Evaluation Doc

Run 5 clients on your local machine. Measure the latency seen by each client for different type requests. Change the probability p of a follow up purchase request from 0 to 80%, with an increment of 20%, and record the result for each p setting. Make simple plots showing the values of p on the X-axis and the latency of different types of request on the y-axis. Also do the same experiments but with caching turned off, estimate how much benefits does caching provide by comparing the results.

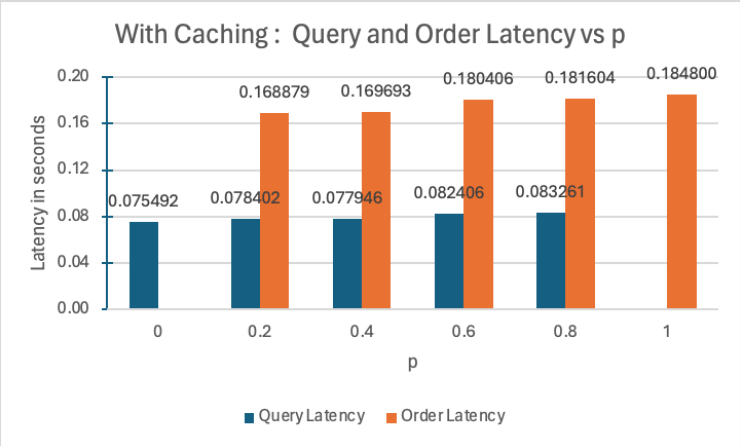
Exhaustive list of latency observed for query type and order type requests for 5 clients. The below values are for each query/order request found by finding the average by sending requests from 20 requests each from 5 clients concurrently.

	With Caching		Without Caching	
	Query latency (in s)	Order latency (in s)	Query latency (in s)	Order latency (in s)
p=0	0.073631108	0.0000000000000000	0.081780565	0.0000000000000000
	0.07515887	0.0000000000000000	0.081780744	0.0000000000000000
	0.075315523	0.0000000000000000	0.081977928	0.0000000000000000
	0.076190364	0.0000000000000000	0.081977749	0.0000000000000000
	0.07716558	0.0000000000000000	0.081994081	0.0000000000000000
Average	**0.0754922890600000**	**0.0000000000000000**	**0.0819022131000000**	**0.0000000000000000**
p=0.2	0.077089056	0.162109494	0.081783985	0.174446285
	0.079065993	0.16519177	0.083048314	0.168288708
	0.081454694	0.16502279	0.080162704	0.184055269
	0.079914793	0.165761948	0.083559677	0.180149257
	0.074487492	0.186311066	0.082432121	0.186619043
Average	**0.0784024059800000**	**0.1688794136000000**	**0.0821973601600000**	**0.1787117123800000**
p=0.4	0.0783547805000000	0.1669656549000000	0.0824590921000000	0.1834296286000000
	0.0759830674000000	0.1644881368000000	0.0827781161000000	0.1841931343078610
	0.0789847175000000	0.1732487381000000	0.0804781715000000	0.1793672144000000
	0.0792522430000000	0.1721877456000000	0.0831579367000000	0.1809922254000000
	0.0771528780000000	0.1715734005000000	0.0829825004000000	0.1801206172000000
Average	**0.0779455372800000**	**0.1696927351800000**	**0.0823711633600000**	**0.1816205639815720**
p=0.6	0.0776163472000000	0.1811812574000000	0.0837524235000000	0.1833530186000000
	0.0874659717000000	0.1758310994000000	0.0894112885000000	0.1821032284000000
	0.0801984999000000	0.1848504327000000	0.0789058506000000	0.1893003130000000
	0.0841222074000000	0.1824065772000000	0.0897875130000000	0.1863121776580810
	0.0826267302000000	0.1777589719000000	0.0826627612000000	0.1881412609000000
Average	**0.0824059512800000**	**0.1804056677200000**	**0.0849039673600000**	**0.1858419997116160**
p=0.8	0.0810780048000000	0.1811994531000000	0.0836060047000000	0.1899714118000000
	0.0846138573000000	0.1774730086000000	0.0952307582000000	0.1885766983032220
	0.0773153305000000	0.1829216480000000	0.0940433741000000	0.1834118366241450
	0.0842600431000000	0.1845607209000000	0.0836060047000000	0.1861200332641600
	0.0890360475000000	0.1818627119000000	0.0855371952000000	0.1900589466094970
Average	**0.0832606566400000**	**0.1816035085000000**	**0.0884046673800000**	**0.1876277853202050**
p=1	0.0000000000000000	0.1816719747000000	0.0000000000000000	0.1901948451995840
	0.0000000000000000	0.1849879503000000	0.0000000000000000	0.1851592063903800
	0.0000000000000000	0.1855507708000000	0.0000000000000000	0.1855301856994620
	0.0000000000000000	0.1852005696000000	0.0000000000000000	0.1937782764434810
	0.0000000000000000	0.1865879039764400	0.0000000000000000	0.1915509700775140
Average	**0.0000000000000000**	**0.1847998338752880**	**0.0000000000000000**	**0.1892426967620840**

