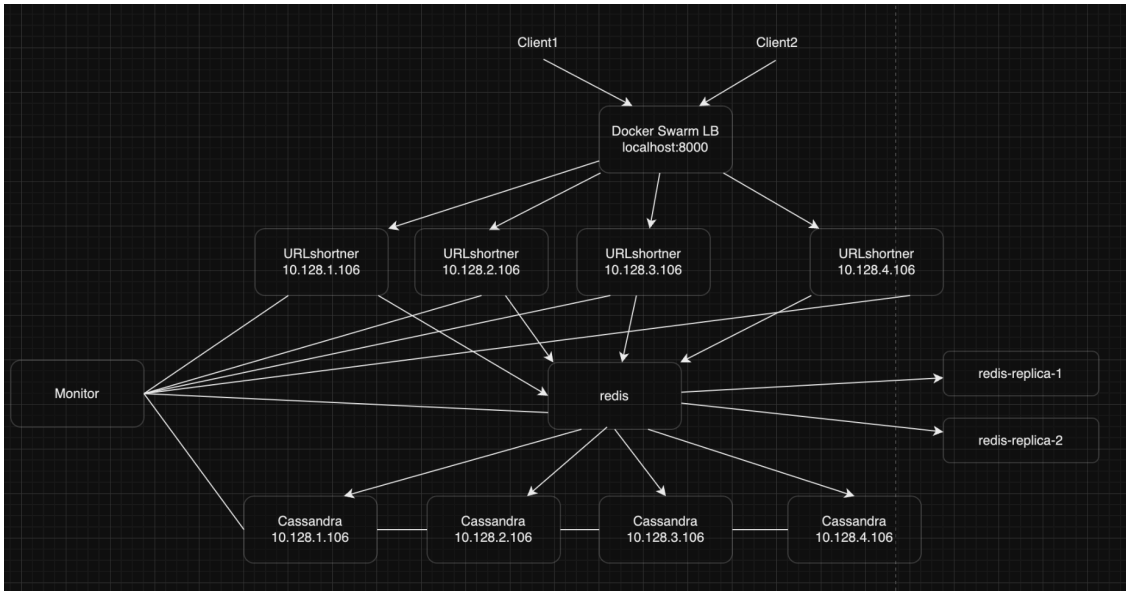
CSC409 A2 - group45 - yangx234, yaolian1

System Diagram:



Overall Structure:

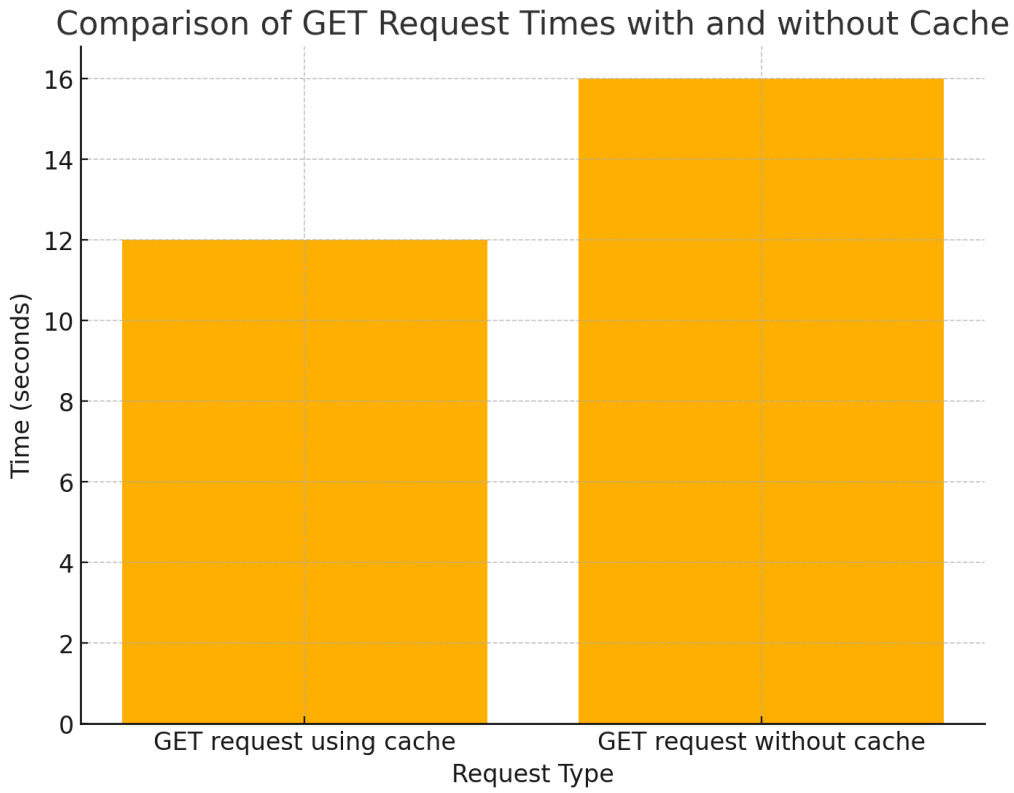
We used a variety of services in our system, such as Python, Flask, Cassandra, Redis, Gunicorn and Docker. Our application, urlshortner, is achieved with Python and Flask. Cassandra is what we used for our database, and we used Redis for caching, since data retrieval is much faster with Redis than with Cassandra. We also included Gunicorn in our implementation to help with processing more than one request, since one single Flask instance can only handle one request at a time.

