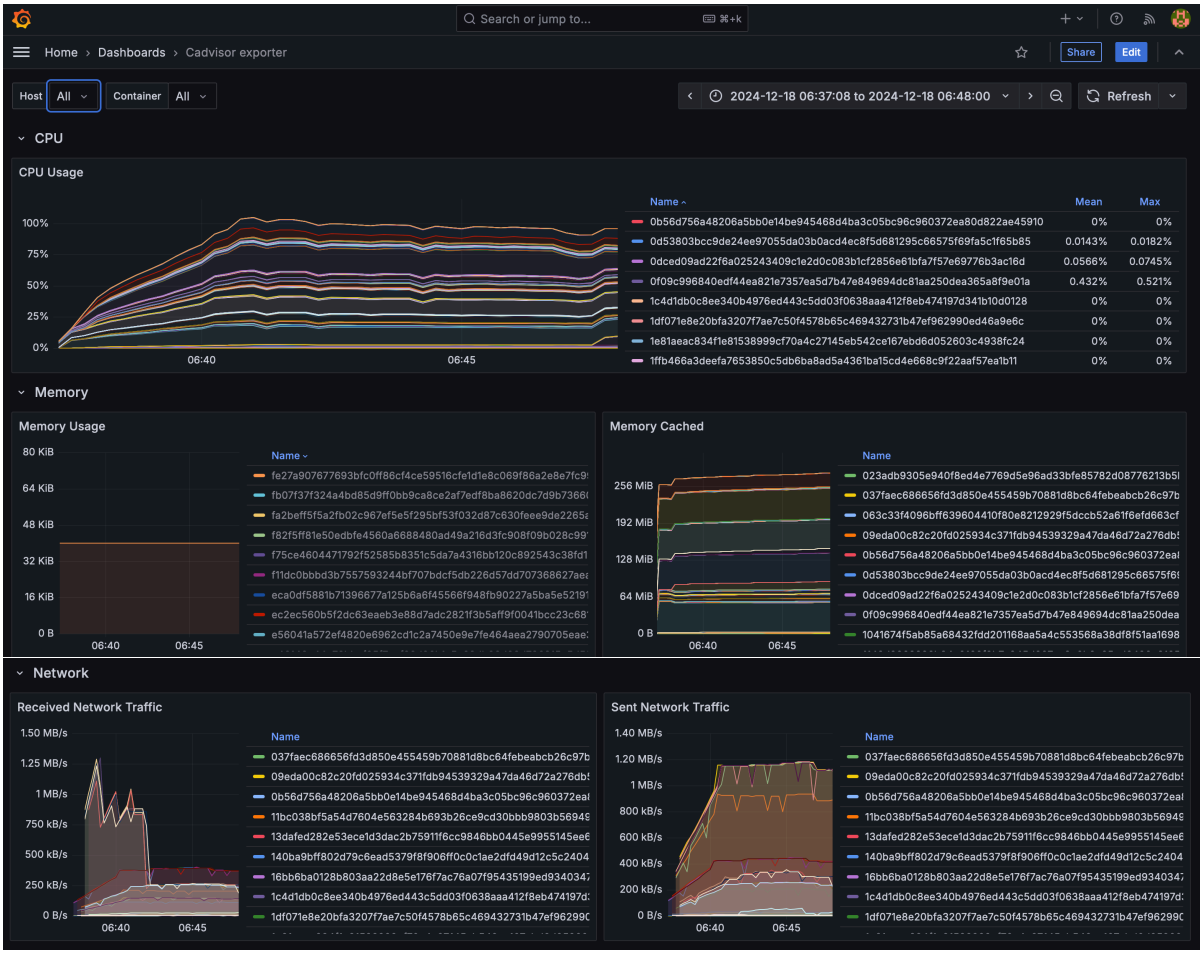
Grafana dashboards:
I imported pre-configured dashboards for visualizing the collecting metrics:

- Dashboard ID = 14282: Kubernetes cAdvisor metrics
- Dashboard ID = 1860: node exporter metrics

Results: Here are the results I obtained







2) Performance evaluation

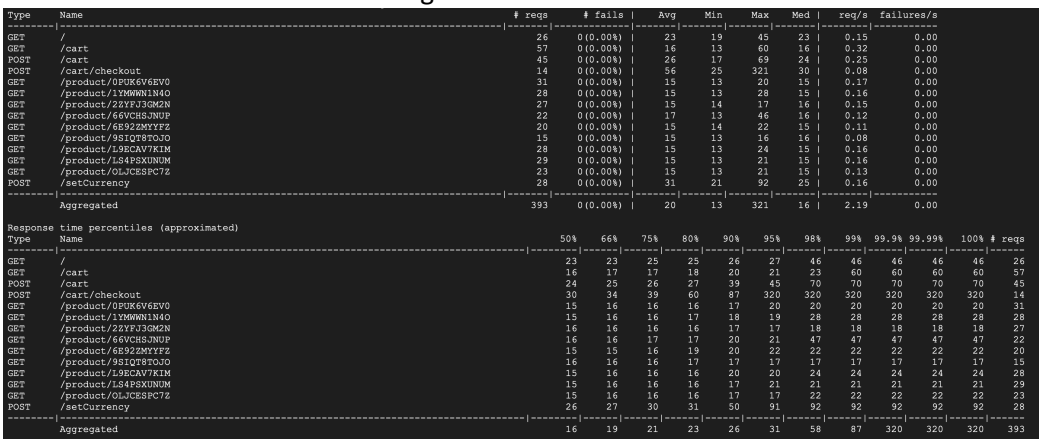
For the performance evaluation, I did multiple tests to analyze the system behavior.

a) First Test

The first test focused on using locust as the load testing tool to evaluate system performance.
I progressively increased the number of simulated users, starting with 10 users, then 100 users and finally 5000 users. The test duration was set to 3 minutes for each load level.
Although, I wanted to use locust's CVS export feature to save the test result, I encountered difficulties in generating the CVS files. Instead, I took screenshots of the final results at the end of each test.

