Amazon Web Service deployment steps:

I started by copying the credentials and saving them to .aws/credentials.

After the credentials, I created the EC2 instance using the Ubuntu 18.04 server image. I made the instance type m5a.large (since xlarge was not allowed). I used the following command:

```
aws ec2 run-instances --image-id ami-0d73480446600f555 --instance-type m5a.large --key-name
vockey > instance.json
```

This gave me an instance ID.

Using the following command, I retrieved the public DNS name to access the instance:

```
aws ec2 describe-instances --instance-id {instance ID}
```

I then adjusted the PEM key permissions:

```
chmod 400 labsuser.pem
```

Then authorized port 22 for ssh:

```
aws ec2 authorize-security-group-ingress --group-name default --protocol tcp --port 22 --cidr
0.0.0.0/0
```

Then authorized port 1234 for HTTP so that the frontend service can be reached from the outside client:

```
aws ec2 authorize-security-group-ingress --group-name default --protocol http --port 1234 --cidr
0.0.0.0/0
```

Then SSH connect to the client using the public DNS name:

```
ssh -i labsuser.pem ubuntu@ec2-3-227-253-129.compute-1.amazonws.com
```

I zipped the frontend, order, catalog, database, and order log files and sent them to the AWS server using scp:

```
scp storefiles.zip ubuntu@ec2-3-227-253-129.compute-1.amazonws.com:/store
```

Now all the files needed to run the store on the AWS instance were ready and the store could be run.

Evaluation Questions:

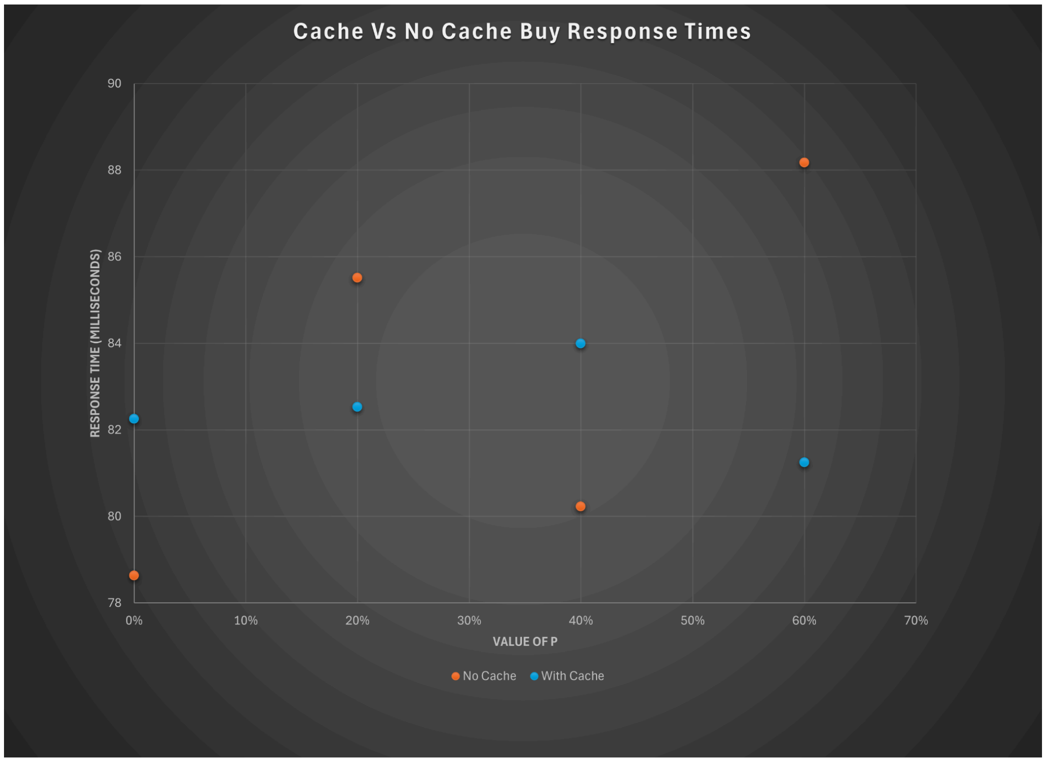
Can the clients notice the failures (either during order requests or the final order checking phase) or are they transparent to the clients?

Clients cannot notice the failures on the server side. Placing orders while the leader order replica is down is handled on the server side and the client is unable to see that anything has occurred. The only thing the client may notice is a very slight delay in the request being filled since when the request comes in, the front end has to perform the leader election again and find an order service that is online. Killing the leader replica then making an order check for an order that was serviced on the (now down) replica still returns the correct order information to the client. This can be seen clearly in the output pdf where all these scenarios are tested and the outputs are shown. The client never notices any failure in the order service replicas or leaders.

