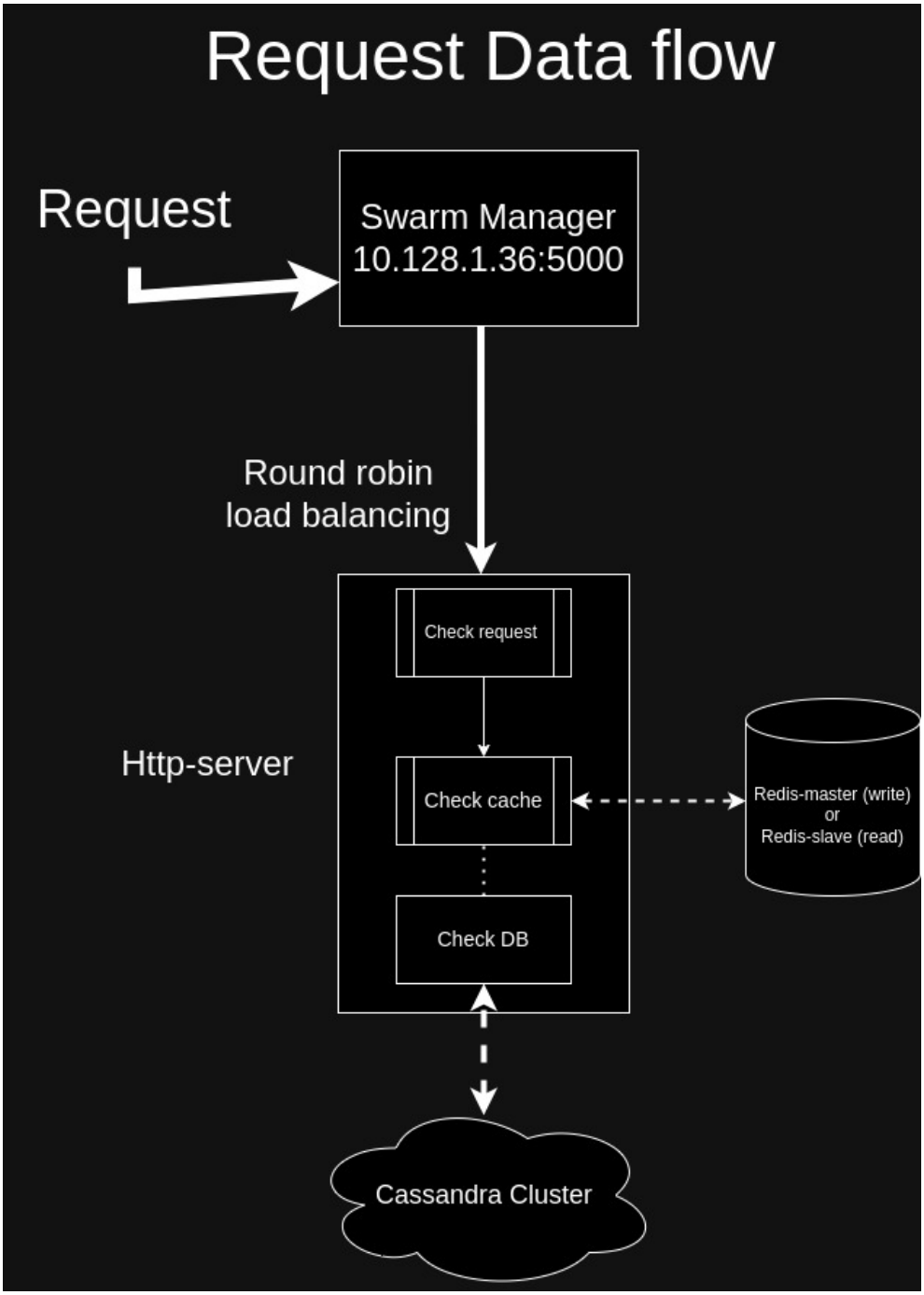
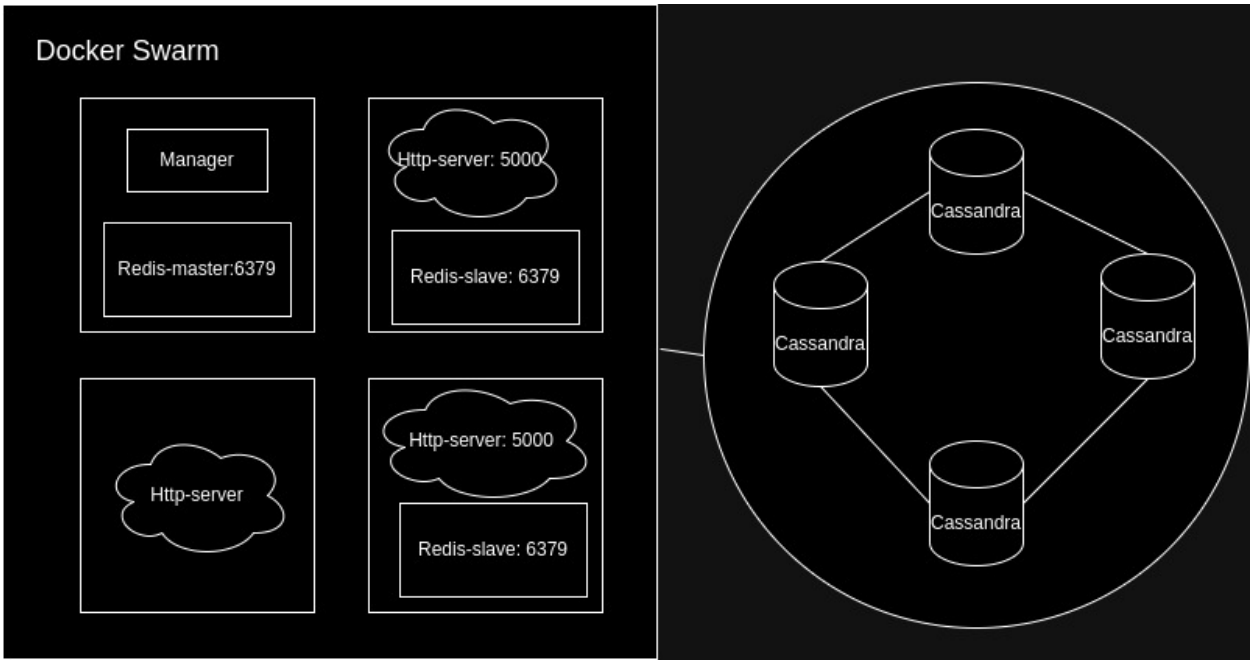
CSC409 A2 Report