Test Result:

- With Users = 10 and Running time = 3 min

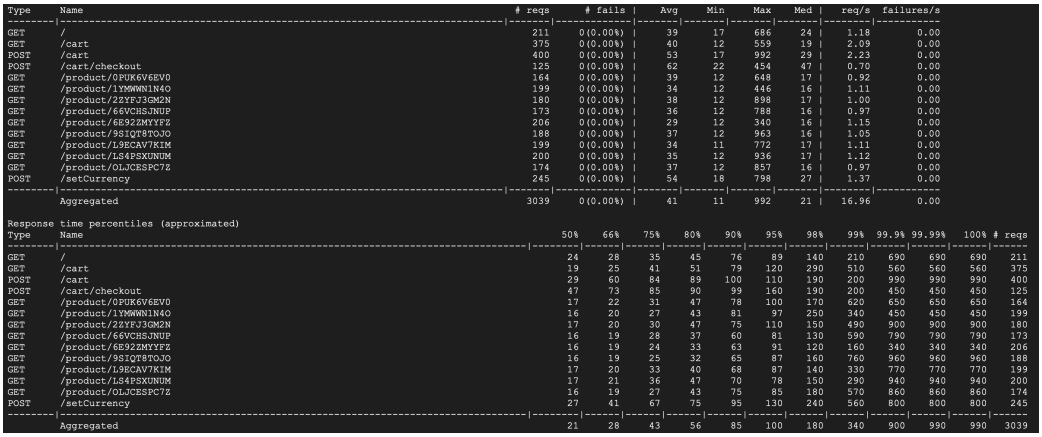


Type	Name	# reqs	# fails	Avg	Min	Max	Med	req/s	failures/s
GET	/	26	0(0.00%)	23	19	45	23	0.15	0.00
GET	/cart	57	0(0.00%)	16	13	60	16	0.32	0.00
POST	/cart	45	0(0.00%)	26	17	69	24	0.25	0.00
POST	/cart/checkout	14	0(0.00%)	56	25	321	30	0.08	0.00
GET	/product/0PUK6V6EV0	31	0(0.00%)	15	13	20	15	0.17	0.00
GET	/product/1YMWWN1N4O	28	0(0.00%)	15	13	28	15	0.16	0.00
GET	/product/2ZYFJ3GM2N	27	0(0.00%)	15	14	17	16	0.15	0.00
GET	/product/66VCHSJNUP	22	0(0.00%)	17	13	46	16	0.12	0.00
GET	/product/6E92ZMYYFZ	20	0(0.00%)	15	14	22	15	0.11	0.00
GET	/product/9SIQT8TOJO	15	0(0.00%)	15	13	16	16	0.08	0.00
GET	/product/L9ECAV7KIM	28	0(0.00%)	15	13	24	15	0.16	0.00
GET	/product/LS4PSXUNUM	29	0(0.00%)	15	13	21	15	0.16	0.00
GET	/product/OLJCESPC7Z	23	0(0.00%)	15	13	21	15	0.13	0.00
POST	/setCurrency	28	0(0.00%)	31	21	92	25	0.16	0.00
	Aggregated	393	0(0.00%)	20	13	321	16	2.19	0.00

Response time percentiles (approximated)

Type	Name	50%	66%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%	# reqs
GET	/	23	23	25	25	26	27	46	46	46	46	46	26
GET	/cart	16	17	17	18	20	21	23	60	60	60	60	57
POST	/cart	24	25	26	27	39	45	70	70	70	70	70	45
POST	/cart/checkout	30	34	39	60	87	320	320	320	320	320	320	14
GET	/product/0PUK6V6EV0	15	16	16	16	17	20	20	20	20	20	20	31
GET	/product/1YMWWN1N4O	15	16	16	17	18	19	28	28	28	28	28	28
GET	/product/2ZYFJ3GM2N	16	16	16	16	17	17	18	18	18	18	18	27
GET	/product/66VCHSJNUP	16	16	17	17	20	21	47	47	47	47	47	22
GET	/product/6E92ZMYYFZ	15	15	16	19	20	22	22	22	22	22	22	20
GET	/product/9SIQT8TOJO	16	16	16	17	17	17	17	17	17	17	17	15
GET	/product/L9ECAV7KIM	15	16	16	16	20	20	24	24	24	24	24	28
GET	/product/LS4PSXUNUM	15	16	16	16	17	21	21	21	21	21	21	29
GET	/product/OLJCESPC7Z	15	16	16	16	17	17	22	22	22	22	22	23
POST	/setCurrency	26	27	30	31	50	91	92	92	92	92	92	28
	Aggregated	16	19	21	23	26	31	58	87	320	320	320	393

