

COMPSCI 677 Spring 2022

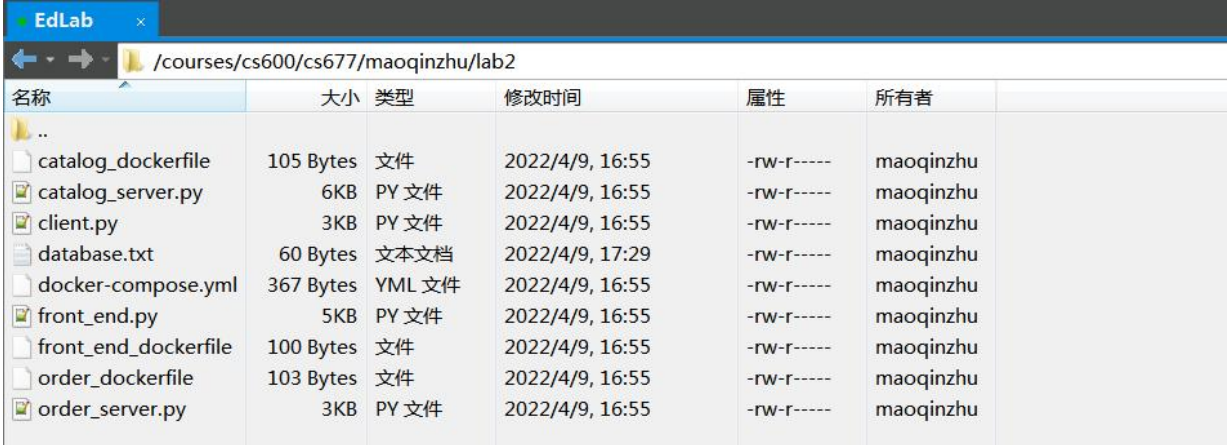
Lab 2: Tiered Microservices-Based Toy Store

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.

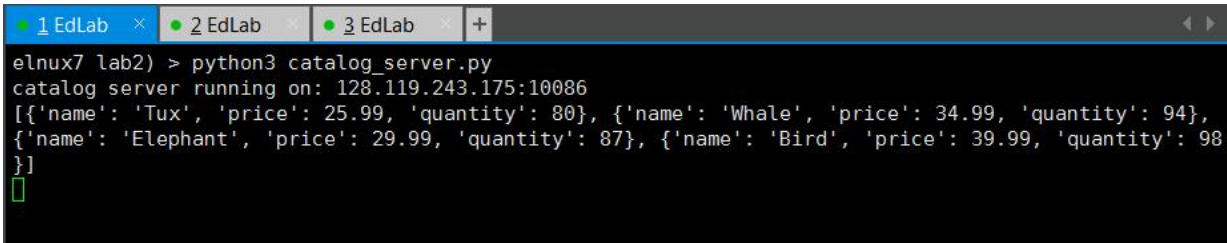


名称	大小	类型	修改时间	属性	所有者
..					
catalog_dockerfile	105 Bytes	文件	2022/4/9, 16:55	-rw-r----	maoqinzhu
catalog_server.py	6KB	PY 文件	2022/4/9, 16:55	-rw-r----	maoqinzhu
client.py	3KB	PY 文件	2022/4/9, 16:55	-rw-r----	maoqinzhu
database.txt	60 Bytes	文本文档	2022/4/9, 17:29	-rw-r----	maoqinzhu
docker-compose.yml	367 Bytes	YML 文件	2022/4/9, 16:55	-rw-r----	maoqinzhu
front_end.py	5KB	PY 文件	2022/4/9, 16:55	-rw-r----	maoqinzhu
front_end_dockerfile	100 Bytes	文件	2022/4/9, 16:55	-rw-r----	maoqinzhu
order_dockerfile	103 Bytes	文件	2022/4/9, 16:55	-rw-r----	maoqinzhu
order_server.py	3KB	PY 文件	2022/4/9, 16:55	-rw-r----	maoqinzhu

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



```
eLinux7 lab2) > python3 catalog_server.py
catalog server running on: 128.119.243.175:10086
[{'name': 'Tux', 'price': 25.99, 'quantity': 80}, {'name': 'Whale', 'price': 34.99, 'quantity': 94},
{'name': 'Elephant', 'price': 29.99, 'quantity': 87}, {'name': 'Bird', 'price': 39.99, 'quantity': 98}]
[]
```

Step2: start order server

\$ **CATALOG=128.119.243.175** python3 order_server.py

```
1 EdLab 2 EdLab 3 EdLab +
eLinux7 lab2) > CATALOG=128.119.243.175 python3 order_server.py
128.119.243.175
<class 'str'>
order server running on: 128.119.243.175:10010
█
```

Step3: start front-end server

\$ **CATALOG=128.119.243.175 ORDER=128.119.243.175** python3 front_end.py

```
1 EdLab 2 EdLab 3 EdLab +
eLinux7 lab2) > CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 front_end.py
128.119.243.175
128.119.243.175
<class 'str'>
front_end server running on: 128.119.243.175:6060
█
```

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
```

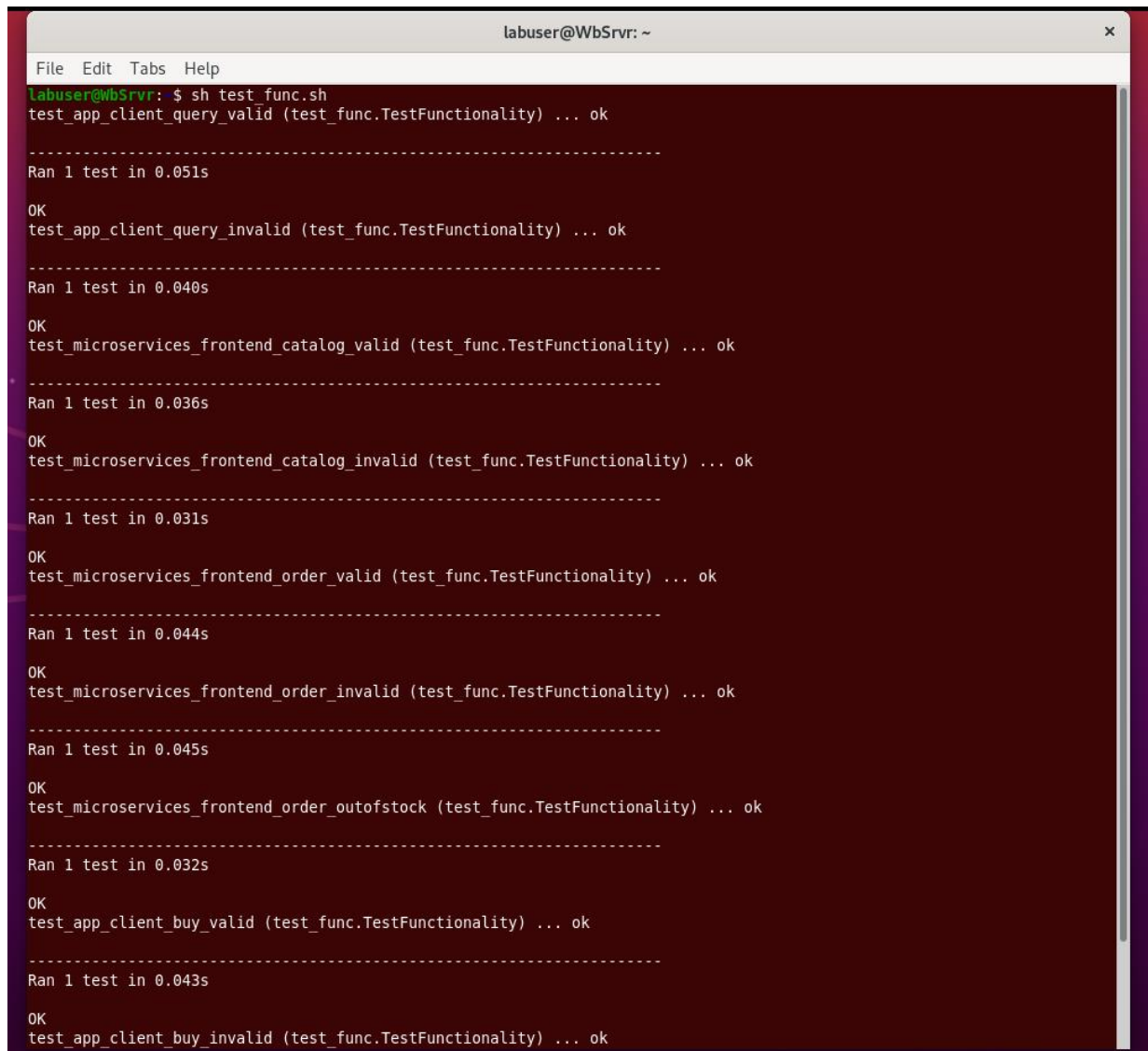
Looking at “**test_func.sh**”, this shell file will help us run all the 13 test cases.

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

