This is the scenario for testing the retry injection on the istio virtual service:
The below Tables show the configuration of the cluster that is used in this practice:

Our OSL in this project:
95% of all responses in 0~100 ms
5% of all responses in more than 90ms

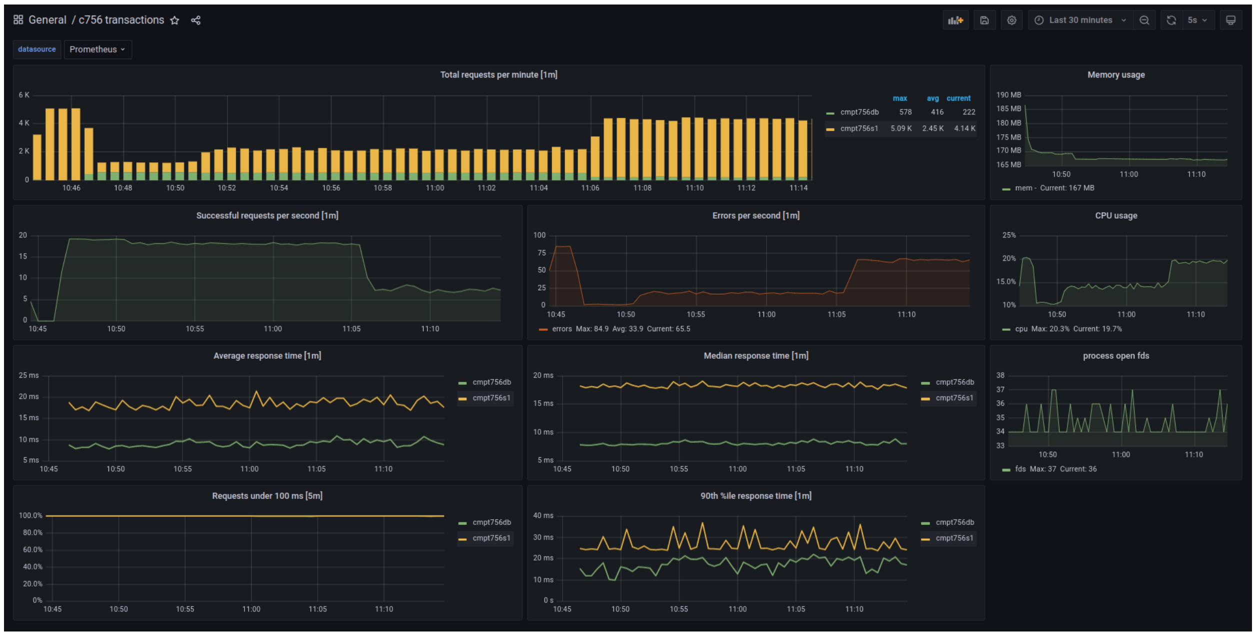
The number of users			Esk AWS	Number of replicas			
Playlist service	User service	Music service	Number of nodes Aws t3.medium	S1 service	S2 service	db service	Playlist service
0	10	0	2	1	1	1	1

TableName	ReadCapacityUnits	WriteCapacityUnits
Music-bagherireza	5	5
User-bagherireza	5	5
Playlist-bagherireza	5	5

The below table shows the retry configuration on s1 and fault injection on the db service:

Phase No.	Time	Retry no., perTryTimeout	Fault percentage
1	10:38	5, 2S	0%
2	10:44	5, 2S	100%
3	10:46	5, 2S	10%
4	10:50	5, 2S	50%
5	11:05	5, 2S	90%

The below diagram shows the overall response of clutter over whole scenario:


General / c756 transactions
Last 30 minutes
5s
datasource
Prometheus
Total requests per minute [1m]
Memory usage
Successful requests per second [1m]
Errors per second [1m]
CPU usage
Average response time [1m]
Median response time [1m]
process open fds
Requests under 100 ms [5m]
90th %ile response time [1m]
cmpt756db
cmpt756s1

	max	avg	current
cmpt756db	578	416	222
cmpt756s1	5.09 K	2.45 K	4.14 K

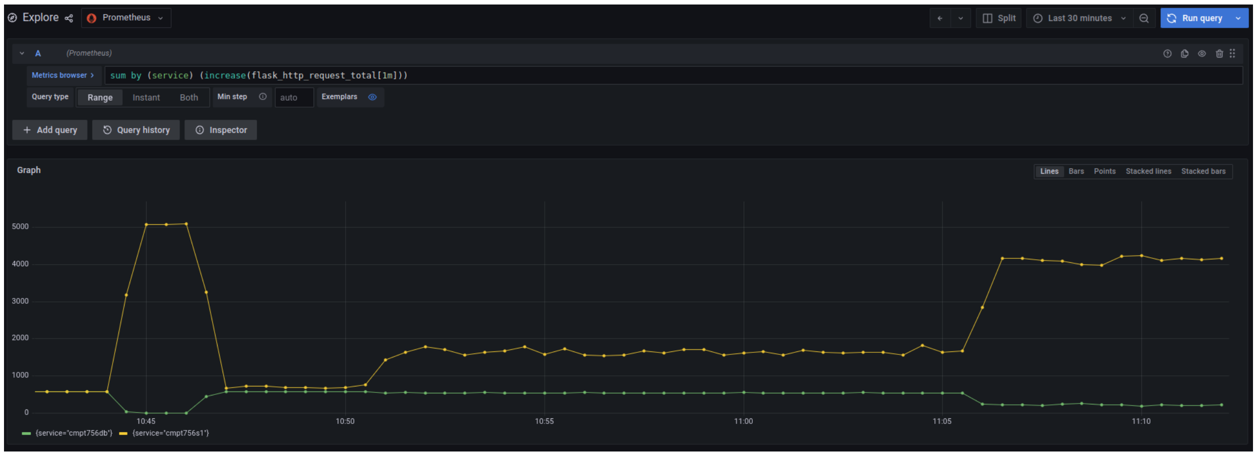
mem - Current: 167 MB

errors Max: 84.9 Avg: 33.9 Current: 65.5

cpu Max: 20.3% Current: 19.7%

fds Max: 37 Current: 36

Let's review the total request No.:



```
sum by (service) (increase(flask_http_request_total[1m]))
```

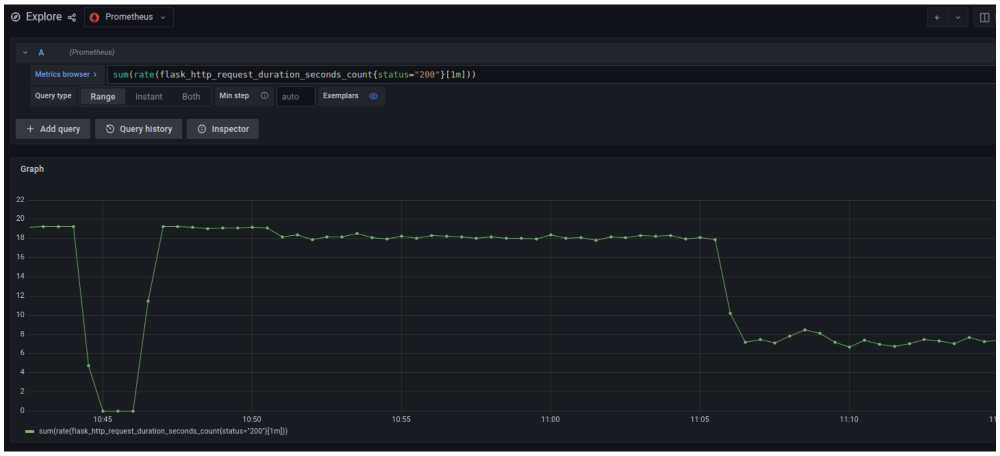
Query type: Range — Min step: auto — Exemplars