- With Users = 100 and Running time = 3 min

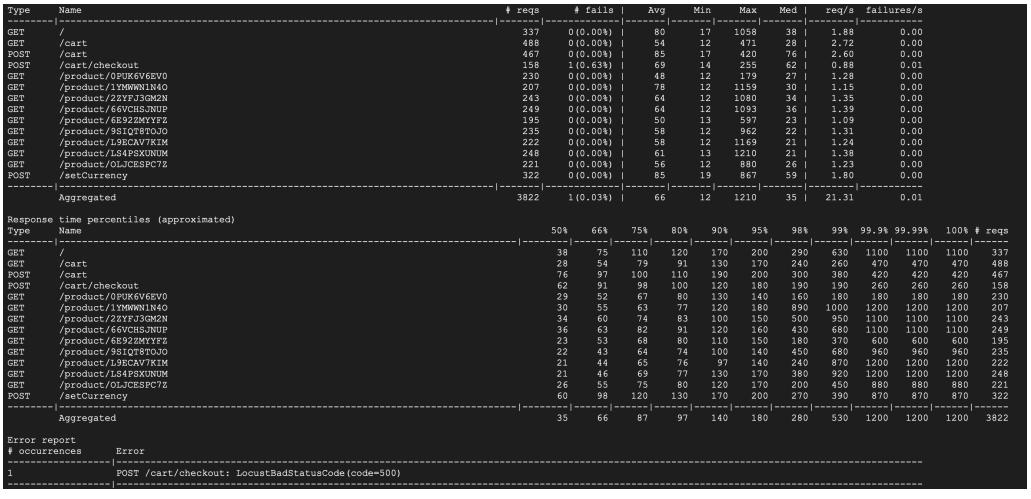


Type	Name	# reqs	# fails	Avg	Min	Max	Med	req/s	failures/s
GET	/	211	0(0.00%)	39	17	686	24	1.18	0.00
GET	/cart	375	0(0.00%)	40	12	559	19	2.09	0.00
POST	/cart	400	0(0.00%)	53	17	992	29	2.23	0.00
POST	/cart/checkout	125	0(0.00%)	62	22	454	47	0.70	0.00
GET	/product/0PUK6V6EV0	164	0(0.00%)	39	12	648	17	0.92	0.00
GET	/product/1YMWWN1N4O	199	0(0.00%)	34	12	446	16	1.11	0.00
GET	/product/2ZYFJ3GM2N	180	0(0.00%)	38	12	898	17	1.00	0.00
GET	/product/66VCHSJNUP	173	0(0.00%)	36	12	788	16	0.97	0.00
GET	/product/6E92ZMYYFZ	206	0(0.00%)	29	12	340	16	1.15	0.00
GET	/product/9SIQT8TOJO	188	0(0.00%)	37	12	963	16	1.05	0.00
GET	/product/L9ECAV7KIM	199	0(0.00%)	34	11	772	17	1.11	0.00
GET	/product/LS4PSXUNUM	200	0(0.00%)	35	12	936	17	1.12	0.00
GET	/product/OLJCESPC7Z	174	0(0.00%)	37	12	857	16	0.97	0.00
POST	/setCurrency	245	0(0.00%)	54	18	798	27	1.37	0.00
	Aggregated	3039	0(0.00%)	41	11	992	21	16.96	0.00

Response time percentiles (approximated)

Type	Name	50%	66%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%	# reqs
GET	/	24	28	35	45	76	89	140	210	690	690	690	211
GET	/cart	19	25	41	51	79	120	290	510	560	560	560	375
POST	/cart	29	60	84	89	100	110	190	200	990	990	990	400
POST	/cart/checkout	47	73	85	90	99	160	190	200	450	450	450	125
GET	/product/0PUK6V6EV0	17	22	31	47	78	100	170	620	650	650	650	164
GET	/product/1YMWWN1N4O	16	20	27	43	81	97	250	340	450	450	450	199
GET	/product/2ZYFJ3GM2N	17	20	30	47	75	110	150	490	900	900	900	180
GET	/product/66VCHSJNUP	16	19	28	37	60	81	130	590	790	790	790	173
GET	/product/6E92ZMYYFZ	16	19	24	33	63	91	120	160	340	340	340	206
GET	/product/9SIQT8TOJO	16	19	25	32	65	87	160	760	960	960	960	188
GET	/product/L9ECAV7KIM	17	20	33	40	68	87	140	330	770	770	770	199
GET	/product/LS4PSXUNUM	17	21	36	47	70	78	150	290	940	940	940	200
GET	/product/OLJCESPC7Z	16	19	27	43	75	85	180	570	860	860	860	174
POST	/setCurrency	27	41	67	75	95	130	240	560	800	800	800	245
	Aggregated	21	28	43	56	85	100	180	340	900	990	990	3039

- With Users = 5000 and Running time = 3 min



Type	Name	# reqs	# fails	Avg	Min	Max	Med	req/s	failures/s
GET	/	337	0(0.00%)	80	17	1058	38	1.88	0.00
GET	/cart	488	0(0.00%)	54	12	471	28	2.72	0.00
POST	/cart	467	0(0.00%)	85	17	420	76	2.60	0.00
POST	/cart/checkout	158	1(0.63%)	69	14	255	62	0.88	0.01
GET	/product/0PUK6V6EV0	230	0(0.00%)	48	12	179	27	1.28	0.00
GET	/product/1YMWWN1N4O	207	0(0.00%)	78	12	1159	30	1.15	0.00
GET	/product/2ZYFJ3GM2N	243	0(0.00%)	64	12	1080	34	1.35	0.00
GET	/product/66VCHSJNUP	249	0(0.00%)	64	12	1093	36	1.39	0.00
GET	/product/6E92ZMYYFZ	195	0(0.00%)	50	13	597	23	1.09	0.00
GET	/product/9SIQT8TOJO	235	0(0.00%)	58	12	962	22	1.31	0.00
GET	/product/L9ECAV7KIM	222	0(0.00%)	58	12	1169	21	1.24	0.00
GET	/product/LS4PSXUNUM	248	0(0.00%)	61	13	1210	21	1.38	0.00
GET	/product/OLJCESPC7Z	221	0(0.00%)	56	12	880	26	1.23	0.00
POST	/setCurrency	322	0(0.00%)	85	19	867	59	1.80	0.00
	Aggregated	3822	1(0.03%)	66	12	1210	35	21.31	0.01

Response time percentiles (approximated)

Type	Name	50%	66%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%	# reqs
GET	/	38	75	110	120	170	200	290	630	1100	1100	1100	337
GET	/cart	28	54	79	91	130	170	240	260	470	470	470	488
POST	/cart	76	97	100	110	190	200	300	380	420	420	420	467
POST	/cart/checkout	62	91	98	100	120	180	190	190	260	260	260	158
GET	/product/0PUK6V6EV0	29	52	67	80	130	140	160	180	180	180	180	230
GET	/product/1YMWWN1N4O	30	55	63	77	120	180	890	1000	1200	1200	1200	207
GET	/product/2ZYFJ3GM2N	34	60	74	83	100	150	500	950	1100	1100	1100	243
GET	/product/66VCHSJNUP	36	63	82	91	120	160	430	680	1100	1100	1100	249
GET	/product/6E92ZMYYFZ	23	53	68	80	110	150	180	370	600	600	600	195
GET	/product/9SIQT8TOJO	22	43	64	74	100	140	450	680	960	960	960	235
GET	/product/L9ECAV7KIM	21	44	65	76	97	140	240	870	1200	1200	1200	222
GET	/product/LS4PSXUNUM	21	46	69	77	130	170	380	920	1200	1200	1200	248
GET	/product/OLJCESPC7Z	26	55	75	80	120	170	200	450	880	880	880	221
POST	/setCurrency	60	98	120	130	170	200	270	390	870	870	870	322
	Aggregated	35	66	87	97	140	180	280	530	1200	1200	1200	3822

Error report

# occurences	Error
1	POST /cart/checkout: LocustBadStatusCode(code=500)

Result Analysis:
As the number of users increased, the system scaled well, handling a higher number of requests efficiently. The throughput (requests per second) also increased proportionally with the user load, showing the system's ability to manage growing traffic.