```
test_func.sh front_end.py order_server.py catalog_server.py client.py database.txt test_func.py test_load.py
1 #!/bin/bash
2
3 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_app_client_query_valid
4 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_app_client_query_invalid
5 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_microservices_frontend_catalog_valid
6 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_microservices_frontend_catalog_invalid
7 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_microservices_frontend_order_valid
8 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_microservices_frontend_order_invalid
9 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_microservices_frontend_order_outofstock
10 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_app_client_buy_valid
11 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_app_client_buy_invalid
12 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_microservices_order_data_valid
13 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_microservices_order_data_invalid
14 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_microservices_order_data_outofstock
15 FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v test_func.TestFunctionality.test_microservices_order_data_outofstock
16
17 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.

A terminal window titled 'labuser@WbSrvr: ~' showing the execution of a script 'test_func.sh'. The script runs 13 tests, each passing with 'ok'. The tests are: test_app_client_query_valid, test_app_client_query_invalid, test_microservices_frontend_catalog_valid, test_microservices_frontend_catalog_invalid, test_microservices_frontend_order_valid, test_microservices_frontend_order_invalid, test_microservices_frontend_order_outofstock, test_app_client_buy_valid, and test_app_client_buy_invalid. Each test is preceded by a separator line of dashes and a 'Ran 1 test in' message with a timestamp.

```
labuser@WbSrvr: ~
File Edit Tabs Help
labuser@WbSrvr: $ sh test_func.sh
test_app_client_query_valid (test_func.TestFunctionality) ... ok
-----
Ran 1 test in 0.051s

OK
test_app_client_query_invalid (test_func.TestFunctionality) ... ok
-----
Ran 1 test in 0.040s

OK
test_microservices_frontend_catalog_valid (test_func.TestFunctionality) ... ok
-----
Ran 1 test in 0.036s

OK
test_microservices_frontend_catalog_invalid (test_func.TestFunctionality) ... ok
-----
Ran 1 test in 0.031s

OK
test_microservices_frontend_order_valid (test_func.TestFunctionality) ... ok
-----
Ran 1 test in 0.044s

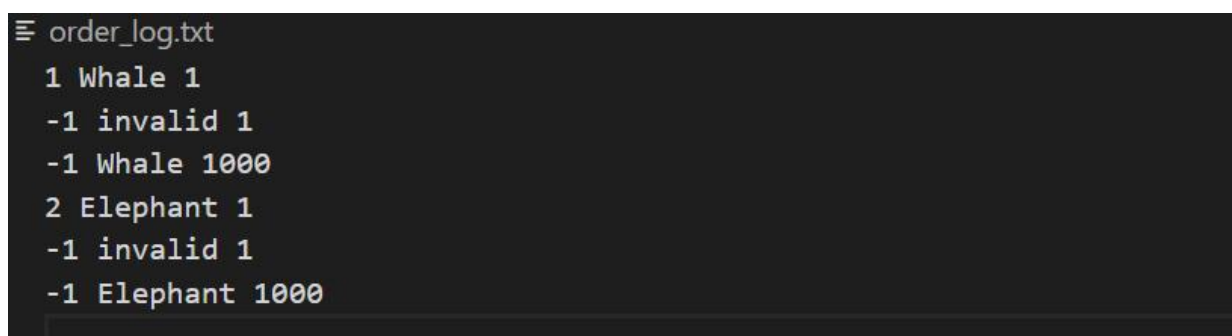
OK
test_microservices_frontend_order_invalid (test_func.TestFunctionality) ... ok
-----
Ran 1 test in 0.045s

OK
test_microservices_frontend_order_outofstock (test_func.TestFunctionality) ... ok
-----
Ran 1 test in 0.032s

OK
test_app_client_buy_valid (test_func.TestFunctionality) ... ok
-----
Ran 1 test in 0.043s

OK
test_app_client_buy_invalid (test_func.TestFunctionality) ... ok
```

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.

A dark-themed text editor showing the content of 'order_log.txt'. The file contains two entries for 'Whale' and two entries for 'Elephant'. Each entry has a quantity, an order ID (-1), and a price (1000).

```
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
```

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

```
1 EdLab x 2 EdLab x 3 EdLab x +
Connected to : 128.119.243.175 : 39852
Tux
GET /Tux HTTP/1.1
Host: 128.119.243.175:10086
User-Agent: python-requests/2.22.0
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive

Connected to : 128.119.243.175 : 39854
invalid
GET /invalid HTTP/1.1
Host: 128.119.243.175:10086
User-Agent: python-requests/2.22.0
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive

Connected to : 76.74.66.19 : 52797
Tux
GET /Tux HTTP/1.1
Host: 128.119.243.175:10086
User-Agent: python-requests/2.22.0
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive

Connected to : 76.74.66.19 : 52798
invalid
GET /invalid HTTP/1.1
Host: 128.119.243.175:10086
User-Agent: python-requests/2.22.0
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
```

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.

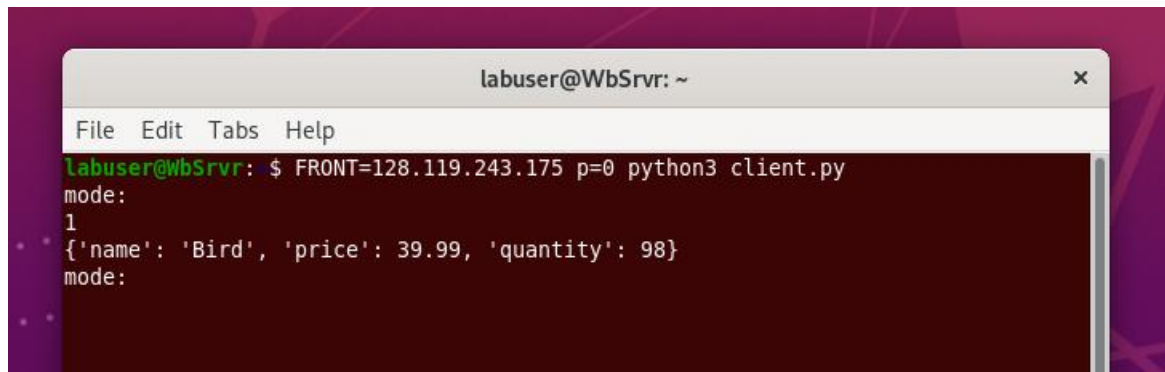
Mode 2: Initiate a serials of Query

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

Mode 3: Initiate a serials of Buy

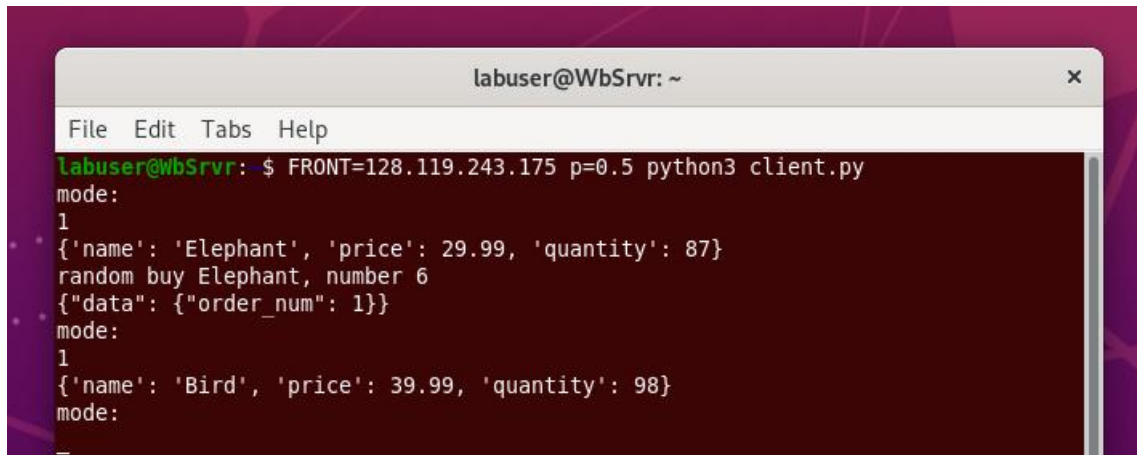
You can specify the toy name, quantity and number of requests as you want.