duvallou, hokinmat, tabbahch
Louis-Phillippe Duval, Matthew Ho Kin, Christian Tabbah

Architecture:





- In our architecture we have multiple components, including Flask, for the URLShortener service, Redis for caching and Cassandra for the database. Firstly we are using docker swarm to host our URLShortener services, our Redis services as well as our visualizer. Our URLShortener service is replicated across every node excluding the manager. Our Redis service consists of one Redis-master, on the manager node, as well as two redis-slaves on the other nodes. Our cassandra cluster is run separately from the docker swarm, and hosts a cassandra node on every node. We are also able to host this on a separate machine to increase throughput. In cassandra, data is partitioned and replicated(with a replication factor of 3) across multiple nodes. Each node in the cassandra cluster contains a subset of the data, ensuring horizontal scalability and fault tolerance. We chose this distributed setup as it makes use of our nodes, and enhances performance, availability and fault tolerance by partitioning and replicating data across several nodes.

Proxy:

- Instead of having a reverse proxy using a tool like nginx, we decided to instead use the load balancing and proxying features of docker swarm. Docker swarm has built-in load balancing system, as it automatically distributes incoming requests among the containers in a swarm using D. It also helps us handle automatic retries, as if one of the services goes down, we are able to configure docker such that the service is immediately restarted. Finally, the swarm also handles scalability, since docker swarm can dynamically adjust the number of replicas for a service, and distribute the traffic accordingly. On the other hand, nginx does provide some additional features like fine grained load balancing(where you can choose the algorithm), URL rewriting and specified redirection, however for the purposes of the assignment, these features are not necessary to us.

Cache:

- We are caching through the use of Redis.
    - When we get a PUT request, we cache it, then we send it to the redis writer, and the writer will send that to cassandra.
    - When we get a GET request, we first check if it is in the cache. If it is, then we can simply return that to the client. Otherwise, we directly send the query to cassandra, as well as cache the GET request result to redis, with a TTL of 60 seconds. We are capable of modifying this number depending on the load as well.

