PROJECT 2: SCALABILITY REPORT

Contents

❖ Abstract

This report is prepared by Team Scapsulators on WeatherApp developed for the NexRad dataset. This is a detailed report of performance of the existing system for different load sizes. Then we have suggested some architecture changes and implementations for a scalable system and reported the results.

❖ Load Test Plan

- **Hardware Setup :**

The total RAM requirement for our application (With one replica per microservice) is 6GB. We had tested the microservices on our which is a 4 core intel i5-9300H @2.4 Ghz, and 16Gb RAM. Now assuming there are other processes running too during our testing we can assume that we have 70% of the CPU and 60% of the RAM available to us for testing. We have used one master node and 3 worker node, 16 GB memory each.

- **System Requirements :**

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
98839dec7c0a	audit-micro	0.33%	224.1MiB / 12.36GiB	1.77%	1.16kB / 0B	0B / 0B	30
360f9fe84593	react-frontend	0.00%	6.867MiB / 12.36GiB	0.05%	978B / 0B	0B / 0B	9
093d246854be	mongodb	1.07%	245MiB / 12.36GiB	1.94%	11.9kB / 30.5kB	0B / 0B	35
2fc9c634fa62	weather-reporter	0.02%	263.2MiB / 4GiB	6.43%	978B / 0B	0B / 0B	3
78638a411cad	database-connect	1.20%	209.9MiB / 12.36GiB	1.66%	31.7kB / 11kB	0B / 0B	33
1af528c7cd17	authentication-micro	0.30%	191.4MiB / 12.36GiB	1.51%	1.07kB / 0B	0B / 0B	30
75f471d8361d	gateway_node	0.00%	14.07MiB / 12.36GiB	0.11%	978B / 0B	0B / 0B	7

This is strictly the docker container requirements/usage for hosting the containers is local. Now, in case of hosting it on K8s will require some additional networking and orchestration overhead that also needs to be accounted for.

- **Estimated Number of Users :**

Our User base is very niche and selective. Our application appeals to the scientific community that research on weather and atmospheric phenomena. We expect around 1000 users, and 200 concurrent users for our application.

Java Microservices - We expect JAVA microservice to be able to handle a lot of requests, it has 2 write heavy requests, which is used to register the user and update the password. We expect it to handle up to 2000 concurrent requests based on our existing setup.

Python Microservice - This is the bottleneck of our system, because of the complex and heavy tasks that it runs in fetching the nexrad data and making plots using heavy scientific libraries. Typically one request takes upto 20 seconds to execute, and using multithreading, we are estimating that the python container can handle upto 200 requests at a time without failing.

Gateway Overall System - Due to a single threaded runtime nodejs environment that uses synchronous api calls for all the microservices, we are estimating that our gateway can also handle 200 concurrent requests currently.

- **Expected Throughput of each microservice :**

Java Microservice (Audit/Authenticate) - 50 requests per second
Python Microservice - 200 requests per minute

- **Estimated Usage of System :**

As mentioned above, based on the number of requests, the usage of our system will vary. Therefore if we calculate on the basis of our expected number of users which is 150. We will have the following usage of our system.

1) JAVA microservice - 10% usage
2) Python microservice - 6,666% usage

As we can see based on our initial assessment we can see for sure that our python microservice will not be able to handle a high workload.

- **Tool to be used for Load Testing** : Jmeter

- **Container Orchestration System to be used:** Kubernates

❖ Load Testing Results

- **Capacity of our system :**

WEATHERPEDIA APPLICATION				
SR No	**Title**	**Results**	**Reason**	**Improvements**
1	Determine the user limit for the Web application.			
a	The user limit is the maximum number of concurrent users that the system can support while remaining stable and providing reasonable response time to users as they perform a variety of typical business transactions.	Replica = 1 : Endurance Testing - 1200 requests/minute Spike Testing - 1900 requests in a span of 60 secs with an acceptable error rate of 0.04%. Replica = 3 : Endurance Testing - 3000 requests/minute Spike Testing - 4300 requests in a span of 60 secs with an acceptable error rate of 0.01% Replica = 5 : Endurance Testing - 6000 requests/minute with an acceptable error rate of 0.01% Spike Testing - 7600 requests in a span of 60 secs with an acceptable error rate of 0.01% AutoScaling : Endurance Testing - 7300 requests/minute Spike Testing - 8600 requests in a span of 60	1) We have implemented synchronous API which is affecting the capacity. 2) The microservice which fetches weather data is taking a significant amount of time to execute a single request.	1) We are planning to implement asynchronous API's in our future sprints. 2) We are assigning more RAM and CPU to our weather data fetch microservice which is enhancing our performance.
b	The user limit should be higher than the required number of concurrent users that the application must support when it is deployed.	As our application appeals to the scientific community that research on weather and atmospheric phenomena, We expect around 1000 users a day, and 200 concurrent users for our application.		

- **How does the system degrade beyond maximum capacity :**

2	Determine client-side degradation and end user experience under load.			
a	Can users get to the Web appli	Yes.	The Web application takes < 0.3s to load.	
b	Are users able to conduct business or perform a transaction within an acceptable time?	Yes.	Our slowest microservice is taking <20s to execute and other microservices are taking <10 secs to execute	
c	How does the time of day, number of concurrent users, transactions and usage affect the performance of the Web application?	Time of day -> As our application does not have real users curently, we were not able to observe this. Number of Concurrent Users -> We have observed that as the number of concurrent users increase, the error rate for our application also increases. The maximun load of our system is specified above where the error rate is acceptable. For Autoscaling, More the number of concurrent users, more the number of replicas of pods are created through autoscaling. Lesser number of concurrent users, less number of replicas of pods are created through autoscaling		
d	Is the degradation "graceful?" Under heavy loading conditions, does the application behave correctly in "slow motion," or do components crash or send erroneous/incomplete pages to the client?	Under heavy load conditions (> than the capacity of the system mentioned above), the clients receive an error message as the requests time out.	Under extreme high loads the application containers are up but the requests later in the queue eventually time out and those requests are not served.	We are planning to implement asynchronous API's in our future sprints which will help alleviate this issue.
e	What is the failure rate that users observe? Is it within acceptable limits? Under heavy loading conditions do most users continue to complete their business transactions or do a large number of users receive error messages?	Under extremely high loads (more than the system capacity mentioned above), the error rates are significantly higher and beyond acceptable rates.The requests time out and the users receive an error message.	Under extreme high loads the application containers are up but the requests later in the queue eventually time out and those requests are not served.	We are planning to implement asynchronous API's in our future sprints which will help alleviate this issue.

- **How do the servers behave beyond the maximum capacity of the application :**

3	Determine server-side robustness and degradation.			
a	Does my Web server crash under heavy load?	No		
b	Does my application server crash under heavy load?	No		
c	Do other middle-tier servers crash or slow down under heavy load?	No		
d	Does my database server crash under heavy load?	No		
e	Does my system load require balancing, or if a load balancing system is in place, is it functioning correctly?	We have implemented kubernates that handles load balancing for multiple replicas.		
f	Can my current architecture be	Yes		
g	Should hardware changes be made for improved performance?			
h	Are there any resource deadlocks in my system?	Yes. The database server.	As we have only one database server that multiple instances of one microservice will be connecting to to maintain strict consistency, the database server can potentially cause deadlocks.	Spawn multiple instances of the database server and implement eventual consistency instead of strict consistency.

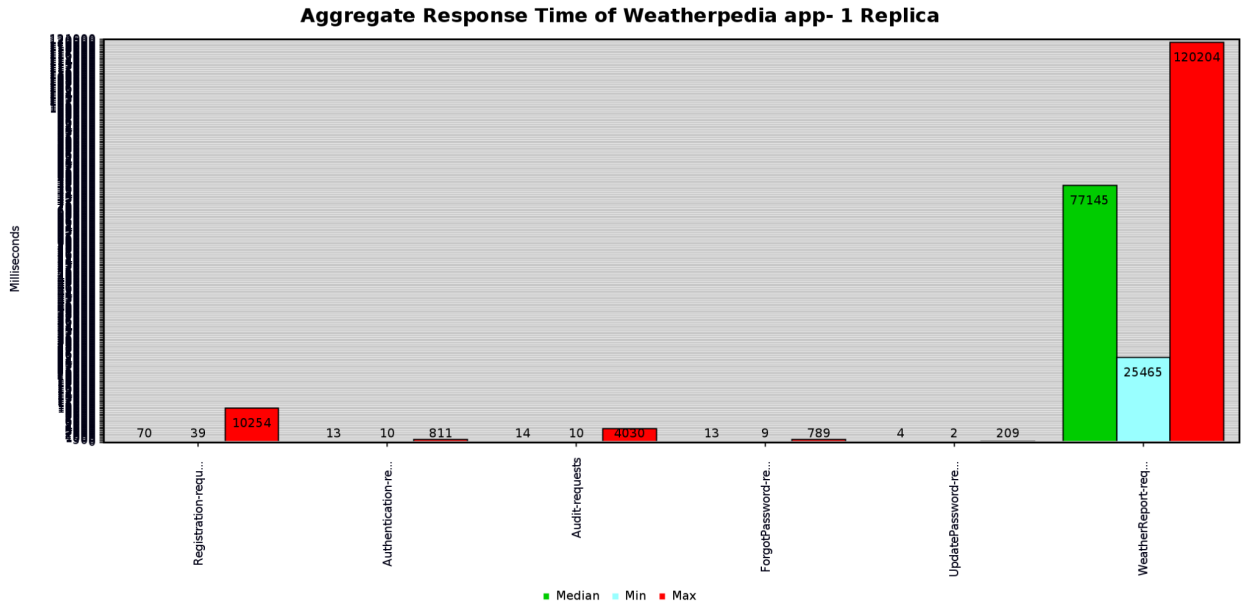
- **Plots and Analysis For Load Testing - Replica = 1 :**