Mode 1 Examples:



A terminal window titled 'labuser@WbSrvr: ~' with a menu bar (File, Edit, Tabs, Help). The prompt is 'labuser@WbSrvr: \$'. The command entered is 'FRONT=128.119.243.175 p=0 python3 client.py'. The output shows 'mode: 1' followed by a JSON object: {'name': 'Bird', 'price': 39.99, 'quantity': 98}. The prompt 'mode:' is shown again.

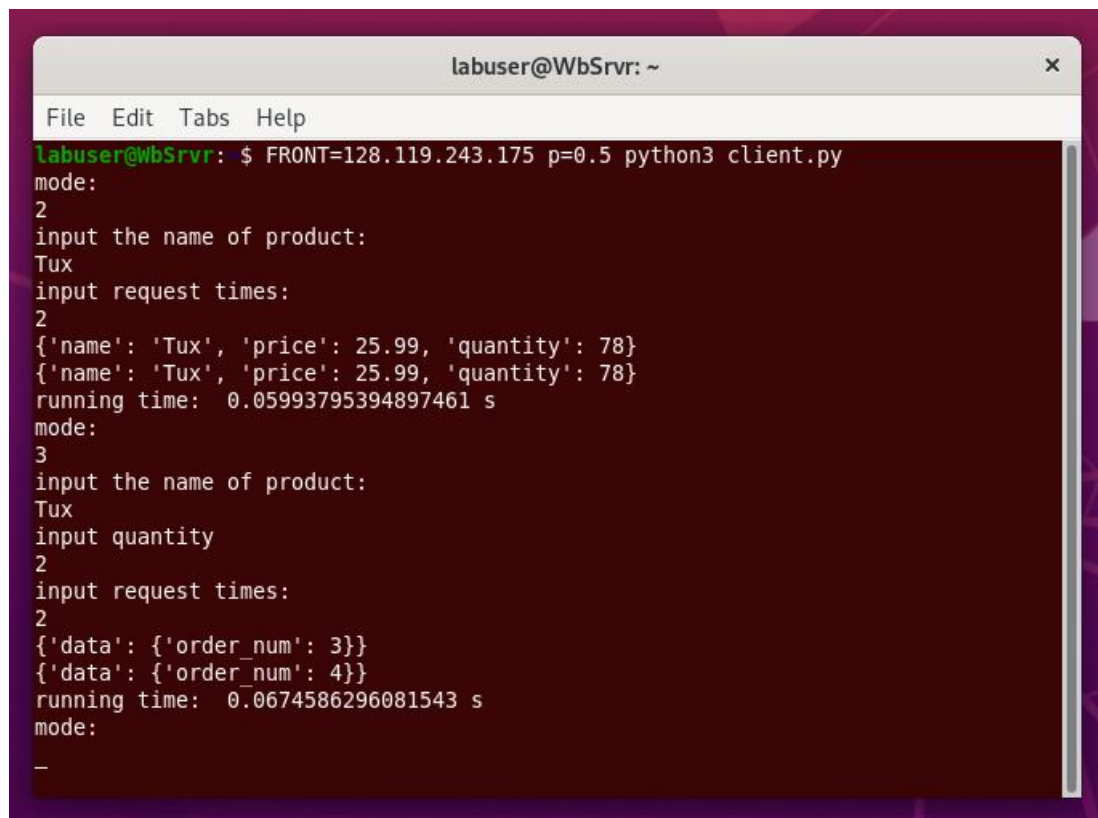
```
labuser@WbSrvr: ~
File Edit Tabs Help
labuser@WbSrvr: $ FRONT=128.119.243.175 p=0 python3 client.py
mode:
1
{'name': 'Bird', 'price': 39.99, 'quantity': 98}
mode:
```



A terminal window titled 'labuser@WbSrvr: ~' with a menu bar (File, Edit, Tabs, Help). The prompt is 'labuser@WbSrvr: \$'. The command entered is 'FRONT=128.119.243.175 p=0.5 python3 client.py'. The output shows 'mode: 1' followed by a JSON object: {'name': 'Elephant', 'price': 29.99, 'quantity': 87}. Then it says 'random buy Elephant, number 6' followed by a JSON object: {'data': {'order_num': 1}}. The prompt 'mode:' is shown again, followed by 'mode: 1' and another JSON object: {'name': 'Bird', 'price': 39.99, 'quantity': 98}. The prompt 'mode:' is shown again.

```
labuser@WbSrvr: ~
File Edit Tabs Help
labuser@WbSrvr: $ FRONT=128.119.243.175 p=0.5 python3 client.py
mode:
1
{'name': 'Elephant', 'price': 29.99, 'quantity': 87}
random buy Elephant, number 6
{"data": {"order_num": 1}}
mode:
1
{'name': 'Bird', 'price': 39.99, 'quantity': 98}
mode:
```

Mode 2 & Mode 3 Examples:



A terminal window titled 'labuser@WbSrvr: ~' with a menu bar (File, Edit, Tabs, Help). The prompt is 'labuser@WbSrvr: \$'. The command entered is 'FRONT=128.119.243.175 p=0.5 python3 client.py'. The output shows 'mode: 2', then 'input the name of product:' followed by 'Tux', then 'input request times:' followed by '2'. Then it shows two identical JSON objects: {'name': 'Tux', 'price': 25.99, 'quantity': 78}. Then it shows 'running time: 0.05993795394897461 s'. The prompt 'mode:' is shown again, followed by 'mode: 3', then 'input the name of product:' followed by 'Tux', then 'input quantity' followed by '2', then 'input request times:' followed by '2'. Then it shows two JSON objects: {'data': {'order_num': 3}} and {'data': {'order_num': 4}}. Then it shows 'running time: 0.0674586296081543 s'. The prompt 'mode:' is shown again.

```
labuser@WbSrvr: ~
File Edit Tabs Help
labuser@WbSrvr: $ FRONT=128.119.243.175 p=0.5 python3 client.py
mode:
2
input the name of product:
Tux
input request times:
2
{'name': 'Tux', 'price': 25.99, 'quantity': 78}
{'name': 'Tux', 'price': 25.99, 'quantity': 78}
running time: 0.05993795394897461 s
mode:
3
input the name of product:
Tux
input quantity
2
input request times:
2
{'data': {'order_num': 3}}
{'data': {'order_num': 4}}
running time: 0.0674586296081543 s
mode:
```

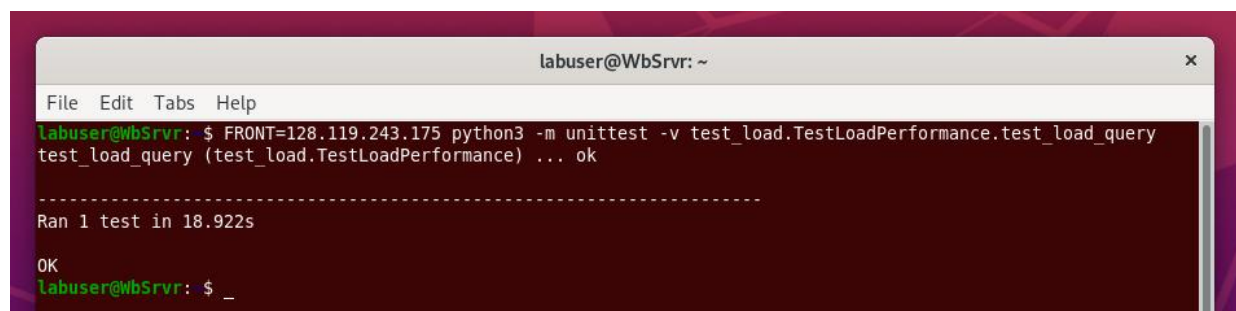
3. Load Test Output

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:

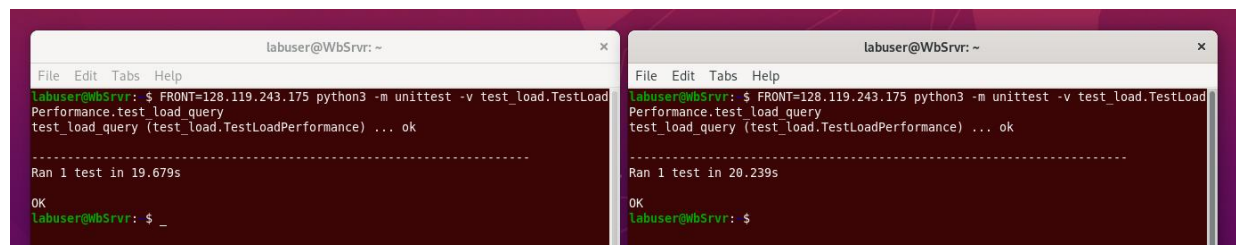
\$ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query

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



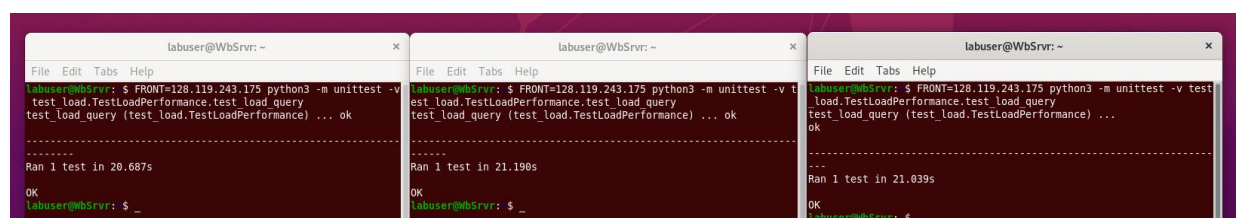
```
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
-----  
Ran 1 test in 18.922s  
OK  
labuser@WbSrvr: $ _
```

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



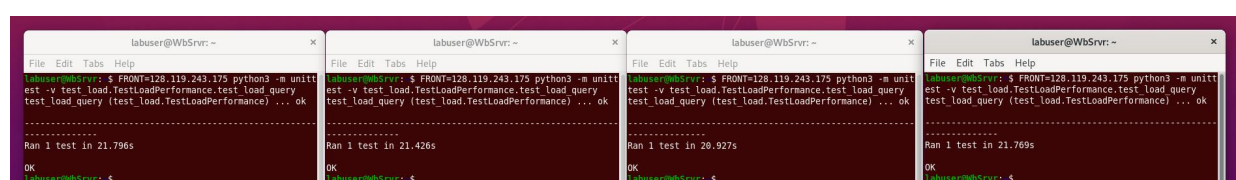
```
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
-----  
Ran 1 test in 19.679s  
OK  
labuser@WbSrvr: $ _  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
-----  
Ran 1 test in 20.239s  
OK  
labuser@WbSrvr: $
```

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



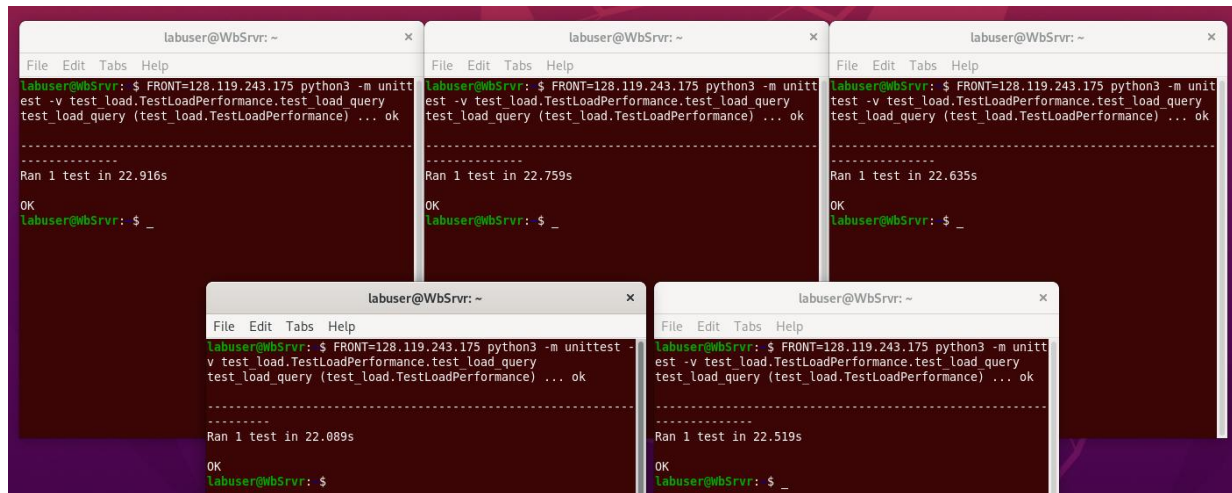
```
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
-----  
Ran 1 test in 20.687s  
OK  
labuser@WbSrvr: $ _  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
-----  
Ran 1 test in 21.198s  
OK  
labuser@WbSrvr: $ _  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
-----  
Ran 1 test in 21.039s  
OK  
labuser@WbSrvr: $
```

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



```
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
-----  
Ran 1 test in 21.796s  
OK  
labuser@WbSrvr: $ _  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
-----  
Ran 1 test in 21.426s  
OK  
labuser@WbSrvr: $ _  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
-----  
Ran 1 test in 20.927s  
OK  
labuser@WbSrvr: $ _  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=128.119.243.175 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
-----  
Ran 1 test in 21.769s  
OK  
labuser@WbSrvr: $
```

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



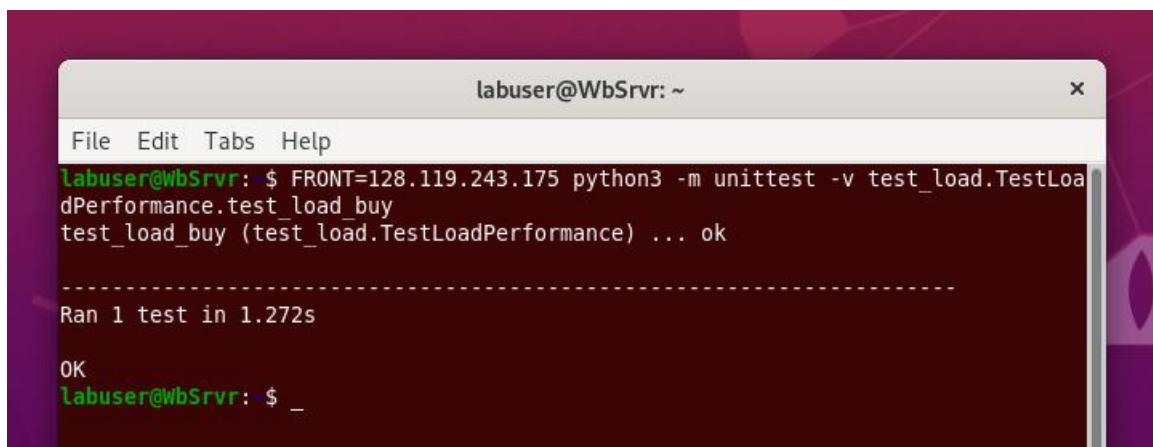
We have analyzed the average latency of each request in different scenarios in evaluation document. Please check out the details there.

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**.