- Performance improvement:
    - **This can be found in the load testing section of this report**

Load Balancing:

- We are load balancing through the use of docker swarm. Docker swarm has built-in load balancing that utilizes DNS, as it automatically distributes incoming requests among the containers running the URLShortener services. Docker Swarm also balances the load when requesting from a redis-slave, also through DNS resolution.

- Performance Improvement:
    - **This is demonstrated and explained in the load testing section of this report**

Fault Tolerance:

- Docker swarm allows you to deploy a “replica” amount of services. If one or more replicas experience failures, the remaining healthy replicas continue to handle requests.

- Docker swarm also monitors the health of nodes in the swarm. If a node fails, or becomes unreachable, the swarm reschedules the tasks running on that node to healthy nodes ensuring availability.

- Because of docker, we also have network isolation, so failures in one service won't affect the others.

- Finally, we also ensure that if a URLshortener goes down, it will restart on failure. Like this even if we have services go down, after requests being redirected to other shorteners, the services that went down will restart, so losing several nodes will not be an issue.

Healing:

- It depends. If the node in the swarm is a worker node, then yes, we can simply introduce a new node to the cluster and replace the failed node, and the orchestration will all be handled automatically by the manager node. However, if the node is the only manager node of our swarm, then our whole swarm fails and the whole cluster goes down because manager nodes are responsible for orchestration and are necessary for the functioning of a docker swarm. One way to mitigate this issue would be to introduce multiple manager nodes to the swarm such that in the event of a failure of a manager node, the whole cluster does not fail so long as there’s at least one manager node that is running on the cluster. If a worker node fails, or becomes unreachable, the swarm reschedules the tasks running on that node to healthy nodes ensuring availability.

  When a service in the swarm fails, docker swarm detects the failure and will attempt to restart those services to bring them back to a healthy state. If an entire worker node fails, the services will be orchestrated on another healthy node in the swarm. The services that go down automatically come back up due to the configuration of the swarm (see docker-compose.yml).

Monitoring System:

- Yes, the manager in the docker swarm keeps track of all services along with their health in an up-to-date manner. Each node has an agent process which reports the status of the services running on the local machine. If a server goes down, it will keep track of that fact, and if a new service goes up, it will know as well.

Monitoring UI:

- For the UI, we are using a docker visualizer tool. Here is the link to the tool:
- https://github.com/yandeu/docker-swarm-visualizer
- In the docker-compose.yml file we put in the image of the visualizer in the manager node, allowing us to monitor all nodes in a visual manner
- Yes, a human can easily state the system on their browser by going to the following link(for us): http://dh2020pc02-vm-a.taild476f.ts.net:9500/ (using tailscale, a meshed vpn solution based on wireguard)

Logging:

- Yes, we log every time we receive a request. The system events which are logged are any errors with http requests, cache hits and database fetches in the Http-server. On the writer application, all errors are logged. Both log files are stored in volumes on their respective docker containers.
- We log everytime a service starts up, hence if a service goes down, and an automatic retry occurs, that event will be logged.

Availability:

- Our system is designed to always be available. The use of the Docker Swarm gives us fault tolerance by distributing services across nodes. Cassandra's replication factor of 3 ensures data redundancy. Redis employs a master-slave architecture, giving us availability even if some nodes fail. The URL shortener is designed to gracefully handle errors, but persistent issues could arise if Redis caches reach their limits. Hence, our system aims for high availability through these design choices.

Consistency:

- If a short URL is requested shortly after an update, there is a high likelihood of receiving the latest data due to the redis cache. However, within a 5-second window, it's possible for some requests to see slightly outdated information, especially during heavy load or concurrent updates. This is due to our design choice of not choosing to have a high level of consistency(like choosing QUORUM or LVL 2), as it would affect performance heavily. Hence our design balances performance and consistency.
  Note: You can find our results on performance per level of consistency below in the load testing and performance section of this report.

- Additionally, while our system is distributed across multiple nodes, we have a cassandra data center, which has a replication factor of 3, meaning that data will essentially never be totally lost (unless a catastrophic event occurs).

