Assignment 1

Name 1: Ahmik Virani
Roll Number 1 : ES22BTECH11001

Name 2 : Divya Rajparia
Roll Number 2 : ES22BTECH11013

Question 4:

Brief explanation of our approach:

The **PUT** request from client to server simply communicates the key-value pair to the server, which the server stores in its storage.

Syntax followed: PUT /assignment1/key1/val1 HTTP/1.1

We chose the text file approach, as it would be possible to check the contents of storage even after the program finishes its execution, and hence a more realistic approach.
Please note, putting the same key with a different value updates the key-value pair to the new value.

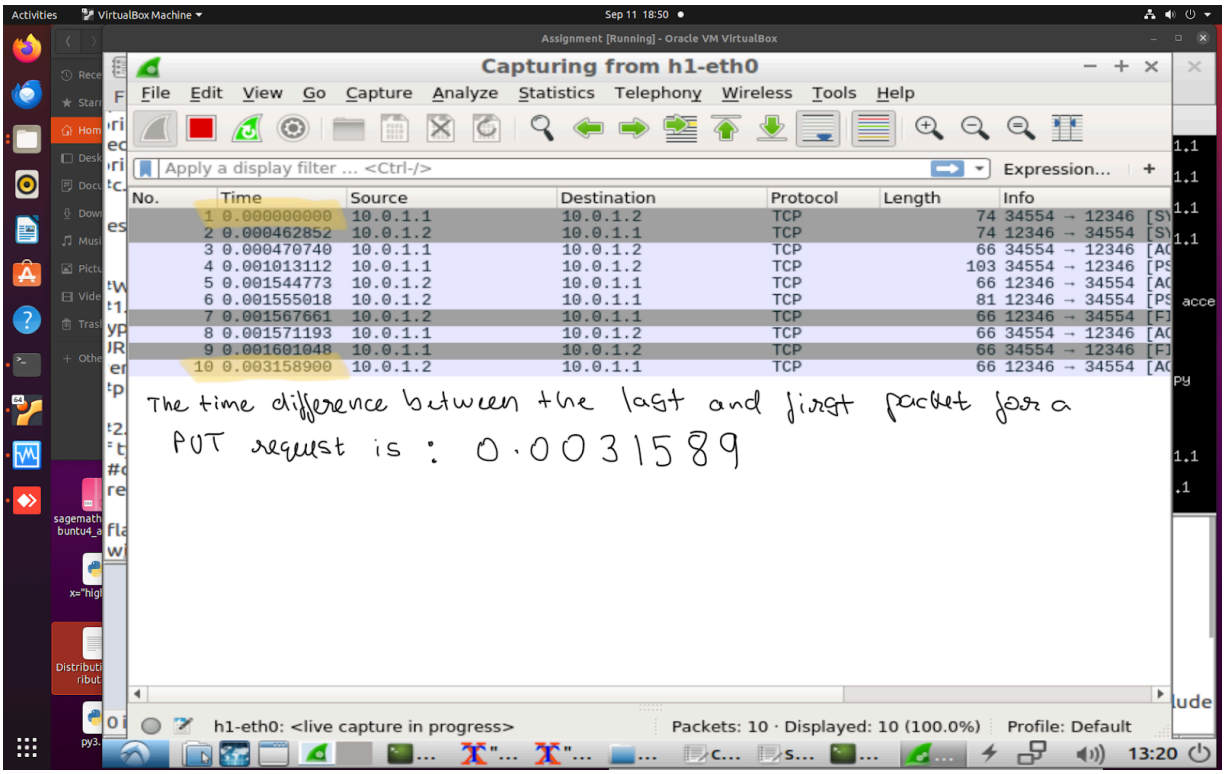
The **GET** request from client to server checks the storage of the server for the key. If found, it returns the key value pair, and if not, a 404 NOT FOUND error is returned.