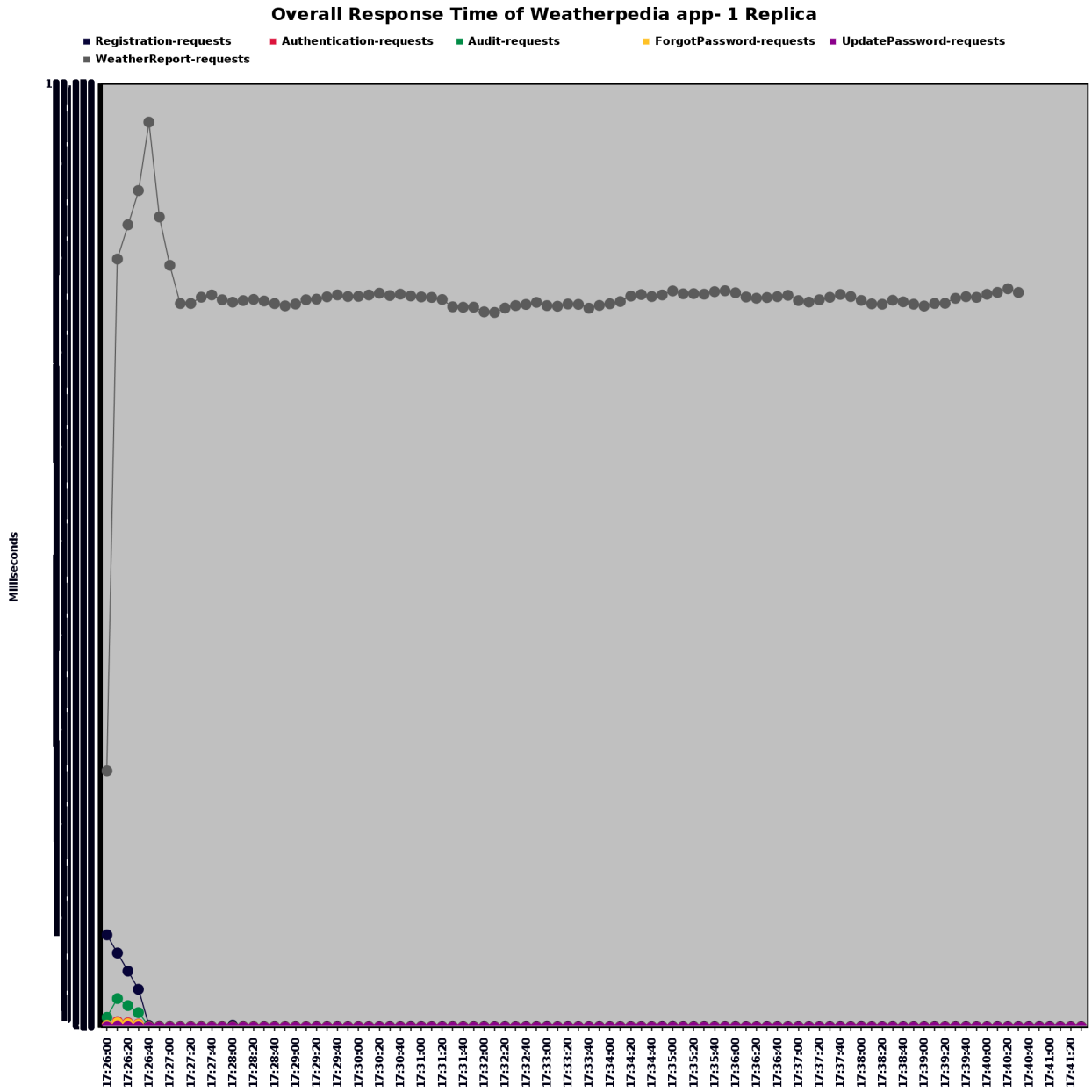
  ➔ **Endurance Testing :**

  Results : Our system can handle a maximum consistent load of 1200 requests/minute for 30 minutes with 0% error rate and a throughput of 14.9 requests/sec.

  Analysis : We observed that our system was able to handle a load of 1200 requests/minute for 30 minutes with 0% error rate. When we increased the load, we were getting a significant error rate. From the graphs, we can infer that our python microservice is taking maximum time to execute requests because we have used in-built libraries to implement our logic that is causing significant delays in processing requests. Also, the synchronous behavior of REST API's implemented for inter microservice communication is causing delays in our system and limiting the capacity of our system.

  Plots :


Aggregate Response Time of Weatherpedia app- 1 Replica
Milliseconds
70
39
10254
13
10
811
14
10
4030
13
9
789
4
2
209
77145
25465
120204
Registration-requ...
Authentication-re...
Audit-requests
ForgotPassword-re...
UpdatePassword-re...
WeatherReport-req...
Median
Min
Max

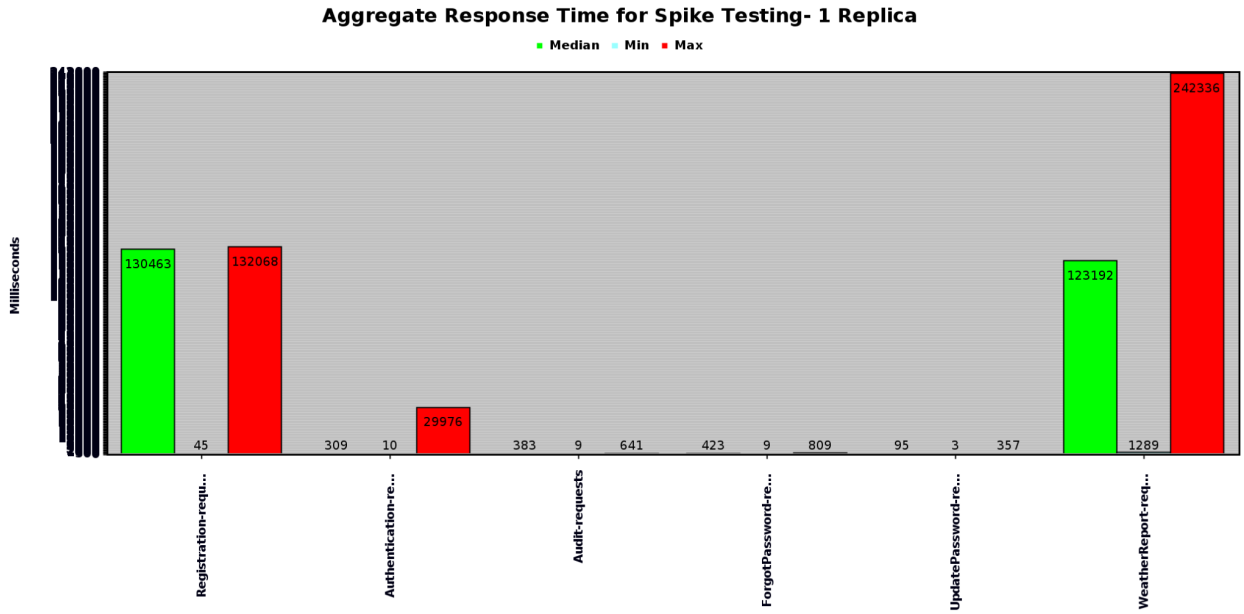

Overall Response Time of Weatherpedia app- 1 Replica
Registration-requests
Authentication-requests
Audit-requests
ForgotPassword-requests
UpdatePassword-requests
WeatherReport-requests
Milliseconds
17:26:00
17:26:20
17:26:40
17:27:00
17:27:20
17:27:40
17:28:00
17:28:20
17:28:40
17:29:00
17:29:20
17:29:40
17:30:00
17:30:20
17:30:40
17:31:00
17:31:20
17:31:40
17:32:00
17:32:20
17:32:40
17:33:00
17:33:20
17:33:40
17:34:00
17:34:20
17:34:40
17:35:00
17:35:20
17:35:40
17:36:00
17:36:20
17:36:40
17:37:00
17:37:20
17:37:40
17:38:00
17:38:20
17:38:40
17:39:00
17:39:20
17:39:40
17:40:00
17:40:20
17:40:40
17:41:00
17:41:20