1. Phase1:
    a. It is shown that In the beginning the number of requests in the s1 and the db are equal as we expected and it is around 580.
2. Phase2:
    a. When the fault injection percentage is 100, as we can see the no. of requests of db reach zero because it is not receiving and request. then the number of requests reaches more than 5000 which shows that the retry function worked.
3. Phase3:
    a. Then when the percentage is decreased to 10%, as we can see the number of request from s1 increases while db just receive the same amount of request and generates the same amount of request to DynamoDB.
4. Phase 4:
    a. Even after increasing the percentage of failure by 50%, we can see that the number of requests is increasing while the number of db requests is the same as

normal. it means that with help of the retry function we are able to recover the services.

5. Phase 5:
    a. When the percentage of failure increased to 90%, we can observe that the retry function can't recover because the number of db requests is decreased.

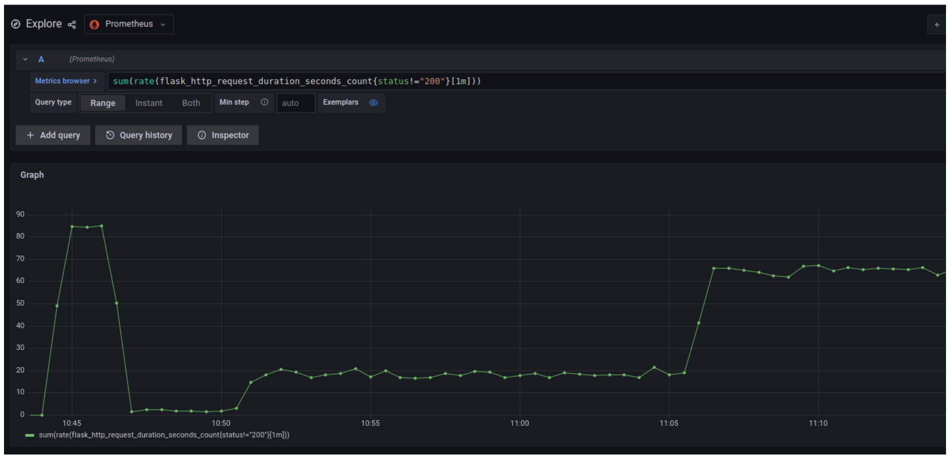
Successful rate:



```
sum(rate(flask_http_request_duration_seconds_count{status="200"}[1m]))
```

6. Phase1:
    a. It is shown that there are around 19 successful requests each minute.
7. Phase2:
    a. When the fault injection percentage is 100, as we can see the no. of successful requests is decreased to zero.
8. Phase3:
    a. Then when the percentage is decreased to 10%, as we can see the number of successful requests back to 19.
9. Phase 4:
    a. Even after increasing the percentage of failure by 50%, we can see the number of successful requests back to 19.
10. Phase 5:
    a. When the percentage of failure increased to 90%, we can observe the number of successful requests is dropped to less than 8.

Failure rate:



```
sum(rate(flask_http_request_duration_seconds_count{status!="200"}[1m]))
```