Syntax followed: GET /assignment1?request=key1 HTTP/1.1

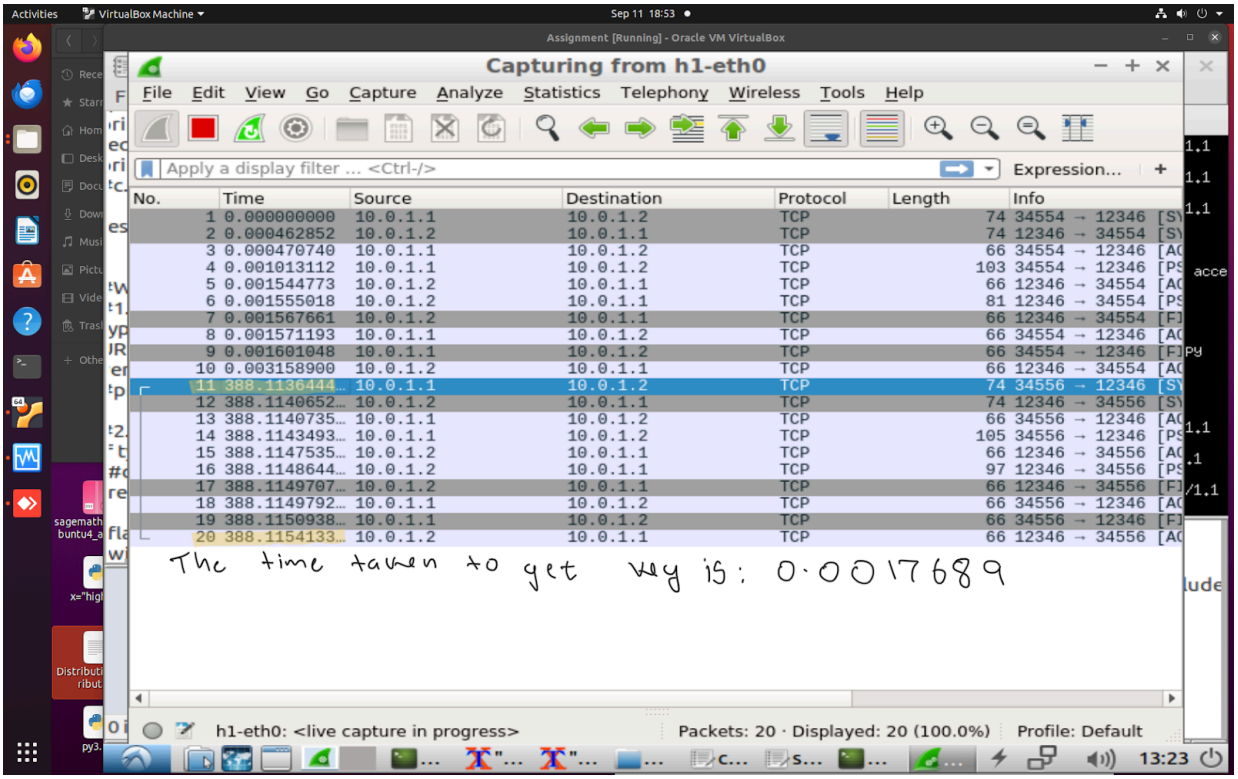
The **DELETE** request simply deletes the key-value pair from the server's storage.

