COMPSCI 677 Spring 2022

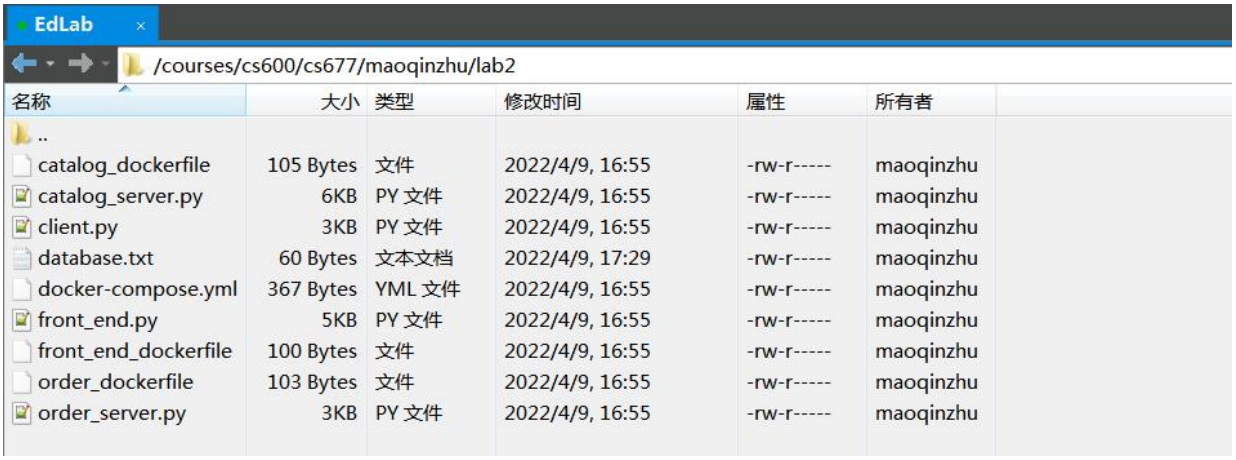
Lab 2: Tiered Microservices-Based Toy Store

Team Members: Maoqin Zhu, Yixiang Zhang

Part 1 - Implement Your Multi-Tiered Toy Store as Microservices

1. Server Startup Screenshots

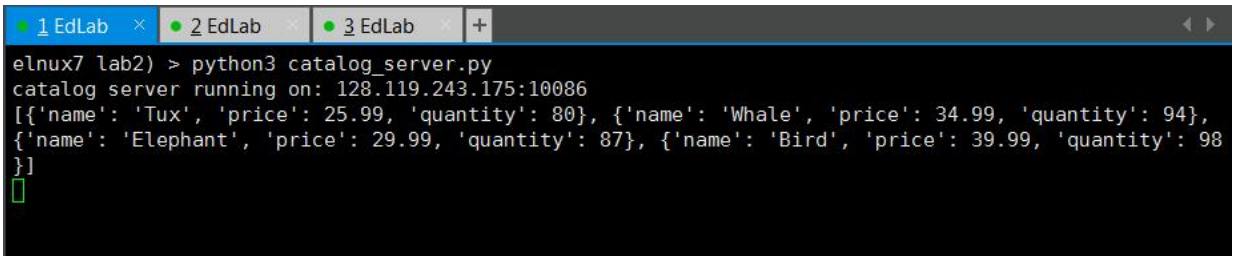
EdLab View- Log in the UMass EdLab remote server, and upload our source code files as follows.



Server Startup- Use following command to start up servers in specific order as described in design document. Note that when starting a server, we will print its IP address on your screen. You should type that IP address as environment variable for the next one.

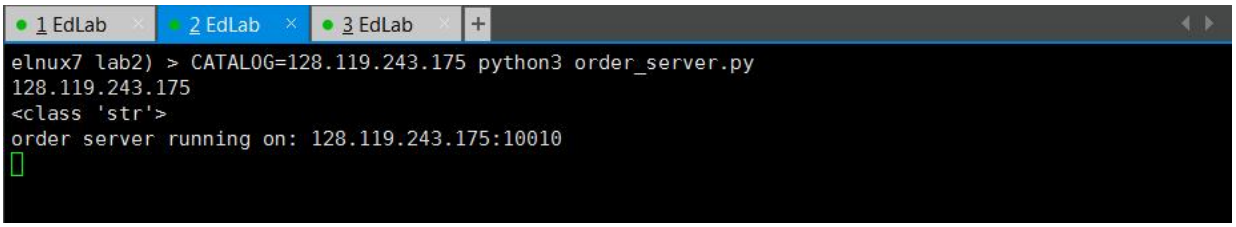
Step1: start catalog server

$ python3 catalog_server.py



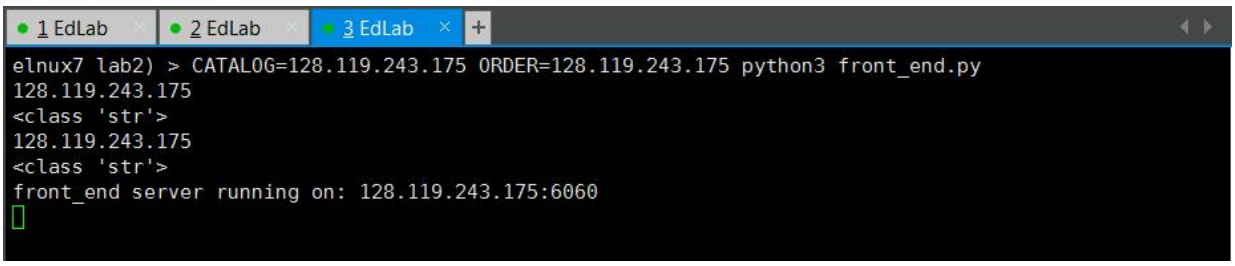
Step2: start order server

$ CATALOG=128.119.243.175 python3 order_server.py



Step3: start front-end server

$ CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 front_end.py



2. Functional Test Output

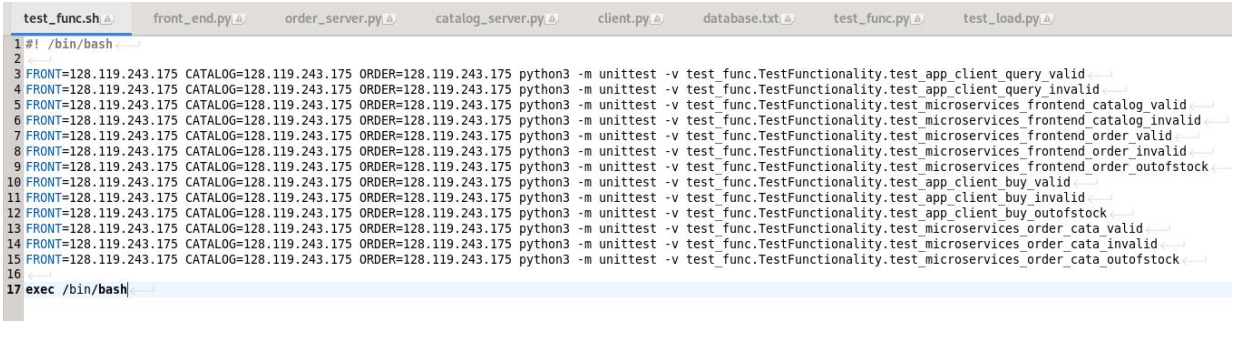
Automated Testing- Looking at **"test_func.py"**, for different HTTP GET / HTTP POST, we created 13 test cases which correspond to 13 possible HTTP responses described in design document Part 1 Section 2.1.

Notice that our test cases are effective only when database is in initial state, because expected response is configured statically in testing codes. Of course, you can also run your own test case simply by configuring request parameters and expected responses in the method. The initial state of database should be:

```
1  Tux 25.99 80
2  Whale 34.99 94
3  Elephant 29.99 87
4  Bird 39.99 98
```

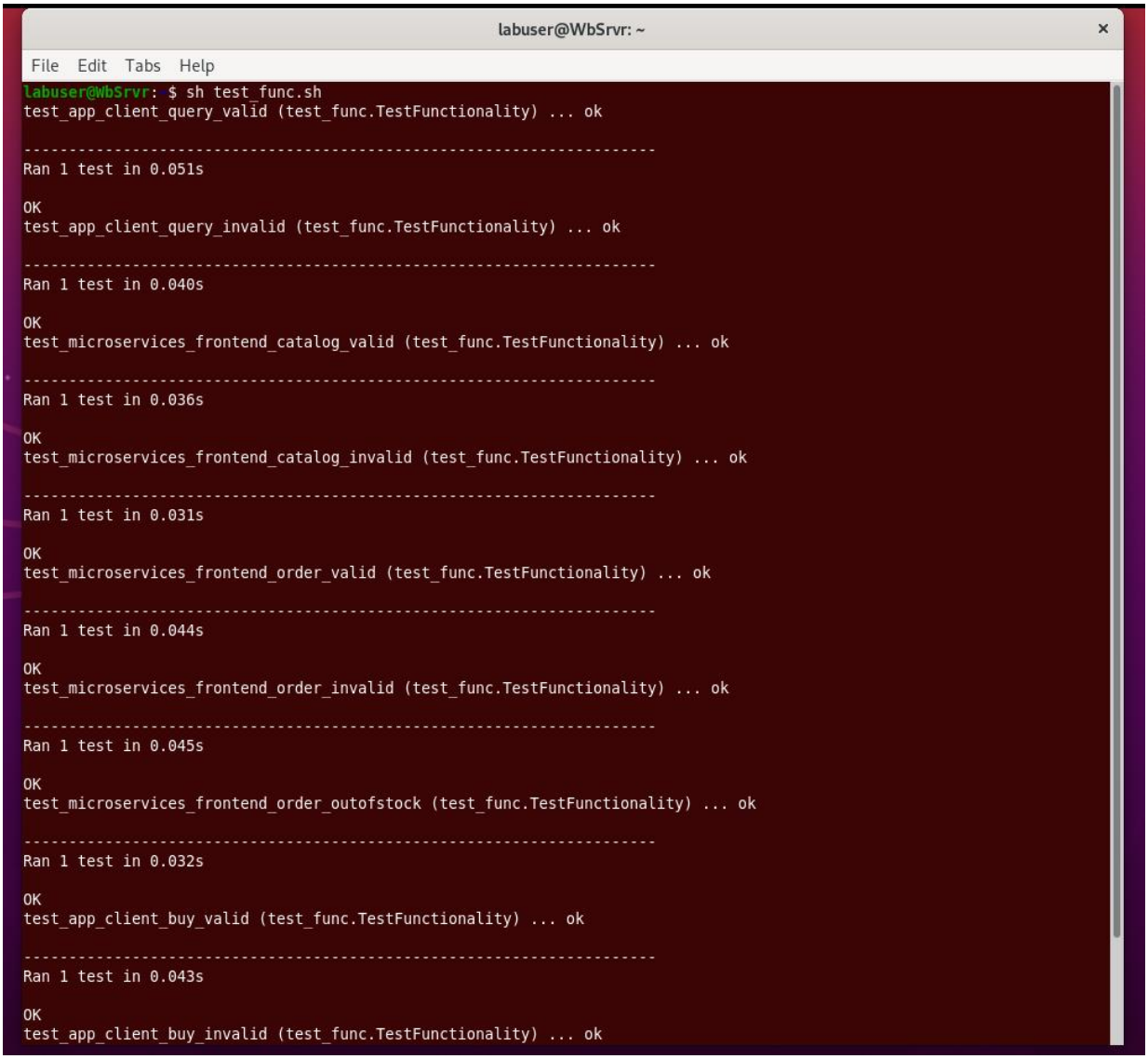
Looking at **"test_func.sh"**, this shell file will help us run all the 13 test cases.