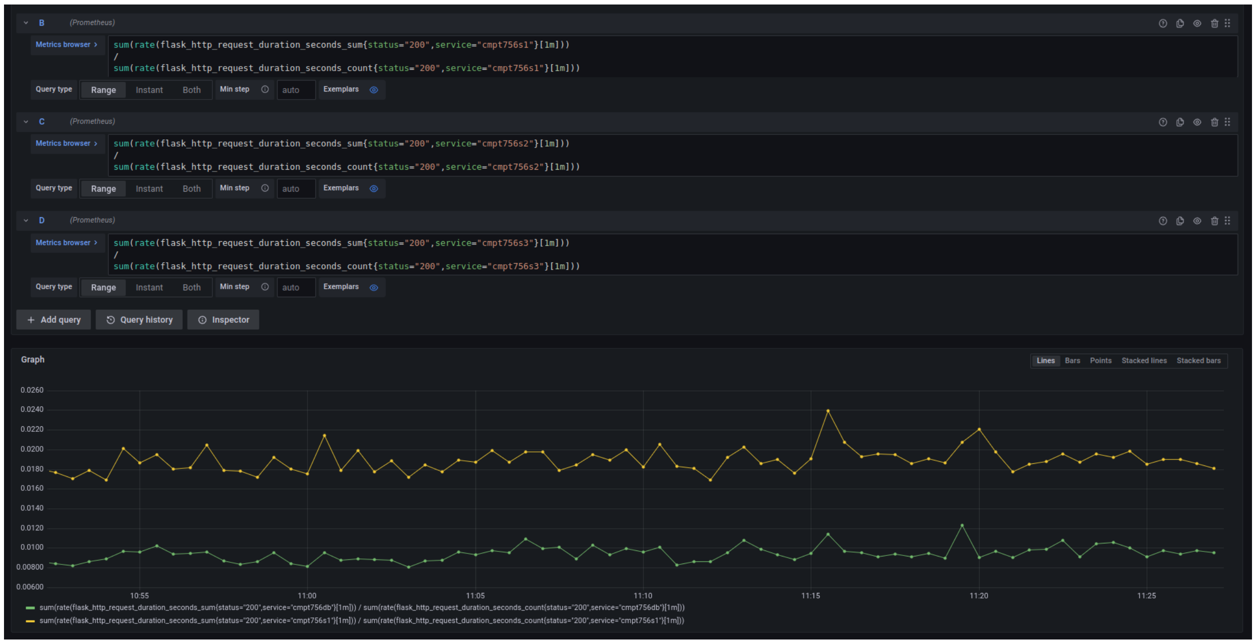
Query type: Range; Min step: auto

sum(rate(flask_http_request_duration_seconds_count{status!="200"}[1m]))

11. Phase1:
    a. It is shown that there is no failure.
12. Phase2:
    a. When the fault injection percentage is 100, as we can see there are around 85 failures.
13. Phase3:
    a. Then when the percentage is decreased to 10%, as we can see the number of failures 2.
14. Phase 4:
    a. After increasing the percentage of failure by 50%, we can see the number of failures is increasing to 20.
15. Phase 5:
    a. When the percentage of failure increased to 90%, we can observe the number of failures is dropped to less than 70.

In the end, there is a report regarding the number of failures and requests and their relation to the number of retries.

Average response:



B (Prometheus)

```
sum(rate(flask_http_request_duration_seconds_sum{status="200",service="cmpt756s1"}[1m]))
/
sum(rate(flask_http_request_duration_seconds_count{status="200",service="cmpt756s1"}[1m]))
```

Query type: Range | Min step: auto | Exemplars

C (Prometheus)

```
sum(rate(flask_http_request_duration_seconds_sum{status="200",service="cmpt756s2"}[1m]))
/
sum(rate(flask_http_request_duration_seconds_count{status="200",service="cmpt756s2"}[1m]))
```

Query type: Range | Min step: auto | Exemplars

D (Prometheus)

```
sum(rate(flask_http_request_duration_seconds_sum{status="200",service="cmpt756s3"}[1m]))
/
sum(rate(flask_http_request_duration_seconds_count{status="200",service="cmpt756s3"}[1m]))
```