Failure Testing:

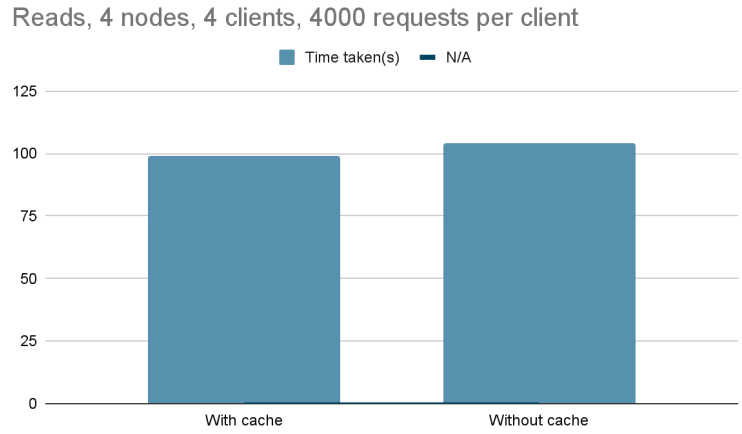
- Yes, we can. We are able to kill nodes and services explicitly, and see them get put back up through docker swarms automatic reboot mechanism.
- We have scripts to pause/restart cassandra nodes and we can demonstrate that in the presence of a cassandra node going down, we can recover from this failure because of our data partitioning and replication of 3 hosts in the cassandra cluster.

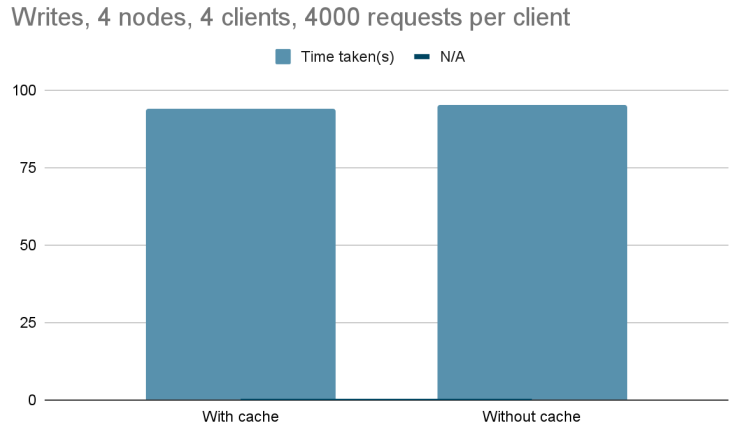
Load Testing AND Performance

- Below are our several load tests which also show our performance guarantees.

Load test 1: Testing performance via CACHE vs. NO CACHE

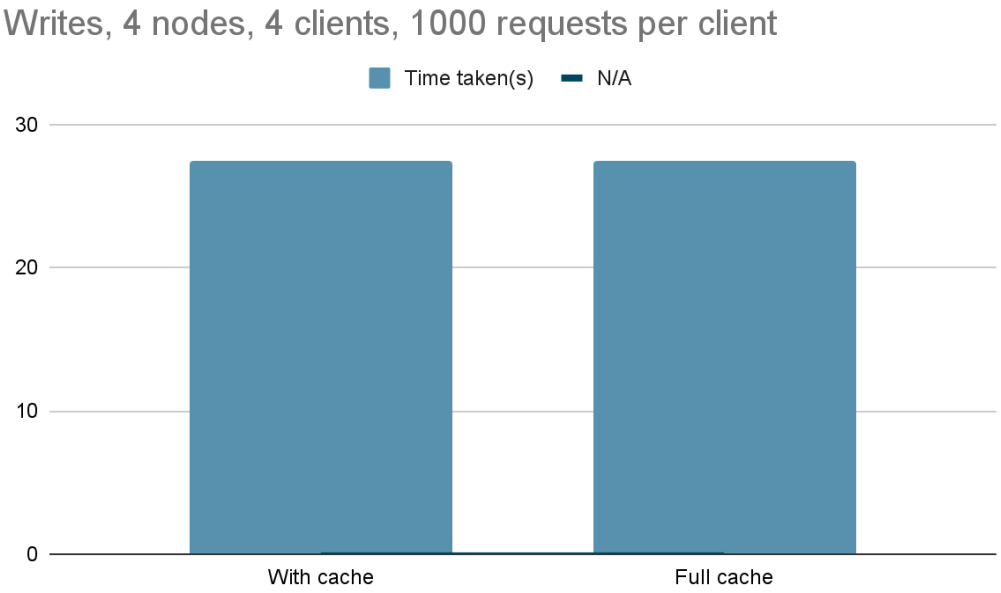
We first decided to test the difference in performance between having a cache, and not having a cache. The time that it takes to make a request in cassandra is the same as it takes in redis as cassandra has default caching mechanisms. Additionally, the latency it takes for the request to reach the redis cache is the same for it to reach the cassandra node as our network is virtualized, and we are accessing another machine on the same domain. We know this as pinging redis and cassandra from the http server has approximately the same latency (1-3ms). Hence, as we can see from the graphs below, actually having a cache does not inherently affect the latency due to the reasons mentioned above.

