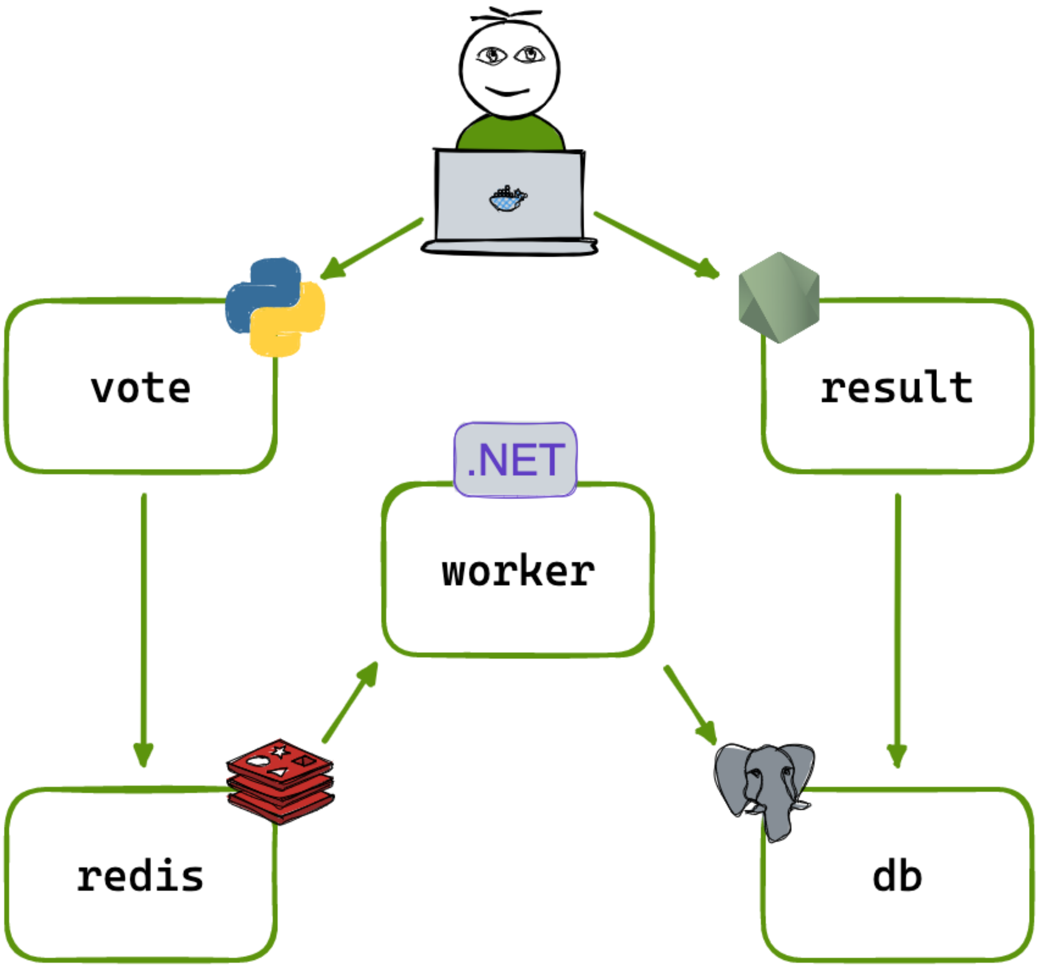
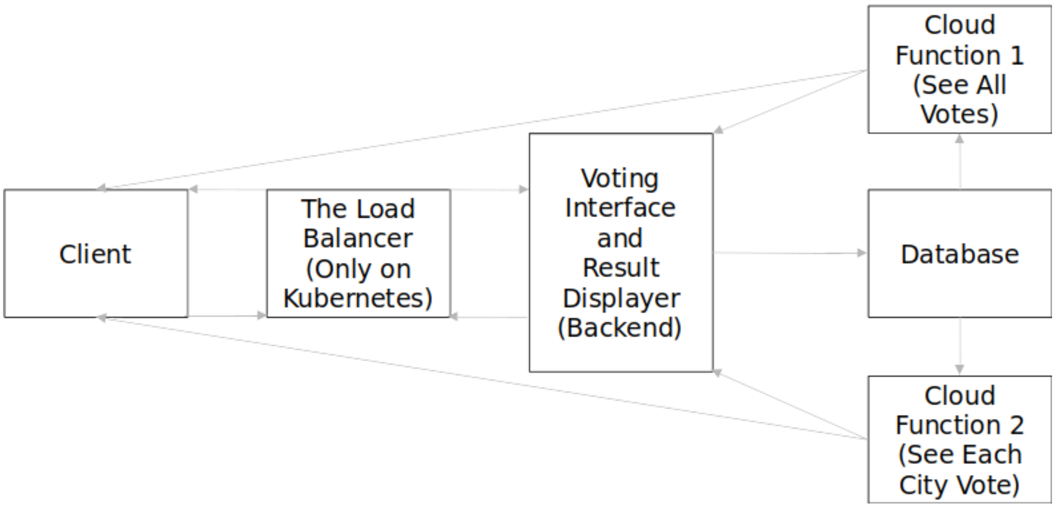
Introduction