Syntax followed: DELETE /assignment1/key1 HTTP/1.1

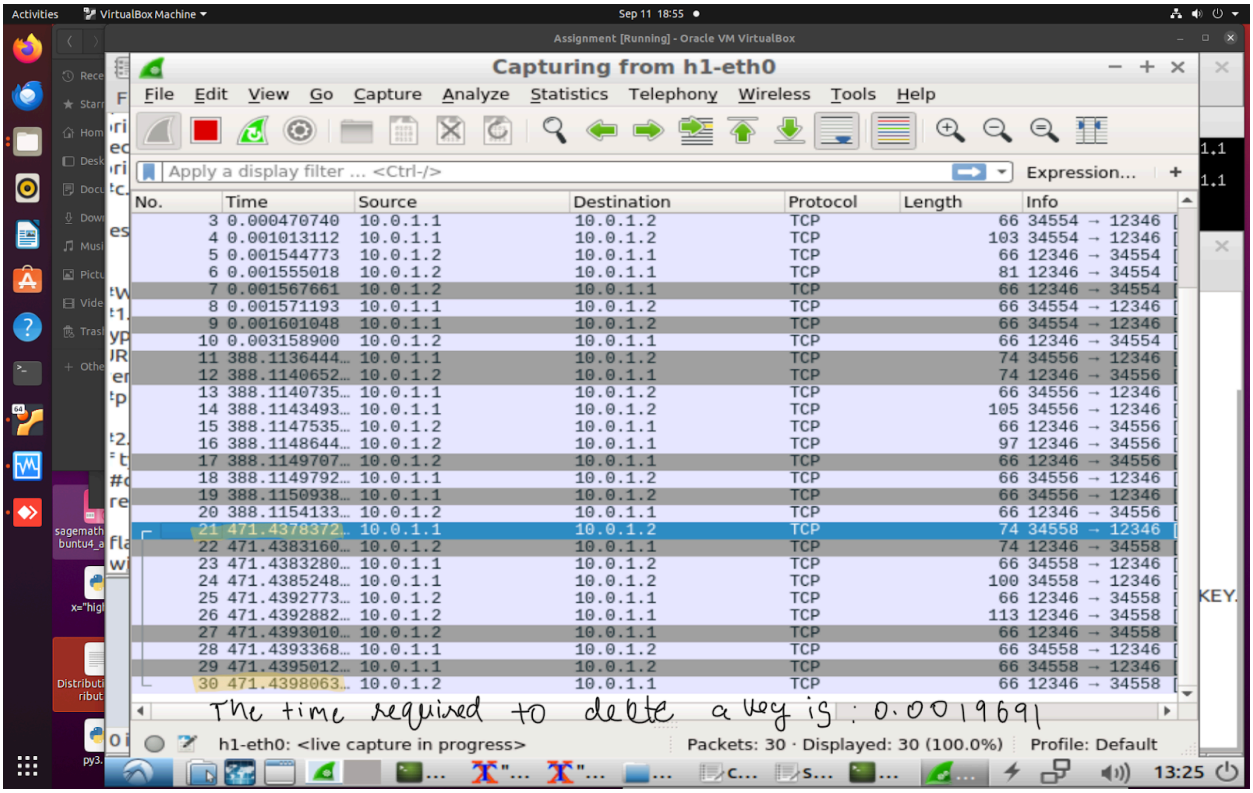
Q4H-a) Wireshark pcap trace for putting a key-value pair:



Wireshark pcap trace for getting a key-value pair:



Wireshark pcap trace for a delete request:



Each of these 3 traces have 10 messages exchanged between the source and destination. These can be classified as follows:
Packets 1 to 3: A 3 way handshake to establish the TCP connection between source and destination
Packets 4 to 7: Data exchange between client and server.
Packets 8-10: Terminating the TCP connection

Q4H-b)

Key	Req1 (first time)	Req2 (second time)	Req3 (third time)	Average Time
Key1	0.002245	0.001949	0.002664	0.002286
Key2	0.002112	0.002323	0.002565	0.002333333333
Key3	0.002369	0.002608	0.002588	0.002521666667
Key4	0.001783	0.002423	0.002872	0.002359333333
Key5	0.001782	0.001835	0.001883	0.001833333333
Key6	0.001458	0.001666	0.002283	0.001802333333

We can see that the time is quite variable, which can be due to various factors such as network latency, server load at that time, TCP connection overheads, etc. Overall, the end to end time is of the order of milliseconds.

Question 5:

Brief explanation of approach:

The fundamental functions of PUT, GET, and DELETE remain the same as before, but now, we have a slight twist - we have a cache! So here is how it will work now:

When we do a **PUT**, the client communicates a request to the cache for PUT. The cache simply forwards the same request to the server, which stores(or updates in the case when the same key, but new value is being putted) the key-value pair in the server storage.

Syntax followed: PUT /assignment1/key1/val1 HTTP/1.1

The **GET** request is sent from client to cache. If the cache has the key-value pair, it is returned from the cache itself, and no communication is made to the server.
But, if the cache does not possess the requested key-value pair, the cache requests the server for it. If the server has the pair, it returns it to the cache. The cache stores this pair in its own storage viz cache storage, and forwards it to the client as well.
However, if the server also does not have the requested key-value pair, it send a 404 NOT FOUND error to the cache, which is forwarded to the client.