However, as the user load grew, the response time also increased. This indicates that while the system can handle higher traffic, it experiences slight delays under heavy load.

- GET /cart and POST /cart: These endpoints handled the most requests. They showed a slight increase in response times but no failures.
- POST /cart/checkout: This endpoint encountered a 500 error under 5000 users, suggesting a potential bottleneck or resource issue that needs further investigation.

b) Second test

In the second test I did, I keep 5000 users but extended the duration to 10 minutes to observe if failures occurred over time.

Test Result:

Type	Name	# reqs	# fails	Avg	Min	Max	Med	req/s	failures/s
GET	/	2017	0(0.00%)	1364	18	5433	1000	3.37	0.00
GET	/cart	4275	0(0.00%)	1092	11	5797	820	7.13	0.00
POST	/cart	4154	0(0.00%)	1506	17	6004	1200	6.93	0.00
POST	/cart/checkout	1400	6(0.43%)	823	23	3401	610	2.34	0.01
GET	/product/0PUK6V6EV0	1957	0(0.00%)	955	13	3283	750	3.27	0.00
GET	/product/1YMWWN1N4O	1927	0(0.00%)	971	12	3374	810	3.21	0.00
GET	/product/2ZYFJ3GM2N	2035	0(0.00%)	985	13	3404	820	3.40	0.00
GET	/product/66VCHSJNUP	2020	0(0.00%)	994	12	3373	870	3.37	0.00
GET	/product/6E92ZMYYFZ	1982	0(0.00%)	985	12	3156	810	3.31	0.00
GET	/product/9SIQT8TOJO	1974	0(0.00%)	968	12	3130	790	3.29	0.00
GET	/product/L9ECAV7KIM	2002	0(0.00%)	948	13	3151	760	3.34	0.00
GET	/product/LS4PSXUNUM	2057	0(0.00%)	970	13	3393	800	3.43	0.00
GET	/product/OLJCESPC7Z	1997	0(0.00%)	995	13	3421	830	3.33	0.00
POST	/setCurrency	2698	0(0.00%)	1559	20	5178	1300	4.50	0.00
	Aggregated	32495	6(0.02%)	1124	11	6004	890	54.21	0.01

Response time percentiles (approximated)

Type	Name	50%	66%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%	# reqs
GET	/	1000	1800	2300	2600	3200	3500	3900	4300	5200	5400	5400	2017
GET	/cart	820	1400	1700	1900	2500	3000	3600	4000	5000	5800	5800	4275
POST	/cart	1200	2000	2500	2700	3300	3800	4400	4700	5600	6000	6000	4154
POST	/cart/checkout	610	1100	1400	1500	1800	2000	2300	2500	3000	3400	3400	1400
GET	/product/0PUK6V6EV0	750	1300	1600	1800	2100	2300	2600	2700	3100	3300	3300	1957
GET	/product/1YMWWN1N4O	810	1300	1600	1800	2100	2300	2600	2700	3100	3400	3400	1927
GET	/product/2ZYFJ3GM2N	820	1400	1600	1800	2100	2400	2600	2800	3100	3400	3400	2035
GET	/product/66VCHSJNUP	870	1400	1600	1800	2100	2400	2600	2800	3300	3400	3400	2020
GET	/product/6E92ZMYYFZ	810	1400	1600	1800	2100	2400	2700	2800	3100	3200	3200	1982
GET	/product/9SIQT8TOJO	790	1300	1600	1800	2100	2300	2600	2800	3100	3100	3100	1974
GET	/product/L9ECAV7KIM	760	1300	1600	1700	2100	2300	2600	2700	3000	3200	3200	2002
GET	/product/LS4PSXUNUM	800	1400	1600	1800	2100	2400	2600	2800	3300	3400	3400	2057
GET	/product/OLJCESPC7Z	830	1300	1600	1800	2200	2400	2700	2800	3300	3400	3400	1997
POST	/setCurrency	1300	2100	2600	2800	3400	3700	4000	4200	5000	5200	5200	2698
	Aggregated	890	1500	1800	2000	2500	3000	3600	3900	5000	5800	6000	32495

Error report

# occurrences	Error
6	POST /cart/checkout: LocustBadStatusCode(code=500)

Result Analysis:
We can see that the endpoints POST /cart/checkout continues to fail under load, with 6 instances of 500 errors. This indicates a potential bottleneck or an issue with resource allocation.

c) Third test

The third test I did was by changing the number of spawn_rate to simulate a big spike, to see how the system behave.
I used user=5000 and spawn_rate=50

Test result:

Type	Name	# reqs	# fails	Avg	Min	Max	Med	req/s	failures/s
GET	/	4747	4111(86.60%)	53489	1	133107	60000	26.38	22.85
GET	/cart	926	662(71.49%)	28544	0	118496	7300	5.15	3.68
POST	/cart	608	464(76.32%)	20367	1	123032	6200	3.38	2.58
POST	/cart/checkout	135	92(68.15%)	8537	5	117722	42	0.75	0.51
GET	/product/0PUK6V6EV0	454	326(71.81%)	31005	1	116451	8500	2.52	1.81
GET	/product/1YMWWN1N4O	435	312(71.72%)	31288	1	117287	7900	2.42	1.73
GET	/product/2ZYFJ3GM2N	428	309(72.20%)	28812	1	116965	7800	2.38	1.72
GET	/product/66VCHSJNUP	468	333(71.15%)	31278	1	115035	8200	2.60	1.85
GET	/product/6E92ZMYYFZ	462	353(76.41%)	28251	1	115186	7700	2.57	1.96
GET	/product/9SIQT8TOJO	490	359(73.27%)	27884	1	117209	7700	2.72	2.00
GET	/product/L9ECAV7KIM	443	316(71.33%)	30637	1	114966	7800	2.46	1.76
GET	/product/LS4PSXUNUM	469	335(71.43%)	32466	1	116442	7800	2.61	1.86
GET	/product/OLJCESPC7Z	449	334(74.39%)	29938	1	114334	7800	2.50	1.86
POST	/setCurrency	525	488(92.95%)	40682	1	129875	15000	2.92	2.71
	Aggregated	11039	8794(79.66%)	39754	0	133107	18000	61.36	48.88