We can visualize the observations better with some simple plots:

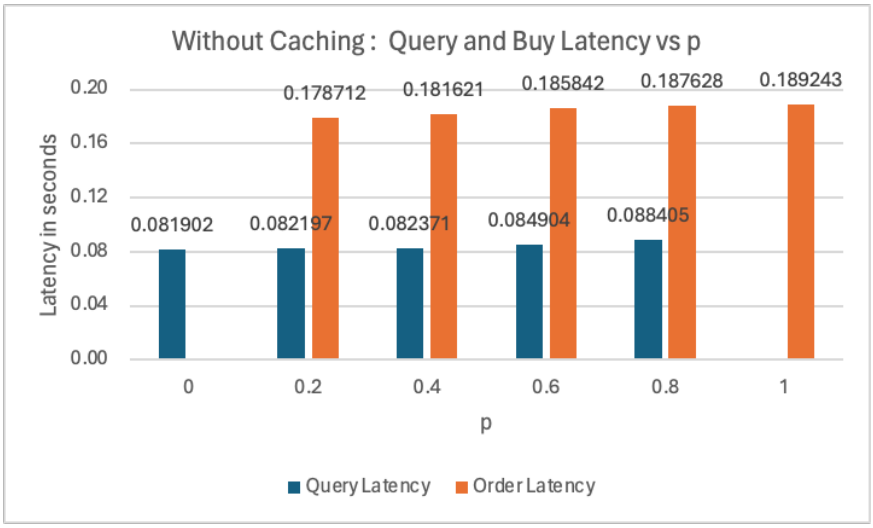
Comparison between Query and Order Request Latency (in seconds) for different values of p, **with Caching**:

p	Query Latency (in s)	Order Latency (in s)
0	**0.0754922890600000**	**0.0000000000000000**
0.2	**0.0784024059800000**	**0.1688794136000000**
0.4	**0.0779455372800000**	**0.1696927351800000**
0.6	**0.0824059512800000**	**0.1804056677200000**
0.8	**0.0832606566400000**	**0.1816035085000000**
1	**0.0000000000000000**	**0.1847998338752880**



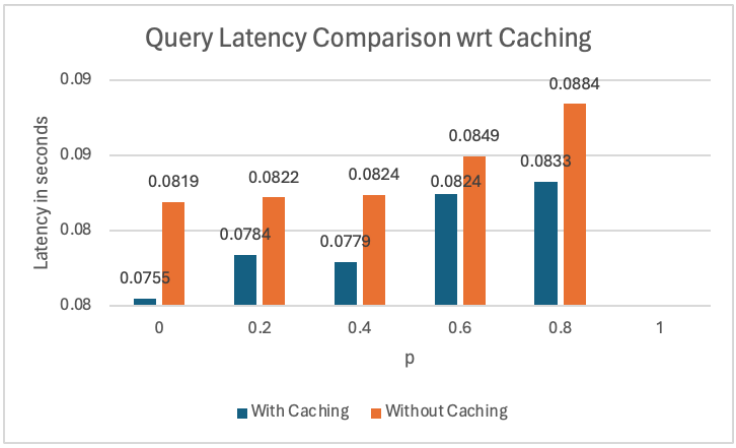
Comparison between Query and Order Request Latency (in seconds) for different values of p, **without Caching**:

p	Query Latency (in s)	Order Latency (in s)
0	**0.0819022131000000**	**0.0000000000000000**
0.2	**0.0821973601600000**	**0.1787117123800000**
0.4	**0.0823711633600000**	**0.1816205639815720**
0.6	**0.0849039673600000**	**0.1858419997116160**
0.8	**0.0884046673800000**	**0.1876277853202050**
1	**0.0000000000000000**	**0.1892426967620840**