- You can find our github repo at https://github.com/hks1444/cmpe48a.
- You can find the whole report at https://github.com/hks1444/cmpe48a/wiki/Cmpe48A-Term-Project-Final-Report.
- You can find our video link for VMs here and for Kubernetes Redis here.
- Kubernetes related stuff is located at Kubernetes_part folder and VM related stuff is located at VM_part fold.
- In our project we decided to deploy a voting app. First we found the following repo: example-voting-app. This repo uses Python for its voting interface, Redis for collecting new votes, a .NET worker for consuming votes to store them in a Postgres database backed by a Docker volume, and Node.js web app which shows the results of the voting in real time.



- However we decided to implement the system from scratch because it we did not have any experience in .NET and it would be hard to modify an existing codebase someone else wrote if we need to tweak the code or add new features like cloud functions.

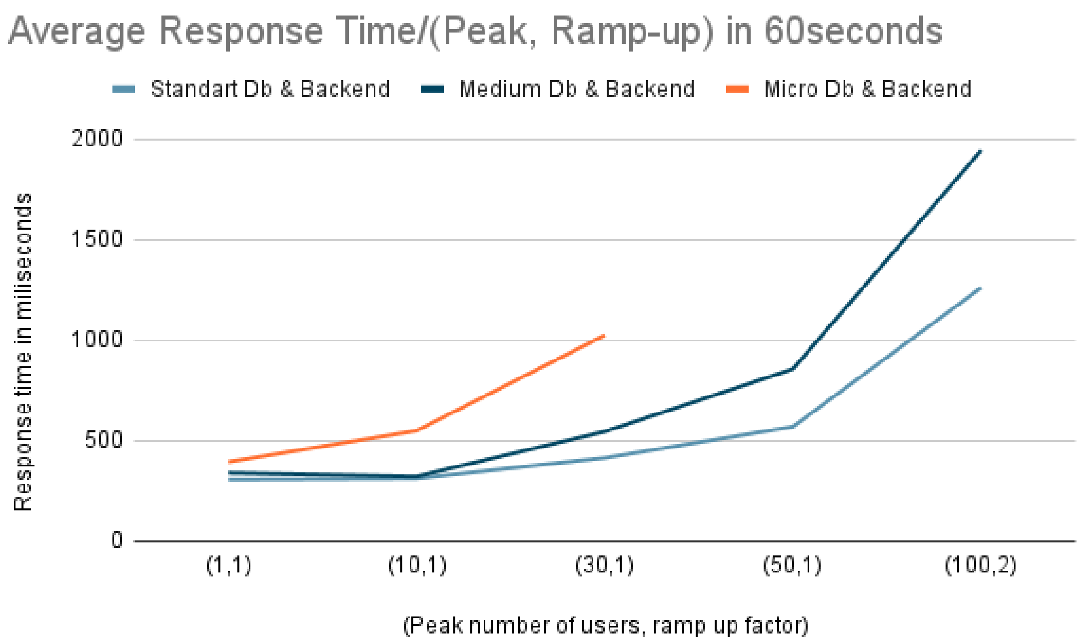
- Implementing the app from scratch allowed us understand the underlying system thoroughly. In our initial implementation we have a voting interface implemented in Django, Redis for collecting new votes, a python script as worker for consuming votes to store them in a Postgres database. Instead of Node.js for showing results, we created two cloud functions to see overall results and see results for each city.
- Later we decided to remove Redis beacuse the repsonse time for the voting system did not provide a full measure to inspect a request's impact on the system. The system could quickly queue many requests and return answers in a short time but we would not be able to see each votes impact at the response time. In other words inserting votes to the database takes more time than just queueing the request. We could poll the database periodically to detect changes but this would create an extra load on the system affecting performance and this approach might have not capture each request (in a checking period the more than one request can be processed).