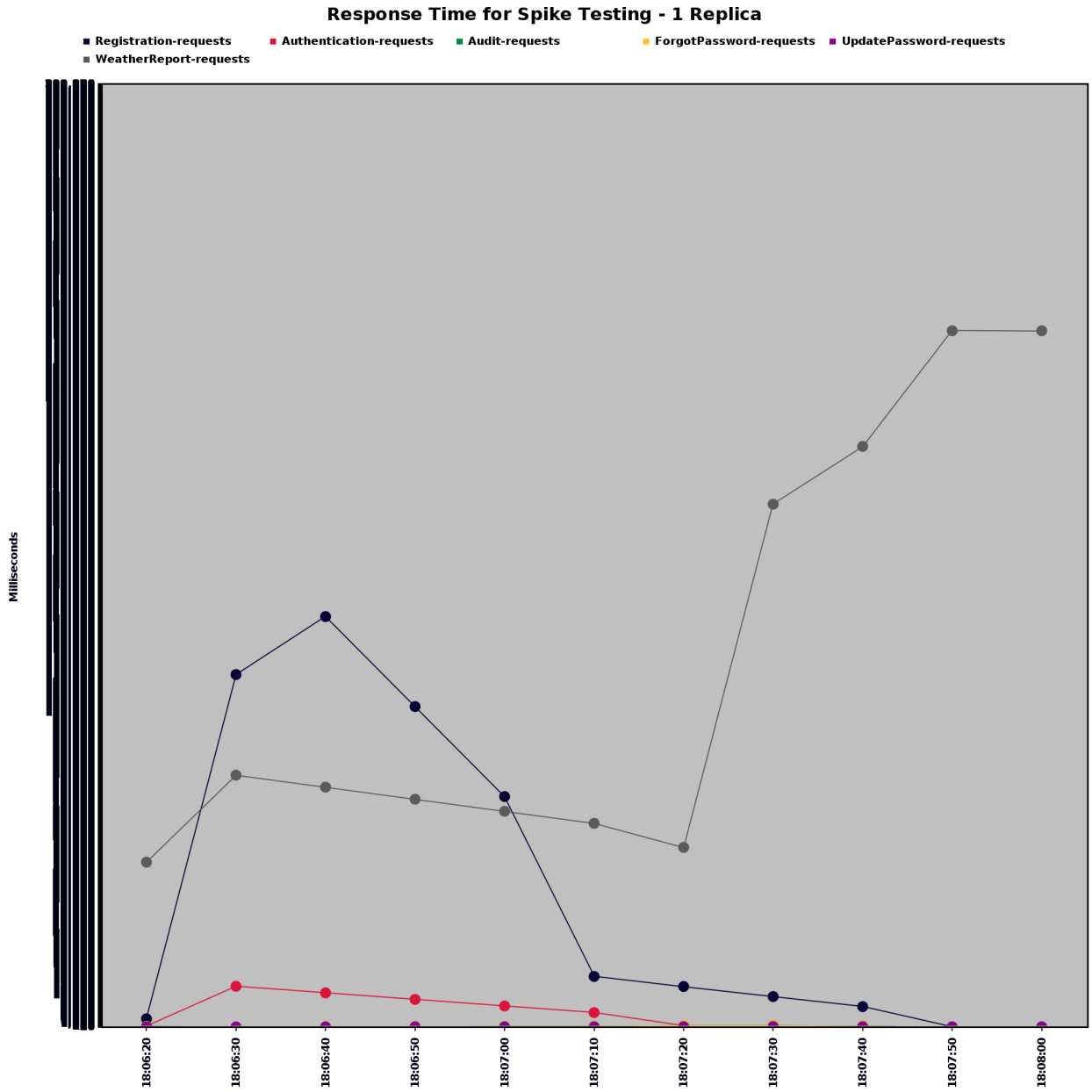
➔ **Spike Testing :**

Results : Our system can handle a maximum of 1900 requests in a span of 60 secs with an acceptable error rate of 0.04% and a throughput of 15.87 requests/sec.

Analysis : We observed that our system was able to handle a load of 1900 requests/minute for a span 60 secs with 0.04% error rate. When we increased the load, we were getting a significant error rate. From the graphs, we can infer that our python API and registration API are taking maximum time to execute requests. For python API, we have used in-built libraries to implement our logic that is causing significant delays in processing requests. The registration API involved writing a chinook of user information to the database which is time consuming. Also, the synchronous behavior of REST API's implemented for inter microservice communication is causing delays in our system and limiting the capacity of our system.

Plots :



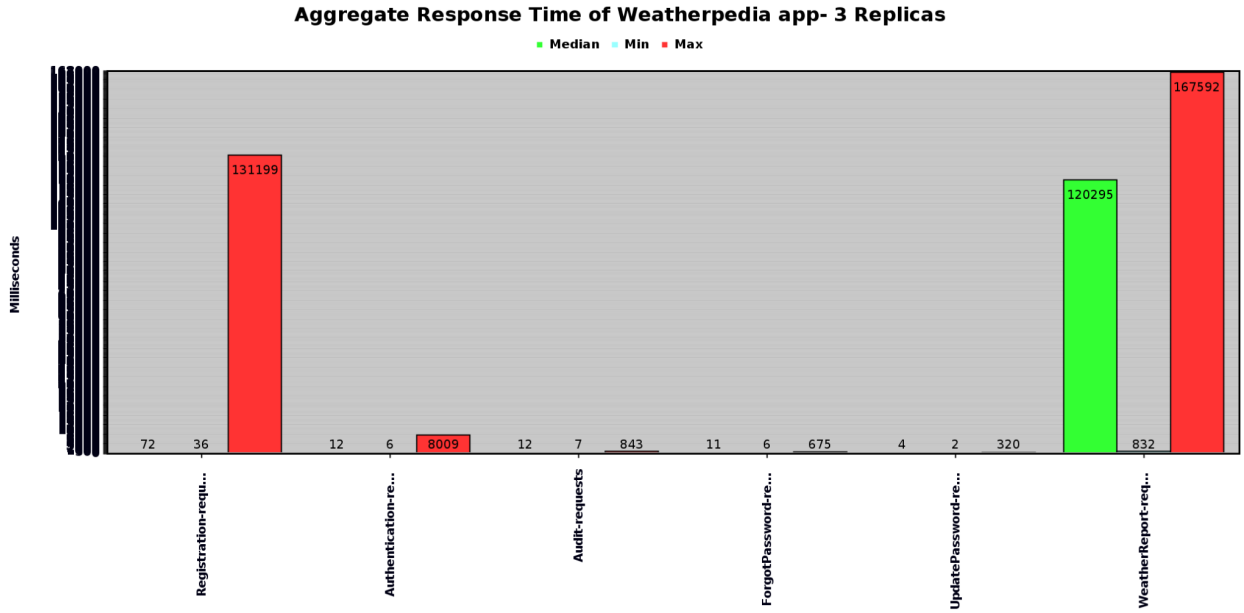

Response Time for Spike Testing - 1 Replica
Registration-requests
Authentication-requests
Audit-requests
ForgotPassword-requests
UpdatePassword-requests
WeatherReport-requests
Milliseconds
18:06:20
18:06:30
18:06:40
18:06:50
18:07:00
18:07:10
18:07:20
18:07:30
18:07:40
18:07:50
18:08:00

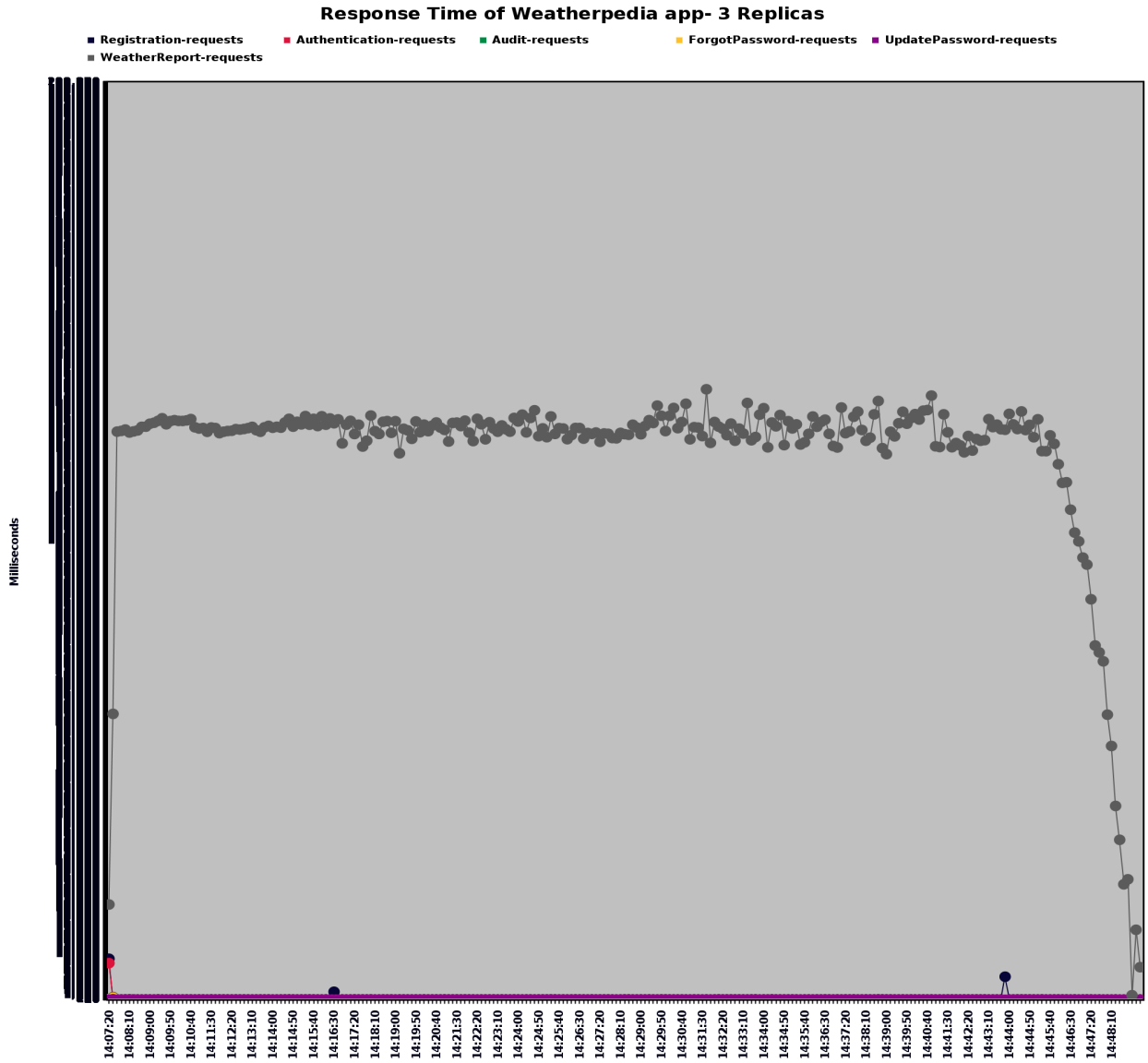
- **Plots and Analysis For Load Testing - Replica = 3 :**
  - ➔ **Endurance Testing :**

    Results : Our system can handle a maximum consistent load of 3000 requests/minute for 30 minutes with 0.02% error rate and a throughput of 50.3 requests/sec.