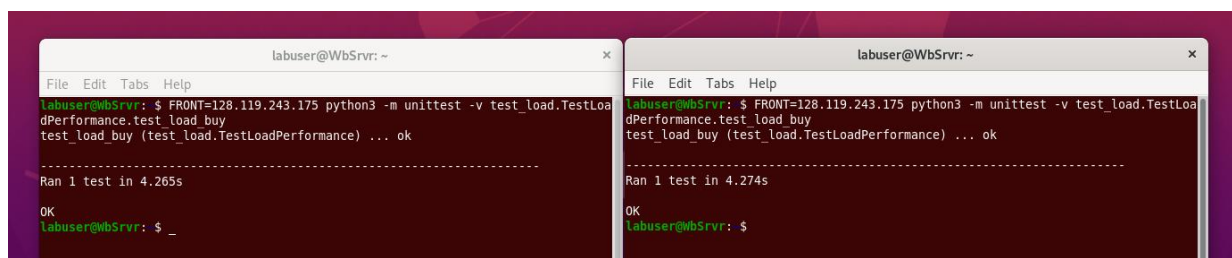
Here we vary the number of clients from 1 to 5 and measure the total latency as the load goes up. For each client terminal, we type the following command:

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

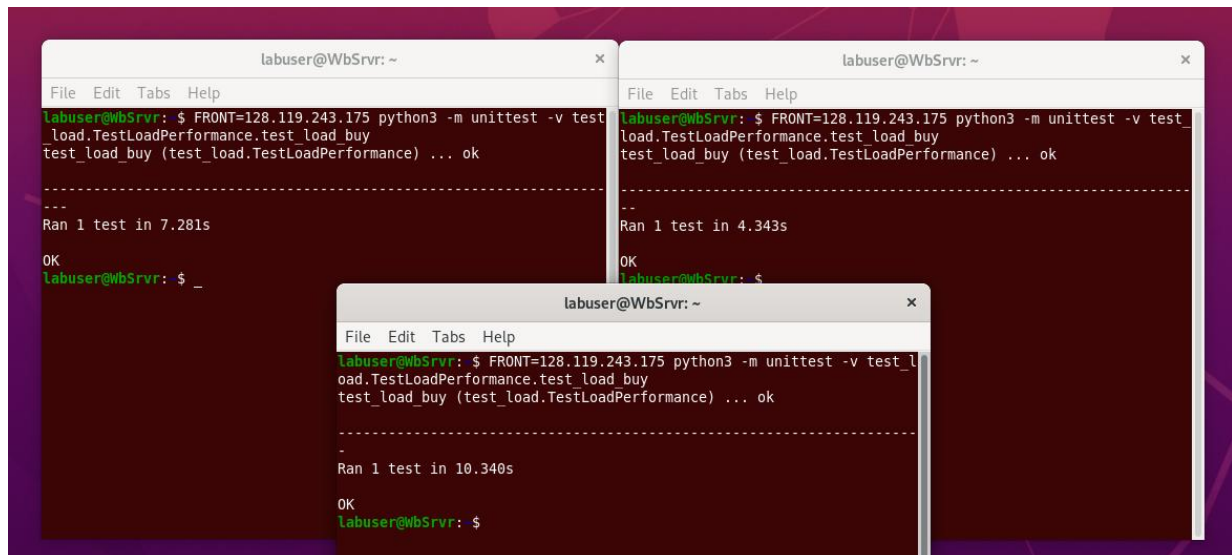
1 Client connected screenshot: terminal shows total latency of 100 Buy calls



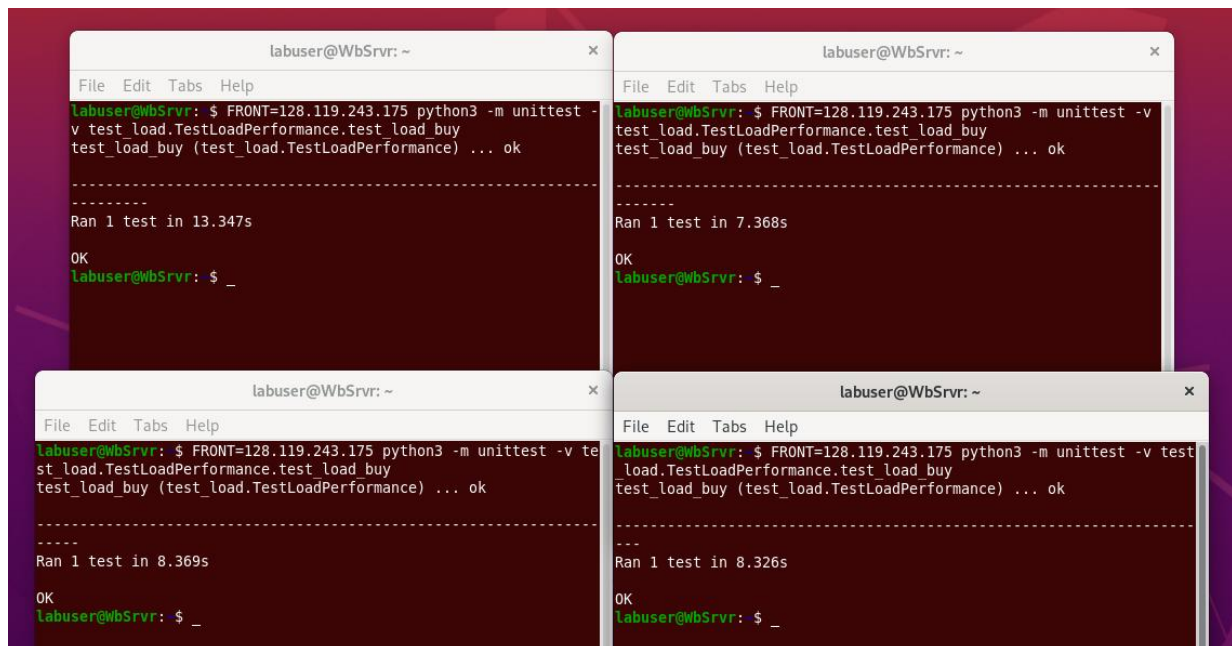
2 Clients connected screenshot: terminal shows total latency of 100 Buy calls



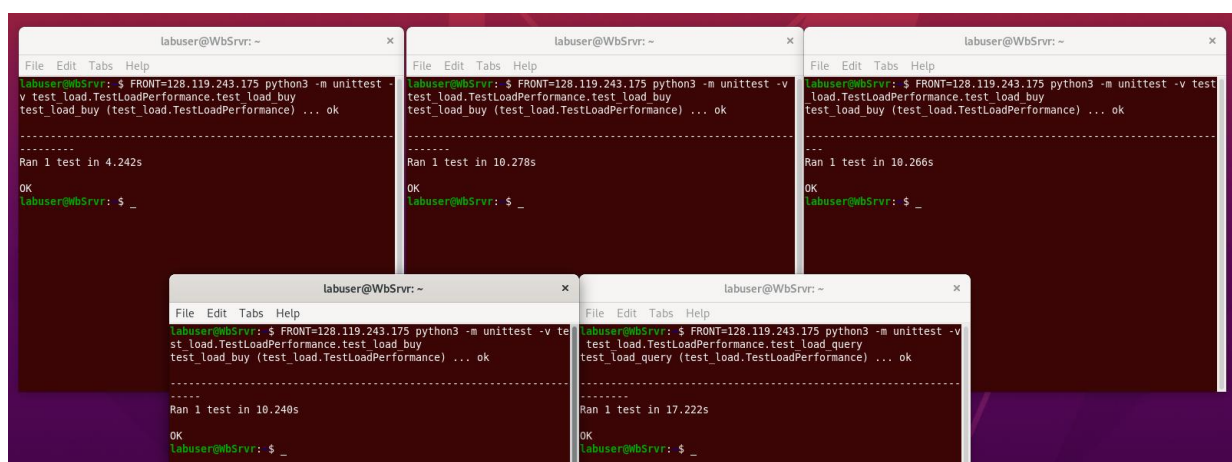
3 Clients connected screenshot: terminal shows total latency of 100 Buy calls