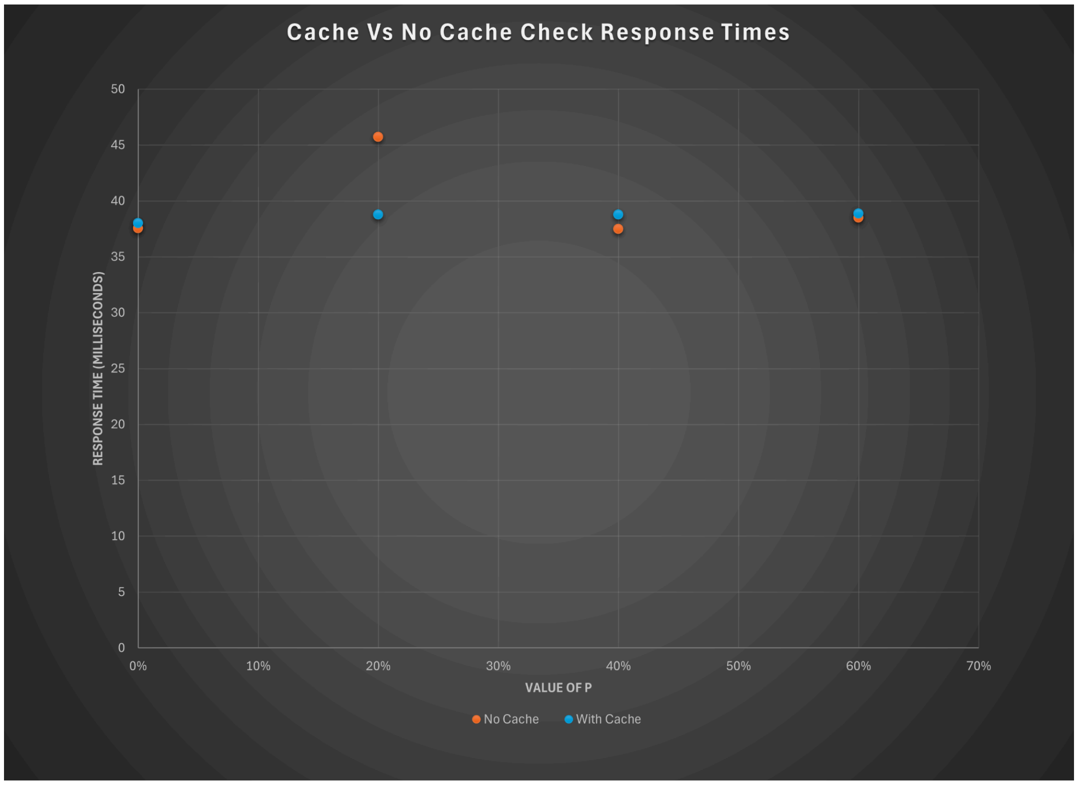
Do all the order service replicas end up with the same database file?

Yes, all the order service replicas end up with the same database file. After killing, restarting and performing orders while replicas are down, at the end, all the order service order log files end up exactly the same. This can be seen clearly in the output pdf as well.