Notice that each time if you are running this shell, please configure those IP addresses(environment variables) manually. Thank you!!!



Type the command: **$ sh test_func.sh**

For each test case(valid/invalid requests), if our application or micro-services work correctly, Python unittest will tell **“ok”** on your terminal. As you can see, all the functionalities is working correctly as follows.

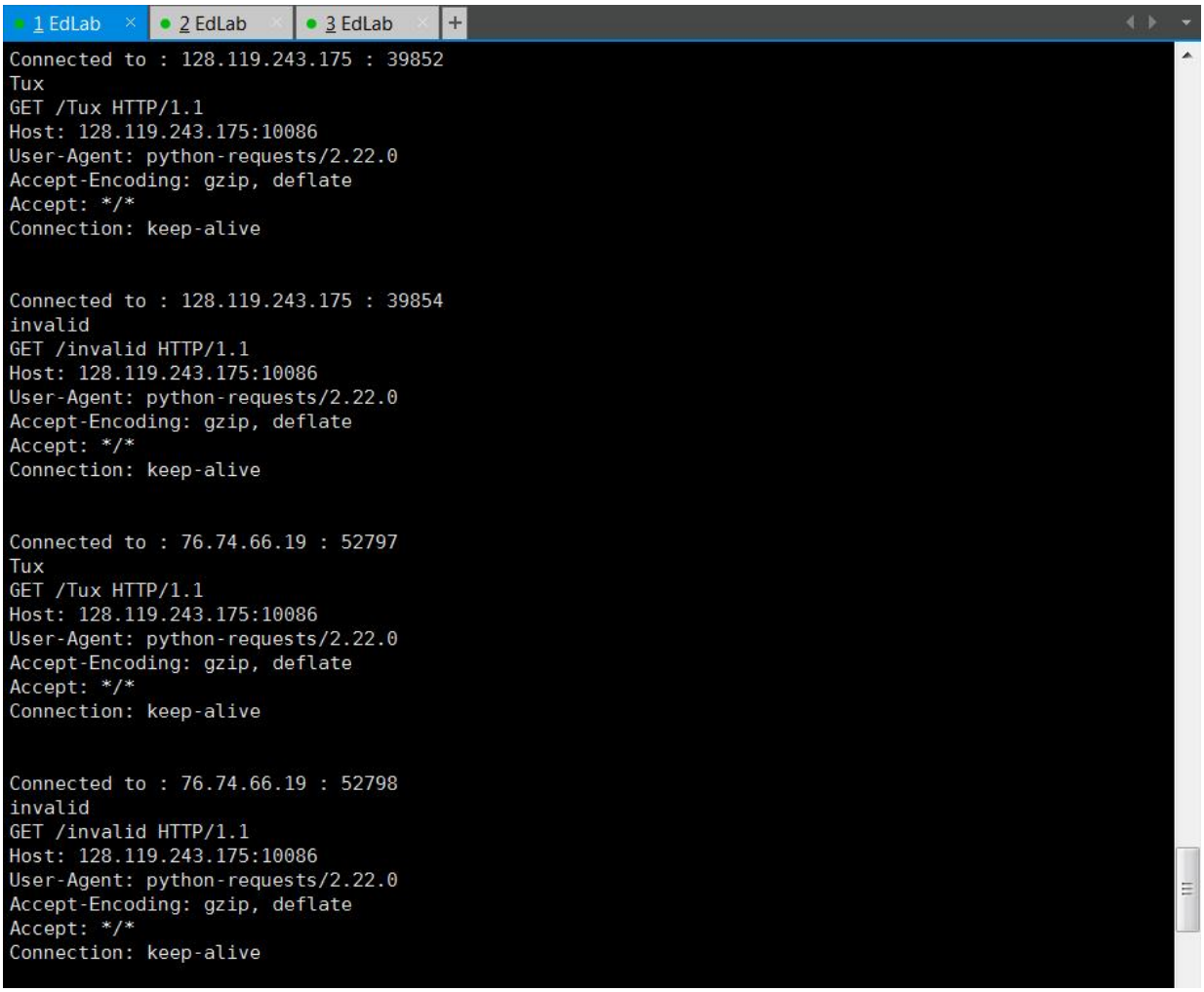


And also, after finishing those 13 test cases, we can see the order log has been recorded correctly (**order ID is -1** if the order has not been placed), and the database has been persisted too.

database.txt

```
Tux 25.99 80
Whale 34.99 93
Elephant 29.99 86
Bird 39.99 97
```

In terms of server terminal, its output roughly looks like this.



Client Functional Testing

Looking at **“client.py”**, we implemented 3 modes for you.

Mode 1: Query and Buy randomly: It randomly queries an item, if the returned quantity is greater than 0, with probability “p”(environment variable initialized in terminal) it will send an order request.

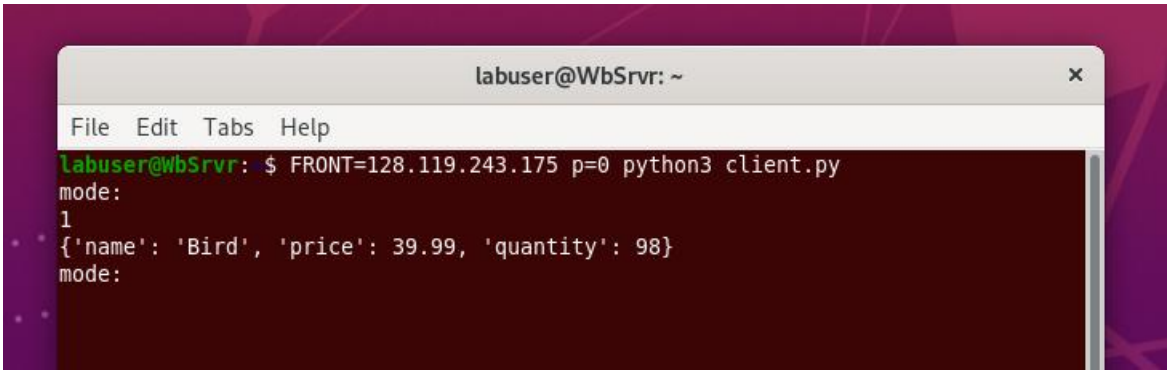
Mode 2: Initiate a serials of Query

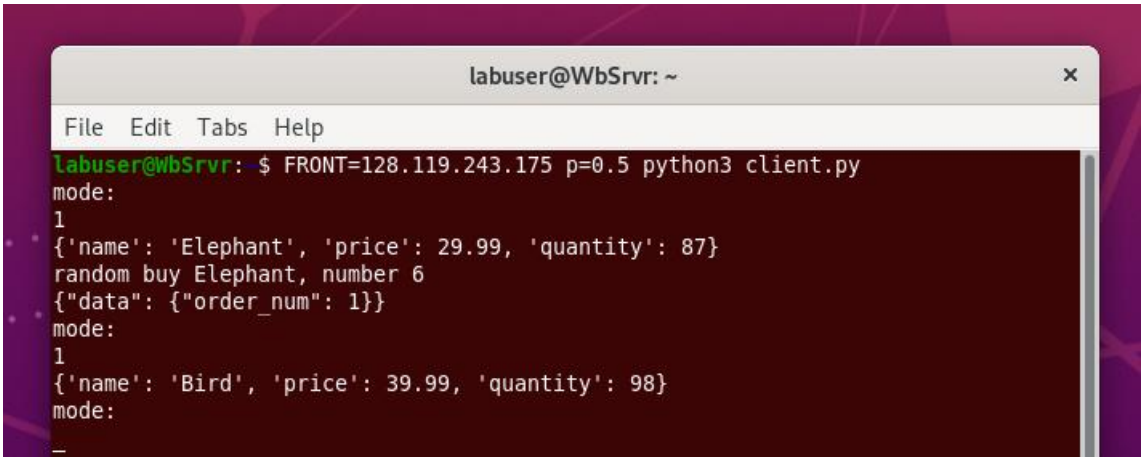
You can specify the toy name and query times as you want.