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

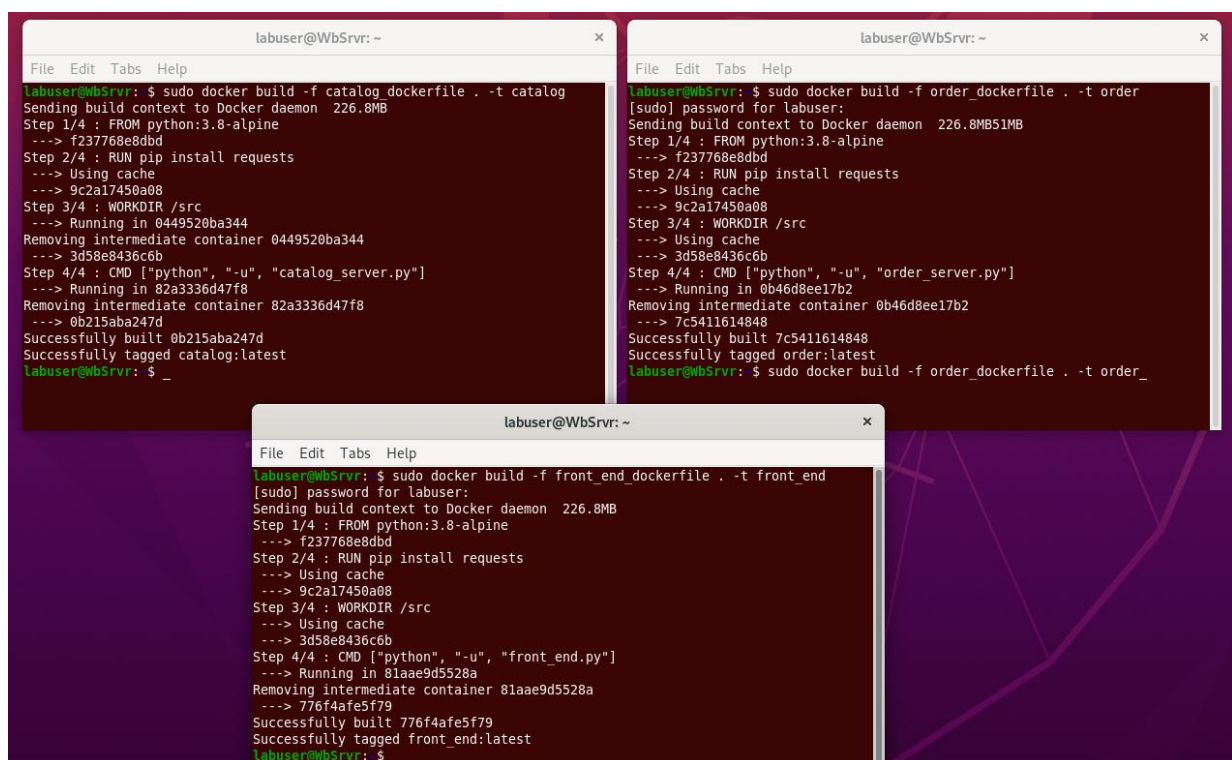
Consider that we can not containerize our application using Docker on remote Edlab server. So this time we provide a local test demo.

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
```

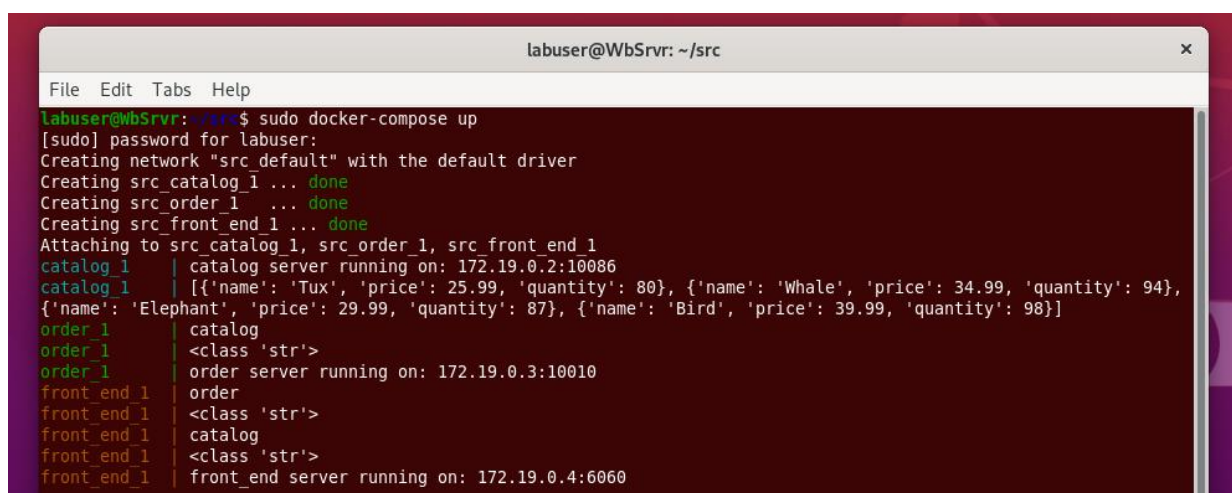


The image shows three terminal windows from a user named 'labuser' on a host named 'WbSrvr'. Each window shows the execution of a 'sudo docker build' command for a different service. The first window shows the build for 'catalog', the second for 'order', and the third for 'front_end'. Each build process follows a similar pattern: sending build context, pulling the base image (python:3.8-alpine), installing dependencies with pip, and running the application. All three builds are successful and tagged as 'latest'.

```
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ sudo docker build -f catalog_dockerfile . -t catalog  
Sending build context to Docker daemon 226.8MB  
Step 1/4 : FROM python:3.8-alpine  
----> f237768e8dbd  
Step 2/4 : RUN pip install requests  
----> Using cache  
----> 9c2a17450a08  
Step 3/4 : WORKDIR /src  
----> Running in 0449520ba344  
Removing intermediate container 0449520ba344  
----> 3d58e8436c6b  
Step 4/4 : CMD ["python", "-u", "catalog_server.py"]  
----> Running in 82a3336d47f8  
Removing intermediate container 82a3336d47f8  
----> 0b215aba247d  
Successfully built 0b215aba247d  
Successfully tagged catalog:latest  
labuser@WbSrvr: $  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ sudo docker build -f order_dockerfile . -t order  
[sudo] password for labuser:  
Sending build context to Docker daemon 226.8MB51MB  
Step 1/4 : FROM python:3.8-alpine  
----> f237768e8dbd  
Step 2/4 : RUN pip install requests  
----> Using cache  
----> 9c2a17450a08  
Step 3/4 : WORKDIR /src  
----> Using cache  
----> 3d58e8436c6b  
Step 4/4 : CMD ["python", "-u", "order_server.py"]  
----> Running in 0b46d8ee17b2  
Removing intermediate container 0b46d8ee17b2  
----> 7c5411614848  
Successfully built 7c5411614848  
Successfully tagged order:latest  
labuser@WbSrvr: $ sudo docker build -f order_dockerfile . -t order_  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ sudo docker build -f front_end_dockerfile . -t front_end  
[sudo] password for labuser:  
Sending build context to Docker daemon 226.8MB  
Step 1/4 : FROM python:3.8-alpine  
----> f237768e8dbd  
Step 2/4 : RUN pip install requests  
----> Using cache  
----> 9c2a17450a08  
Step 3/4 : WORKDIR /src  
----> Using cache  
----> 3d58e8436c6b  
Step 4/4 : CMD ["python", "-u", "front_end.py"]  
----> Running in 81aae9d5528a  
Removing intermediate container 81aae9d5528a  
----> 776f4afe5f79  
Successfully built 776f4afe5f79  
Successfully tagged front_end:latest  
labuser@WbSrvr: $
```

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

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



The image shows a terminal window from a user named 'labuser' on a host named 'WbSrvr'. The user has navigated to the 'src' directory and executed the 'sudo docker-compose up' command. The output shows the creation of a network named 'src default' and the creation of three containers: 'src_catalog_1', 'src_order_1', and 'src_front_end_1'. The containers are attached to the network and their IP addresses are displayed. The output also shows the command used to run each container and the IP address of the host.