    Analysis : We observed that our system was able to handle a load of 3000 requests/minute for 30 minutes with 0.02% error rate. When we increased the load, we were getting a significant error rate. From the graphs, we can infer that our python microservice is taking maximum time to execute requests as compared to other microservices because we have used in-built libraries to implement our logic that is causing significant delays in processing requests. Also, the synchronous behavior of REST API's implemented for inter microservice communication is causing delays in our system and limiting the capacity of our system.

    Plots :





Load Balanced in all the replicas :

```
Every 2.0s: kubectl top pods                                                    js-170-205.jetstream-cloud.org: Sat Mar  5 13:29:58 2022

NAME                                          CPU(cores)   MEMORY(bytes)
audit-deployment-7888f6bddb-89kch             22m          308Mi
audit-deployment-7888f6bddb-qmgdx             18m          308Mi
audit-deployment-7888f6bddb-scrcb             20m          317Mi
authentication-deployment-5555885cb4-9wwts    26m          321Mi
authentication-deployment-5555885cb4-cgxcf    20m          326Mi
authentication-deployment-5555885cb4-p56vp    36m          345Mi
database-connect-deployment-5465944748-68s1v  13m          281Mi
database-connect-deployment-5465944748-6xh5n  39m          285Mi
database-connect-deployment-5465944748-x99w4  25m          290Mi
gateway-6b48d5bb7c-6bww6                      210m         38Mi
gateway-6b48d5bb7c-c2gbr                      150m         42Mi
gateway-6b48d5bb7c-mh7xf                      201m         43Mi
mongodb-deployment-789c9fd5b7-nhfwn           29m          190Mi
react-frontend-74787b9948-s6tq6               0m           5Mi
weather-reporter-deployment-7b45cbd79c-6z7qp  396m         960Mi
weather-reporter-deployment-7b45cbd79c-jxfmc  390m         967Mi
weather-reporter-deployment-7b45cbd79c-tvxk2  447m         1080Mi
```

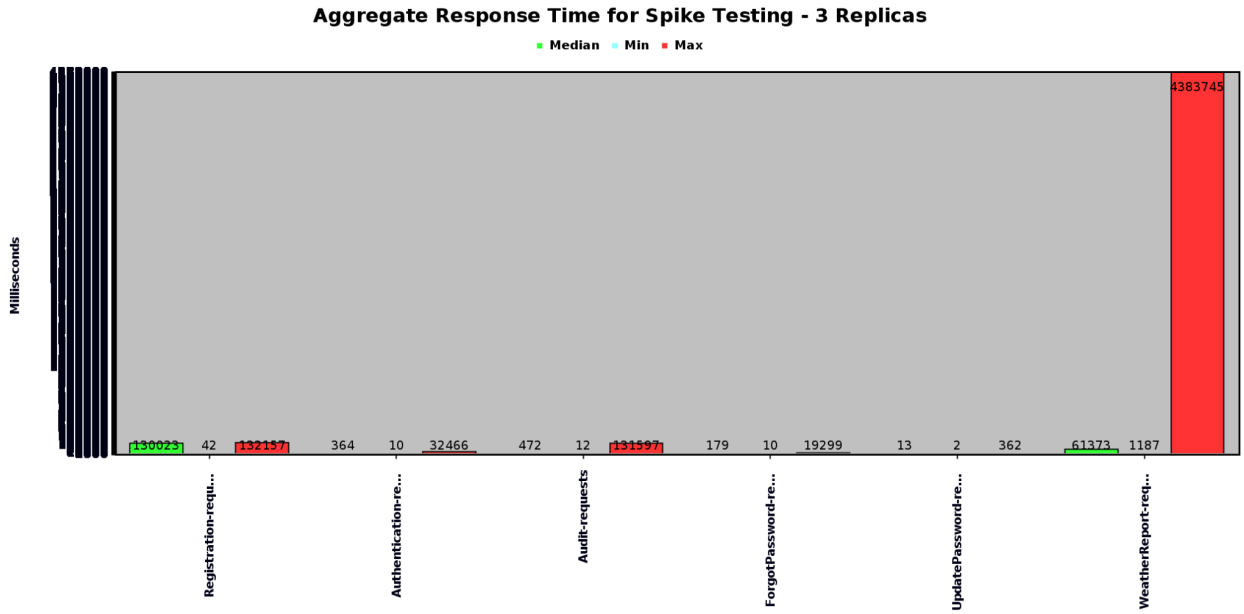
As we can see in the above image, the CPU (cores) and memory (bytes) are equally distributed among all the 3 replicas of each microservice which demonstrates that the load is balanced across all the replicas.

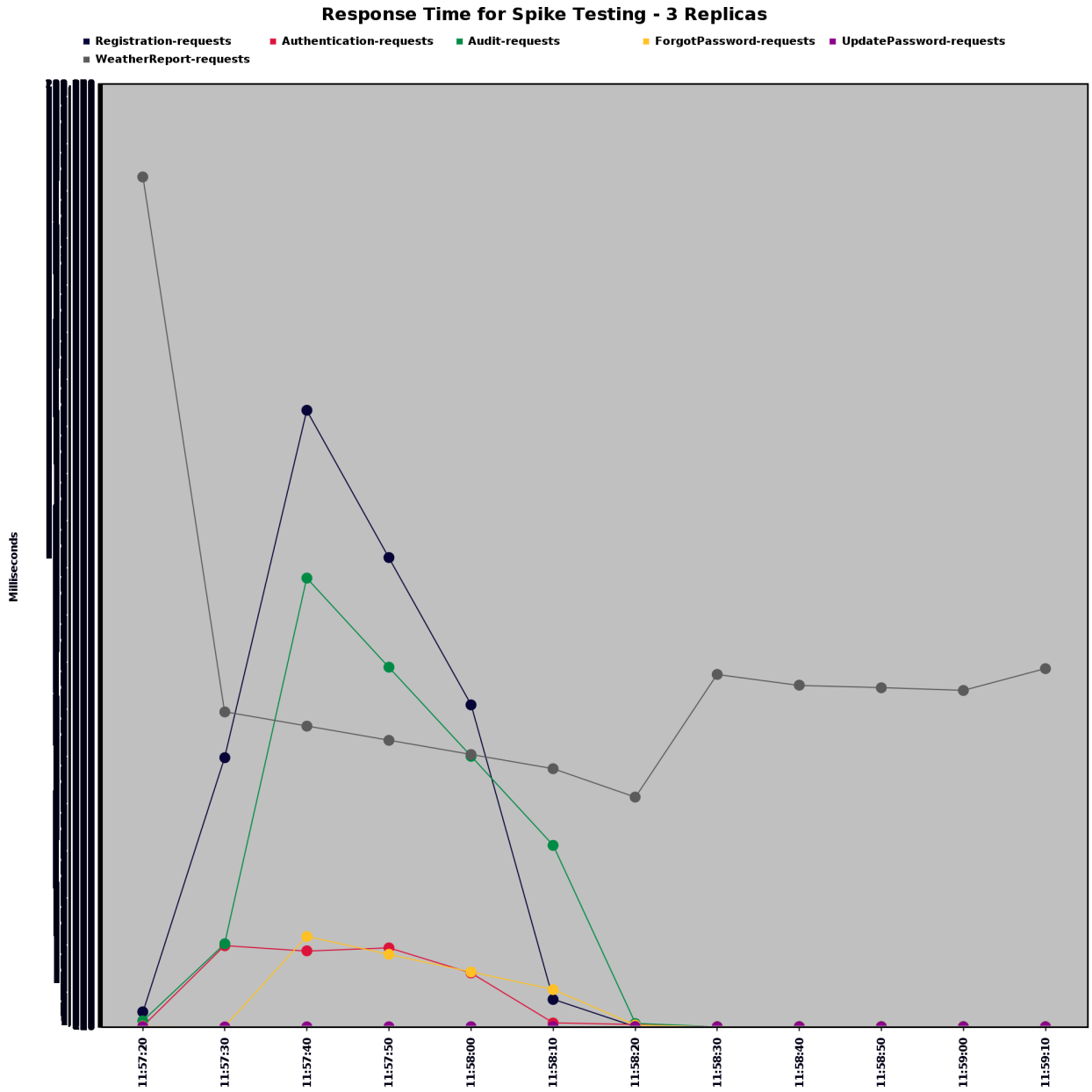
➔ **Spike Testing :**

Results : Our system can handle a maximum of 4300 requests in a span of 60 secs with an acceptable error rate of 0.01% and a throughput of 48.6 requests/sec.