Mode 3: Initiate a serials of Buy

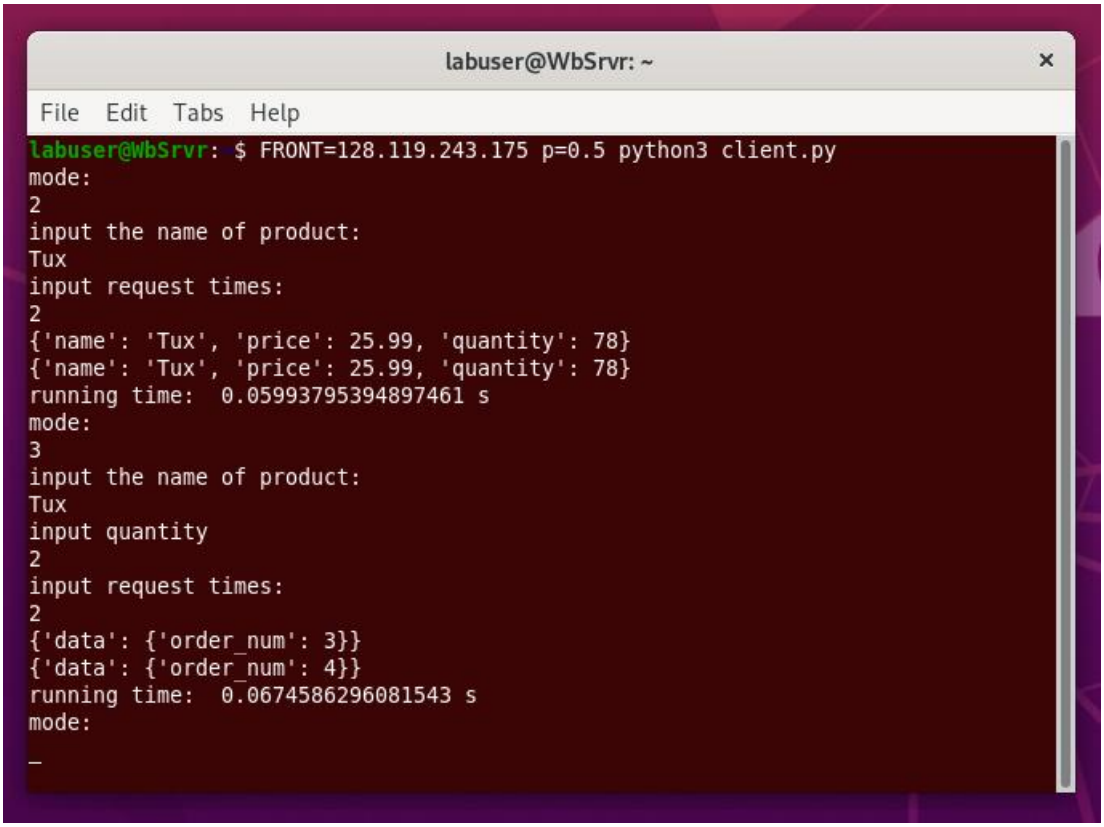
You can specify the toy name, quantity and number of requests as you want.

Mode 1 Examples:





Mode 2 & Mode 3 Examples:



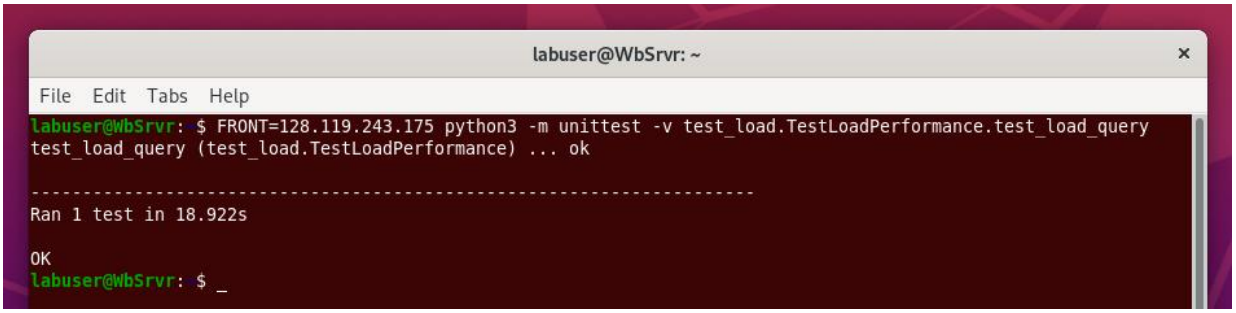
3. Load Test Output

Concurrent Queue- Looking at **“test_load.py”**, it automatically sends **1000** HTTP GET. Python unittest can help measure the total latency seen by clients in this case. Hence, in terms of average latency for each request, we should divide the total time by **1000**.

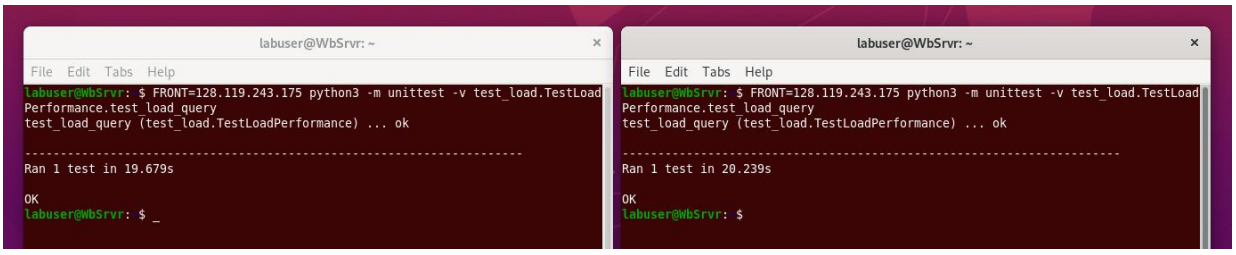
Here we vary the number of clients from 1 to 5 and measure the total latency as the load goes up. For each client terminal, we type the following command:

$ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query

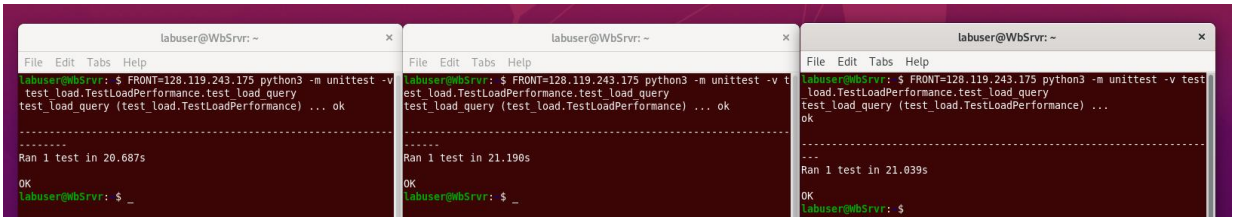
1 Client connected screenshot: terminal shows total latency of 1000 Query calls



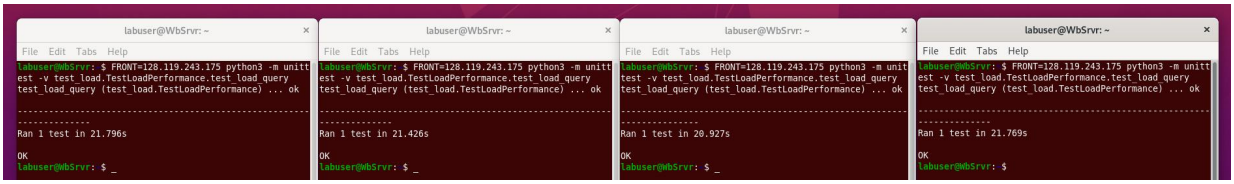
2 Clients connected screenshot: terminal shows total latency of 1000 Query calls



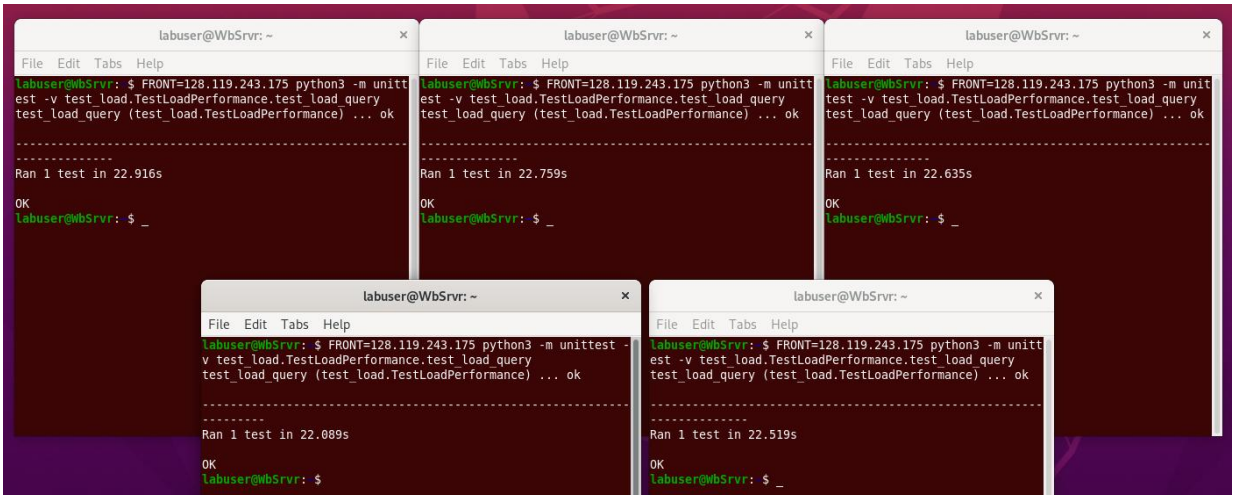
3 Clients connected screenshot: terminal shows total latency of 1000 Query calls



4 Clients connected screenshot: terminal shows total latency of 1000 Query calls



5 Clients connected screenshot: terminal shows total latency of 1000 Query calls



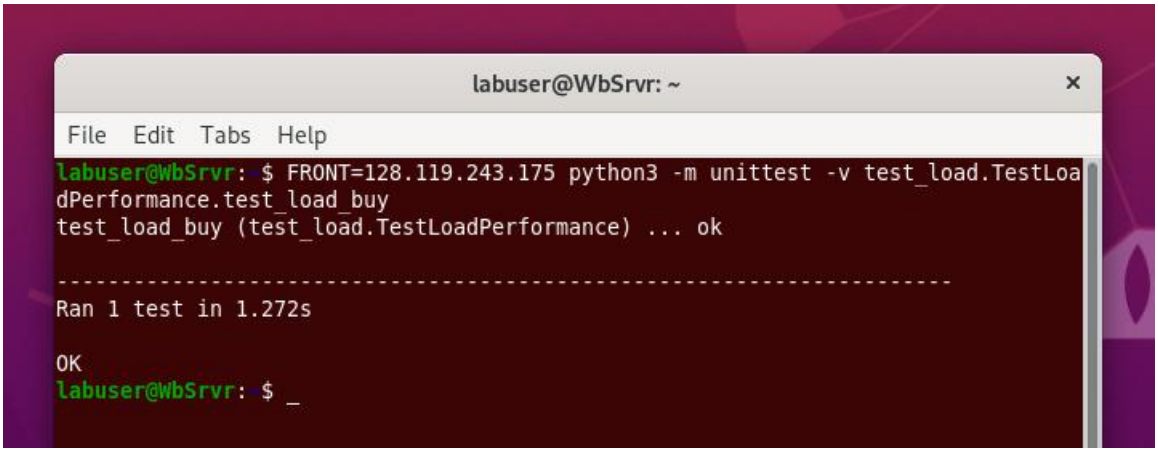
We have analyzed the average latency of each request in different scenarios in evaluation document. Please check out the details there.