Response time percentiles (approximated)

Type	Name	50%	66%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%	# reqs
GET	/	60000	72000	86000	98000	115000	124000	129000	132000	133000	133000	133000	4747
GET	/cart	7300	25000	59000	65000	100000	108000	112000	115000	118000	118000	118000	926
POST	/cart	6200	13000	23000	33000	69000	109000	111000	117000	123000	123000	123000	608
POST	/cart/checkout	42	4000	7500	8700	16000	62000	116000	117000	118000	118000	118000	135
GET	/product/0PUK6V6EV0	8500	31000	62000	68000	99000	106000	109000	110000	116000	116000	116000	454
GET	/product/1YMWWN1N4O	7900	43000	64000	69000	99000	108000	112000	113000	117000	117000	117000	435
GET	/product/2ZYFJ3GM2N	7800	26000	61000	67000	98000	107000	108000	111000	117000	117000	117000	428
GET	/product/66VCHSJNUP	8200	30000	64000	71000	99000	107000	109000	113000	115000	115000	115000	468
GET	/product/6E92ZMYYFZ	7700	25000	60000	66000	98000	105000	109000	112000	115000	115000	115000	462
GET	/product/9SIQT8TOJO	7700	25000	60000	65000	98000	108000	112000	116000	117000	117000	117000	490
GET	/product/L9ECAV7KIM	7800	30000	62000	68000	99000	108000	112000	114000	115000	115000	115000	443
GET	/product/LS4PSXUNUM	7800	41000	65000	71000	102000	108000	112000	114000	116000	116000	116000	469
GET	/product/OLJCESPC7Z	7800	28000	62000	67000	98000	105000	109000	112000	114000	114000	114000	449
POST	/setCurrency	15000	66000	73000	97000	108000	121000	127000	129000	130000	130000	130000	525
	Aggregated	18000	61000	71000	81000	105000	115000	126000	129000	133000	133000	133000	11039

Error report

# occurrences	Error
30	POST /cart/checkout: LocustBadStatusCode(code=500)
870	GET /: LocustBadStatusCode(code=500)
53	GET /product/66VCHSJNUP: LocustBadStatusCode(code=500)
56	GET /product/L9ECAV7KIM: LocustBadStatusCode(code=500)
50	GET /product/LS4PSXUNUM: LocustBadStatusCode(code=500)
98	POST /cart: LocustBadStatusCode(code=500)
64	GET /product/6E92ZMYYFZ: LocustBadStatusCode(code=500)
101	POST /setCurrency: LocustBadStatusCode(code=500)
71	GET /product/9SIQT8TOJO: LocustBadStatusCode(code=500)
138	GET /cart: LocustBadStatusCode(code=500)
58	GET /product/2ZYFJ3GM2N: LocustBadStatusCode(code=500)
57	GET /product/OLJCESPC7Z: LocustBadStatusCode(code=500)
61	GET /product/1YMWWN1N4O: LocustBadStatusCode(code=500)
58	GET /product/0PUK6V6EV0: LocustBadStatusCode(code=500)
736	GET /: ConnectionRefusedError(111, 'Connection refused')
147	GET /product/LS4PSXUNUM: ConnectionRefusedError(111, 'Connection refused')
146	GET /product/66VCHSJNUP: ConnectionRefusedError(111, 'Connection refused')
140	GET /product/0PUK6V6EV0: ConnectionRefusedError(111, 'Connection refused')
193	POST /setCurrency: ConnectionRefusedError(111, 'Connection refused')
27	GET /: ConnectionResetError(104, 'Connection reset by peer')
1	GET /product/0PUK6V6EV0: ConnectionResetError(104, 'Connection reset by peer')
1	GET /cart: ConnectionResetError(104, 'Connection reset by peer')
129	GET /product/L9ECAV7KIM: ConnectionRefusedError(111, 'Connection refused')
290	GET /cart: ConnectionRefusedError(111, 'Connection refused')
130	GET /product/1YMWWN1N4O: ConnectionRefusedError(111, 'Connection refused')
167	GET /product/9SIQT8TOJO: ConnectionRefusedError(111, 'Connection refused')
154	GET /product/OLJCESPC7Z: ConnectionRefusedError(111, 'Connection refused')
140	GET /product/2ZYFJ3GM2N: ConnectionRefusedError(111, 'Connection refused')
171	GET /product/6E92ZMYYFZ: ConnectionRefusedError(111, 'Connection refused')
234	POST /cart: ConnectionRefusedError(111, 'Connection refused')
53	POST /cart/checkout: ConnectionRefusedError(111, 'Connection refused')
1	POST /cart: ConnectionResetError(104, 'Connection reset by peer')
1	GET /product/LS4PSXUNUM: ConnectionResetError(104, 'Connection reset by peer')
1	GET /product/6E92ZMYYFZ: ConnectionResetError(104, 'Connection reset by peer')
1	POST /setCurrency: ConnectionResetError(104, 'Connection reset by peer')
1	GET /product/1YMWWN1N4O: ConnectionResetError(104, 'Connection reset by peer')
2478	GET /: RetriesExceeded('http://34.65.53.172/', 0, original=timed out)
81	POST /cart: RetriesExceeded('http://34.65.53.172/cart', 0, original=timed out)
229	GET /cart: RetriesExceeded('http://34.65.53.172/cart', 0, original=timed out)
111	GET /product/2ZYFJ3GM2N: RetriesExceeded('http://34.65.53.172/product/2ZYFJ3GM2N', 0, original=timed out)
193	POST /setCurrency: RetriesExceeded('http://34.65.53.172/setCurrency', 0, original=timed out)
131	GET /product/L9ECAV7KIM: RetriesExceeded('http://34.65.53.172/product/L9ECAV7KIM', 0, original=timed out)
121	GET /product/9SIQT8TOJO: RetriesExceeded('http://34.65.53.172/product/9SIQT8TOJO', 0, original=timed out)
123	GET /product/OLJCESPC7Z: RetriesExceeded('http://34.65.53.172/product/OLJCESPC7Z', 0, original=timed out)
137	GET /product/LS4PSXUNUM: RetriesExceeded('http://34.65.53.172/product/LS4PSXUNUM', 0, original=timed out)
120	GET /product/1YMWWN1N4O: RetriesExceeded('http://34.65.53.172/product/1YMWWN1N4O', 0, original=timed out)
127	GET /product/0PUK6V6EV0: RetriesExceeded('http://34.65.53.172/product/0PUK6V6EV0', 0, original=timed out)
134	GET /product/66VCHSJNUP: RetriesExceeded('http://34.65.53.172/product/66VCHSJNUP', 0, original=timed out)
117	GET /product/6E92ZMYYFZ: RetriesExceeded('http://34.65.53.172/product/6E92ZMYYFZ', 0, original=timed out)
7	POST /cart/checkout: RetriesExceeded('http://34.65.53.172/cart/checkout', 0, original=timed out)
50	POST /cart: HTTPParseError('Incomplete response.')
4	GET /cart: HTTPParseError('Incomplete response.')
2	POST /cart/checkout: HTTPParseError('Incomplete response.')