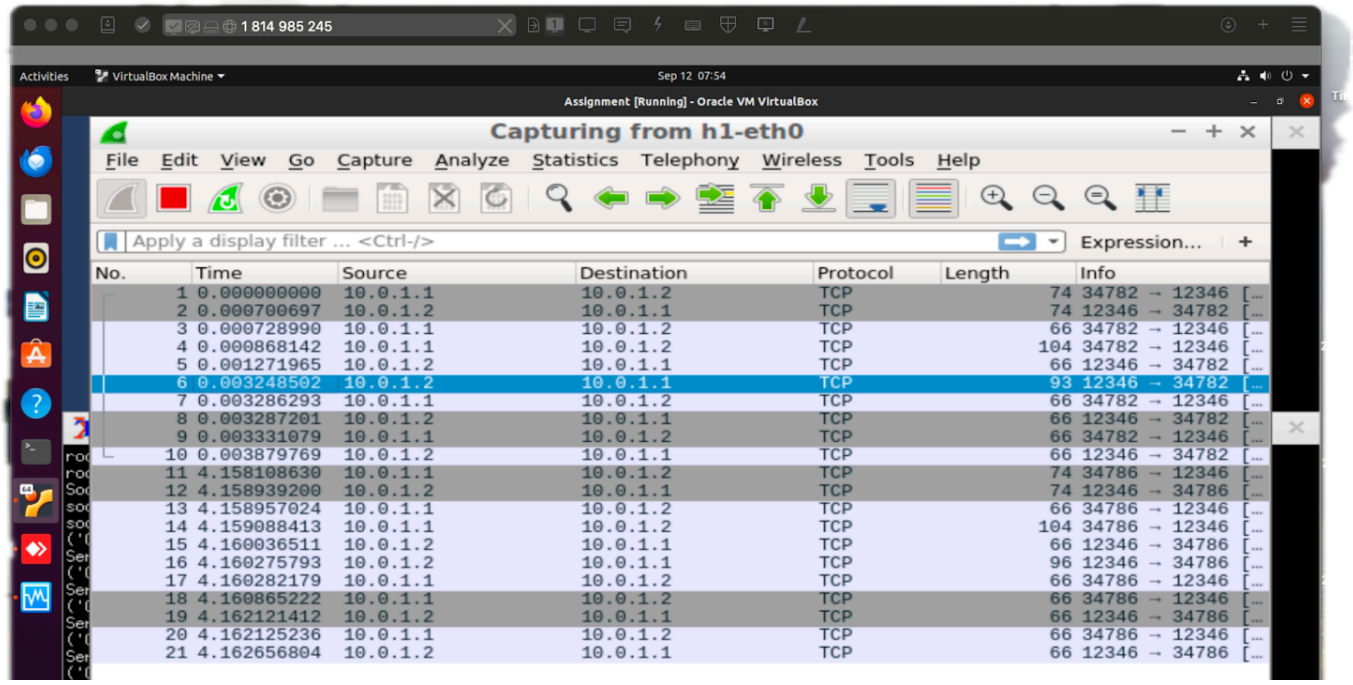
Syntax followed: GET /assignment1?request=key1 HTTP/1.1

If the client sends a **DELETE** request to the cache, the cache first forwards the query to the server. If the key is present, it is removed from the server storage, and if is not present, we return the NOT FOUND error to the cache, which forwards it to the client.
After this, we check if this value is present in the cache or not. Please note, it is important to delete the key-value pair from the cache also, as if not deleted, we might give an incorrect response in the future.