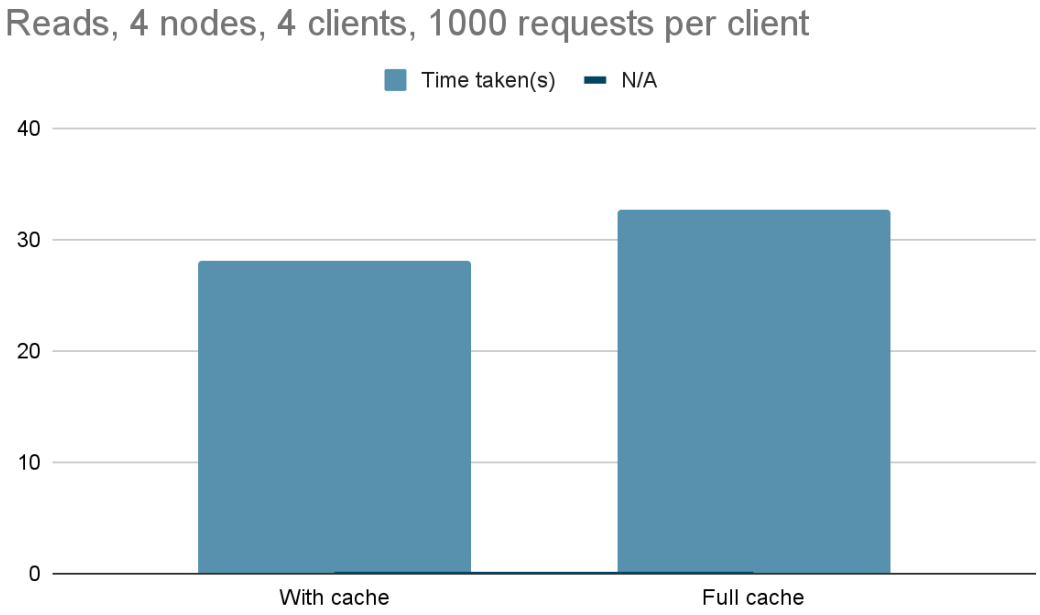




Load test 2: Testing performance via CACHE vs. FULL CACHE

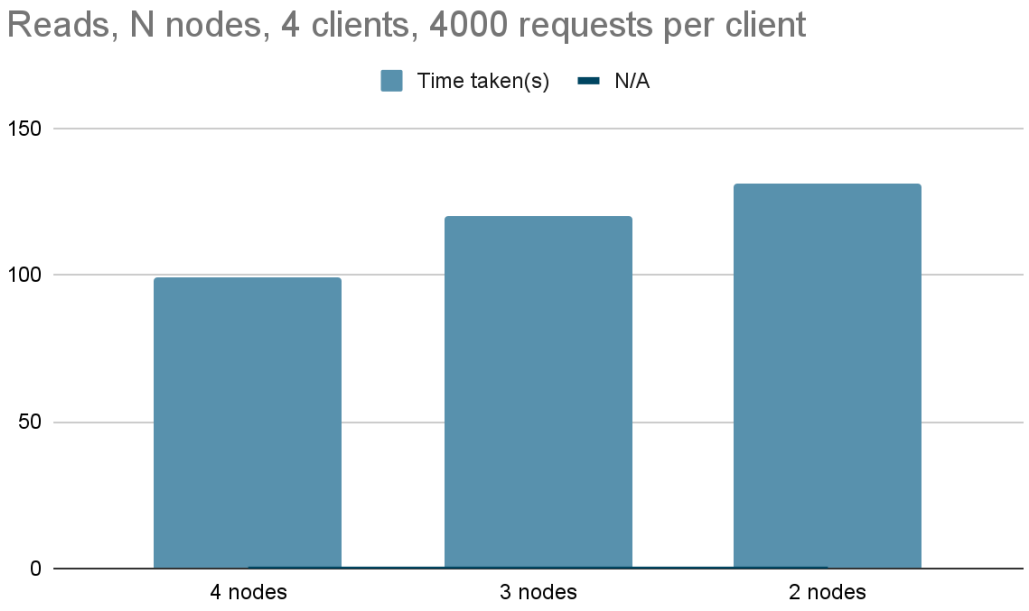
Now we will test the latency we get when we have a full cache vs. when we have a cache with space. For write requests, we should see no notable difference as we have to send the request to the writer to update the request regardless. However for reads, the eviction process in the cache gives some additional latency, making it take longer to get the longs. Here are the graphs to show this result:

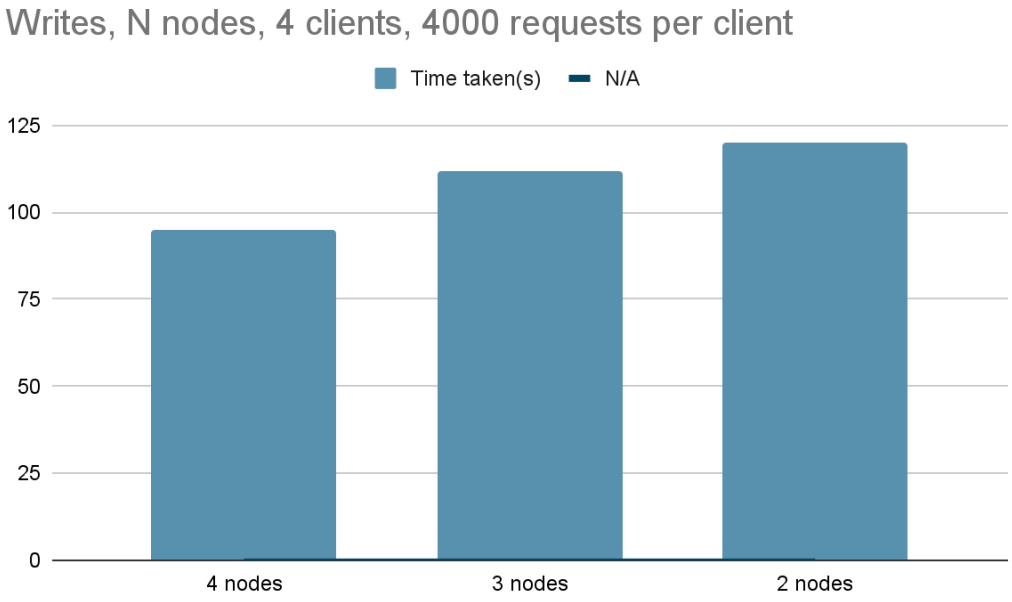




Load test 3: Testing Performance via number of nodes

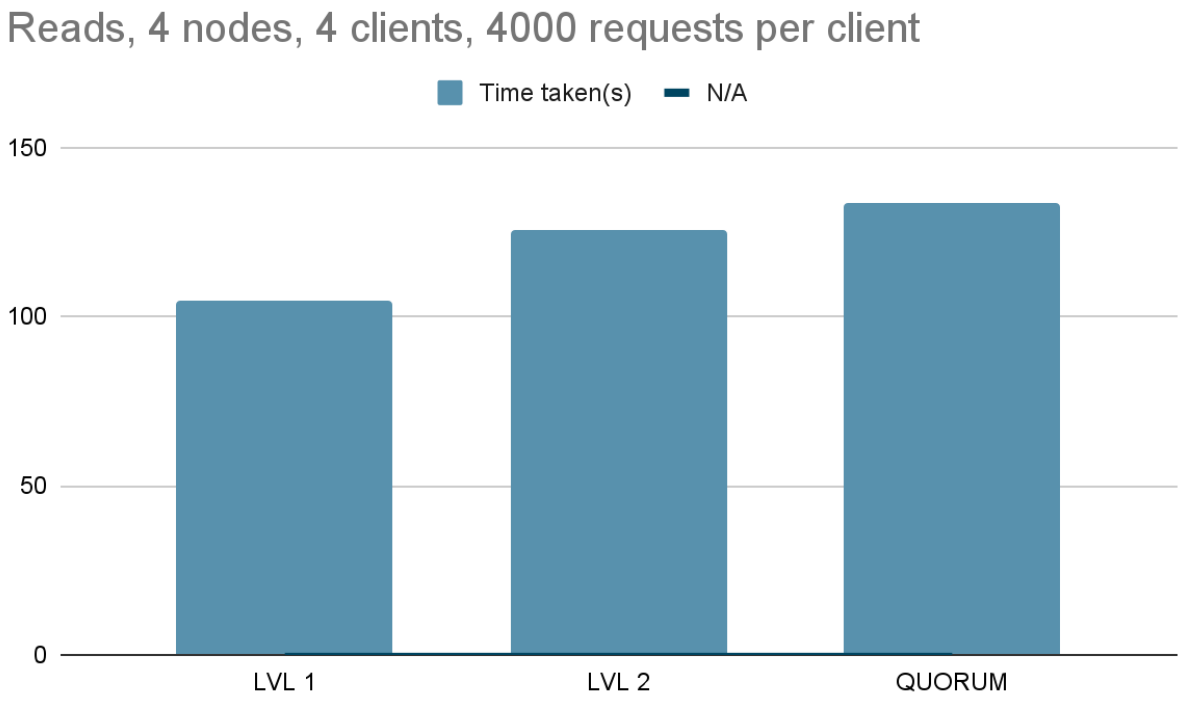
In this test we want to show how well our architecture makes use of having additional nodes. Hence, we will now test the effect of removing nodes on our performance latency. Obviously, we expect that the less nodes we use, the slower our requests will take for both reads and writes, as we utilize each node through the use of docker swarm's load balancing feature. Here are the graphs that represent this:





Load test 4: Testing Performance via consistency level

In this test we wanted to see the performance impact if we increased the consistency level. We explicitly chose to not have a high degree of consistency, as in our architecture we only chose the default of 1 level of consistency for reads. We decided on this design choice as we knew that having higher levels of consistency would have an impact on performance, as the cassandra nodes would have to communicate with each other more to agree on the data being requested, hence causing more delay. Here is a graph on the performance impact of having different consistency levels:



Bug Testing:

- While we already made a script for testing our performance, we added another short script specifically to meet this demand. Essentially it just does a few PUTs and GETs. We can then simply check our logs to see if we got any errors. Additionally, we can see from the requests what response code we get.

Data Scalability:

- Yes, In cassandra, horizontal scalability is achieved by adding more nodes to the cluster, which we have created scripts for. You can easily scale the system by adding new machines, allowing cassandra to distribute and balance the data across the nodes automatically. Hence, we can increase storage capacity and handle growing amounts of data by expanding the cluster with more resources.