Result Analysis: The system showed significant struggles under a sudden spike in load with a spawn rate of 50:

- The sharp increase in users caused significant delays and failures.
- Response times increased dramatically, with an average response time of 39.7 seconds and a median response time of 18 seconds.
- Multiple types of errors were observed, including:
    - 500 errors (server-side failures)
    - Connection refused
    - Connection reset
    - Timeout errors
- The error report highlights that POST /cart/checkout experienced a high failure rate, primarily due to 500 errors and connection resets, indicating a critical bottleneck in the system.

To identify potential bottleneck, I look at multiple metrics in the dashboard by simulating user=5000:

a) CPU usage

I look at the CPU Busy and System Load at the Node Exporter Dashboard and the CPU usage in the Cadvisor dashboard. I tried to identify which service or container is consuming the most CPU and depending on that we can use horizontal scaling.

b) Memory Usage

I looked for the RAM Used and the memory info in Node Exporter dashboard and the memory usage and cached in cadvisor dashboard. I did that to identify containers or nodes using the most memory. And depending on that we can use vertical scaling by adding more RAM.

c) Network traffic

I look for network info in both dashboards to find a sudden drop in network traffic that may indicate a network bottleneck where the system stops processing request due to resource exhaustion. And depending on that we can use load balancing to distribute traffic across multiple instances

d) System Load

A greater system load than the number of CPU cores can indicate CPU contention. If so, we can use horizontal scaling by adding more CPU cores or additional nodes.

Here are the results from the different dashboards:

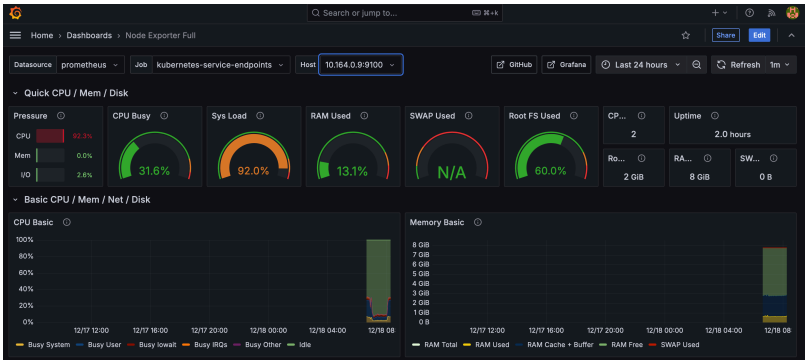
Node Exporter Result:

CPU Pressure and System Load:

- Host 10.164.0.9:9100: CPU pressure reach 92%, and System Load over 101%, indicating that the CPU's capacity is passed.

Memory and I/O:

- RAM usage remained below 15% on all host, indicating that memory is not a bottleneck.
- I/O utilization was also low (<7%), confirming that disk input/output operation did not contribute to delays.



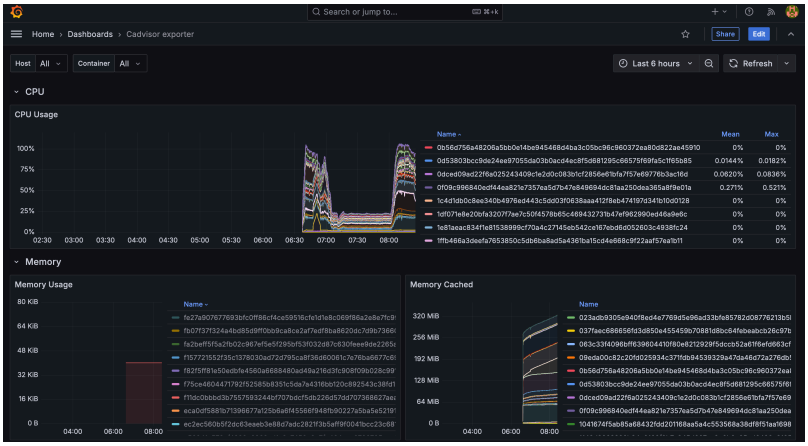
cAdvisor Result:

CPU usage:

- Pool 6a7a090c-9510: Increase progressively and passed 40%
- Pool 6a7a090c-g8bq: had a remaining CPU usage at around 38%
- After a while: a sharp spike in CPU usage across multiple containers occurred after a period of stability (as see in figure: around 7:30-08:00)
- This spike was at the same time when failures was observed in the Docker logs.

Memory usage:

- Memory consumption remained relatively low but memory cached increase slightly over time.