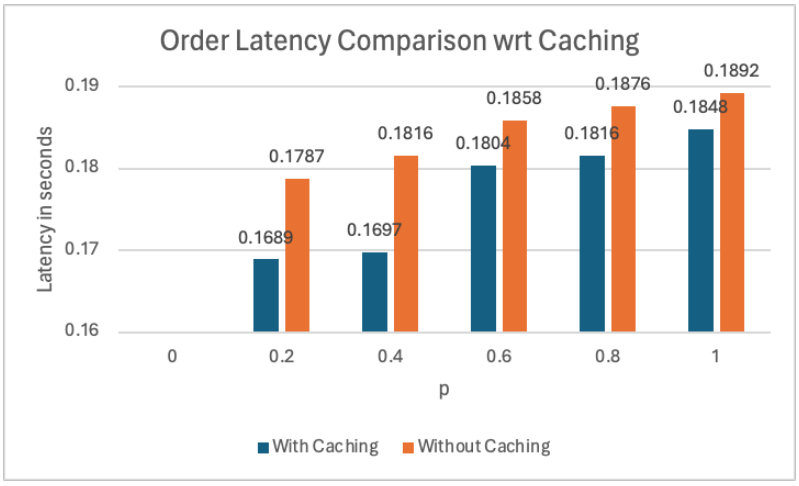
Comparison of how caching benefits Query Latency (in seconds) for different values of p:

	Latency in seconds		
p	**With Caching**	**Without Caching**	**% benefit**
0	0.0754922890600000	0.0819022131000000	7.8263136
0.2	0.0784024059800000	0.0821973601600000	4.61688085
0.4	0.0779455372800000	0.0823711633600000	5.37278569
0.6	0.0824059512800000	0.0849039673600000	2.94216649
0.8	0.0832606566400000	0.0884046673800000	5.81870946
1	0.0000000000000000	0.0000000000000000	
Average benefit			5.31537122



Comparison of how caching benefits Order Latency (in seconds) for different values of p:

	Latency in seconds		
p	**With Caching**	**Without Caching**	**% benefit**
0	0.0000000000000000	0.0000000000000000	0
0.2	0.1688794136000000	0.1787117123800000	5.501765189
0.4	0.1696927351800000	0.1816205639815720	6.567443983
0.6	0.1804056677200000	0.1858419997116160	2.925244025
0.8	0.1816035085000000	0.1876277853202050	3.210759435
1	0.1847998338752880	0.1892426967620840	2.347706391
Average benefit			4.110583805



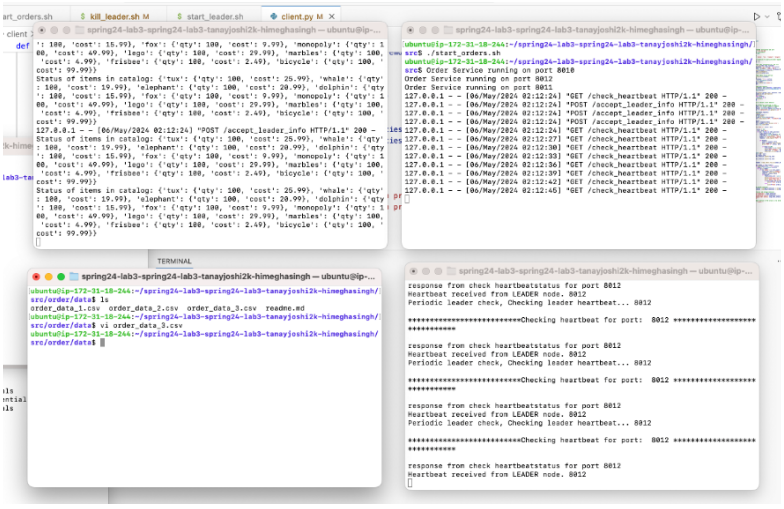
It can be observed that latency observed per query and order request is noticeably lower when caching is implemented.

In case of a query request, there is an approximate 5% benefit, whereas in the case of order request, there is an approximate 4% benefit. This benefit is because we are using caching, in the form of least recently used cache, where the server first checks the in-memory cache to see whether it can be served from the cache. If found in the cache, the server can quickly respond without needing to fetch the data from the underlying service, resulting in reduced response times.
The above observations are a result of 20 requests sent per client, with a total of 5 clients, sending order requests with probability in increments of 0.2. As the application scales or load on the system increases, caching is expected to provide even more significant benefits in terms of reducing latency. The probability of cache hits (requests for data that are already present in the cache) would tend to increase, further amplifying the benefits of caching.

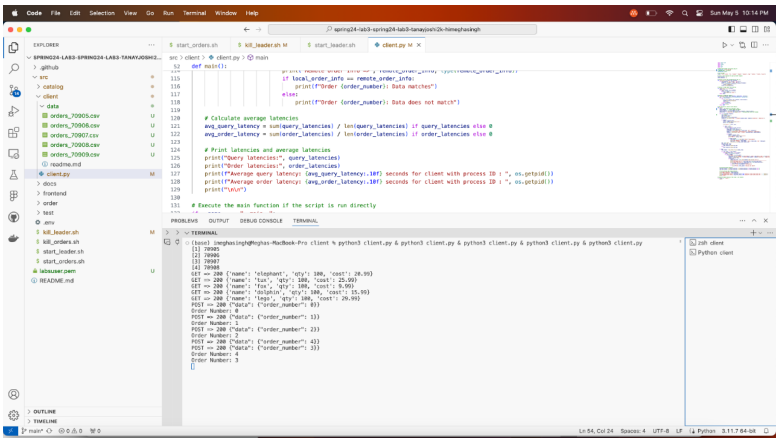
Finally, simulate crash failures by killing a random order service replica while the clients is running, and then bring it back online after some time. Repeat this experiment several times and make sure that you test the case when the leader is killed.