CPU/Node Scalability:

- Yes, the distributed nature of Cassandra allows it to effectively utilize additional computational resources by horizontally scaling across multiple nodes. Additionally, adding more nodes will allow us to make use of having more URLShorteners, as we replicate them per node. Overall, this means that adding more nodes will allow us to have more URLShorteners, and as docker swarm load manages, the system's capacity to handle incoming requests increases.

Admin Scalability:

- It is very easy to make new ressources a part of our system. We have created scripts to add and remove nodes from the cassandra cluster, as well as reset the cassandra nodes. We are also able to easily add nodes to the swarm by getting the token and calling the join command on the new node. Additionally, docker swarm will make use of the node immediately by starting all necessary services on it.

Orchestration:

- It is very easy to stop and start the system. To start the docker swarm services, there are a few scripts provided to build, deploy, and remove the application stack. It is also seamless to start/stop (startCluster.sh/stopCluster.sh) and add/remove (addCassandraNodes.sh/decomissionCassandraNode.sh)

cassandra nodes from a cassandra cluster. We also provide scripts to pause/restart (pauseCassandraNode.sh/restartCassandraNode.sh) a cassandra node. Lastly, we have a script which easily runs ‘nodetool-status.sh’.
See /cassandra for all the Cassandra orchestration scripts.

- Yes, they can be found in /docker and /cassandra. The names of the scripts explain their usage. Typically one would do, docker-build-push-all.sh > docker-stack-deploy.sh, to fully initialize the service.