Concurrent Buy- Looking at **"test_load.py"**, it automatically sends **100** HTTP POST. Python unittest can help measure the total latency seen by clients in this case. Hence, in terms of the average latency for each request, we should divide the total time by **100**.

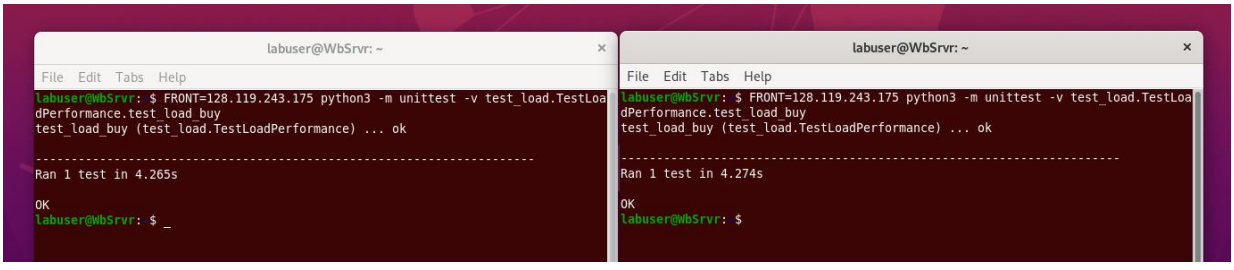
Here we vary the number of clients from 1 to 5 and measure the total latency as the load goes up. For each client terminal, we type the following command:

$ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_buy

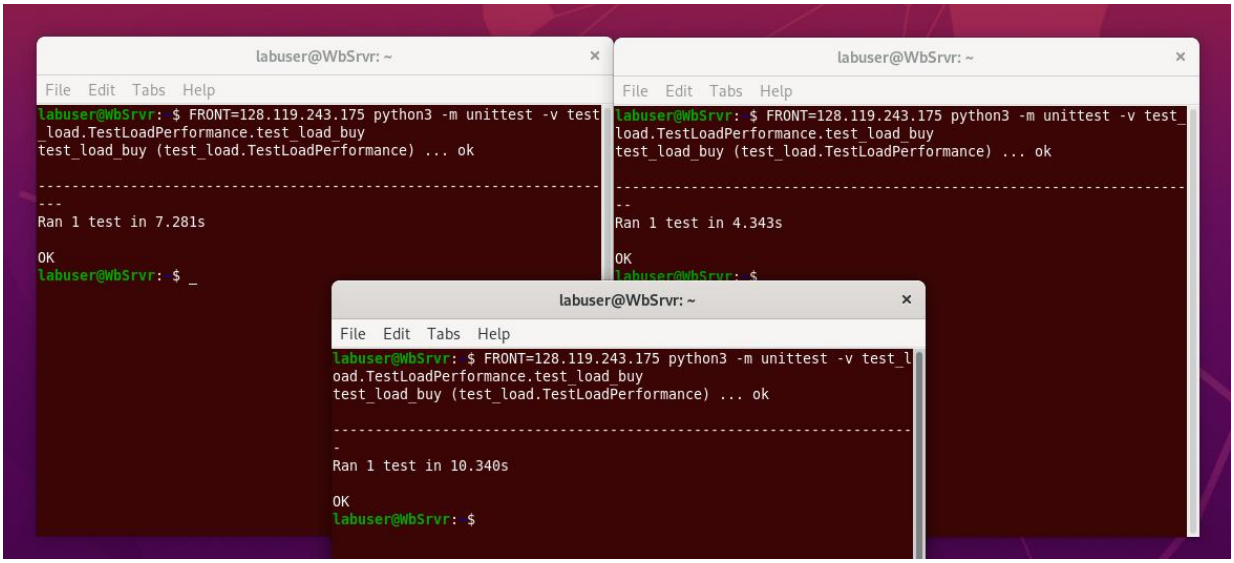
1 Client connected screenshot: terminal shows total latency of 100 Buy calls



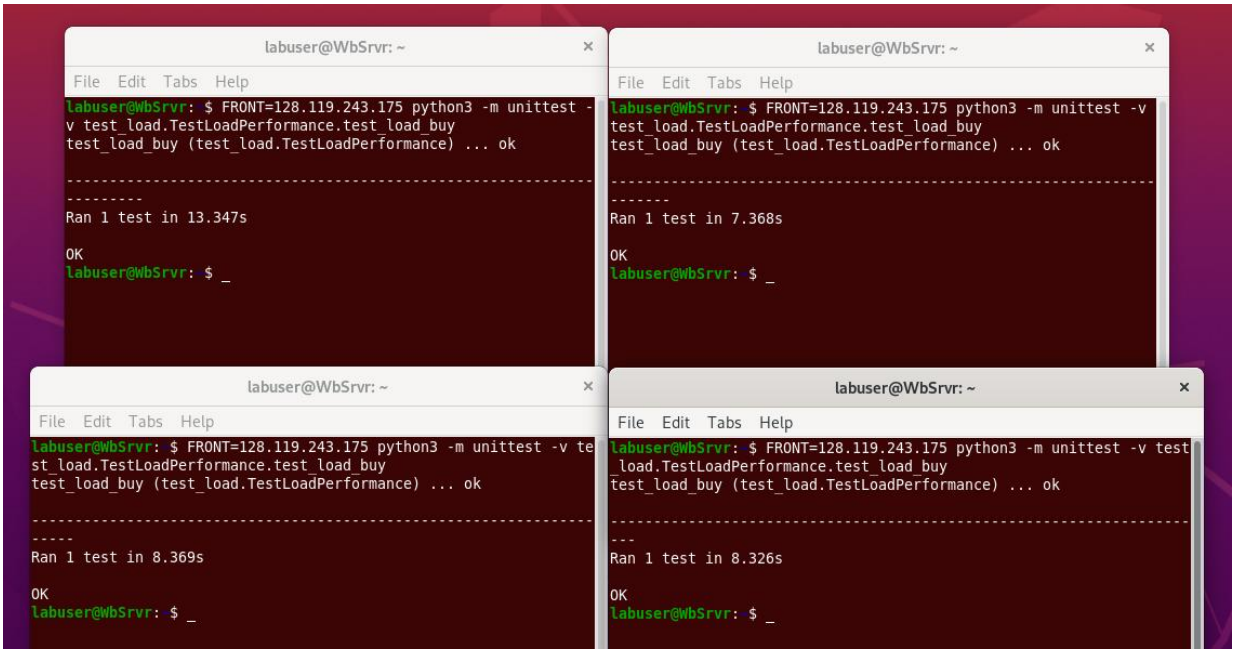
2 Clients connected screenshot: terminal shows total latency of 100 Buy calls



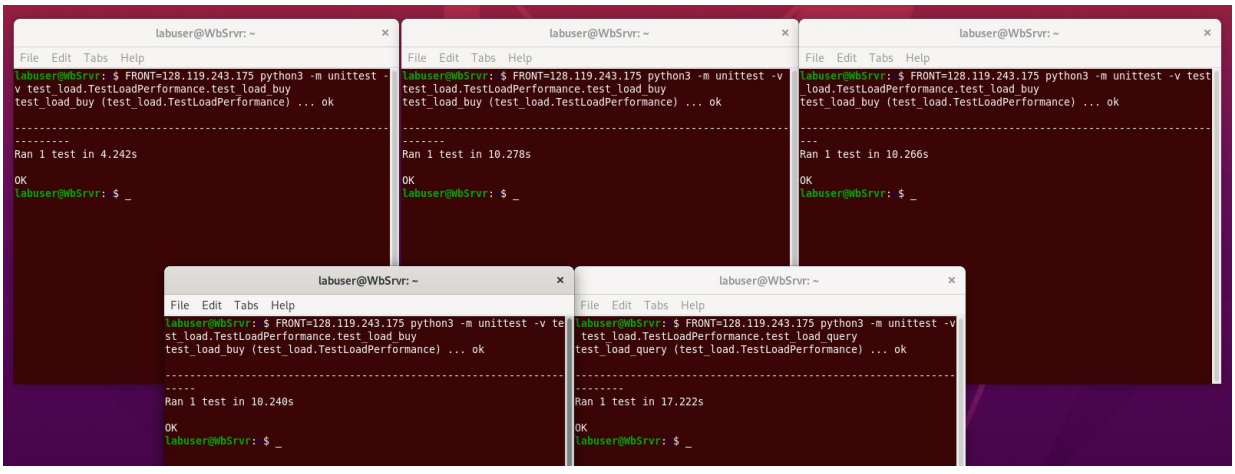
3 Clients connected screenshot: terminal shows total latency of 100 Buy calls