To simulate this, I started a 5 concurrent clients, and I have 3 order replicas as defined below with their respective priorities and ports:
1: 8010
2: 8011
3: 8012
As I choose the replica with the highest ID as the leader, Replica 3 on port 8012 is elected as the leader.

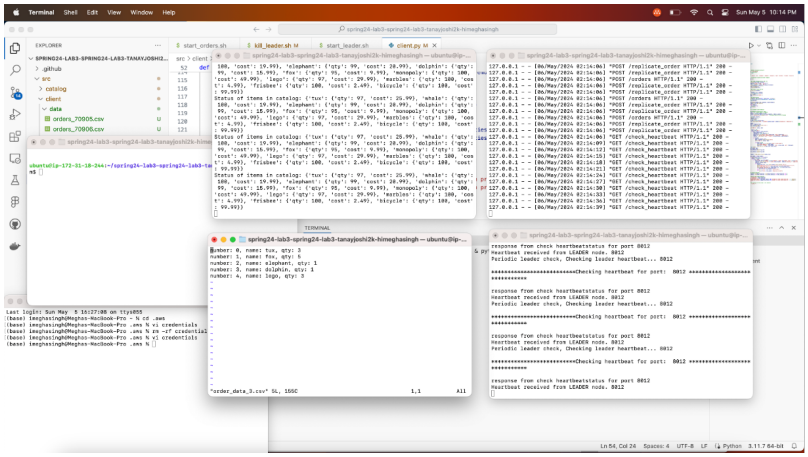
1. **Services start and port 8012 is elected as the leader:**



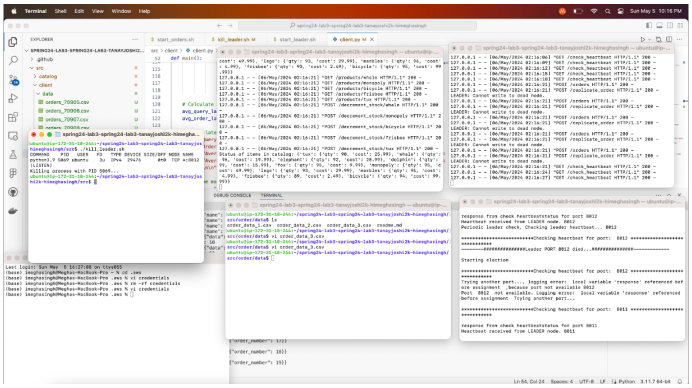
2. Orders start getting sent and an orders csv file is created for each client:



3. **Here, we can see that 8012 is the leader and its corresponding file order_data_3.csv is getting updated:**

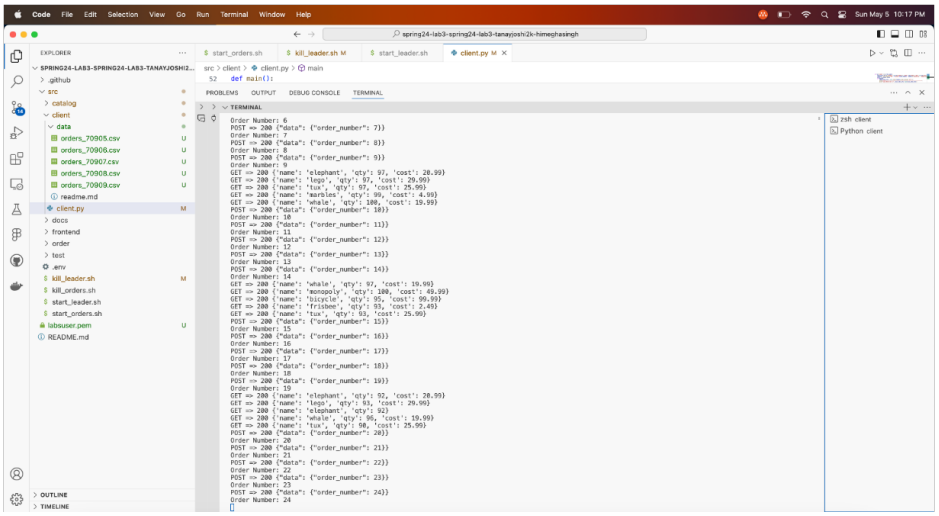


4. **After the clients have sent a few requests, I killed the Replica 3 (8012) using my bash script kill_leader.sh:**