Query type: Range | Min step: auto | Exemplars

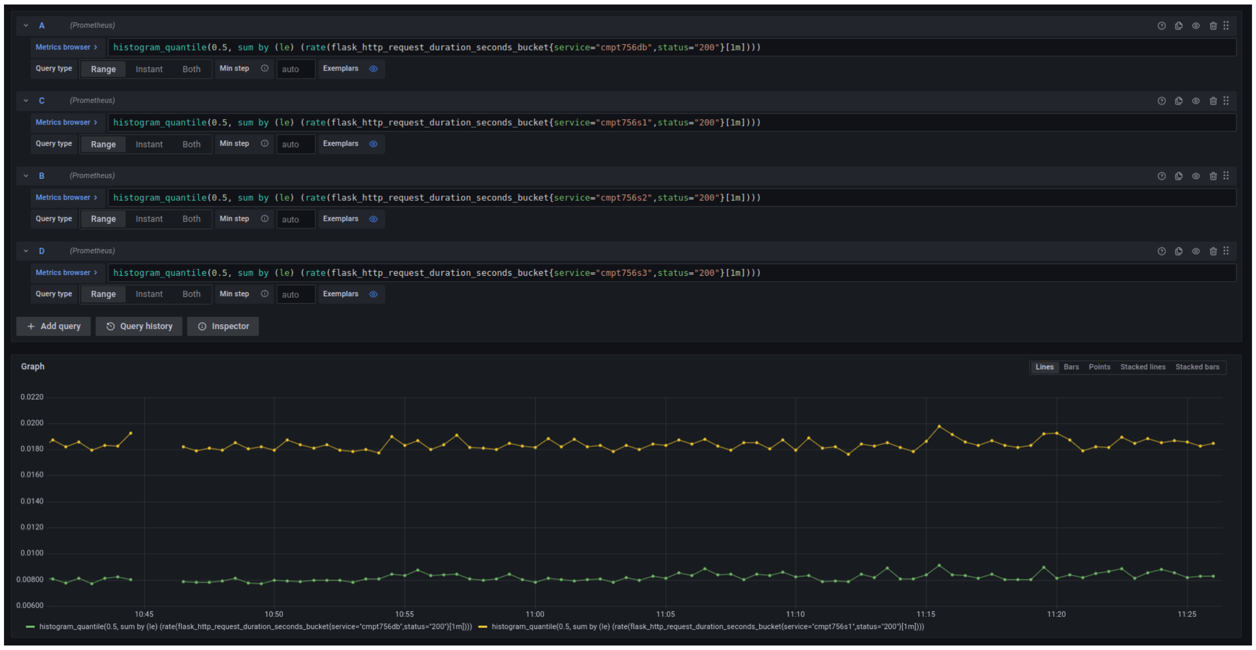
Graph

🟩 sum(rate(flask_http_request_duration_seconds_sum{status="200",service="cmpt756db"}[1m])) / sum(rate(flask_http_request_duration_seconds_count{status="200",service="cmpt756db"}[1m]))

🟨 sum(rate(flask_http_request_duration_seconds_sum{status="200",service="cmpt756s1"}[1m])) / sum(rate(flask_http_request_duration_seconds_count{status="200",service="cmpt756s1"}[1m]))

Because the fault is added by istio, for the packets that are replied with status 200, the average response time is the same as normal.

Median response:



A (Prometheus)

```
histogram_quantile(0.5, sum by (le) (rate(flask_http_request_duration_seconds_bucket{service="cmpt756db",status="200"}[1m])))
```

Query type: Range | Min step: auto | Exemplars

C (Prometheus)

```
histogram_quantile(0.5, sum by (le) (rate(flask_http_request_duration_seconds_bucket{service="cmpt756s1",status="200"}[1m])))
```

Query type: Range | Min step: auto | Exemplars

B (Prometheus)

```
histogram_quantile(0.5, sum by (le) (rate(flask_http_request_duration_seconds_bucket{service="cmpt756s2",status="200"}[1m])))
```

Query type: Range | Min step: auto | Exemplars

D (Prometheus)

```
histogram_quantile(0.5, sum by (le) (rate(flask_http_request_duration_seconds_bucket{service="cmpt756s3",status="200"}[1m])))
```