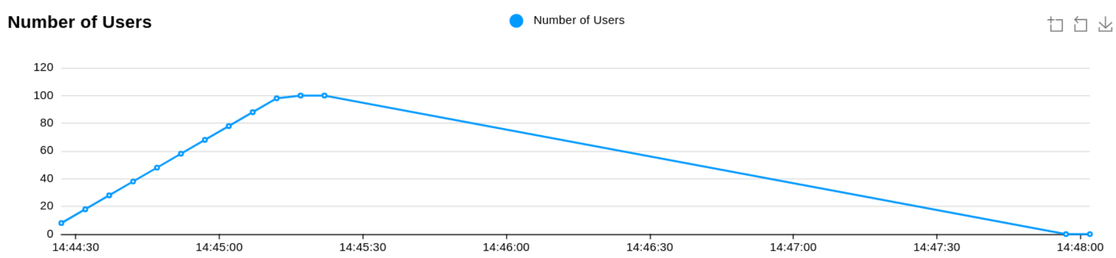
- Nevertheless, the new architecture described above had abysmal results even when it was lightly loaded. Even the lightest loads had response time in magnitude of hundreds of millisecond at best, easily surpassing seconds on the average. After seeing the architecture deployed on Kubernetes outperformed the VM based architecture, we decided to deploy Redis based version on the Kubernetes. This version had the best response time around 70 miliseconds. This is expected because the response time is only dependent on the queueing time thus not directly reflecting each request's effect on the database immediately.
- Not directly reflecting results directly is not a real issue though. In the real world examples (i.e. elecdtions) results are not immediately shown to the public when the votes in a single ballot box are tallied, instead results shown are updated when certain amount of new votes entered to the system.

For Vm Part

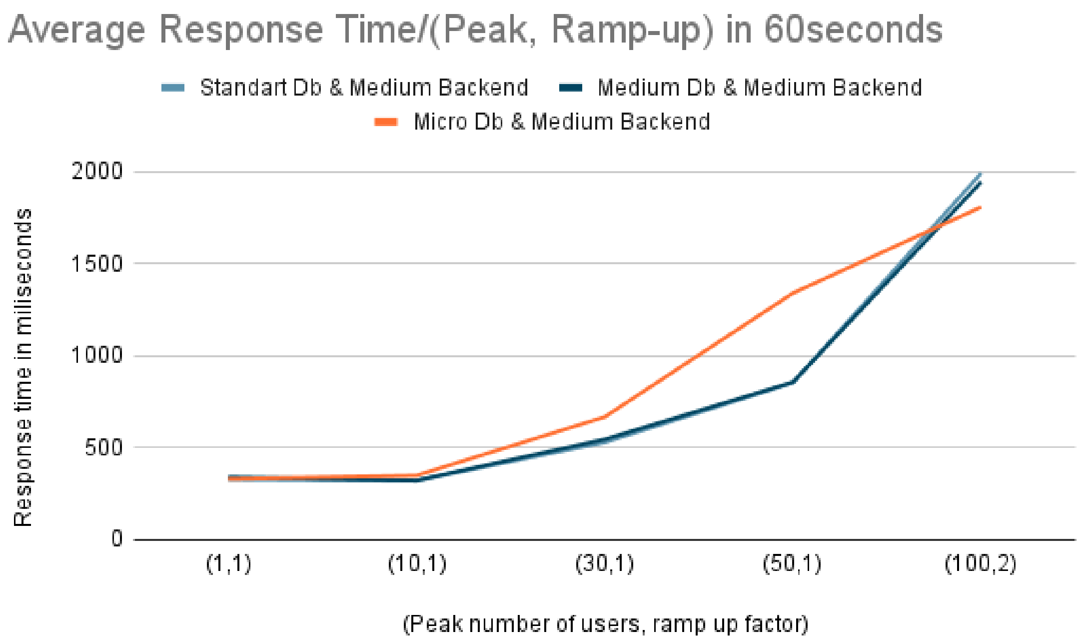
In this part we used two VMs. One of them was running the backend and the other one was hosting the database. We vertically scaled them and applied different loads. We tested backend on three different instances: e2-micro(0.25-2 vCPU(1 shared core), 1 GB memory), e2-medium(1-2 vCPU(1 shared core), 4 GB memory), e2-standard-4(2 vCPU(2 core), 16 GB memory). For each backend configuration we tested three different database configuration: e2-micro(0.25-2 vCPU(1 shared core), 1 GB memory), e2-medium(1-2 vCPU(1 shared core), 4 GB memory), e2-standard-4(2 vCPU(2 core), 16 GB memory). You can see the detailed results [here](). Each one of the nine files you see here contains the test results for a different configuration. In each file you will see files like 100_2.html. This files are test results. 100 means peak number of users and 2 means ramp up factor. In each second two more users are created until the number of users are 100. We run this tests for 60 seconds. The results of the tests showed us scaling both VMs vertically at once improved the response time:

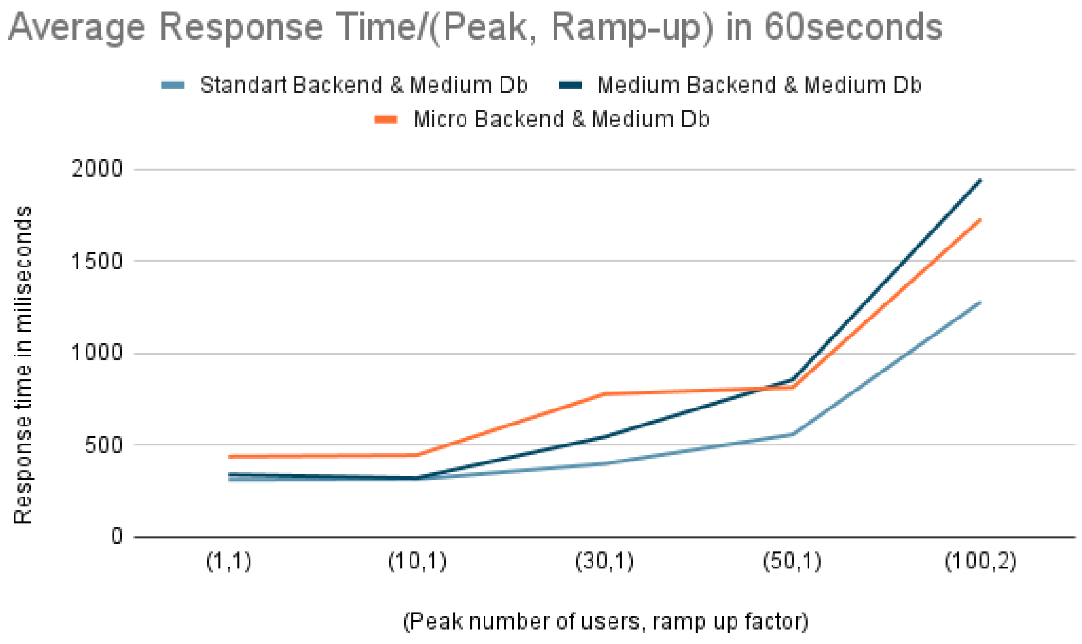


The y-axis of the graph is repsonse time in miliseconds and x-axis tuples are tuples of peak user number and ramp up factor which are explained above. We can look at the chart below to understand the (Peak number of users, Ramp-up factor) tuple. In each two more users are spwaned and at the fiftieth second number of users reached to the peak of 100 and it stayed the same for ten seconds. Then the experiment is finished.

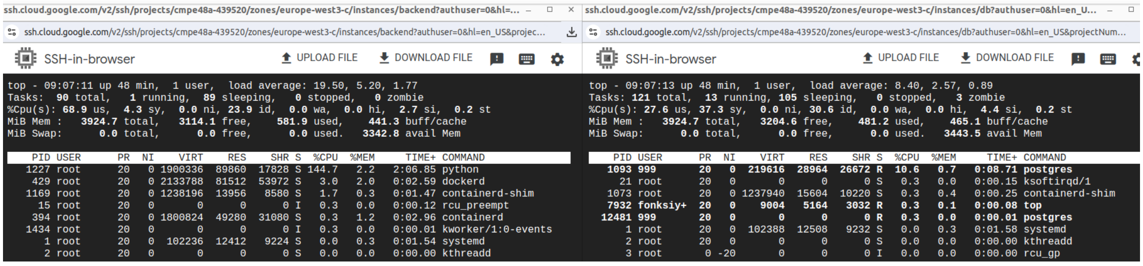


Another result we derived from the experiment is the backend is the limiting factor if one of the VM is kept at e2-medium and the other one is upscaled or downscaled. Because scaling database did not yield improvement while the backend is kept same and scaling backend yielded an improvement while database is kept same.