Result Analysis: To investigate the high CPU pressure and system load observed on the host 10.164.0.9, I follow different steps:

a) Identify the Node

I listed the node in the cluster using: kubectl get nodes -o wide

```
NAME                                          STATUS   ROLES    AGE   VERSION             INTERNAL-IP   EXTERNAL-IP      OS-IMAGE                             KERNEL-VERSION
CONTAINER-RUNTIME
gke-online-boutique-default-pool-6a7a090c-5mqm   Ready    <none>   3h1m   v1.30.6-gke.1125000   10.164.0.11   34.13.192.61     Container-Optimized OS from Google   6.1.112+
containerd://1.7.23
gke-online-boutique-default-pool-6a7a090c-8p7p   Ready    <none>   3h1m   v1.30.6-gke.1125000   10.164.0.10   34.90.17.141     Container-Optimized OS from Google   6.1.112+
containerd://1.7.23
gke-online-boutique-default-pool-6a7a090c-9510   Ready    <none>   3h1m   v1.30.6-gke.1125000   10.164.0.9    34.90.144.202    Container-Optimized OS from Google   6.1.112+
containerd://1.7.23
gke-online-boutique-default-pool-6a7a090c-g8bq   Ready    <none>   3h1m   v1.30.6-gke.1125000   10.164.0.12   34.13.129.66     Container-Optimized OS from Google   6.1.112+
containerd://1.7.23
```

This host belong to the node Pool-6a7a090c-9510, which align with earlier observations of high CPU usage on this node.

b) Identify Pods

Using kubectl get pods -o wide, I identified the pods running on this host:

- Redis-cart
- Frontend
- Productcatalogservice
- Shippingservice

```
NAME                                     READY   STATUS    RESTARTS      AGE   IP          NODE                                              NOMINATED NODE   READINESS GATES
adservice-766c974448-krxlt               1/1     Running   0             161m  10.80.3.3   gke-online-boutique-default-pool-6a7a090c-g8bq    <none>           <none>
cartservice-77549b5fd5-ghfvh             1/1     Running   0             161m  10.80.2.3   gke-online-boutique-default-pool-6a7a090c-5mqm    <none>           <none>
checkoutservice-7998f957b-lrg5n          1/1     Running   0             161m  10.80.2.4   gke-online-boutique-default-pool-6a7a090c-5mqm    <none>           <none>
currencyservice-6b8bcdcb56-grp8s         1/1     Running   1 (39m ago)   161m  10.80.3.4   gke-online-boutique-default-pool-6a7a090c-g8bq    <none>           <none>
emailservice-b745c6cb4-2zs5h             1/1     Running   0             161m  10.80.2.5   gke-online-boutique-default-pool-6a7a090c-5mqm    <none>           <none>
frontend-58f4b9c4c9-jlr8s                1/1     Running   11 (37m ago)  161m  10.80.0.4   gke-online-boutique-default-pool-6a7a090c-9510    <none>           <none>
loadgenerator-5595ff587d-fjmsm           1/1     Running   0             161m  10.80.2.6   gke-online-boutique-default-pool-6a7a090c-5mqm    <none>           <none>
paymentservice-68f5d8498d-8rg4s          1/1     Running   0             161m  10.80.3.5   gke-online-boutique-default-pool-6a7a090c-g8bq    <none>           <none>
productcatalogservice-7bc9878974-j8td8   1/1     Running   0             161m  10.80.0.5   gke-online-boutique-default-pool-6a7a090c-9510    <none>           <none>
recommendationservice-56c9f8f46b-pzvfk   1/1     Running   0             161m  10.80.3.6   gke-online-boutique-default-pool-6a7a090c-g8bq    <none>           <none>
redis-cart-7ff8f4d6ff-vqgc6              1/1     Running   0             161m  10.80.0.6   gke-online-boutique-default-pool-6a7a090c-9510    <none>           <none>
shippingservice-6cf6fcd654-wrsh2         1/1     Running   0             161m  10.80.0.7   gke-online-boutique-default-pool-6a7a090c-9510    <none>           <none>
```

c) Resource usage analysis during a spike

To determine CPU usage for each pod during a spike (when failure occur), I ran: kubectl top pods

```
NAME                                     CPU(cores)   MEMORY(bytes)
adservice-766c974448-krxlt               13m          105Mi
cartservice-77549b5fd5-ghfvh             60m          66Mi
checkoutservice-7998f957b-lrg5n          17m          10Mi
currencyservice-6b8bcdcb56-grp8s         161m         34Mi
emailservice-b745c6cb4-2zs5h             6m           41Mi
frontend-58f4b9c4c9-jlr8s                200m         65Mi
loadgenerator-5595ff587d-fjmsm           6m           49Mi
paymentservice-68f5d8498d-8rg4s          5m           26Mi
productcatalogservice-7bc9878974-j8td8   72m          9Mi
recommendationservice-56c9f8f46b-pzvfk   115m         48Mi
redis-cart-7ff8f4d6ff-vqgc6              10m          4Mi
shippingservice-6cf6fcd654-wrsh2         9m           8Mi
```

I observed that the Frontend pod reaches 200m CPU usage, which matches its CPU limit defined in the YAML configuration file.