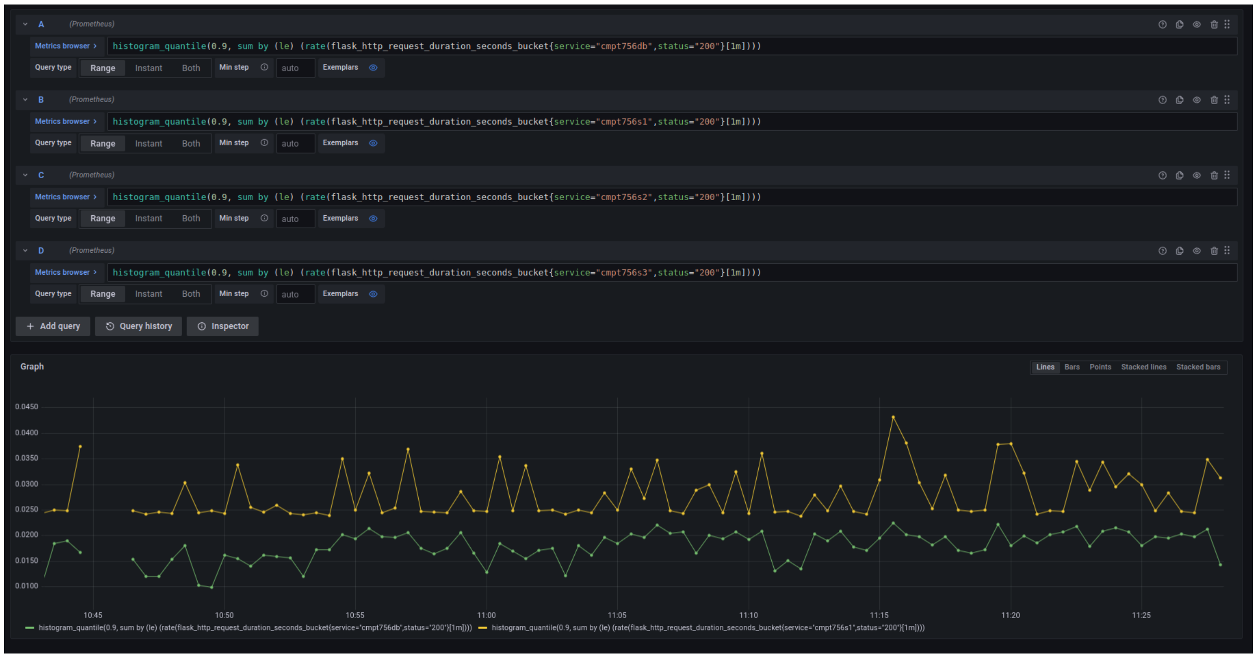
Query type: Range | Min step: auto | Exemplars

Graph

🟩 histogram_quantile(0.5, sum by (le) (rate(flask_http_request_duration_seconds_bucket{service="cmpt756db",status="200"}[1m]))) 🟨 histogram_quantile(0.5, sum by (le) (rate(flask_http_request_duration_seconds_bucket{service="cmpt756s1",status="200"}[1m])))

Because the fault is added by istio, for the packets that are replied with status 200, the median response time is the same as normal.

90% ile response time:



```
A (Prometheus)
histogram_quantile(0.9, sum by (le) (rate(flask_http_request_duration_seconds_bucket{service="cmpt756db",status="200"}[1m])))
B (Prometheus)
histogram_quantile(0.9, sum by (le) (rate(flask_http_request_duration_seconds_bucket{service="cmpt756s1",status="200"}[1m])))
C (Prometheus)
histogram_quantile(0.9, sum by (le) (rate(flask_http_request_duration_seconds_bucket{service="cmpt756s2",status="200"}[1m])))
D (Prometheus)
histogram_quantile(0.9, sum by (le) (rate(flask_http_request_duration_seconds_bucket{service="cmpt756s3",status="200"}[1m])))
```