The y-axis of thees two graphs is repsonse time in miliseconds and x-axis tuples are tuples of peak user number and ramp up factor as the previous graph. You can see in the e2-medium backend and database configuration the limiting factor is the backend VM since its CPU usage(%CPU(s):68.9 us) is more than the database VM(%CPU(s):27.6 us). The screenshot below is taken when the both VMs are in stable state while their test configuration is 100 peak number of users with 2 ramp up factor.



For each average response time/(peak, ramp-up) graph you can check appendix A for more detailed results.

The backend VMs has [35.246.137.147]() as its static ip address. The database VMs has [34.159.36.35]() as its static ip address.

For Kubernetes Part

The Kubernetes has [34.110.171.4]() as its static ip address.

Note : In the result tables every axis metrics same with the VM part so it is not mentioned again.

Although the VM system can be useful for simple tasks, using VM with 2 machines was not giving us the performance we were looking for. For comparison, we decided to build a kubernetes system with 2 nodes. Kubernetes gave us a difference with its robust load balance feature. Unlike VMs, here we applied the tests using pods. While VMs are more suitable for vertical scaling, the management of pods in kubernetes was done by managing the load in the system with horizontal scaling method. Therefore, the only machine type we can use in this comparison is e2-micro as opposed to VM. The only way to ensure performance gains is to optimally distribute the load between these underpowered machines. In this regard, 3 features of Kubernetes come to our aid.

1 - Improved Scalability: Kubernetes allows dynamic scaling in the system through pod management, enabling the system to handle varying loads more efficiently compared to the original VM-based architecture. 2 - Improved Resource Management: Kubernetes, being able to orchestrate containers properly, ensures the allocation of resources and their utilization across all system components much better than handling parallel voting operations. 3 - Load Distribution: The Kubernetes load balancing in native could help the system distribute the requests among multiple pods, hence increasing its responsiveness and reliability.

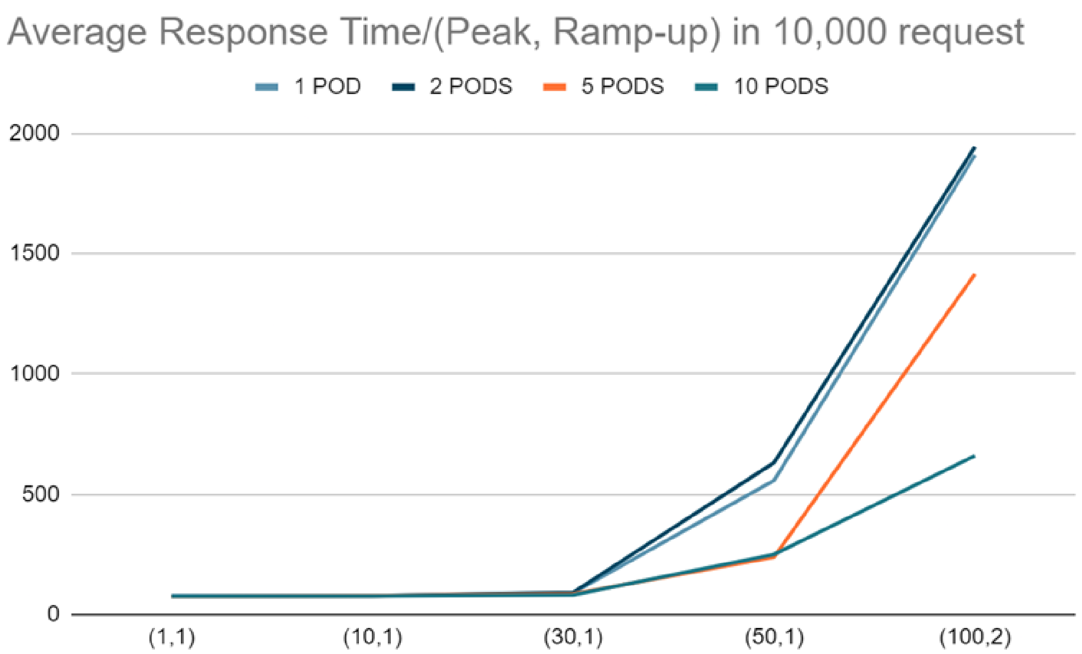
It is worth noting that since our main goal here is performance testing, we did the tests by manually changing the number of pods instead of using the autoscaling offered by Kubernetes. The reason for this decision is that our goal in this test is to measure the performance under the highest loads. In these load tests, the system will already use the highest resources, so the system will use the value at the maximum pod limit. It is also worth noting that the system is capped at 300 dollars in terms of cost. In order to save even more money on top of this, the optimal number of pods should be set as the maximum number of pods, while the lower limit should be chosen by calculating how long the system we will use will never be under load. In the meantime, we need to pay attention to the elasticity of the system so that we don't lose users due to under-provisioning as the system becomes increasingly overloaded. In our voting application, such calculations do not make sense. Because for the voting system, which is done every 2 to 4 years, the cost of the system that will be used only for one day will not be very important for us. Since there will be a continuous voting situation during that day, it is more valuable for us to run a high number of pods with high reliability. For these reasons, it is the best choice to use the maximum number of pods throughout the day instead of switching between minimum and maximum pod values.