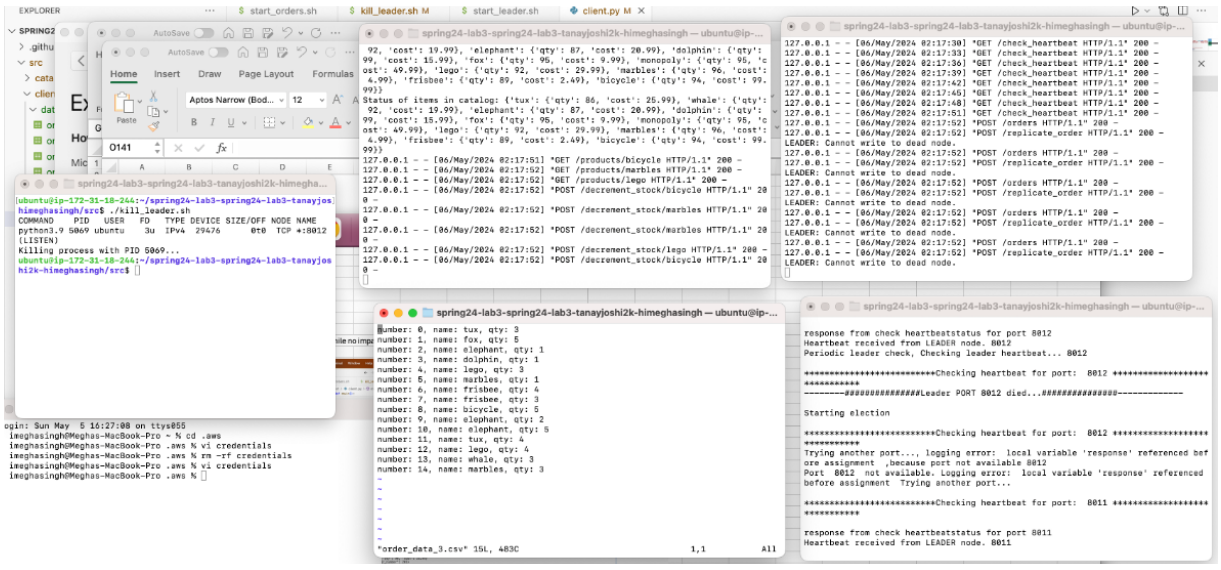
It can be seen in the order and frontend logs that these services realize that they are no longer finding a heartbeat from the Leader which is Replica 3 on port 8012, when performing the periodic heartbeat check. Hence, it starts a re-election and now with Replica 3 crashed, it has options between Replica 1 and Replica 2. Therefore, Replica 2 on port 8011 gets elected as the leader, and the execution on all services continue gracefully.
We should also note that when the new leader Replica 2 on port 8011 tries to propagate its order info to the other replicas, which includes the killed node, the leader realizes that there is no heartbeat from this replica and hence logs “Cannot write to dead node.”