d) Logs Analysis for /cart/checkout failures

As most failure was observed on the /cart/checkout (see earlier analysis), I checked the logs for the Frontend service: kubectl logs frontend-58f4b9c4c9-jlr8s


```
{"http.req.id":"2ff9a270-c6bf-4400-926c-f5dcccf40074","http.req.method":"POST","http.req.path":"/cart/checkout","http.resp.bytes":6999,"http.resp.status":200,"http.resp.took_ms":29
,"message":"request complete","session":"662fb4c0-86b8-4ee6-8fbf-dee50f7146ae","severity":"debug","timestamp":"2024-12-18T09:29:31.461083662Z"}
{"http.req.id":"e51e4b03-68b1-41fc-b0a6-d1c9c4f36ca7","http.req.method":"GET","http.req.path":"/_healthz","message":"request started","session":"x-readiness-probe","severity":"debu
g","timestamp":"2024-12-18T09:29:32.239771504Z"}
{"http.req.id":"e51e4b03-68b1-41fc-b0a6-d1c9c4f36ca7","http.req.method":"GET","http.req.path":"/_healthz","http.resp.bytes":2,"http.resp.status":200,"http.resp.took_ms":0,"message"
:"request complete","session":"x-readiness-probe","severity":"debug","timestamp":"2024-12-18T09:29:32.239844249Z"}
{"http.req.id":"e9d2d278-d462-4ad0-8fff-04ad7b9e6fcf","http.req.method":"GET","http.req.path":"/_healthz","message":"request started","session":"x-liveness-probe","severity":"debug
","timestamp":"2024-12-18T09:29:32.239782146Z"}
{"http.req.id":"e9d2d278-d462-4ad0-8fff-04ad7b9e6fcf","http.req.method":"GET","http.req.path":"/_healthz","http.resp.bytes":2,"http.resp.status":200,"http.resp.took_ms":0,"message"
:"request complete","session":"x-liveness-probe","severity":"debug","timestamp":"2024-12-18T09:29:32.240024555Z"}
{"http.req.id":"cceba82e-d52b-46cb-b4bd-832fdc7f7a58","http.req.method":"GET","http.req.path":"/product/LS4PSXUNUM","message":"request started","session":"44efdca5-d9d4-45b6-9f65-4
f2e051fd2cd","severity":"debug","timestamp":"2024-12-18T09:29:32.519766256Z"}
{"currency":"USD","http.req.id":"cceba82e-d52b-46cb-b4bd-832fdc7f7a58","http.req.method":"GET","http.req.path":"/product/LS4PSXUNUM","id":"LS4PSXUNUM","message":"serving product pa
ge","session":"44efdca5-d9d4-45b6-9f65-4f2e051fd2cd","severity":"debug","timestamp":"2024-12-18T09:29:32.519841362Z"}
{"http.req.id":"cceba82e-d52b-46cb-b4bd-832fdc7f7a58","http.req.method":"GET","http.req.path":"/product/LS4PSXUNUM","http.resp.bytes":8234,"http.resp.status":200,"http.resp.took_ms
":17,"message":"request complete","session":"44efdca5-d9d4-45b6-9f65-4f2e051fd2cd","severity":"debug","timestamp":"2024-12-18T09:29:32.536921792Z"}
{"http.req.id":"81aed45e-8f00-41a3-b2a5-93652427fde9","http.req.method":"GET","http.req.path":"/product/L9ECAV7KIM","message":"request started","session":"1e0782db-f1f3-4052-b2b9-c
a3124f8ce36","severity":"debug","timestamp":"2024-12-18T09:29:32.577573924Z"}
```


Conclusion: The bottleneck seems to be caused by the Frontend service.

Possible solution: To address the high CPU usage observed on the frontend service, I tried horizontal scaling by manually adding 3 replicas using the following command:
kubectl scale deployment frontend --replicas=3

```
NAME                                     CPU(cores)   MEMORY(bytes)
adservice-766c974448-krxlt               42m          102Mi
cartservice-77549b5fd5-ghfvh             179m         66Mi
checkoutservice-7998f957b-lrg5n          17m          10Mi
currencyservice-6b8bcdcb56-grp8s         200m         31Mi
emailservice-b745c6cb4-2zs5h             6m           41Mi
frontend-58f4b9c4c9-2156d                157m         21Mi
frontend-58f4b9c4c9-jlr8s                144m         19Mi
frontend-58f4b9c4c9-mv4cr                145m         18Mi
loadgenerator-5595ff587d-fjmsm           6m           49Mi
paymentservice-68f5d8498d-8rg4s          7m           26Mi
productcatalogservice-7bc9878974-j8td8   181m         9Mi
recommendationservice-56c9f8f46b-pzvfk   162m         48Mi
redis-cart-7ff8f4d6ff-vqgc6              16m          5Mi
shippingservice-6cf6fcd654-wrsh2         22m          8Mi
```

Observation: After scaling, the CPU usage on the initial host 10.164.0.9 decreased significantly, which indicate that the load was distributed across multiple replicas. However, I observe that another host 10.164.0.12 started showing increased CPU usage, suggesting that the load was shifted rather than fully resolved. To improve this method, I considered using HPA to automatically scale the number of replicas based on CPU usage threshold (for example: cpu-percent = 70).

3) Canary releases

To implement the canary release, I did a visible change in the productcatalogservice. Specifically, I modified the product name "Sunglasses" to "Class Sunglasses" in the products.json file. This allowed me to easily verify whether traffic was being routed to the updated version of the service.

I began by creating two Docker images:

- v1, which contained the original product catalog with "Sunglasses".
- v2, which reflected the updated catalog with "Class Sunglasses".

After building and pushing the images, I updated the Kubernetes configuration to define separate deployments for v1 and v2. I then applied the manifests.

To manage traffic between v1 and v2, I configured Istio. This involved defining a DestinationRule with subsets for both versions and a VirtualService to split traffic based on predefined percentages. For example, I initially set 75% of traffic to route to v1 and 25% to v2.

For testing, I refreshed the frontend repeatedly to observe whether "Class Sunglasses" appeared, indicating traffic was being routed to v2. Initially, I encountered some unexpected results. At one point, I was able to see "Class Sunglasses" intermittently, but I also realized I had modified the frontend to default the currency to EUR, which may have influenced the behavior. This introduced some uncertainty into the testing process.

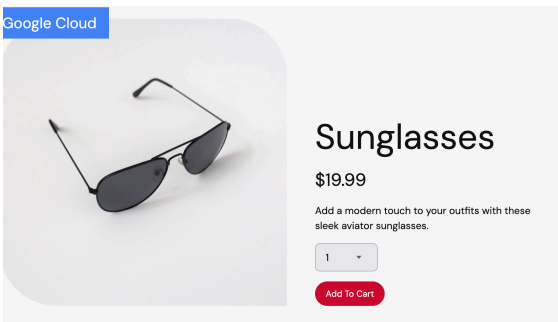
When I tried to replicate the setup later, I faced challenges in reproducing the earlier results. Despite following the same steps, the expected traffic splitting wasn't working.

However, during the initial successful test phase, I did the following tests:

1. With 75% traffic to v1 and 25% to v2, I observed "Class Sunglasses" in 11 out of 50 requests (22%).
2. When adjusting to 50% traffic to v1 and 50% to v2, "Class Sunglasses" appeared in 20 out of 50 requests (40%).
3. Finally, with 100% traffic to v2, all requests displayed "Class Sunglasses".

I suspect the issue is related to the frontend, but I was unable to identify the specific cause.

V1



V2