4 Clients connected screenshot: terminal shows total latency of 100 Buy calls



5 Clients connected screenshot: terminal shows total latency of 100 Buy calls



Part 2 - Containerize Your Application

1. Server Startup Screenshots

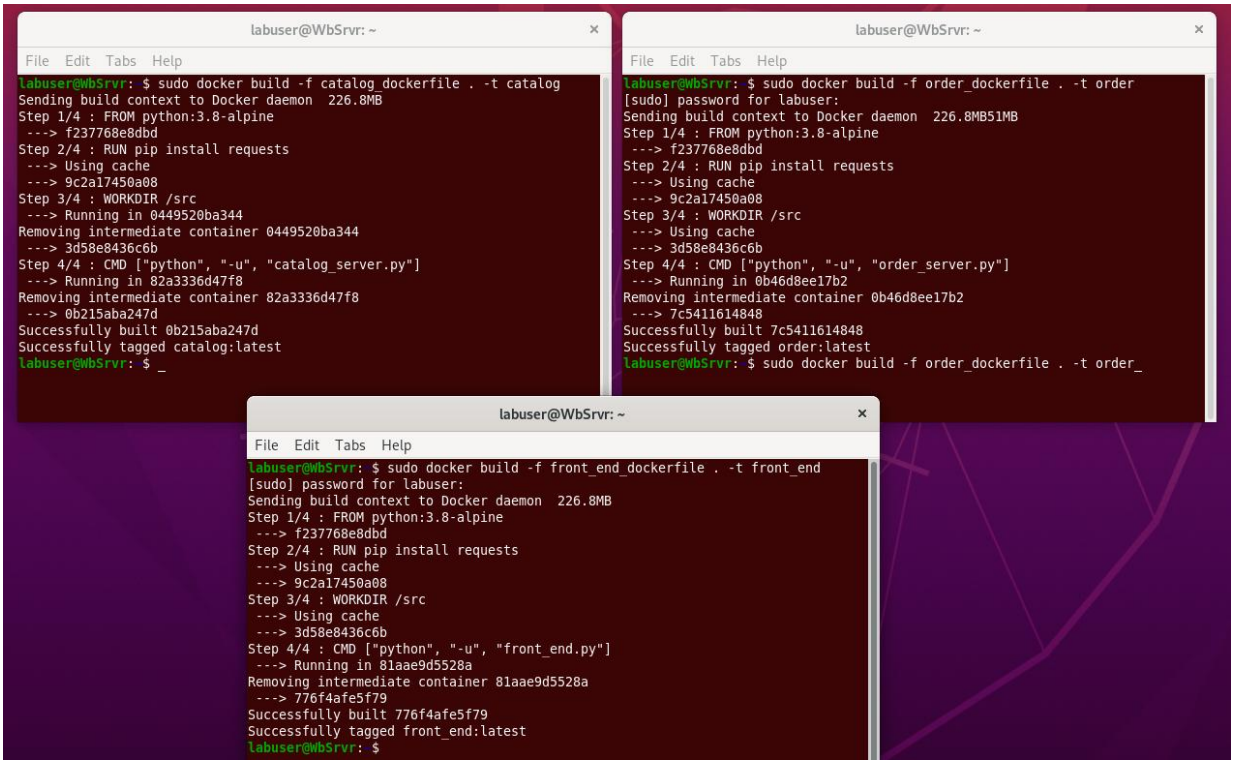
Consider that we can not containerize our application using Docker on remote Edlab server. So this time we provide a local test demo.

Containerize- Before running containers, we should build the Docker images.

$ sudo docker build -f catalog_dockerfile . -t catalog

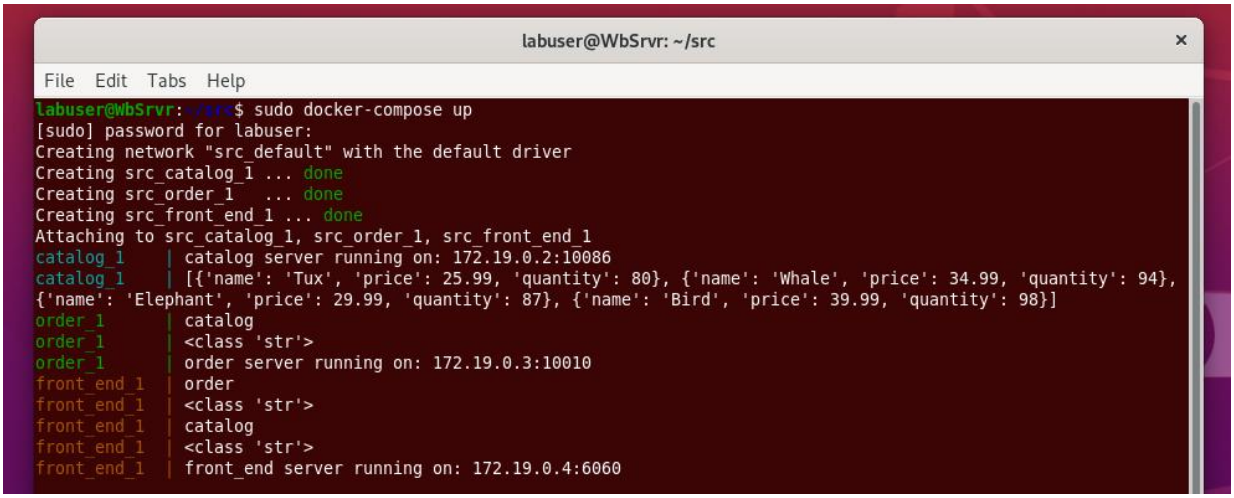
$ sudo docker build -f order_dockerfile . -t order

$ sudo docker build -f front_end_dockerfile . -t front_end



As described in design document, we can run dockers using Docker compose.

$ cd src and **$ sudo docker-compose up**



2. Functional Test Output

Automated Testing- Looking at **“test_func.py”**, for different HTTP GET / HTTP POST, we created 13 test cases which correspond to 13 possible HTTP responses described in design document Part 1 Section 2.1.

Notice that our test cases are effective only when database is in initial state, because expected response is configured statically in testing codes. Of course, you can also run your own test case simply by configuring request parameters and expected responses in the method. The initial state of database should be:

```
Tux 25.99 80
Whale 34.99 94
Elephant 29.99 87
Bird 39.99 98
```

Looking at **“test_func.sh”**, this shell file will help us run all the 13 test cases.

Notice that each time if you are running this shell, please configure those IP addresses(environment variables) manually. Thank you!!!

```
#! /bin/bash

FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_app_client_query_valid
FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_app_client_query_invalid
FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_microservices_frontend_catalog_valid
FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_microservices_frontend_catalog_invalid
FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_microservices_frontend_order_valid
FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_microservices_frontend_order_invalid
FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_microservices_frontend_order_outofstock
FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_app_client_buy_valid
FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_app_client_buy_invalid
FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_app_client_buy_outofstock
FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_microservices_order_cata_valid
FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_microservices_order_cata_invalid
FRONT=172.19.0.4 CATALOG=172.19.0.2 ORDER=172.19.0.3 python3 -m unittest -v test_func.TestFunctionality.test_microservices_order_cata_outofstock

exec /bin/bash
```

Type the command: **$ sh test_func.sh**

For each test case(valid/invalid requests), if our application or micro-services work correctly, Python unittest will tell **“ok”** on your terminal. As you can see, all the functionalities is working correctly as follows.

```
[qqmima@localhost test]$ FRONT=172.20.0.4 CATALOG=172.20.0.2 ORDER=172.20.0.3 python -m unittest -v test_func.TestFunctionality.test_app_client_buy_valid
test_app_client_buy_valid (test_func.TestFunctionality) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.012s

OK
[qqmima@localhost test]$ FRONT=172.20.0.4 CATALOG=172.20.0.2 ORDER=172.20.0.3 python -m unittest -v test_func.TestFunctionality.test_app_client_buy_invalid
test_app_client_buy_invalid (test_func.TestFunctionality) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.032s

OK
[qqmima@localhost test]$ FRONT=172.20.0.4 CATALOG=172.20.0.2 ORDER=172.20.0.3 python -m unittest -v test_func.TestFunctionality.test_app_client_buy_outofstock
test_app_client_buy_outofstock (test_func.TestFunctionality) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.010s

OK
[qqmima@localhost test]$ FRONT=172.20.0.4 CATALOG=172.20.0.2 ORDER=172.20.0.3 python -m unittest -v test_func.TestFunctionality.test_microservices_order_cata_valid
test_microservices_order_cata_valid (test_func.TestFunctionality) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.007s

OK
[qqmima@localhost test]$ FRONT=172.20.0.4 CATALOG=172.20.0.2 ORDER=172.20.0.3 python -m unittest -v test_func.TestFunctionality.test_microservices_order_cata_invalid
test_microservices_order_cata_invalid (test_func.TestFunctionality) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
[qqmima@localhost test]$ FRONT=172.20.0.4 CATALOG=172.20.0.2 ORDER=172.20.0.3 python -m unittest -v test_func.TestFunctionality.test_microservices_order_cata_outofstock
test_microservices_order_cata_outofstock (test_func.TestFunctionality) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
```

```
[qqmima@localhost test]$ FRONT=172.20.0.4 CATALOG=172.20.0.2 ORDER=172.20.0.3 python -m unittest -v test_func.TestFunctionality.test_app_client_query_invalid
test_app_client_query_invalid (test_func.TestFunctionality) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.020s

OK
[qqmima@localhost test]$ FRONT=172.20.0.4 CATALOG=172.20.0.2 ORDER=172.20.0.3 python -m unittest -v test_func.TestFunctionality.test_microservices_frontend_catalog_valid
test_microservices_frontend_catalog_valid (test_func.TestFunctionality) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.021s

OK
[qqmima@localhost test]$ FRONT=172.20.0.4 CATALOG=172.20.0.2 ORDER=172.20.0.3 python -m unittest -v test_func.TestFunctionality.test_microservices_frontend_catalog_invalid
test_microservices_frontend_catalog_invalid (test_func.TestFunctionality) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.020s

OK
[qqmima@localhost test]$ FRONT=172.20.0.4 CATALOG=172.20.0.2 ORDER=172.20.0.3 python -m unittest -v test_func.TestFunctionality.test_microservices_frontend_order_valid
test_microservices_frontend_order_valid (test_func.TestFunctionality) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.027s

OK
[qqmima@localhost test]$ FRONT=172.20.0.4 CATALOG=172.20.0.2 ORDER=172.20.0.3 python -m unittest -v test_func.TestFunctionality.test_microservices_frontend_order_invalid
test_microservices_frontend_order_invalid (test_func.TestFunctionality) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.023s

OK
[qqmima@localhost test]$ FRONT=172.20.0.4 CATALOG=172.20.0.2 ORDER=172.20.0.3 python -m unittest -v test_func.TestFunctionality.test_microservices_frontend_order_outofstock
test_microservices_frontend_order_outofstock (test_func.TestFunctionality) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.023s
```

And also, after finishing those 13 test cases, we can see the order log has been recorded correctly (**order ID is -1** if the order has not been placed), and the database has been persisted too.

order_log.txt

```
1 Whale 1
-1 invalid 1
-1 Whale 1000
2 Elephant 1
-1 invalid 1
-1 Elephant 1000
```

database.txt

```
Tux 25.99 80
Whale 34.99 93
Elephant 29.99 86
Bird 39.99 97
```

Client Functional Testing

Looking at **“client.py”**, we implemented 3 modes for you.