6. Meanwhile, there is no impact on the client:

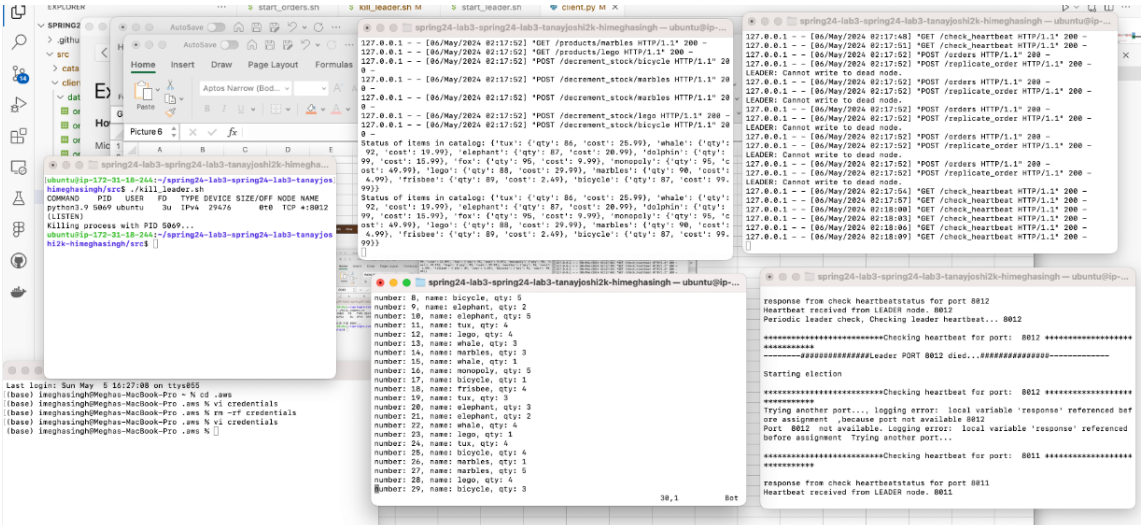


7. Before bringing replica 3 on port 8012 alive, lets compare the csv files on it and the current leader on 8011:

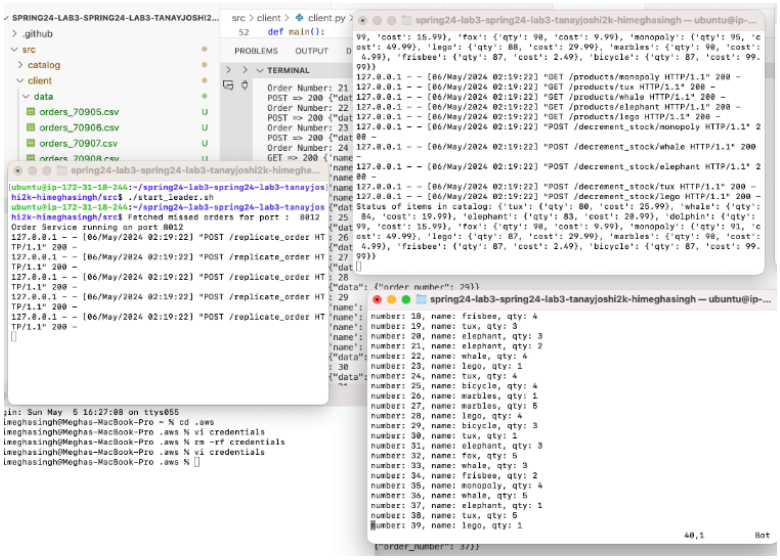
On order_data_3.csv, there are 14 records:



Whereas on order_data_2.csv, there are 29 records:

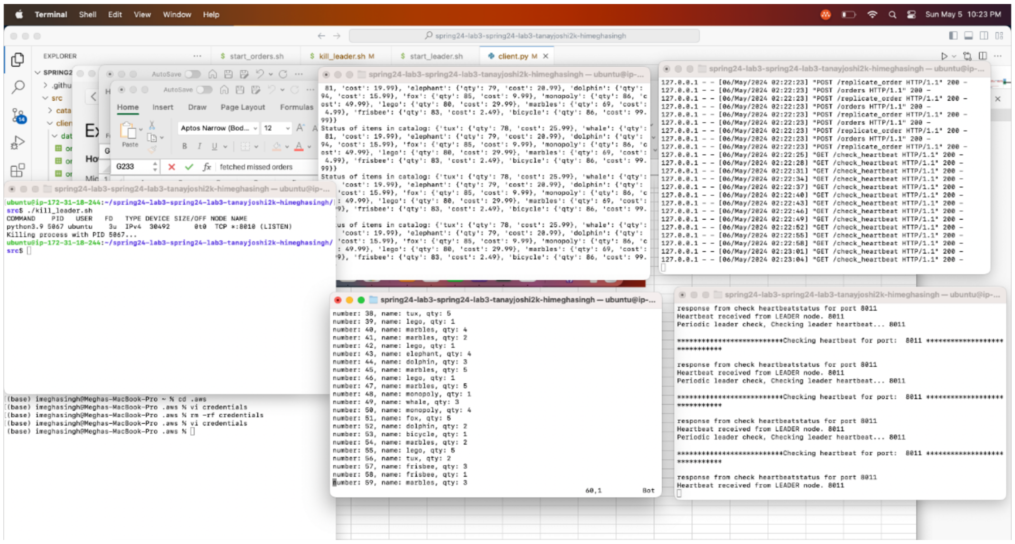


8. Now, we shall bring Replica 3 back alive on port 8012 and notice if it fetches the missed orders:



Above, we see that as soon as the replica 3 comes back alive, it fetches the missed orders, and gets all orders it missed and the latest order it has is order number 39, which is the latest order in real-time.