Syntax followed: DELETE /assignment1/key1 HTTP/1.1

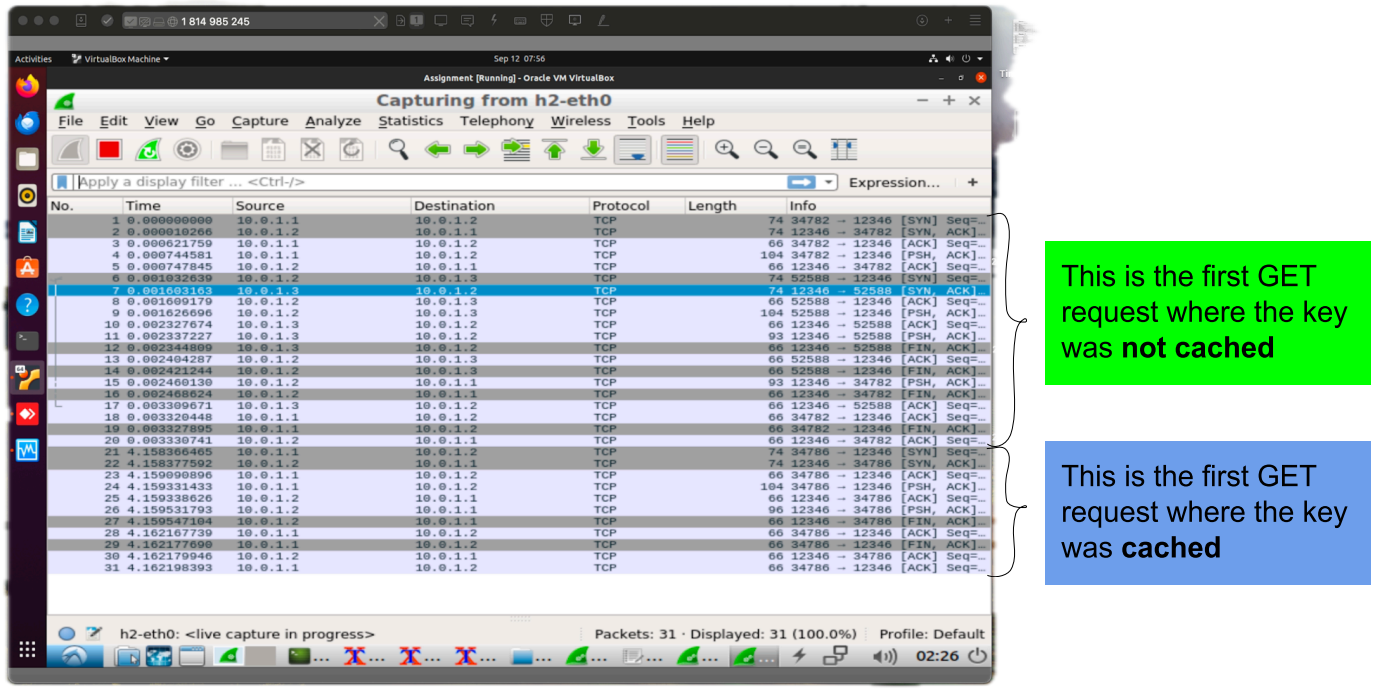
Q5H-a)
In this question, we want to study the pcap traces for GET request at the 3 nodes - H1, H2, H3, for two cases, when the key-value pair is cached, and when the key-value pair is not cached.

Let us first see the traces at the **client side** for the 2 requests:



Here, we can see that we have done 2 GET requests.
In the first request(packets 1-10), the pair is not present in the cache, and hence the request is forwarded to the server(will demonstrate the server's traces in the following images).
Whereas, in the second request, (packets 11-20), the object was present in the cache, and will be returned to the client.

Let us now study the communications at the **cache**.



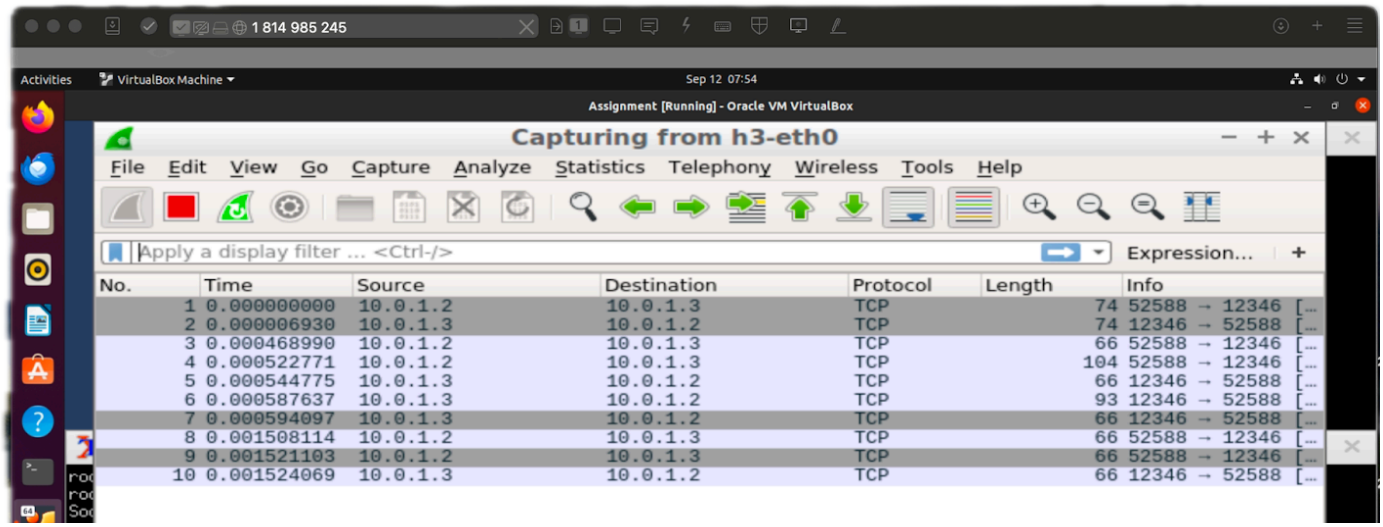
This is the first GET request where the key was **not cached**