We added 1, 2, 5, 10 backend pods inside our 2 nodes, except for 1 Postgress database inside our 2 nodes to examine the pod change of Kubernetes.

For Results please refer to Appendix B.1

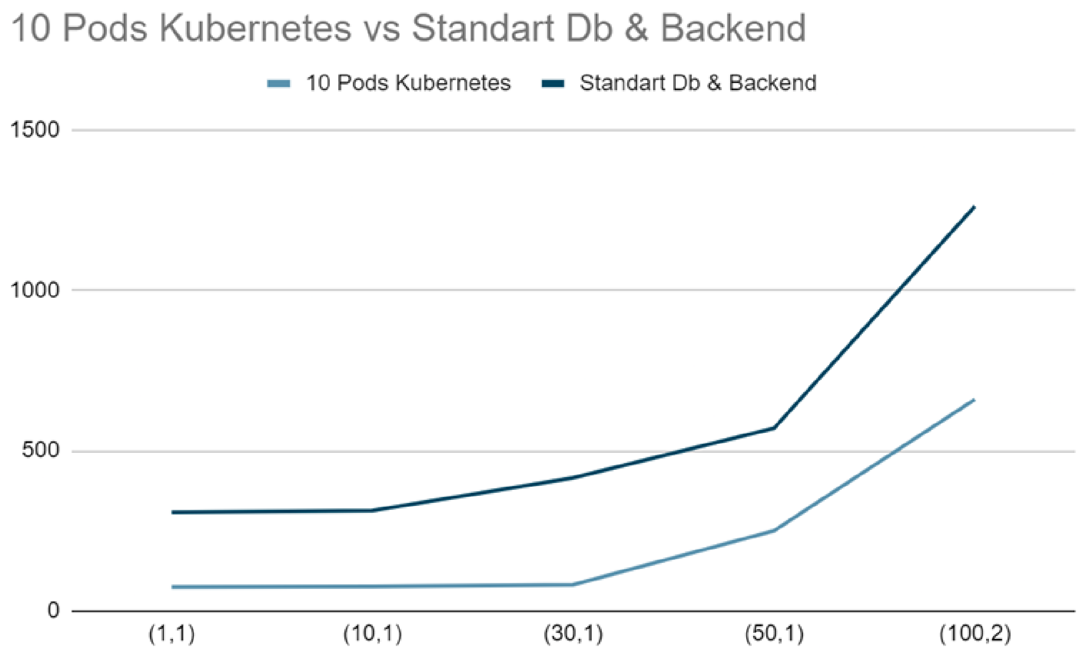
The detail that we can easily notice in these tables is that for 1 user, all of the systems give almost the same performance, but as we approach 100 parallel users, we see that the actual load distribution in the system becomes more difficult and important task. The more pods the Kubernetes load balancer can distribute the load, the sooner it responses to the user.

If we put these results in a table like VMs:



Kubernetes vs VM in overall

In our comparison of Kubernetes and VM structure, we compared 2 node systems and 2 VM machines for a correct comparison. This comparison can be seen as a bit of a vertical scaling and load balancing comparison. The reason for this is basically that while in VMs we develop on the power of the same machine, in Kubernetes our development criteria is not about the number or power of computers, but how much we divide the load on that computer and thus how efficiently we use the capacity.