```
labuser@WbSrvr: ~/src  
File Edit Tabs Help  
labuser@WbSrvr: ~/src$ sudo docker-compose up  
[sudo] password for labuser:  
Creating network "src default" with the default driver  
Creating src_catalog_1 ... done  
Creating src_order_1 ... done  
Creating src_front_end_1 ... done  
Attaching to src_catalog_1, src_order_1, src_front_end_1  
catalog_1 | catalog server running on: 172.19.0.2:10086  
catalog_1 | [{ 'name': 'Tux', 'price': 25.99, 'quantity': 80 }, { 'name': 'Whale', 'price': 34.99, 'quantity': 94 },  
catalog_1 | { 'name': 'Elephant', 'price': 29.99, 'quantity': 87 }, { 'name': 'Bird', 'price': 39.99, 'quantity': 98 }]  
order_1 | catalog  
order_1 | <class 'str'>  
order_1 | order server running on: 172.19.0.3:10010  
front_end_1 | order  
front_end_1 | <class 'str'>  
front_end_1 | catalog  
front_end_1 | <class 'str'>  
front_end_1 | front_end server running on: 172.19.0.4:6060
```

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!!!

```
1 #!/bin/bash
2
3 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
4 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
5 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
6 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
7 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
8 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
9 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
10 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
11 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
12 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
13 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_data_valid
14 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_data_invalid
15 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_data_outofstock
16
17 exec /bin/bash
18
```

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_data_valid
test_microservices_order_data_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_data_invalid
test_microservices_order_data_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_data_outofstock
test_microservices_order_data_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:
```

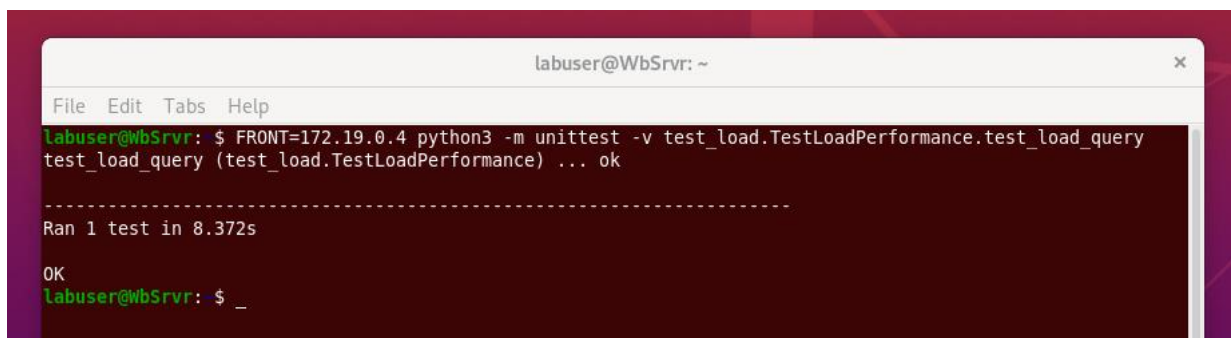
3. Load Test Output

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

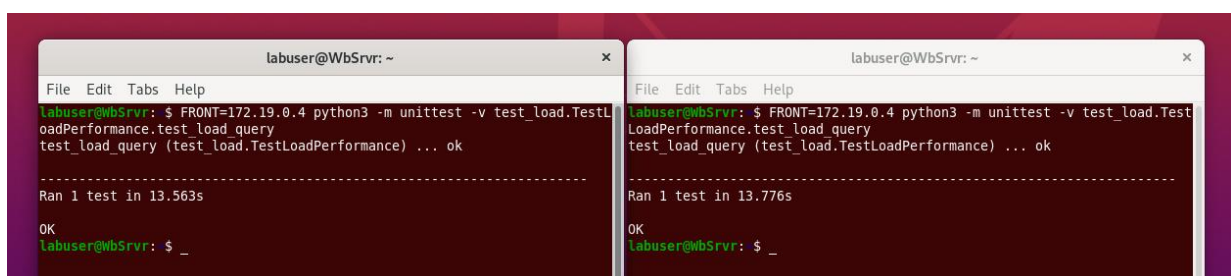


```
labuser@WbSrvr: ~
File Edit Tabs Help
labuser@WbSrvr: $ FRONT=172.19.0.4 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query
test_load_query (test_load.TestLoadPerformance) ... ok

-----
Ran 1 test in 8.372s

OK
labuser@WbSrvr: $ _
```

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



```
labuser@WbSrvr: ~
File Edit Tabs Help
labuser@WbSrvr: $ FRONT=172.19.0.4 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query
test_load_query (test_load.TestLoadPerformance) ... ok

-----
Ran 1 test in 13.563s

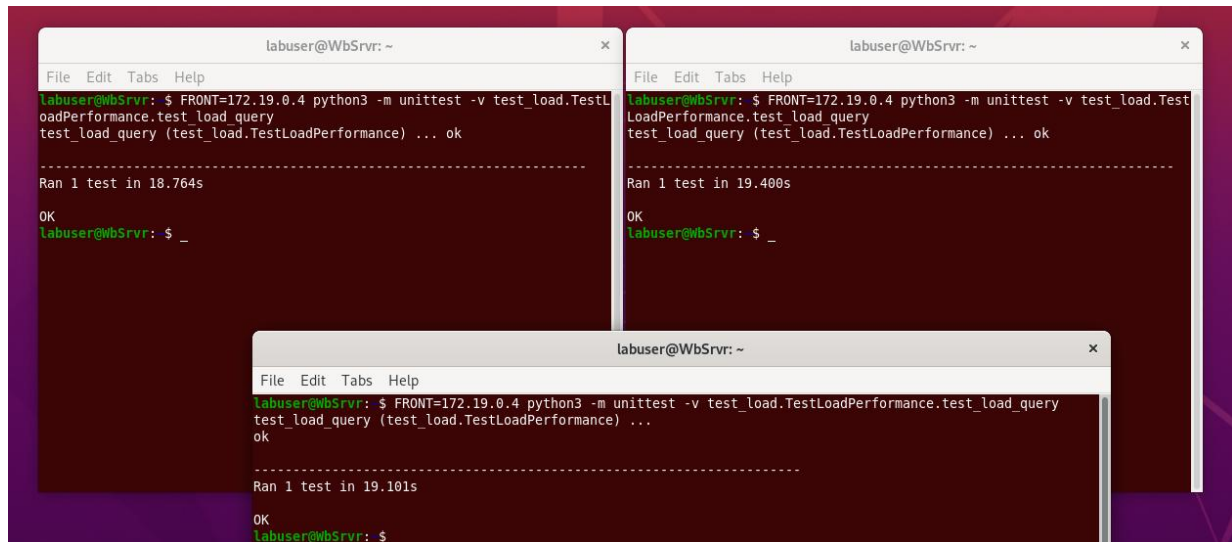
OK
labuser@WbSrvr: $ _

labuser@WbSrvr: ~
File Edit Tabs Help
labuser@WbSrvr: $ FRONT=172.19.0.4 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query
test_load_query (test_load.TestLoadPerformance) ... ok

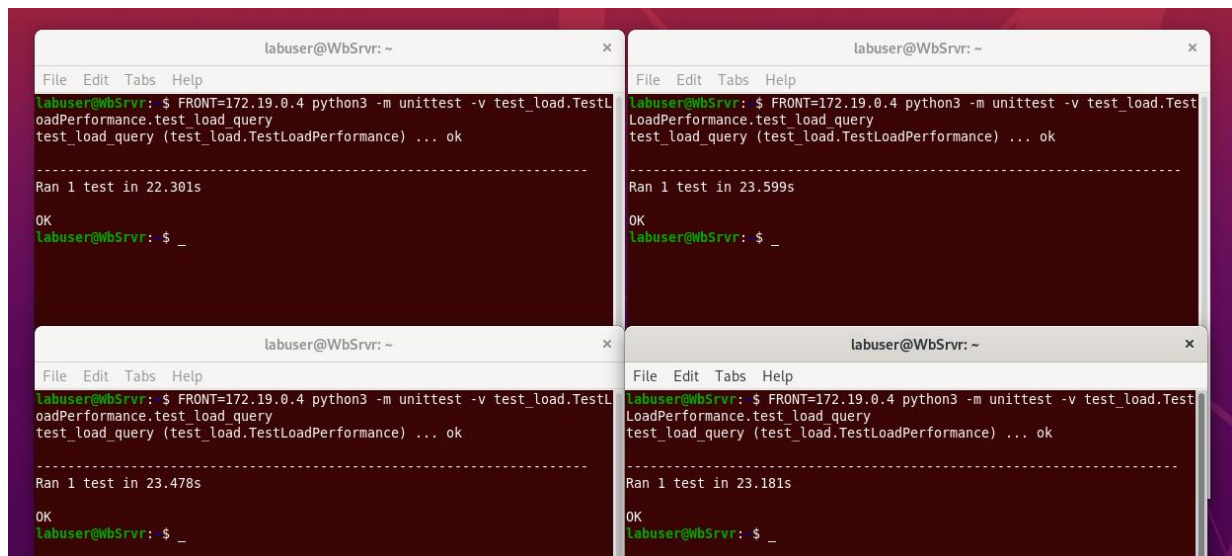
-----
Ran 1 test in 13.776s

OK
labuser@WbSrvr: $ _
```