Analysis : We observed that our system was able to handle a load of 4300 requests/minute for a span of 60 secs with 0.01% error rate. When we increased the load, we were getting a significant error rate. From the graphs, we can infer that our python API and registration API are taking maximum time to execute requests. For python API, we have used in-built libraries to implement our logic that is causing significant delays in processing requests. The registration API involved writing a chinook of user information to the database which is time consuming. Also, the synchronous behavior of REST API's implemented for inter microservice communication is causing delays in our system and limiting the capacity of our system.

Plots :



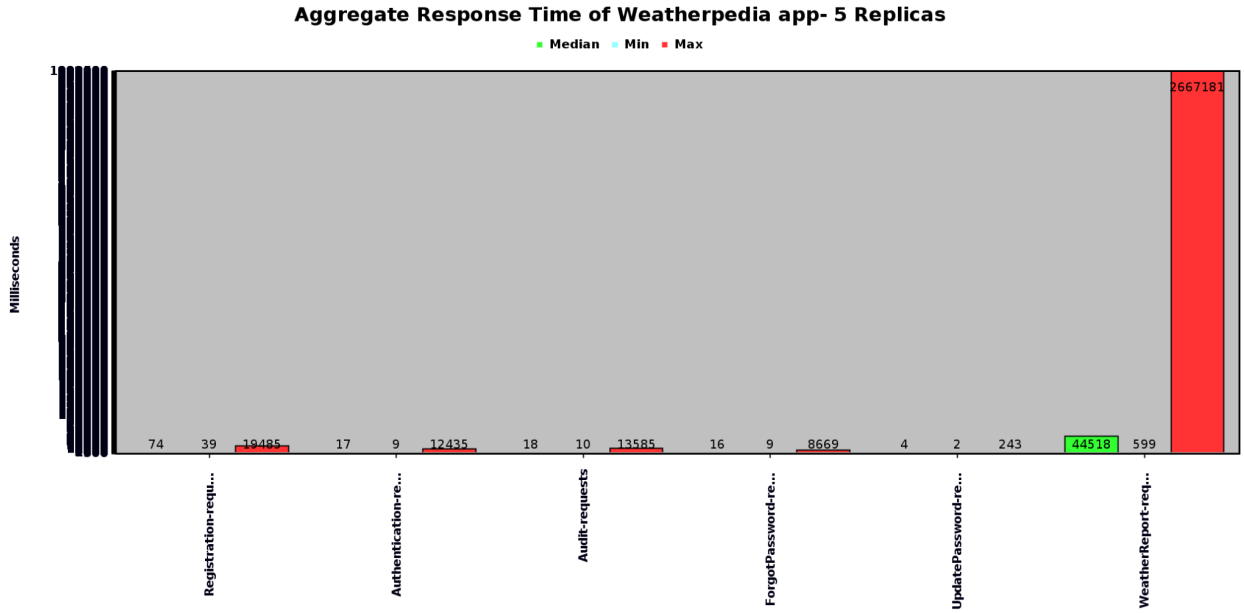

Response Time for Spike Testing - 3 Replicas
Registration-requests
Authentication-requests
Audit-requests
ForgotPassword-requests
UpdatePassword-requests
WeatherReport-requests
Milliseconds
11:57:20
11:57:30
11:57:40
11:57:50
11:58:00
11:58:10
11:58:20
11:58:30
11:58:40
11:58:50
11:59:00
11:59:10

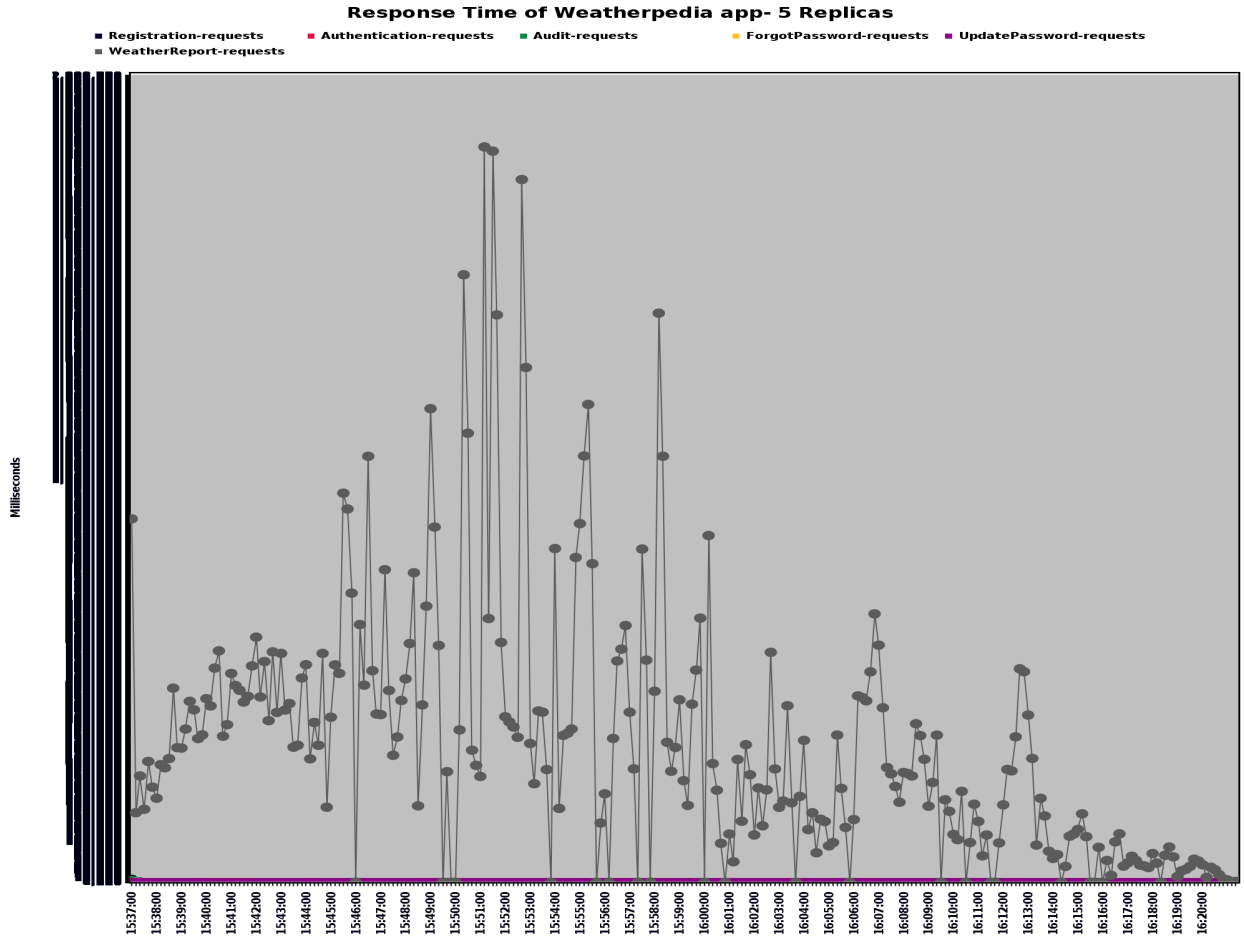
- **Plots and Analysis For Load Testing - Replica = 5 :**
  - ➔ **Endurance Testing :**

    Results : Our system can handle a maximum consistent load of 6000 requests/minute for 30 minutes with 0.01% error rate and a throughput of 17.9 requests/sec.

    Analysis : We observed that our system was able to handle a load of 6000 requests/minute for 30 minutes with 0.01% error rate. When we increased the load, we were getting a significant error rate. From the graphs, we can infer that our python microservice is taking maximum time to execute requests as compared to other microservices because we have used in-built libraries to implement our logic that is causing significant delays in processing requests. Also, the synchronous behavior of REST API's implemented for inter microservice communication is causing delays in our system and limiting the capacity of our system. We are observing a low throughput value for 5 replica set as opposed to a 3 replica set. After inspection, we found out that as we are using only 3 worker nodes to test, each of 16 GB, all our microservices can use a maximum of 48 GB to serve the requests. As we have allocated 4 GB for one python microservice to enhance the performance and decrease the response time of this python microservice, the other microservices are scrambling for more memory and hence we are getting a low throughput value.