Database:

Our core database, the keyspace and the table are implemented using Cassandra. We have 4 hosts, each loaded with Cassandra. Our keyspace's replication factor is 2, meaning that each piece of data will have two copies in our system. In case one host fails, we can still retrieve the data from the other host. Since we only have 4 hosts in total, we believe that 2 copies would be sufficient for fault tolerance.

Caching:

We use Redis to handle caching. When the system receives a GET request, it would first check if the data exists in Redis. If it does, then the request would never reach Cassandra. For PUT requests, if the data exists in Redis, the old data would get updated first, then Cassadra would also get updated. Below is a graph showing the time difference between 20k GET requests.



Load balancing:

Load balancing is achieved through Docker Swarm's routing mesh and Docker's load balancing features. Incoming requests are distributed evenly across multiple instances of the Flask application and Cassandra nodes.

Recovery:

We have a default recovery among Docker Swarm and Cassandra. If a node for any reason fails, it would get restarted automatically. Cassandra would also replicate data to the newly started host to ensure that it has the most up-to-date data.

Logging:

Recovery logging is implemented by the default Docker logging. We log all other system events ourselves. We have one urlshortner container per VM, and we have a specific directory in each container to store the logged results (GET and PUT requests), and we synchronize all of those logs in each container to each of their corresponding locations in their VM.

Monitoring:

We implement monitoring using a container with Flask, which we call an endpoint. It retrieves the current state of the application system. Here is a sample output from the monitor service:

URL Shortener System Status

Node 0 Status
Cassandra 10.128.1.106 UP About an hour ago Up About an hour
URL Shortener 10.128.1.106 UP 2 minutes ago Up 2 minutes
Redis 10.128.1.106 UP 2 minutes ago Up 2 minutes
Node 1 Status
Cassandra 10.128.2.106 UP About an hour ago Up About an hour
URL Shortener 10.128.2.106 UP 2 minutes ago Up 2 minutes
Redis 10.128.2.106 DOWN
Node 2 Status
Cassandra 10.128.3.106 UP About an hour ago Up About an hour
URL Shortener 10.128.3.106 UP 2 minutes ago Up 2 minutes
Redis 10.128.3.106 DOWN
Node 3 Status
Cassandra 10.128.4.106 UP About an hour ago Up About an hour
URL Shortener 10.128.4.106 UP 2 minutes ago Up 2 minutes
Redis 10.128.4.106 DOWN

Data Partition Tolerance:

Cassandra handles partition tolerance by replicating data, allowing for tunable consistency levels, and employing mechanisms to reconcile inconsistencies after partitions are resolved.

Vertical Scalability:

We can utilize the start-stack function, which is implemented using docker commands, in our orchestration.sh to add potential functional layers to enhance the system.

Admin Scalability:

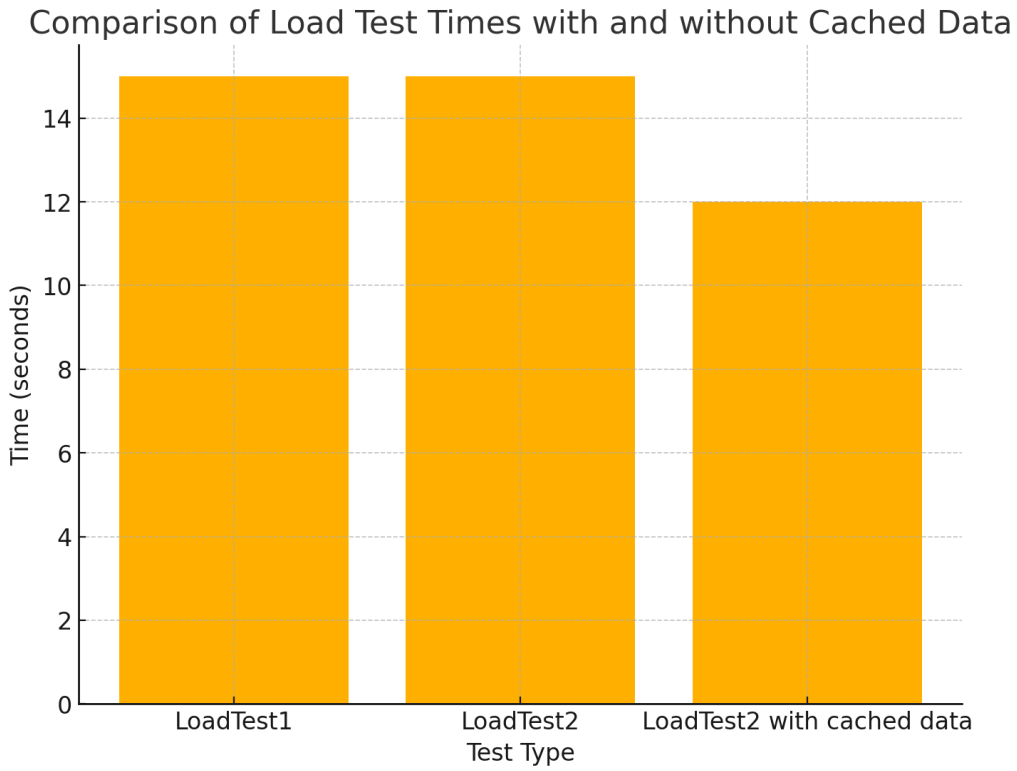
New resources can be added with minimal configuration. Docker Swarm and Cassandra handle the orchestration and data distribution automatically.