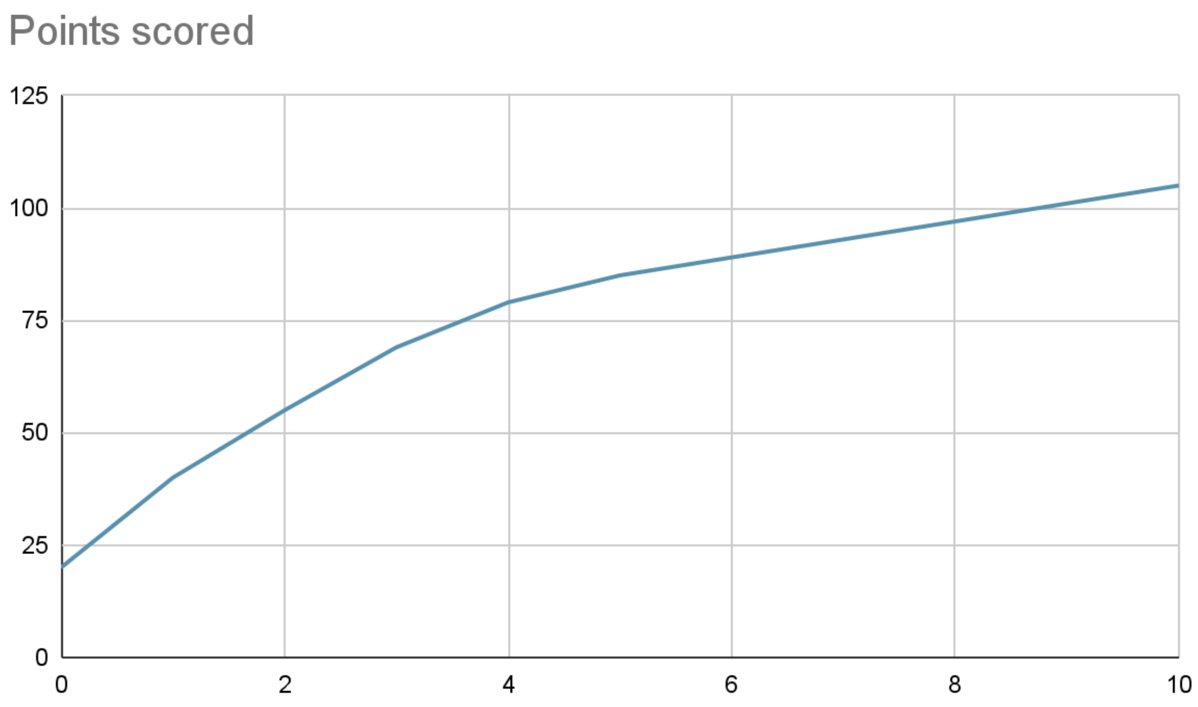
Query type: Range; Min step: auto

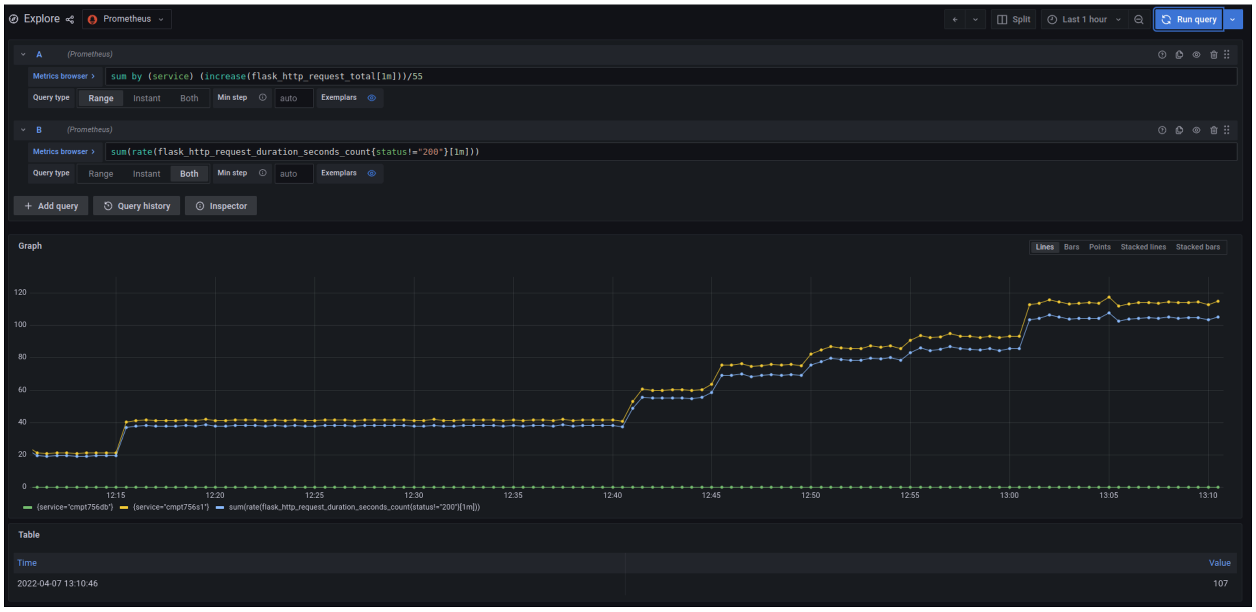
🟩 histogram_quantile(0.9, sum by (le) (rate(flask_http_request_duration_seconds_bucket{service="cmpt756db",status="200"}[1m]))) 🟨 histogram_quantile(0.9, sum by (le) (rate(flask_http_request_duration_seconds_bucket{service="cmpt756s1",status="200"}[1m])))

Because the fault is added by istio, we observe normal behavior here too.

Summary:
Based on my observation, when a system faces some percentages of error, with help of the retry function the cluster can recover. And from the client's point of view, it works normally.
From Istio Retries documentation: Default retry is hardcoded and its value is equal to 2. It means any circumstance, it will try two times at least.
From this experiment, We simulate the situation where the db microservice faces some problem and rejects some percentage of requests by error code 500. The fault injection has been done with help of istio. On the other hand, we tried to use the retry function to recover the system.
It is clear that the retry function can help to recover the system, but the efficiency of the retry function depends on the percentage of failure and traffic of the network. The details exploration has been done and reported in this report (https://github.com/scp756-221/term-project-cloudriven/issues/29).