This is the first GET request where the key was **cached**

The packets 1-20 correspond to the first GET request, where the packet was not present in the cache. Hence, the client first requests the cache(**cache is acting like a server to the client)**, and the cache then requests the server(**cache is acting like a client to the origin server)**, which returns the result.

The packers 21-30 correspond to the second GET request, where the key-value pair was **present in the cache itself**. Here, we do not need to communicate to the server, and hence the entire transaction is completed in 10 packets only.

This final image studies the traces at **server**.



In the first request, the key-value pair is not present in the cache, and hence the cache requests this origin server for the value.
But, in the second GET request, the **value is already present in cache**, and hence, the server is not contacted.
We can clearly see that the server is involved in **only one request**. This is because, only the cache is asking the server once, the second time the cache already has the key that was requested by the client, so the involvement of server was not needed.

Q5h-b)
Here, our aim is to study the average time for 3 GET requests.
In the fist GET request, the key-value pair is not present in the cache, and hence the cache reaches out to the server for the pair. But, in the twi subsequent requests, the pair is already cached, and hence the cache returns the pair itself, and no more involvement of server is necessary. Hence, a lot of time can is saved, which is evident from the average times below:

0.0083s when the pairs are not cached, and 0.00319s, 0.00259s when the pair is present in cache.

Key	Req1 (first time)	Req2 (second time)	Req3 (third time)
Key1	0.010617	0.005306	0.00227
Key2	0.005374	0.002059	0.002373
Key3	0.0109212	0.005842	0.002667
Key4	0.004726	0.001882	0.002328
Key5	0.004536	0.002215	0.00262
Key6	0.013488	0.001806	0.003304
Average Time	0.008277033333	0.003185	0.002593666667

Q6)

We did 3 GET requests for 2 cases, one with cache, and one without cache. Our findings and conclusions were:

1. When we do not have a cache, all the 3 requests take comparable time to be serviced from the server
2. When we do have a cache, the 1st request takes a significantly longer time than the 2nd and 3rd request
3. The time taken in the cases where there is no cache is considerably lesser than the 1st GET request with cache

Reasoning for the above observations:

1. When we do not have a cache, all the 3 GET requests have a similar path, they are sent from the client to the server, which searches its storage, and returns the result. Thus, there is very little room for deviations, and all the times are comparable.
2. But, when we do have a cache, it is an entirely different ball game.
   In the first GET request, the cache does not have the pair in its storage, and hence, it sends the request to the server. The server then sends the pair to the cache, which responds back to the client. The important thing to note here is that - the cache saves this pair in its memory.

   Now, in all future requests for the same key, the cache already has the pair saved in its own memory. Hence, it does not have to reach out to the server, and it responds to the request directly.
3. When we do not have a cache, there is only one correspondence path taken, from the client to the server. Whereas, in the first request of the cached scenario, the client

communicates with the cache, which in turn communicates with the server. Hence, the overhead for this is quite high, and thus the time taken for it is more than the simple client-server setting.

Statement regarding PCAP traces: We had to conduct the pcap traces experiment twice due to a virtual machine failure in the lab. Thus, the pcap images in the report and the pcap file submissions have slightly different time values. The overall functioning and conclusions are the same from the new pcap files as well, hence it does not affect our final report and findings.

We certify that this assignment/report is our own work based on our personal study and/or research and that we have acknowledged all material and sources used in its preparation, whether they be books, articles, packages, datasets, reports, lecture notes, and any other kind of document, electronic or personal communication. We also certify that this assignment/report has not previously been submitted for assessment/project in any other course lab, except where specific permission has been granted from all course instructors involved, or at any other time in this course, and that we have not copied in part or whole or otherwise plagiarized the work of other students and/or persons. We pledge to uphold the principles of honesty and responsibility at CSE@IITH. In addition, We understand my responsibility to report honor violations by other students if we become aware of it.

Name: Ahmik Virani
Date: 14/1/24
Signature: AV