As can be seen from the contraction, the effect of not being able to distribute enough power on the powerful machine caused delays of up to 2 times. Therefore, it would be a better choice to use a system with robust load balancing installed with kubernetes.

Approaching real life examples

As a result of our comparisons between Kubernetes and VMs, we realized that having multiple backend pods in Kubernetes and load balancing between them is better than even the most powerful VMs, so we decided to bring this closer to real life. The first step here was to figure out where the bottleneck was in our most powerful configuration, a 10 pod 2 node system.

Type	Name	# Requests	# Fails	Average (ms)
POST	/voting/clear-db/	21455	10	3037.73
GET	/voting/see-all/	21394	35	998.12
GET	/voting/see-cities/	21390	25	920.62
POST	/voting/vote/	21406	35	3160.35
	Aggregated	85645	105	2030.13

As we can see in this photo, the biggest delay is caused by post requests to /voting/vote, which is the process of adding data to the database. The reading process takes much less time. We will return to this detail in later developments. Also notice that some of the votes failed and this means that the vote of a person is vanished which is not acceptable.

```
PS C:\Users\Musta\Desktop\kubernetes\Postgres> kubectl top pods
NAME                               CPU(cores)   MEMORY(bytes)
postgres-79d96cd89f-8h75t          719m         45Mi
voting-pub-84f98567cd-618tp        70m          104Mi
voting-pub-84f98567cd-9zr9z        65m          104Mi
voting-pub-84f98567cd-cbzfn        73m          105Mi
voting-pub-84f98567cd-f2vwn        68m          104Mi
voting-pub-84f98567cd-k97jz        67m          103Mi
voting-pub-84f98567cd-svn9n        59m          104Mi
voting-pub-84f98567cd-wrqln        68m          105Mi
voting-pub-84f98567cd-xq6qd        55m          104Mi
voting-pub-84f98567cd-z7lkv        60m          104Mi
voting-pub-84f98567cd-zd4nf        67m          104Mi
```

When we try to understand the reason for the delay in post requests from the system utilization values, the first thing we easily notice is that the database pod is overloaded, while the backend pods are not strained and create an unbalanced distribution. Since the database tries to write every request as soon as it arrives, it accumulates a lot of requests in its queue and this accumulation causes delays and slowdowns.

To solve this, we decided to reintroduce Redis, which we had initially given up on using.

Redis

Unlike a PostgreSQL pod that writes every incoming data instantly to disk, Redis keeps data in memory and persists it according to a configured schedule. In our implementation, Redis is configured to write data to disk in two ways: 1 - Append-Only File (AOF) Persistence:

- Logs every write operation in memory
- Syncs these operations to disk every second
- Provides a good balance between performance and data safety

2 - Snapshot-based Persistence:

- Takes complete database snapshots based on activity levels
- Snapshots are taken more frequently during high-load periods; provides recovery points for the system.

This approach greatly improved our system performance for the following reasons:

- Memory operations are much faster compared to disk writes.

- Batching writes reduces I/O overhead.

- The system can now handle more concurrent votes without waiting for disk operations.

  In order to see these effects on the tests, when we did the combination of 2 pods and 100 users on the redis system, we reached a hard to believe but real result like 80ms. At the same time our failure rate decreased from 0.1635% to 0.0177%. Of course, that's still almost 14000 games lost in an election with 80 million people. We tried to solve this problem of pods not being able to keep up with the requests by increasing the number of pods (of course, in a real life example, this problem can be solved by using more powerful machines, but we aimed to offer the most logical solution within our limits). Also, since the duration of unsuccessful requests is now much shorter, a solution like resubmitting the same request without changing it could have solved this problem as well. Satisfied with these results, we decided to take it a step further and try using 500 and then 1000 users at the same time. In these experiments, our number of failures and response time started to become a problem again. So instead of trying to solve this with only 2 pods, we increased the number of backend workers to 10 and wanted to test how well we could get a good result when we use the limited machines we have most efficiently.