    Plots :





Load Balanced in all Replicas :

```
Every 2.0s: kubectl top pods                                                    js-170-205.jetstream-cloud.org: Sat Mar  5 15:41:15 2022

NAME                                        CPU(cores)   MEMORY(bytes)
audit-deployment-7888f6bddb-67cbq           65m          329Mi
audit-deployment-7888f6bddb-767vf           31m          292Mi
audit-deployment-7888f6bddb-sfqb2           33m          301Mi
audit-deployment-7888f6bddb-vph4v           37m          316Mi
audit-deployment-7888f6bddb-wbpqx           38m          350Mi
authentication-deployment-5555885cb4-dgpkz  113m         316Mi
authentication-deployment-5555885cb4-156q5  77m          299Mi
authentication-deployment-5555885cb4-m5q5d  66m          312Mi
authentication-deployment-5555885cb4-wdjt8  81m          302Mi
authentication-deployment-5555885cb4-x7lvd  50m          310Mi
database-connect-deployment-5465944748-6g4g6  44m        358Mi
database-connect-deployment-5465944748-gc9jn  46m        335Mi
database-connect-deployment-5465944748-qm2cc  44m        314Mi
database-connect-deployment-5465944748-r9sjb  65m        355Mi
database-connect-deployment-5465944748-xjwww  58m        345Mi
gateway-6b48d5bb7c-9djmn                    152m         37Mi
gateway-6b48d5bb7c-cqpzd                    214m         32Mi
gateway-6b48d5bb7c-g724p                    157m         38Mi
gateway-6b48d5bb7c-kx7x7                    151m         33Mi
gateway-6b48d5bb7c-pcsk2                    146m         37Mi
mongodb-deployment-789c9fd5b7-8xs82         38m          202Mi
react-frontend-74787b9948-fnb19             0m           5Mi
weather-reporter-deployment-7b45cbd79c-85492  186m       938Mi
weather-reporter-deployment-7b45cbd79c-9vv54  425m       957Mi
weather-reporter-deployment-7b45cbd79c-jm9q2  395m       1676Mi
weather-reporter-deployment-7b45cbd79c-p9n9b  340m       950Mi
weather-reporter-deployment-7b45cbd79c-xzbdr  434m       275Mi
```

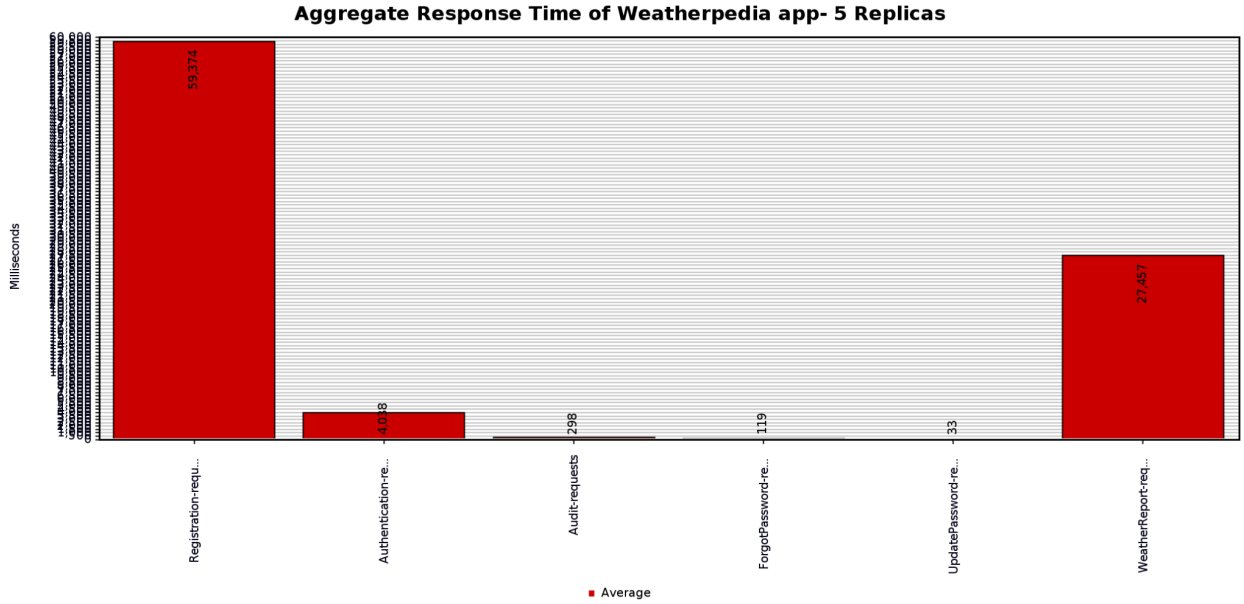
As we can see in the above image, the CPU (cores) and memory (bytes) are equally distributed among all the 5 replicas of each microservice which demonstrates that the load is balanced across all the replicas.

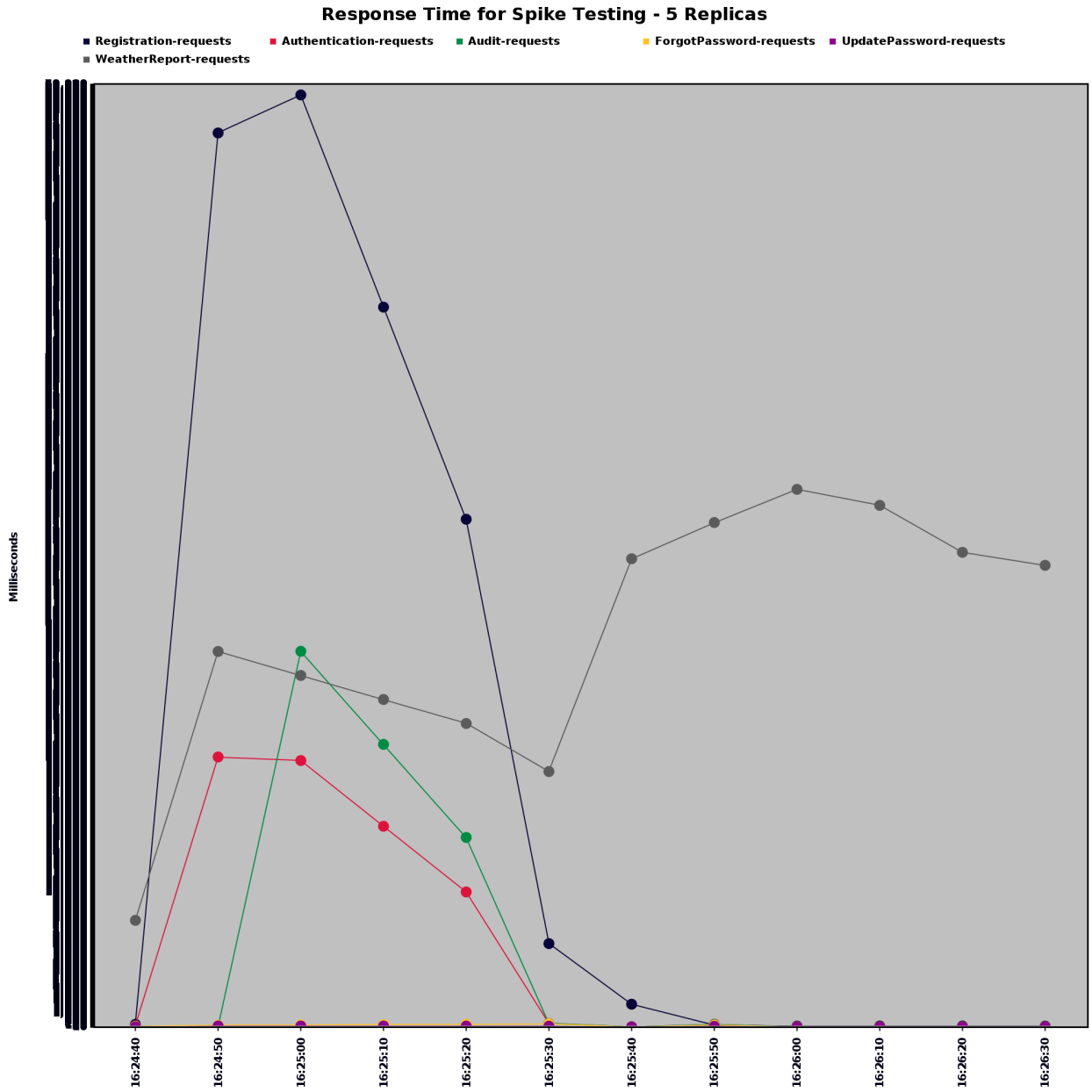
➔ **Spike Testing :**

Results : Our system can handle a maximum of 8600 requests in a span of 60 secs with an acceptable error rate of 0.12% and a throughput of 15.6 requests/sec.

Analysis : We observed that our system was able to handle a load of 8600 requests/minute for a span of 60 secs with 0.012% error rate. When we increased the load, we were getting a significant error rate. From the graphs, we can infer that our python API and registration API are taking maximum time to execute requests. For python API, we have used in-built libraries to implement our logic that is causing significant delays in processing requests. The registration API involves writing a chunk of user information to the database which is time consuming. Also, the synchronous behavior of REST API's implemented for inter microservice communication is causing delays in our system and limiting the capacity of our system.

Plots :




Response Time for Spike Testing - 5 Replicas
Registration-requests
Authentication-requests
Audit-requests
ForgotPassword-requests
UpdatePassword-requests
WeatherReport-requests
Milliseconds
16:24:40
16:24:50
16:25:00
16:25:10
16:25:20
16:25:30
16:25:40
16:25:50
16:26:00
16:26:10
16:26:20
16:26:30

❖ Fault Tolerance Results :

Our implementation relies on the self-healing property of a kubernetes cluster, this is done through the internal kubernetes cluster services that keeps on monitoring the health of the cluster services and its pods. If one of the pods goes down, this is detected and immediately a new pod is deployed and configured.