Mode 1: Query and Buy randomly: It randomly queries an item, if the returned quantity is greater than 0, with probability “p”(environment variable initialized in terminal) it will send an order request.

Mode 2: Initiate a serials of Query

You can specify the toy name and query times as you want.

Mode 3: Initiate a serials of Buy

You can specify the toy name, quantity and number of requests as you want.

Query and Buy Randomly Examples:

```
[qqmima@localhost src]$ FRONT=172.20.0.4 python client.py
mode:
1
{u'price': 25.99, u'name': u'Tux', u'quantity': 80}
mode:
1
{u'price': 39.99, u'name': u'Bird', u'quantity': 94}
random buy Bird, number 1
{"data": {"order_num": 422}}
mode:
1
{u'price': 34.99, u'name': u'Whale', u'quantity': 6591}
random buy Whale, number 6
{"data": {"order_num": 423}}
mode:
```

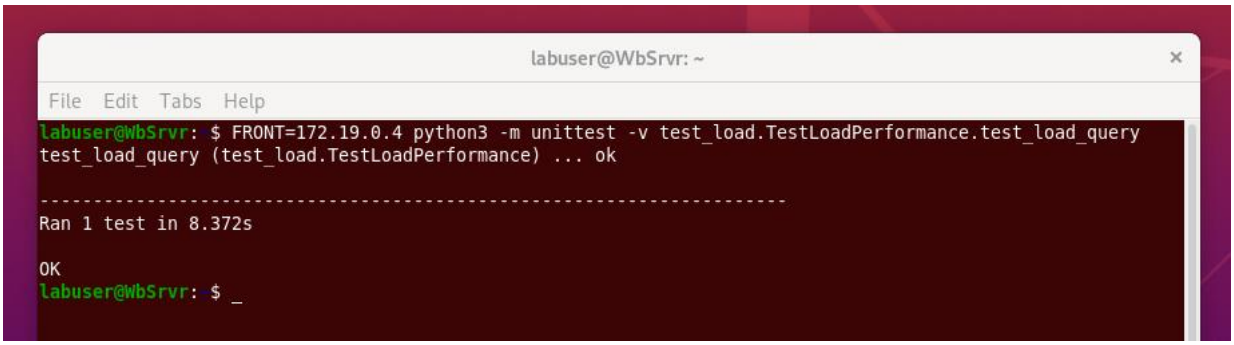
3. Load Test Output

Concurrent Queue- Looking at **"test_load.py"**, it automatically sends **1000** HTTP GET. Python unittest can help measure the total latency seen by clients in this case. Hence, in terms of average latency for each request, we should divide the total time by **1000**.

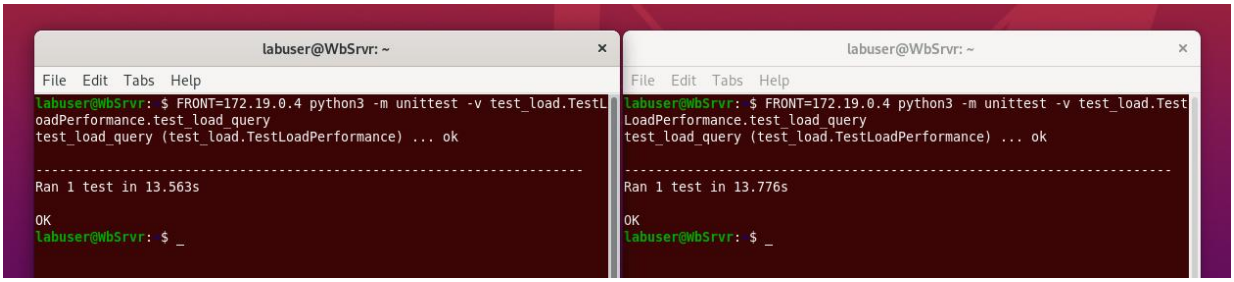
Here we vary the number of clients from 1 to 5 and measure the total latency as the load goes up. For each client terminal, we type the following command:

$ FRONT=172.19.0.4 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query

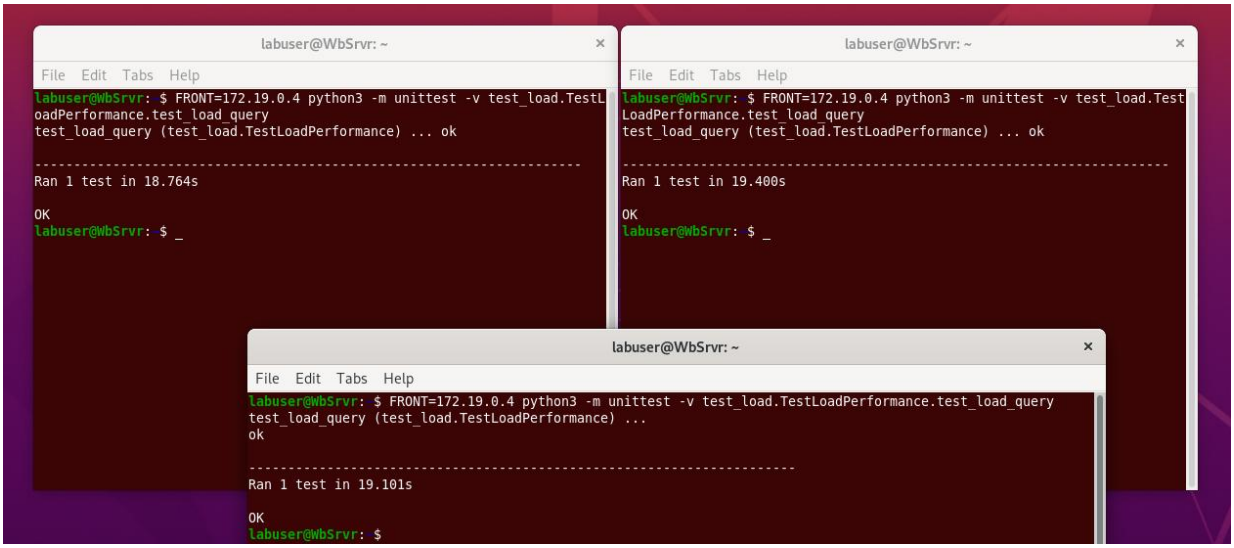
1 Client connected screenshot: terminal shows total latency of 1000 Query calls



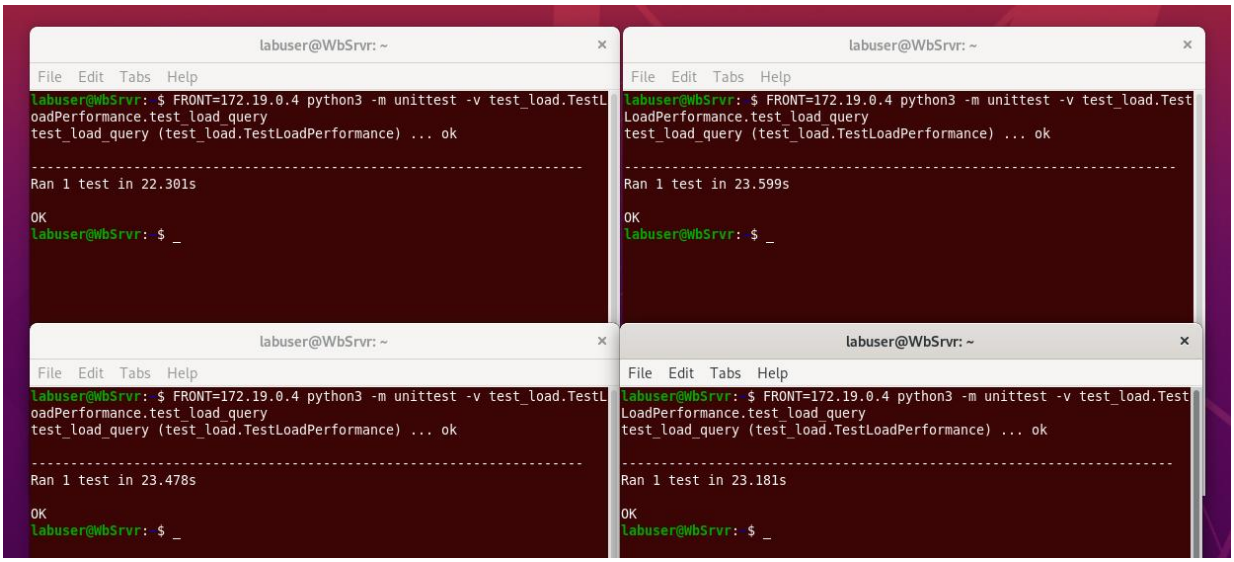
2 Client connected screenshot: terminal shows total latency of 1000 Query calls



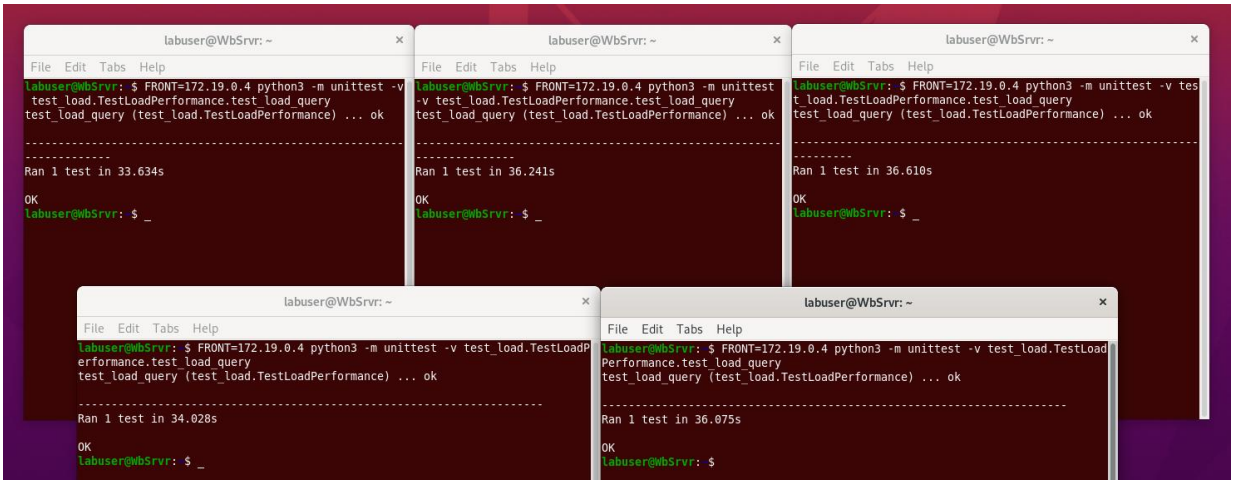
3 Client connected screenshot: terminal shows total latency of 1000 Query calls