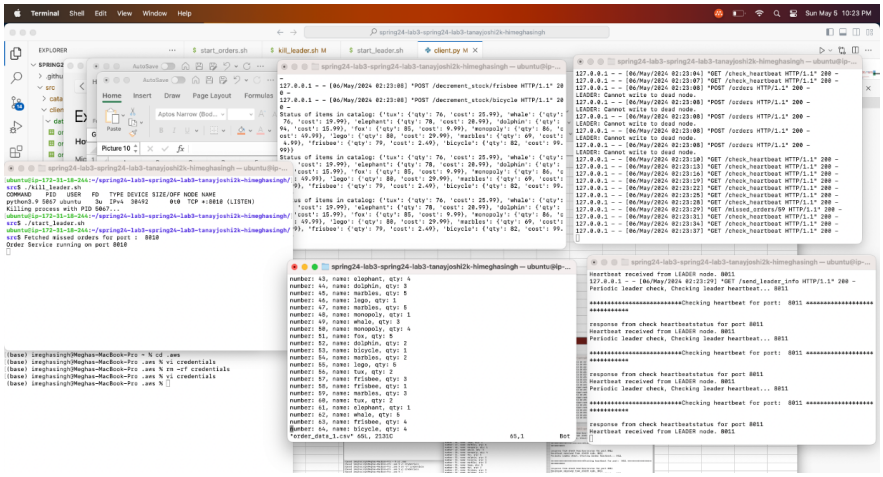
9. Similarly, if we try to kill a random replica(non-leader, say Replica 1 on port 8010, for testing purpose I modified the port number in kill_leader.sh to reuse the same file for killing replica 1):



10. We see that it also has no impact on the service, only the leader would output Cannot write to dead node when trying to replicate information to this Replica 1. Besides that, the client remains unaffected as before.

```
127.0.0.1 - - [06/May/2024 02:22:43] "GET /check_heartbeat HTTP/1.1" 200 -
127.0.0.1 - - [06/May/2024 02:22:46] "GET /check_heartbeat HTTP/1.1" 200 -
127.0.0.1 - - [06/May/2024 02:22:49] "GET /check_heartbeat HTTP/1.1" 200 -
127.0.0.1 - - [06/May/2024 02:22:52] "GET /check_heartbeat HTTP/1.1" 200 -
127.0.0.1 - - [06/May/2024 02:22:55] "GET /check_heartbeat HTTP/1.1" 200 -
127.0.0.1 - - [06/May/2024 02:22:58] "GET /check_heartbeat HTTP/1.1" 200 -
127.0.0.1 - - [06/May/2024 02:23:01] "GET /check_heartbeat HTTP/1.1" 200 -
127.0.0.1 - - [06/May/2024 02:23:04] "GET /check_heartbeat HTTP/1.1" 200 -
127.0.0.1 - - [06/May/2024 02:23:07] "GET /check_heartbeat HTTP/1.1" 200 -
127.0.0.1 - - [06/May/2024 02:23:08] "POST /orders HTTP/1.1" 200 -
LEADER: Cannot write to dead node.
127.0.0.1 - - [06/May/2024 02:23:08] "POST /orders HTTP/1.1" 200 -
LEADER: Cannot write to dead node.
127.0.0.1 - - [06/May/2024 02:23:08] "POST /orders HTTP/1.1" 200 -
LEADER: Cannot write to dead node.
127.0.0.1 - - [06/May/2024 02:23:08] "POST /orders HTTP/1.1" 200 -
LEADER: Cannot write to dead node.
127.0.0.1 - - [06/May/2024 02:23:08] "POST /orders HTTP/1.1" 200 -
LEADER: Cannot write to dead node.
127.0.0.1 - - [06/May/2024 02:23:10] "GET /check_heartbeat HTTP/1.1" 200 -
127.0.0.1 - - [06/May/2024 02:23:13] "GET /check_heartbeat HTTP/1.1" 200 -
127.0.0.1 - - [06/May/2024 02:23:16] "GET /check_heartbeat HTTP/1.1" 200 -
127.0.0.1 - - [06/May/2024 02:23:19] "GET /check_heartbeat HTTP/1.1" 200 -
```

11. When we bring this replica back alive, it also fetches the missed orders as we can see below:



Can the clients notice the failures (either during order requests or the final order checking phase) or are they transparent to the clients?

The failure in the order service (when a replica is killed) is transparent to the client, that is, the client does not notice the failures either during the order requests or during order checking out phase. Even during leader crashes and re-elections, client remains unaware of any disruptions. To achieve this, I've implemented a retry mechanism in the client and added optimized error handling in the server code. When a failure occurs, such as when a replica is killed, the client automatically retries sending the request every 2 seconds, for a maximum of three attempts. This ensures that even if a failure coincides with a client request, the system gracefully handles it without impacting the user experience.

Do all the order service replicas end up with the same database file?

Yes, here the database file is the csv file which each replica maintains. For instance, when Replica 2, operating on port 8011, is still functioning as the leader, and we restore Replica 3 on port 8012, it retrieves any missed orders from the current leader (Replica 2 on port 8011). To accomplish this, Replica 3 compares the last order number in its own CSV file with that of the leader's CSV file. Subsequently, it copies any missed orders from the leader's file. Finally, all order service replicas end up with the same database file.