This is demonstrated below,

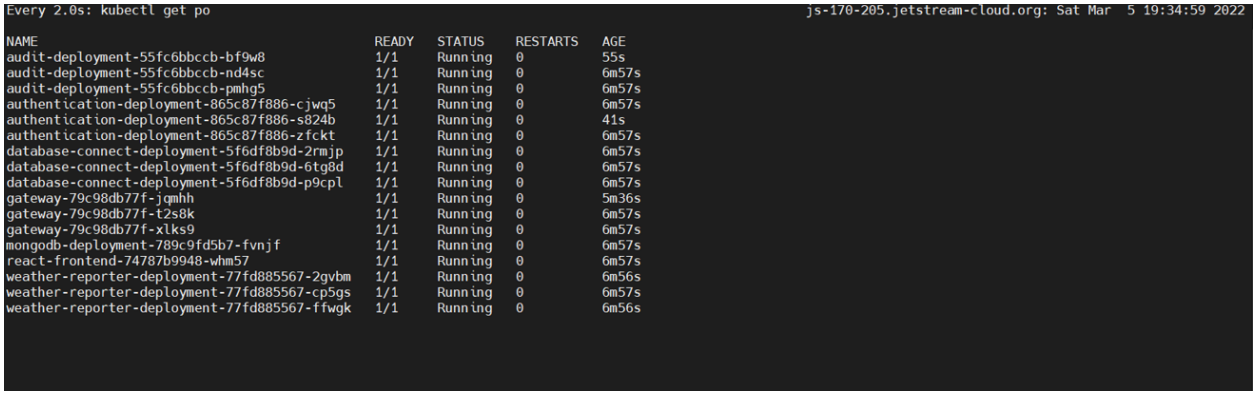
- ➢ As we send a steady load of 1000 users to our system, we can see that all the pods are running.

```
Every 2.0s: kubectl get po                                                    js-170-205.jetstream-cloud.org: Sat Mar  5 19:34:12 2022

NAME                                          READY   STATUS    RESTARTS   AGE
audit-deployment-55fc6bbccb-bf9w8             1/1     Running   0          8s
audit-deployment-55fc6bbccb-nd4sc             1/1     Running   0          6m10s
audit-deployment-55fc6bbccb-pmhg5             1/1     Running   0          6m10s
authentication-deployment-865c87f886-64gft    1/1     Running   0          6m10s
authentication-deployment-865c87f886-cjwq5    1/1     Running   0          6m10s
authentication-deployment-865c87f886-zfckt    1/1     Running   0          6m10s
database-connect-deployment-5f6df8b9d-2rmjp   1/1     Running   0          6m10s
database-connect-deployment-5f6df8b9d-6tg8d   1/1     Running   0          6m10s
database-connect-deployment-5f6df8b9d-p9cpl   1/1     Running   0          6m10s
gateway-79c98db77f-jqmhh                      1/1     Running   0          4m49s
gateway-79c98db77f-t2s8k                      1/1     Running   0          6m10s
gateway-79c98db77f-xlks9                      1/1     Running   0          6m10s
mongodb-deployment-789c9fd5b7-fvnjf           1/1     Running   0          6m10s
react-frontend-74787b9948-whm57               1/1     Running   0          6m10s
weather-reporter-deployment-77fd885567-2gvbm  1/1     Running   0          6m9s
weather-reporter-deployment-77fd885567-cp5gs  1/1     Running   0          6m10s
weather-reporter-deployment-77fd885567-ffwgk  1/1     Running   0          6m9s
```

- ➢ At 7:34 pm, we killed one of the pods, which led to the traffic to be distributed among the other two containers. Therefore we experience no request drops during this time. Also, as the container is terminating, we can see that k8s cluster has already processed the creation of a new container.

```
Every 2.0s: kubectl get po                                                    js-170-205.jetstream-cloud.org: Sat Mar  5 19:34:23 2022

NAME                                          READY   STATUS              RESTARTS   AGE
audit-deployment-55fc6bbccb-bf9w8             1/1     Running             0          19s
audit-deployment-55fc6bbccb-nd4sc             1/1     Running             0          6m21s
audit-deployment-55fc6bbccb-pmhg5             1/1     Running             0          6m21s
authentication-deployment-865c87f886-64gft    0/1     Terminating         0          6m21s
authentication-deployment-865c87f886-cjwq5    1/1     Running             0          6m21s
authentication-deployment-865c87f886-s824b    0/1     ContainerCreating   0          5s
authentication-deployment-865c87f886-zfckt    1/1     Running             0          6m21s
database-connect-deployment-5f6df8b9d-2rmjp   1/1     Running             0          6m21s
database-connect-deployment-5f6df8b9d-6tg8d   1/1     Running             0          6m21s
database-connect-deployment-5f6df8b9d-p9cpl   1/1     Running             0          6m21s
gateway-79c98db77f-jqmhh                      1/1     Running             0          5m
gateway-79c98db77f-t2s8k                      1/1     Running             0          6m21s
gateway-79c98db77f-xlks9                      1/1     Running             0          6m21s
mongodb-deployment-789c9fd5b7-fvnjf           1/1     Running             0          6m21s
react-frontend-74787b9948-whm57               1/1     Running             0          6m21s
weather-reporter-deployment-77fd885567-2gvbm  1/1     Running             0          6m20s
weather-reporter-deployment-77fd885567-cp5gs  1/1     Running             0          6m21s
weather-reporter-deployment-77fd885567-ffwgk  1/1     Running             0          6m20s
```

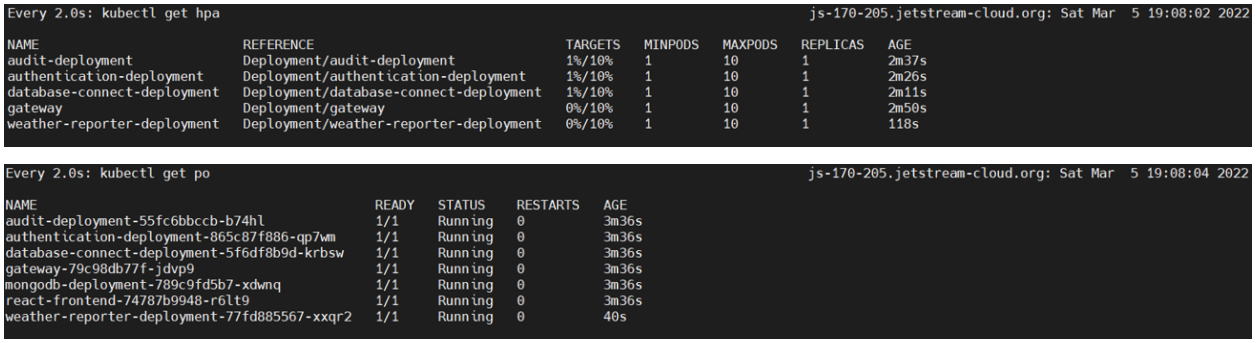
➢ We can see that in under 30 seconds, the new container is up again and the system has reached a steady state and is handling the given load with ease.



❖ Horizontal Pod Autoscaling Results :

For scalability we have implemented k8s horizontal pod autoscaling with a metrics server. The k8s metrics server keeps on monitoring the cpu usage of our defined hpa deployments, and we have set a threshold of 50%. As the cpu-usage reaches 50% of allotted cpu, a new pod will be scaled up. This will continue until it reaches the maximum pod capacity we have set for our hpa.

➢ Initially, we have just 1 replica per pod for a very low traffic to our application.

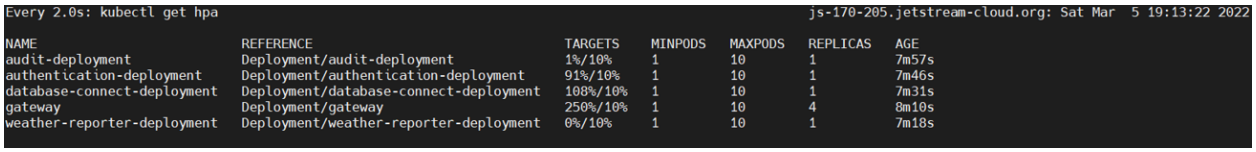


```
Every 2.0s: kubectl get hpa                                                    js-170-205.jetstream-cloud.org: Sat Mar  5 19:08:02 2022

NAME                         REFERENCE                                  TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
audit-deployment             Deployment/audit-deployment                1%/10%    1         10        1          2m37s
authentication-deployment    Deployment/authentication-deployment       1%/10%    1         10        1          2m26s
database-connect-deployment  Deployment/database-connect-deployment     1%/10%    1         10        1          2m11s
gateway                      Deployment/gateway                         0%/10%    1         10        1          2m50s
weather-reporter-deployment  Deployment/weather-reporter-deployment     0%/10%    1         10        1          118s
```



```
Every 2.0s: kubectl get po                                                     js-170-205.jetstream-cloud.org: Sat Mar  5 19:08:04 2022

NAME                                        READY   STATUS    RESTARTS   AGE
audit-deployment-55fc6bbccb-b74hl           1/1     Running   0          3m36s
authentication-deployment-865c87f886-qp7wm  1/1     Running   0          3m36s
database-connect-deployment-5f6df8b9d-krbsw 1/1     Running   0          3m36s
gateway-79c98db77f-jdvp9                    1/1     Running   0          3m36s
mongodb-deployment-789c9fd5b7-xdwnq         1/1     Running   0          3m36s
react-frontend-74787b9948-r6lt9             1/1     Running   0          3m36s
weather-reporter-deployment-77fd885567-xxqr2 1/1    Running   0          40s
```