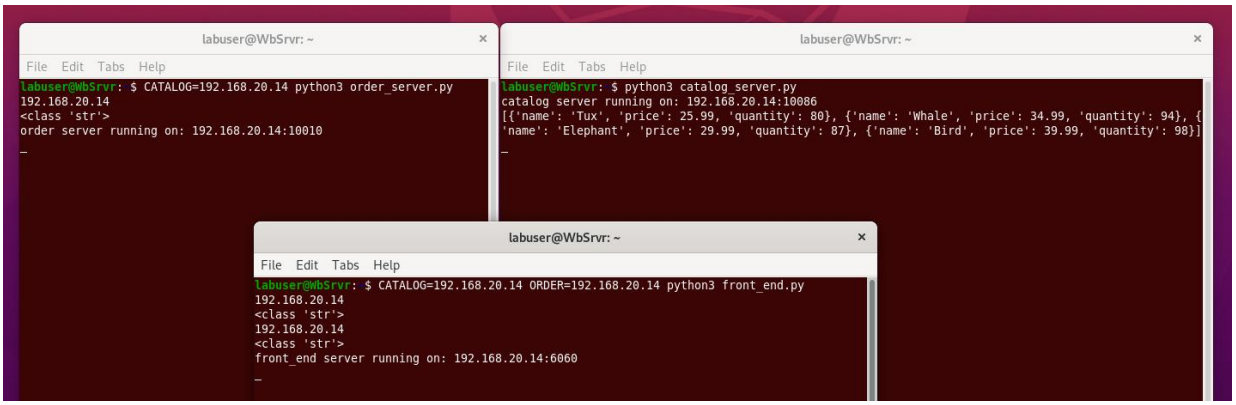
4 Client connected screenshot: terminal shows total latency of 1000 Query calls



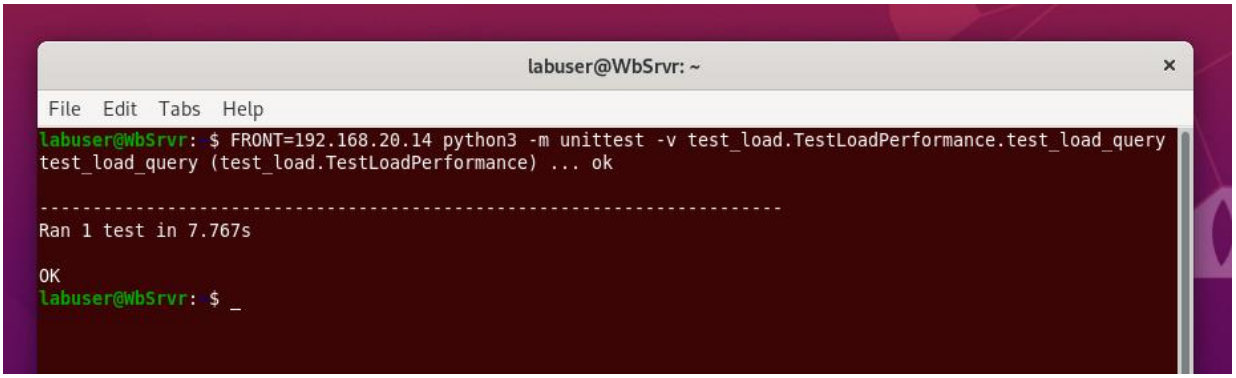
5 Client connected screenshot: terminal shows total latency of 1000 Query calls



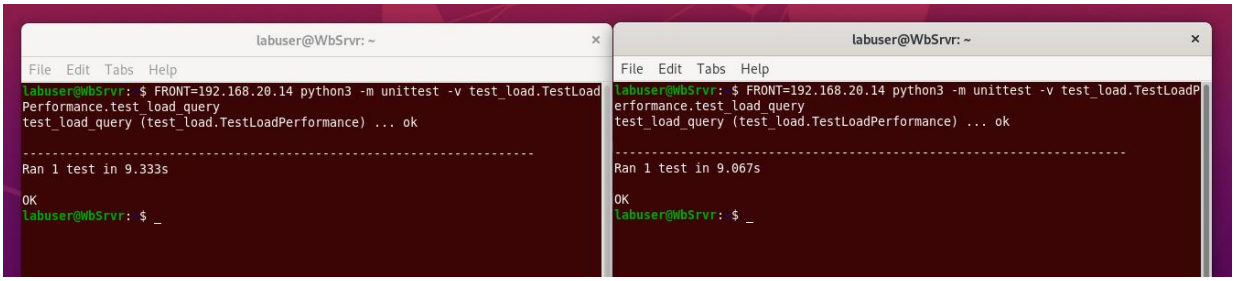
Local Test Without Containers- Since we cannot containerize our application using Docker on remote EdLab server. In order to find out if virtualization add any overheads, we do the local query load test without containers.



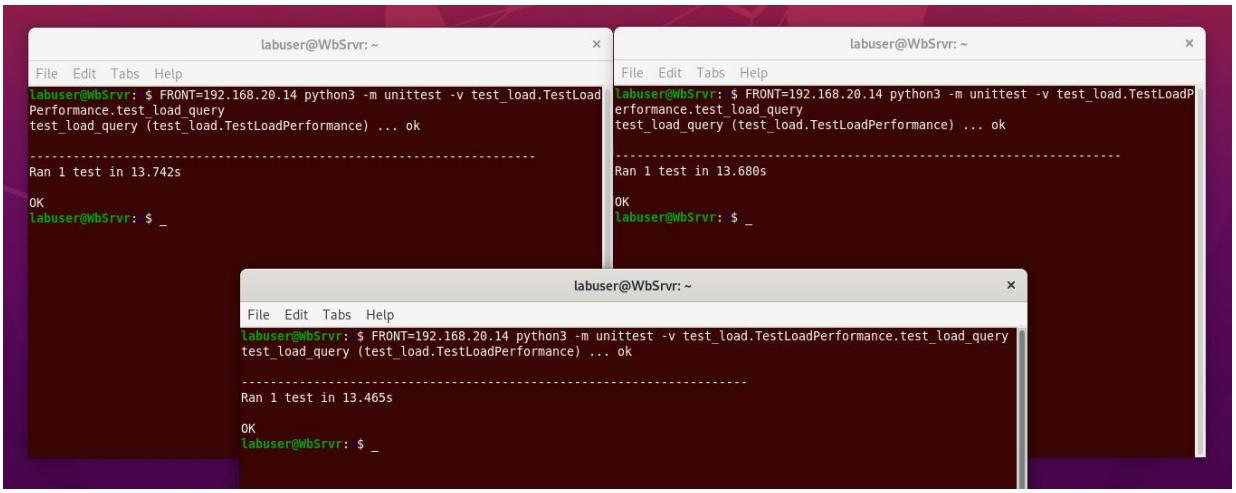
1 Client connected screenshot: terminal shows total latency of 1000 Query calls



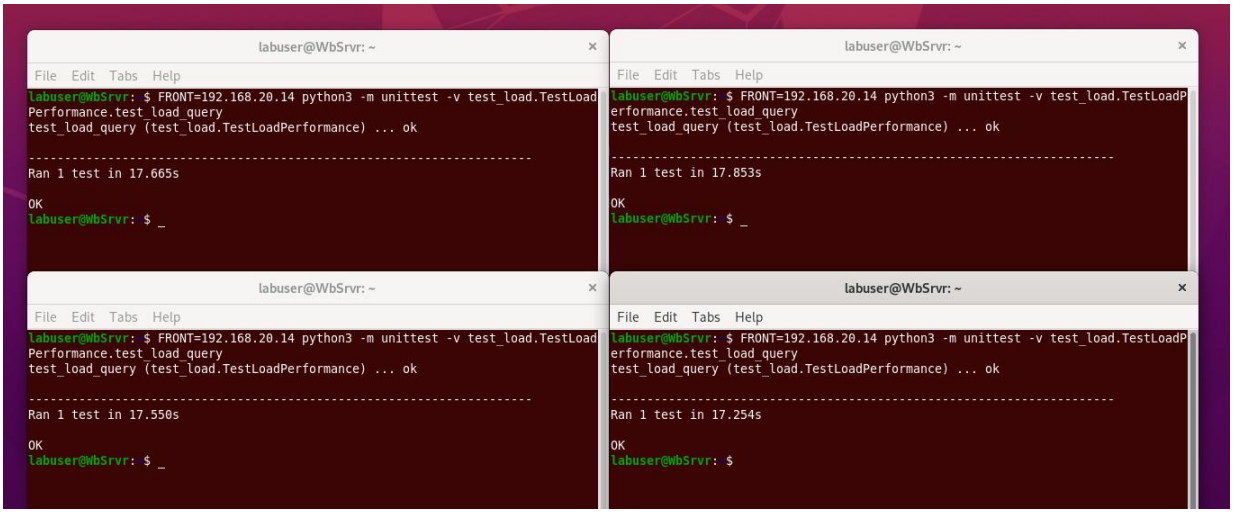
2 Client connected screenshot: terminal shows total latency of 1000 Query calls