How you deployed your application on AWS:

I first started a Lab session on the AWS Learner Academy, then downloaded the .pem file to the working dir and gave it appropriate permissions using *chmod 400 labuser.pem*.
I then copied the output of AWS CLI containing key and token to $HOME/.aws/credentials.
Next, I configured my AWS settings using aws configure.

Next, to start my instance:
aws ec2 run-instances --image-id ami-0d73480446600f555 --instance-type m5d.large --key-name vockey > instance.json

Then, I got my instance ID from the instance.json and went on to find the public DNS of my instance using:
aws ec2 describe-instances --instance-id i-065365bccd1ede37b

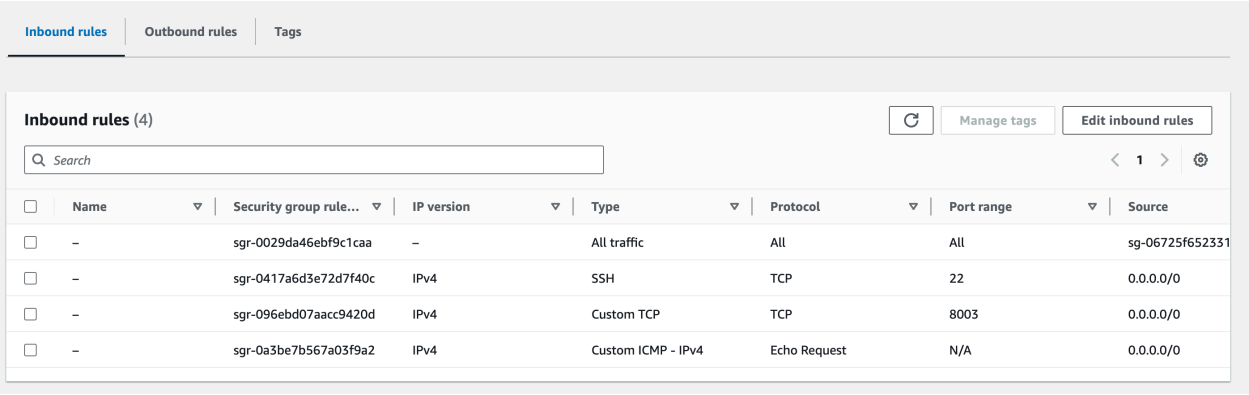
I found a DNS, for example: *ec2-52-91-221-233.compute-1.amazonaws.com*

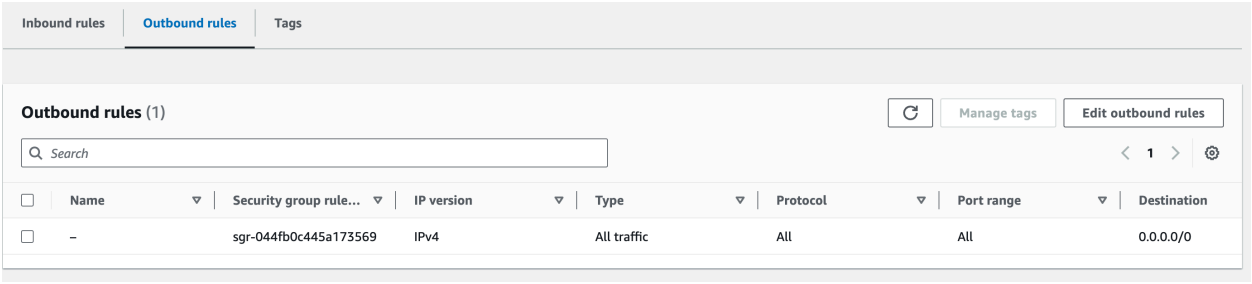
Moving on to providing appropriate network rules, I did the following:

1. Authorize port 22 (used by ssh) in the default security group to allow ssh access from anywhere using: *aws ec2 authorize-security-group-ingress --group-name default --protocol tcp --port 22 --cidr 0.0.0.0/0*
2. I also added 2 additional rules, first one is to authorize port 8003 which is the port used to connect with the frontend service. Second rule is to allow ping testing and network monitoring.
3. Other than that, I am allowing all kinds of traffic inbound and outbound.

☐	–	sgr-096ebd07aacc9420d	IPv4	Custom TCP	TCP	8003	0.0.0.0/0
☐	–	sgr-0a3be7b567a03f9a2	IPv4	Custom ICMP - IPv4	Echo Request	N/A	0.0.0.0/0

Overall, this is how inbound and outbound rules should look like:





Now, the instance is ready to be connected via ssh using:
ssh -i labsuser.pem ubuntu@ec2-52-91-221-233.compute-1.amazonaws.com