➢ Now we have sent a sudden spike of 7000 requests to our gateway, authentication and database services. At first we will observe as all the requests are directed to the gateway first, the gateway will scale up pretty fast and has already scaled up to 4 replicas.



```
Every 2.0s: kubectl get hpa                                                    js-170-205.jetstream-cloud.org: Sat Mar  5 19:13:22 2022

NAME                         REFERENCE                                  TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
audit-deployment             Deployment/audit-deployment                1%/10%     1         10        1          7m57s
authentication-deployment    Deployment/authentication-deployment       91%/10%    1         10        1          7m46s
database-connect-deployment  Deployment/database-connect-deployment     108%/10%   1         10        1          7m31s
gateway                      Deployment/gateway                         250%/10%   1         10        4          8m10s
weather-reporter-deployment  Deployment/weather-reporter-deployment     0%/10%     1         10        1          7m18s
```

➢ Now as the other microservices start receiving more traffic, they will start scaling up as well, we can see that authentication and database microservices have both scaled up to 4. And the gateway microservice has scaled up to 8 replicas.

```
Every 2.0s: kubectl get hpa                                                    js-170-205.jetstream-cloud.org: Sat Mar  5 19:13:37 2022

NAME                          REFERENCE                                   TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
audit-deployment              Deployment/audit-deployment                 1%/10%     1         10        1          8m12s
authentication-deployment     Deployment/authentication-deployment        91%/10%    1         10        4          8m1s
database-connect-deployment   Deployment/database-connect-deployment      108%/10%   1         10        4          7m46s
gateway                       Deployment/gateway                          250%/10%   1         10        8          8m25s
weather-reporter-deployment   Deployment/weather-reporter-deployment      0%/10%     1         10        1          7m33s
```

➢ When all the microservices are processing the 7000 requests, we end up getting 10 replicas per microservice which was the max replica we had set for our hpa.

```
Every 2.0s: kubectl get hpa                                                    js-170-205.jetstream-cloud.org: Sat Mar  5 19:14:41 2022

NAME                          REFERENCE                                   TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
audit-deployment              Deployment/audit-deployment                 1%/10%     1         10        1          9m16s
authentication-deployment     Deployment/authentication-deployment        157%/10%   1         10        10         9m5s
database-connect-deployment   Deployment/database-connect-deployment      115%/10%   1         10        10         8m50s
gateway                       Deployment/gateway                          23%/10%    1         10        10         9m29s
weather-reporter-deployment   Deployment/weather-reporter-deployment      0%/10%     1         10        1          8m37s
```

```
Every 2.0s: kubectl get po                                                     js-170-205.jetstream-cloud.org: Sat Mar  5 19:20:08 2022

NAME                                          READY   STATUS        RESTARTS   AGE
audit-deployment-55fc6bbccb-b74hl             1/1     Running       0          15m
authentication-deployment-865c87f886-4hg4t    1/1     Running       0          6m1s
authentication-deployment-865c87f886-5h8tq    1/1     Running       0          6m46s
authentication-deployment-865c87f886-6w4sw    1/1     Running       0          5m46s
authentication-deployment-865c87f886-dtg9l    1/1     Running       0          6m1s
authentication-deployment-865c87f886-k9wx7    1/1     Running       0          6m46s
authentication-deployment-865c87f886-qp7wm    1/1     Running       0          15m
authentication-deployment-865c87f886-th4pc    1/1     Running       0          6m1s
authentication-deployment-865c87f886-ts64j    1/1     Running       0          6m46s
authentication-deployment-865c87f886-w5vbv    1/1     Running       0          6m1s
authentication-deployment-865c87f886-xxhdb    1/1     Running       0          5m46s
database-connect-deployment-5f6df8b9d-4fxvj   1/1     Running       0          6m1s
database-connect-deployment-5f6df8b9d-5t42t   1/1     Running       0          6m1s
database-connect-deployment-5f6df8b9d-8c54b   1/1     Running       0          6m1s
database-connect-deployment-5f6df8b9d-f87r8   1/1     Running       0          6m1s
database-connect-deployment-5f6df8b9d-j7szf   1/1     Running       0          5m46s
database-connect-deployment-5f6df8b9d-jzgfj   1/1     Running       0          5m46s
database-connect-deployment-5f6df8b9d-krbsw   1/1     Running       0          15m
database-connect-deployment-5f6df8b9d-lgh66   1/1     Running       0          6m46s
database-connect-deployment-5f6df8b9d-rgs25   1/1     Running       0          6m46s
database-connect-deployment-5f6df8b9d-v8dtm   1/1     Running       0          6m46s
gateway-79c98db77f-5ffbz                      0/1     Terminating   0          6m40s
gateway-79c98db77f-65pzp                      0/1     Terminating   0          6m55s
gateway-79c98db77f-ck7t2                      1/1     Running       0          7m10s
gateway-79c98db77f-jdvp9                      0/1     Terminating   0          15m
gateway-79c98db77f-kkvwn                      0/1     Terminating   0          6m55s
gateway-79c98db77f-nswmc                      0/1     Terminating   0          6m40s
gateway-79c98db77f-p4kkn                      0/1     Terminating   0          6m55s
mongodb-deployment-789c9fd5b7-xdwnq           1/1     Running       0          15m
react-frontend-74787b9948-r6lt9               1/1     Running       0          15m
weather-reporter-deployment-77fd885567-xxqr2  1/1     Running       0          12m
```

➢ Finally, after 5 minutes when the traffic load has substantially reduced we can see our containers scaling down and reaching back to the original state. The scaled up containers are maintained for a short period of time in anticipation of more requests. When the application does not receive a large number of requests, the containers are scaled down proportionately to serve the current load of requests.

```
Every 2.0s: kubectl get hpa                                                    js-170-205.jetstream-cloud.org: Sat Mar  5 19:20:32 2022

NAME                          REFERENCE                                   TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
audit-deployment              Deployment/audit-deployment                 1%/10%     1         10        1          15m
authentication-deployment     Deployment/authentication-deployment        1%/10%     1         10        2          14m
database-connect-deployment   Deployment/database-connect-deployment      1%/10%     1         10        3          14m
gateway                       Deployment/gateway                          0%/10%     1         10        1          15m
weather-reporter-deployment   Deployment/weather-reporter-deployment      0%/10%     1         10        1          14m
```

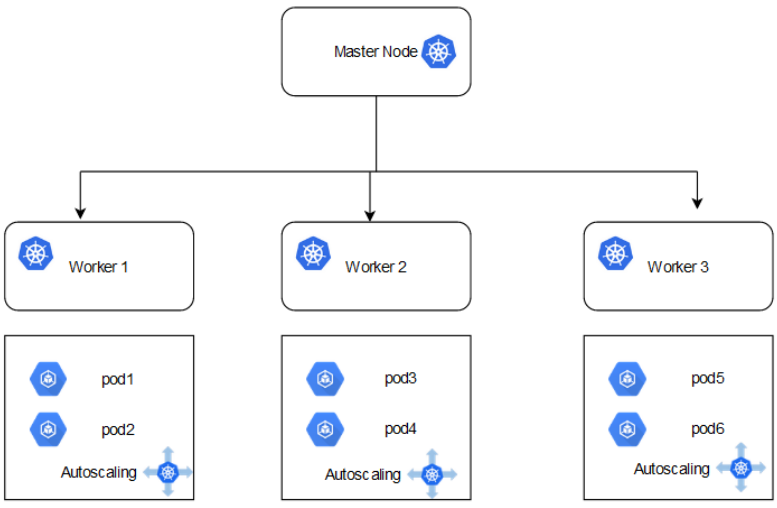
❖ Architecture Changes

- **Challenges faced :**
  1) We were limited by our local system's capacity which was affecting our system capacity.
  2) Selecting an open source container orchestration system that was best suitable for our application.
  3) Fine tuning our python microservice to reduce its response time to increase the overall capacity of our system.

- **How we overcame our challenges :**
  1) We were limited by our local system's capacity which can be substantially improved by moving our system to jetstream.
  2) We set up kubernetes that handles autoscaling, load balancing and self healing aspects of our system. We went for a single master, multiple worker node setup where all the pods are launched in the worker nodes, and master node serves as the control plane and orchestrates the k8s cluster.

     We set this up using kubeadm, kubectl and kubelet configured on Ubuntu 20.04 servers. We have further implemented multiple replicas and autoscaling in the kubernetes cluster configurations.

     The server configurations are the master node is a m1.quad (4 vCPU, 10GB RAM) and the three worker nodes are of the same configuration (6 vCPU, 16GB RAM). All these servers are hosted in the same provider Indiana University and they interact with each other using their private IP's.



3) For our python microservice, we implemented a caching and multiprocessing mechanism to decrease the response time. We also increased the allocated RAM and CPU for python microservice while deploying it as we analyzed that the other microservices used less RAM and CPU as compared to our python microservice.

- **Future Plans to further improve the performance, reliability and capacity of our system :**
    1) Currently we have implemented synchronous API calls between microservices. We will be moving to asynchronous API calls between microservices to increase the throughput.
    2) Research and implement alternative python libraries to decrease the response time of our application even further.
    3) Currently, we only have one database server running to maintain strict consistency. We want to deploy one more replica of our database server and enable replication to preserve the data for database recovery in case of failure.