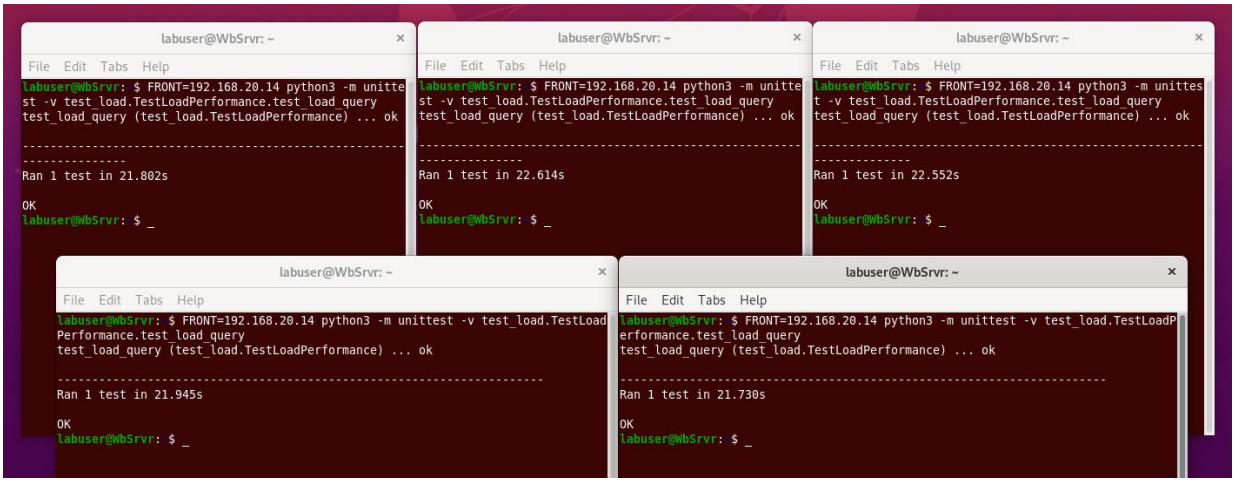
3 Client connected screenshot: terminal shows total latency of 1000 Query calls



4 Client connected screenshot: terminal shows total latency of 1000 Query calls



5 Client connected screenshot: terminal shows total latency of 1000 Query calls



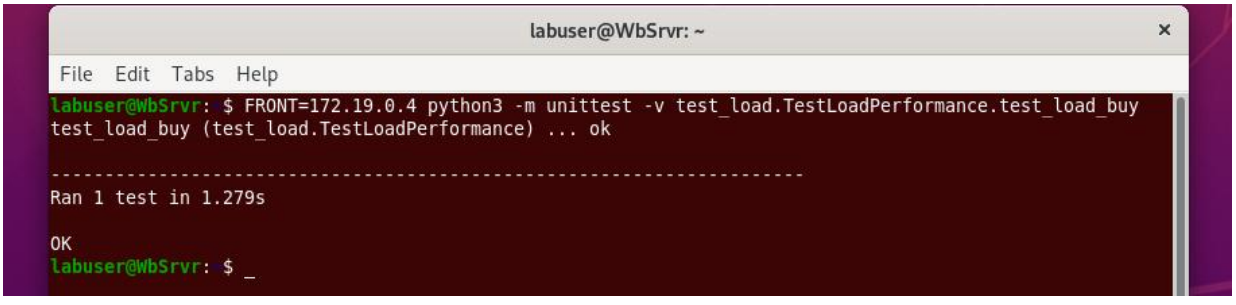
We have analyzed the average latency of each request in different scenarios in evaluation document. Please check out the details there.

Concurrent Buy- Looking at **"test_load.py"**, it automatically sends **100** HTTP POST. Python unittest can help measure the total latency seen by clients in this case. Hence, in terms of the average latency for each request, we should divide the total time by **100**.

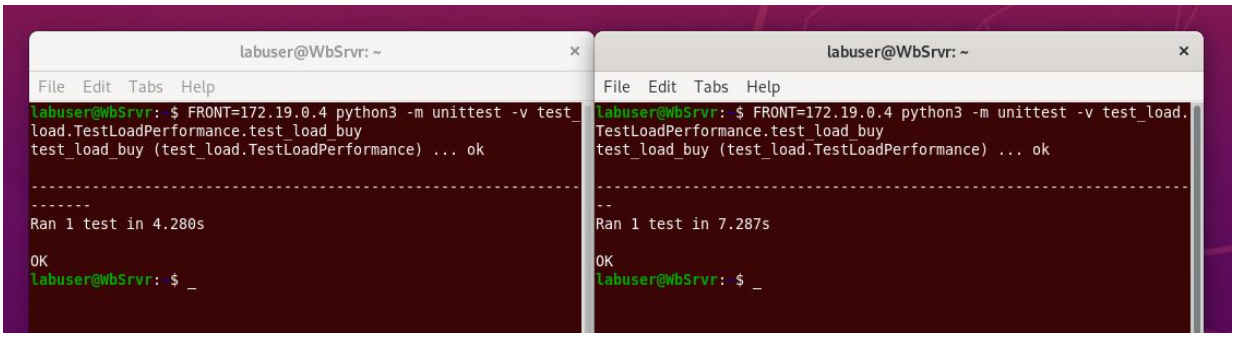
Here we vary the number of clients from 1 to 5 and measure the total latency as the load goes up. For each client terminal, we type the following command:

$ FRONT=172.19.0.4 python3 -m unittest -v test_load.TestLoadPerformance.test_load_buy

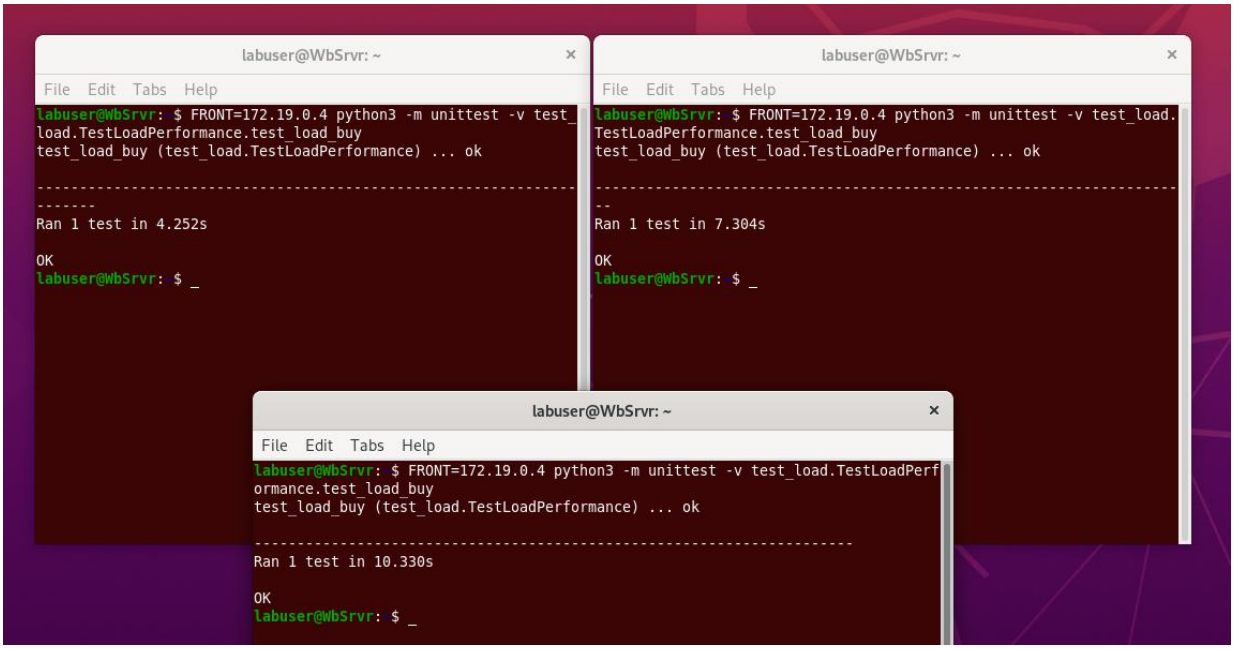
1 Client connected screenshot: terminal shows total latency of 100 Buy calls



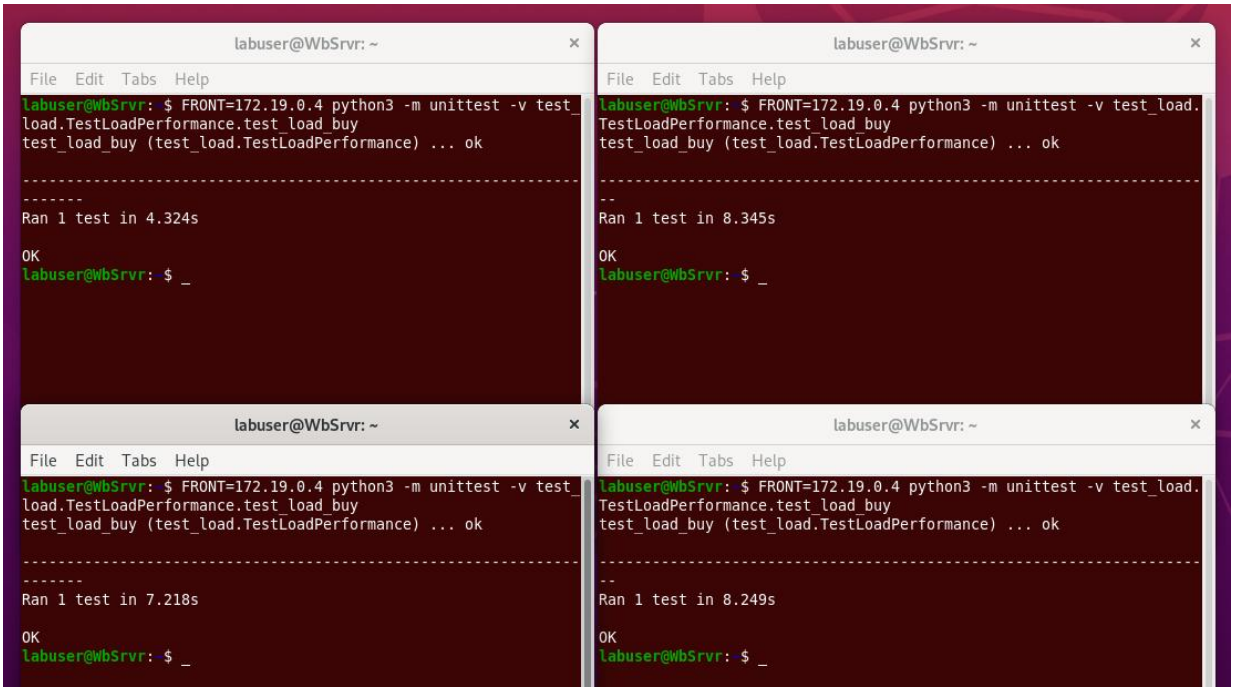
2 Client connected screenshot: terminal shows total latency of 100 Buy calls



3 Client connected screenshot: terminal shows total latency of 100 Buy calls



4 Client connected screenshot: terminal shows total latency of 100 Buy calls



5 Client connected screenshot: terminal shows total latency of 100 Buy calls