Orchestration:

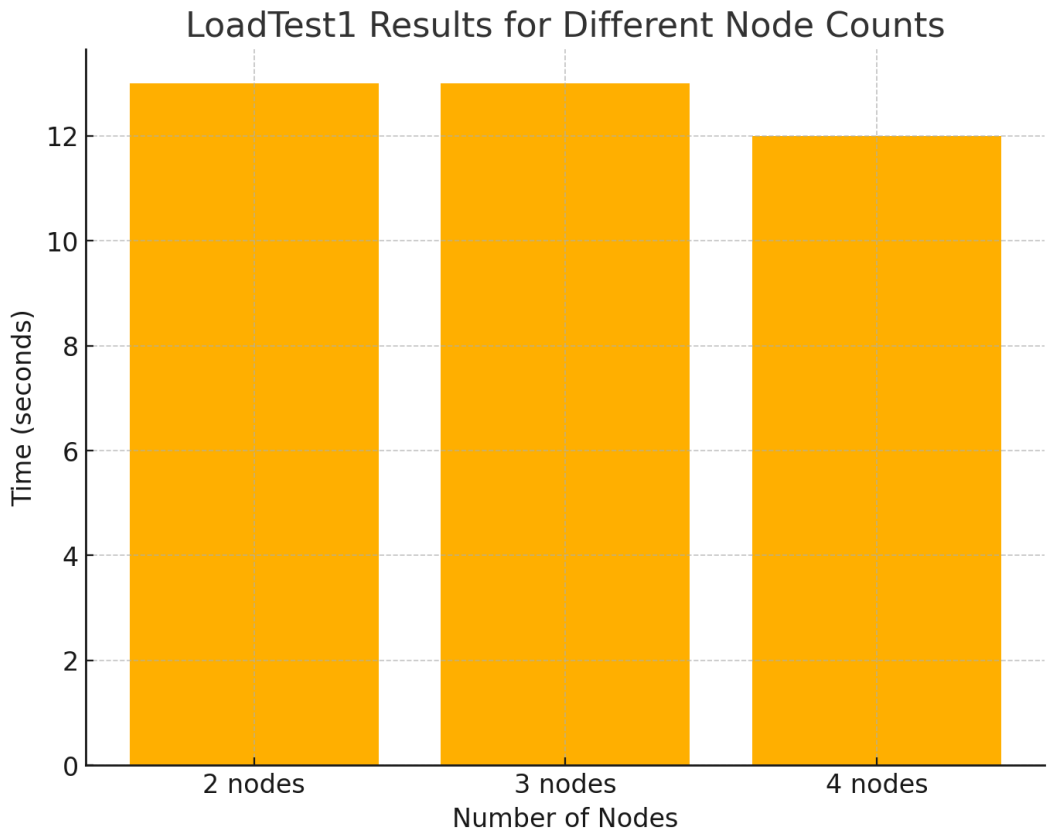
Our orchestration shell script serves as the conductor of our system. In addition to starting and stopping our workflows, it is also capable of adding and removing new nodes. It is capable of scaling horizontally. All of its functionalities are listed below:

```
case "$1" in
    start)    start-cass; start-swarm; start-stack;;
    stop)     stop-stack; stop-cass; stop-swarm;;
    start-cass) start-cass;;
    stop-cass) stop-cass;;
    start-swarm) start-swarm;;
    stop-swarm) stop-swarm;;
    start-stack) start-stack;;
    stop-stack) stop-stack;;
    add-node)    add-node "${@:2}";;
    remove-node) remove-node "${@:2}";;
esac
```

Performance:



The performance of GET and PUT requests are about the same for LoadTest 1 and 2. However, LoadTest2 with cached data is significantly faster since data retrieval from Redis is faster.



Above is a diagram showing the LoadTest1 result using 2, 3 and 4 nodes.