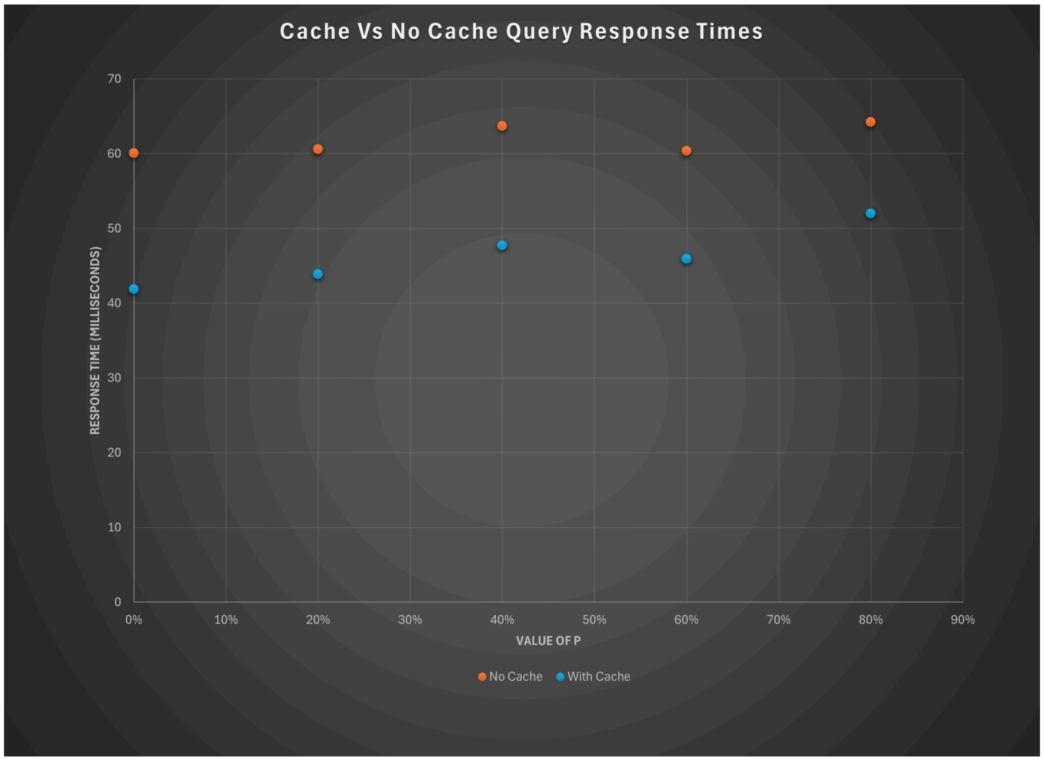
Comparing response times for Caching vs no caching with probability of performing a Buy request P:



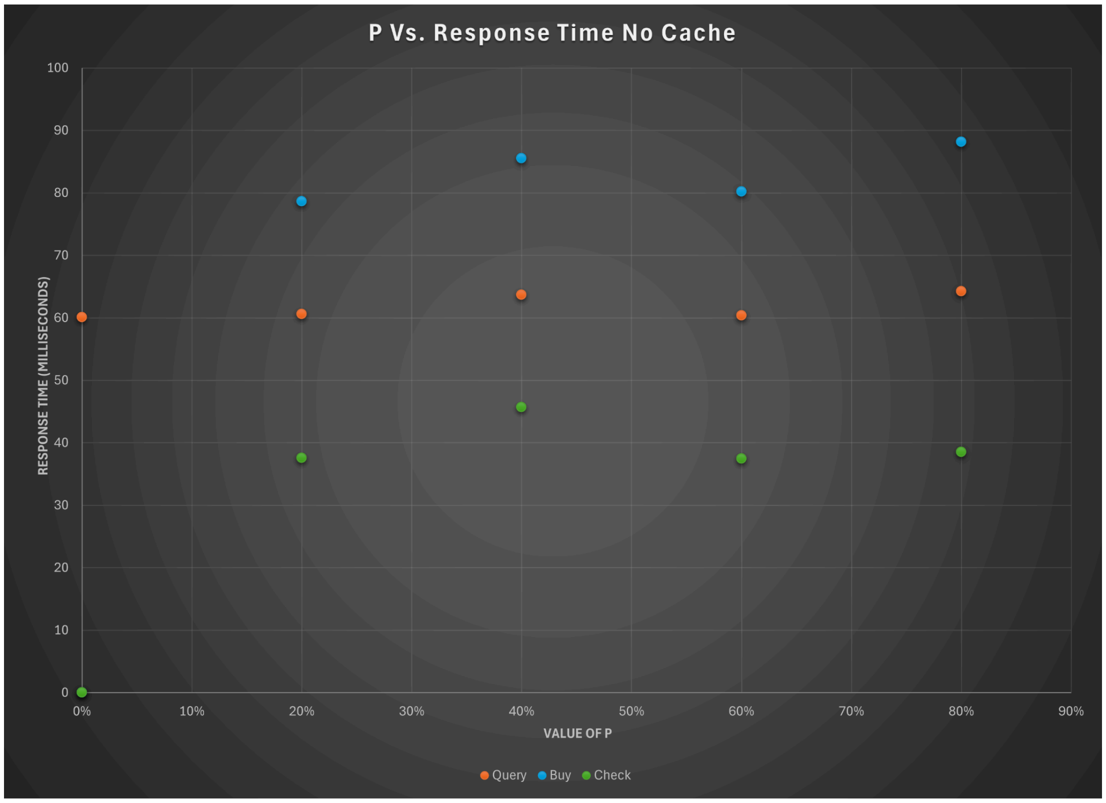
The response times for performing a Buy request is not influenced by the use of the cache. This is due to the implementation of the buy request, the request received by the front end is immediately forwarded to the Order service, the cache is never checked. This graph shows no correlation between the response time of using the cache and the response time not using cache. At each value of P, it is random if the cache or no cache response time was faster.