but these retries can make problems of congestion inside the cluster which can be controlled by the circuit breaker. (https://github.com/scp756-221/term-project-cloudriven/issues/33).
Interesting facts, regarding the number of retries:
I have injected 100% fault in db so I expected to see that s1 retry to send a request based on the number of retries. But my observation was different, In the beginning, there is no retry configuration, but istio as a default configuration retry to send the failed request again, because of that failed requests are 20, two times of origin request which was 10. As you can see in the diagram the number of requests is not following the number of retries. It seems, that there are mechanisms that prevent the retry function to go linear when the number of retries increases.

No.	The number of retries	Failed requests	The total number of requests
1	0	20	1.16K
2	1	40	2.26K
3	2	55	3.3K
4	3	69	4.6K
5	4	79	4.8K
6	5	85	5.12K
7	10	105	6.36K





A (Prometheus)

```
sum by (service) (increase(flask_http_request_total[1m]))/55
```

Query type: [✓ Range] Instant Both — Min step: auto — Exemplars

B (Prometheus)

```
sum(rate(flask_http_request_duration_seconds_count{status!="200"}[1m]))
```

Query type: Range Instant [✓ Both] — Min step: auto — Exemplars

🟩 {service="cmpt756db"} 🟨 {service="cmpt756s1"} 🟦 sum(rate(flask_http_request_duration_seconds_count{status!="200"}[1m]))

Table

Time	Value
2022-04-07 13:10:46	107