2 Pod Redis Results for 1000 User

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/voting/clear-db/	1000	0	1128.37	80	15985	53	3.6	0
GET	/voting/see-all/	11270	1007	7517.14	55	67534	2620.83	40.63	3.63
GET	/voting/see-cities/	10853	908	7148.49	55	67646	375.71	39.12	3.27
POST	/voting/vote/	11735	413	2434.31	54	36760	62.82	42.3	1.49
	Aggregated	34858	2328	5507.94	54	67646	986.99	125.66	8.39

10 Pod Redis Results for 1000 User

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/voting/clear-db/	1000	0	545.54	78	17136	53	0.44	0
GET	/voting/see-all/	84551	102	11882.61	54	66717	7148.29	37.4	0.05
GET	/voting/see-cities/	84111	90	11799.45	54	64618	1007.31	37.2	0.04
POST	/voting/vote/	85012	75	413.42	53	34525	53.25	37.6	0.03
	Aggregated	254674	267	7982.13	53	66717	2723.88	112.64	0.12

As a result of these experiments, the response time of the discarded votes decreased from 2.5 seconds to 0.4 seconds and the failure rate decreased again from 3.5% to 0.088%. The reason why the number looks higher in the table is that the one on the left was run for 35000 votes and the one on the right for 255000 votes. What is striking in this table is that even though the writing process has gotten faster, our read operations taking extremely long time unlike our results in Postgres, causing high latency in overal. This is due to Redis write and read operations differ considerably in performance. The performance of the write operations to the vote/ endpoint is very efficient, having a response time of 413.42ms and an extremely low failure rate of 0.03%, while handling small data packets of about 53.25 bytes. This stellar write performance results from Redis being an in-memory solution; it basically stores the votes instantaneously while persisting data to disk asynchronously on every second, thanks to its AOF mechanism. Reads are different, however. Both the see-all/ and see-cities/ present response times that reach up to 11,800ms with slightly higher failure rates, and far bigger data transfers-particularly the see-all/, which carries over 7,000 bytes per request. This is because the read operations need to do a lot of aggregating and processing on all the stored votes, which may imply disk I/Os on data not in memory and with complex computation to determine vote totals and statistics. While Redis is great for our write-heavy voting operations, its in-memory architecture means that read performance suggests we should consider further optimizations, such as the implementation of result caching, pre-calculating aggregations, or setting up read replicas to better handle demanding read operations.

While we think about real life situations in this case, we should also think about real life solutions. If we think about a real-life voting system, we never see exactly how many votes are cast in real time. On the contrary, the system updates after a certain period of time and gives us an approximate value. So we benefit a lot from the caching mechanism of the results. So instead of scanning the database every time a vote query comes in, if we send the calculated value for a certain period of time and update the value after a certain period of time, we can see much stronger results in terms of our performance.

Type	Name	# Requests	# Fails	Median (ms)
POST	/voting/clear-db/	1000	0	110
GET	/voting/see-all/	73284	143	9800
GET	/voting/see-cities/	72818	134	9700
POST	/voting/vote/	73737	2	210
	Aggregated	220839	279	6700

Interestingly, when I tried this idea and implemented a mechanism that keeps get requests in cache for 1 minute. Here are some of the reasons why I have not seen as much improvement as I expected:

- Network latency between pods
- Time spent serializing large amounts of data
- Lock contention when multiple requests try to access the cache simultaneously

We decided not to go beyond the scope of the project, but to examine the results and finalize the work. Finally, when we examine the resource utilization of our kubernetes system with 1 redis and 10 backend pods, we encountered a much more homogeneous distribution than at the beginning as can be seen below.

```
PS C:\Users\Musta\Desktop\kubernetes\Redis\pub_kubernetes> kubectl top pods
NAME                          CPU(cores)   MEMORY(bytes)
redis-db9cbcdbb-l7k7x         256m         19Mi
voting-pub-66c4454ff4-24gxr   213m         156Mi
voting-pub-66c4454ff4-58hrn   232m         165Mi
voting-pub-66c4454ff4-cm4qx   133m         134Mi
voting-pub-66c4454ff4-l4qnt   120m         130Mi
voting-pub-66c4454ff4-pcntk   135m         131Mi
voting-pub-66c4454ff4-tbvp9   187m         157Mi
voting-pub-66c4454ff4-tjvkd   221m         151Mi
voting-pub-66c4454ff4-z4gsb   133m         131Mi
voting-pub-66c4454ff4-zmsbk   193m         152Mi
voting-pub-66c4454ff4-zzqrk   238m         157Mi
PS C:\Users\Musta\Desktop\kubernetes\Redis\pub_kubernetes>
```

Appendix

You can find the whole appendix at https://github.com/hks1444/cmpe48a/wiki/Appendix.