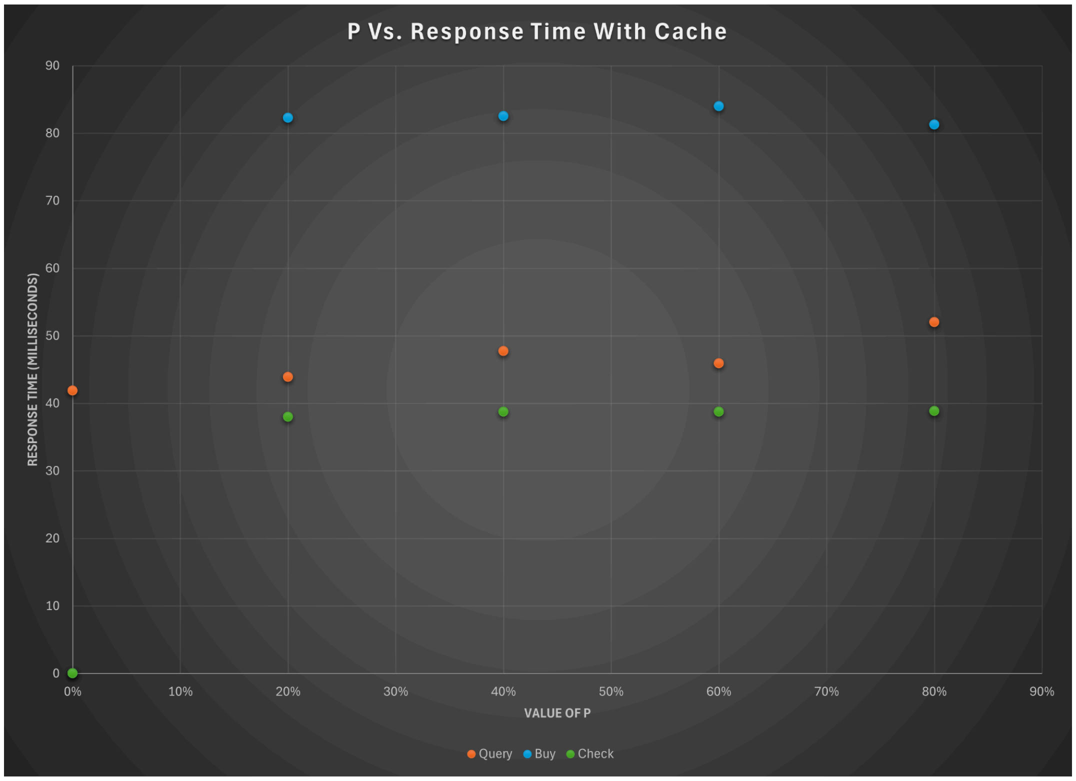
The response times for performing an order check request is not influenced by the use of the cache. This is because an order check request does not rely on the catalog at all; it only relies on accessing the order logs from the order service. As before, this graph shows no correlation between the response time of using the cache and the response time not using cache. At each value of P, the response times are almost identical.



Using the cache shows an improvement in response time for the query responses. This makes sense because the frontend service does not have to ask the catalog for a response if the toy is already in the cache. This saves the time of the frontend connecting to the catalog, waiting for a response, and then forwarding it to the client. The graph clearly shows the blue dots with the cache have lower response times than the orange no cache dots.



This graph shows the difference in response time for different types of requests for different values of P without using the cache. It is clear that the order check has the fastest response time, the query has the second fastest response time and the buy request has the slowest response time. The buy requests having the slowest response time makes sense because the buy requests have to be forwarded to the order service, which then sends a query request to the catalog, then replies to the front end with the order numbers, then the frontend replies to the client. This requires 3 total connections, client to frontend, frontend to order service, order service to catalog, which is 1 more than the check order and query catalog requires. It is interesting that the query response takes longer than the order check response. Looking through my code they make the same number of connections, and have similar operations for each request.



This graph shows the difference in response time for different types of requests for different values of P using the cache. Again, it is clear that the order check has the fastest response time, the query has the second fastest response time and the buy request has the slowest response time. The buy requests still have the slowest response time for the same reason as stated previously. It is clear that using the cache decreased the response time for the query requests, for the reasons stated in the graph for Query requests of cache vs no cache. It is interesting that the query responses still take slightly longer than the order check responses. Despite the query responses occasionally not requiring an additional connection, the order check responses are still the fastes response.