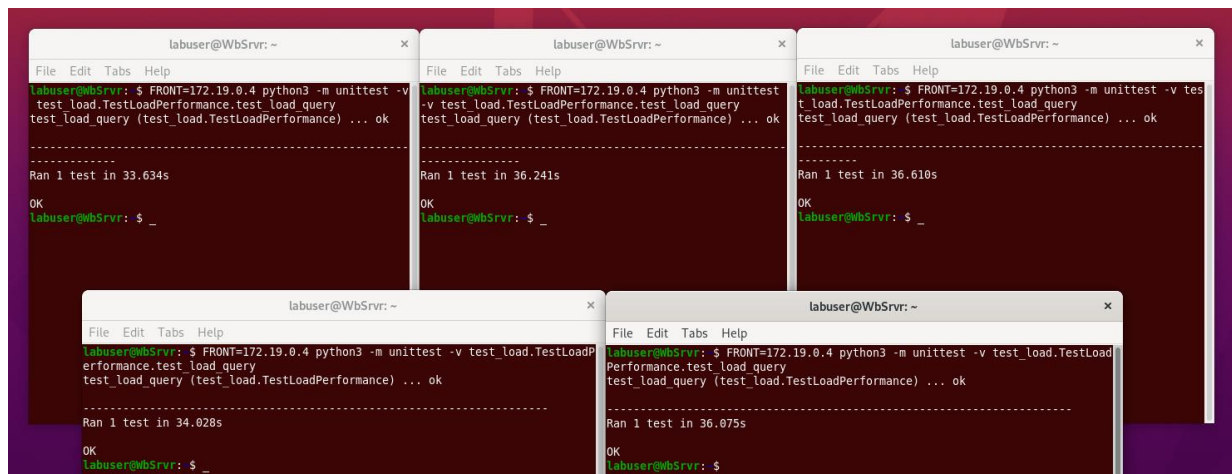

3 Client connected screenshot: terminal shows total latency of 1000 Query calls



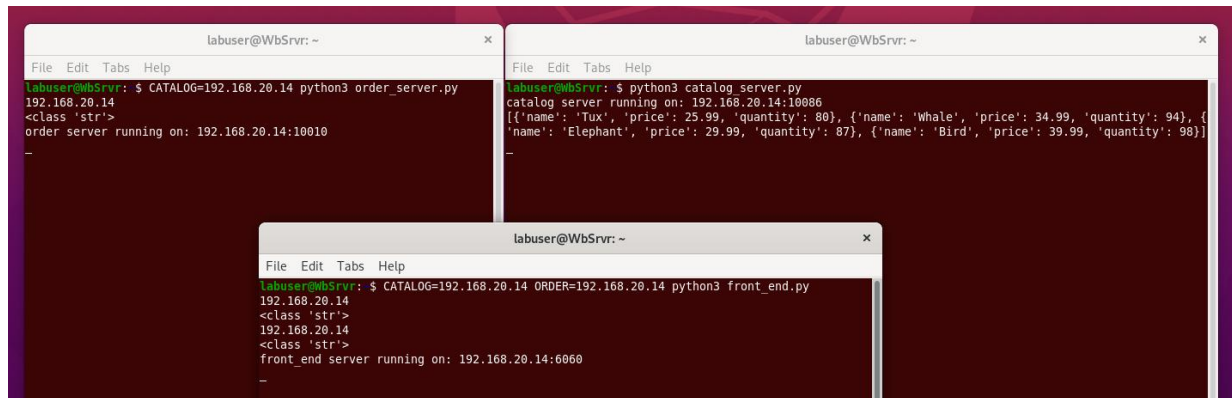
4 Client connected screenshot: terminal shows total latency of 1000 Query calls



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

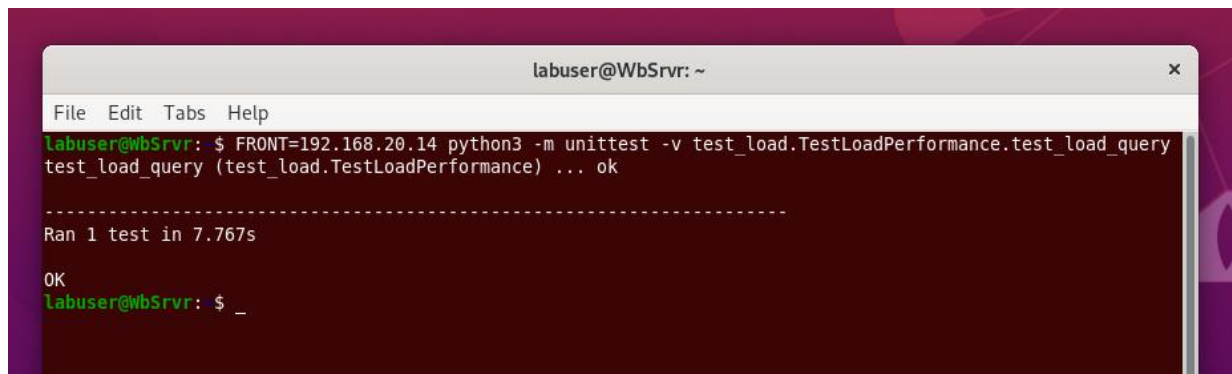


Local Test Without Containers- Since we cannot containerize our application using Docker on remote EdLab server. In order to find out if virtualization add any overheads, we do the local query load test without containers.



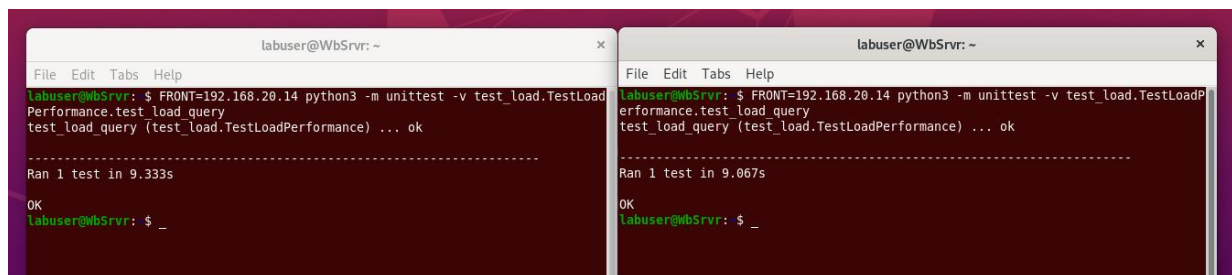
```
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ CATALOG=192.168.20.14 python3 order_server.py  
192.168.20.14  
<class 'str'>  
order server running on: 192.168.20.14:10010  
-  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ python3 catalog_server.py  
catalog server running on: 192.168.20.14:10086  
[{'name': 'Tux', 'price': 25.99, 'quantity': 80}, {'name': 'Whale', 'price': 34.99, 'quantity': 94}, {'name': 'Elephant', 'price': 29.99, 'quantity': 87}, {'name': 'Bird', 'price': 39.99, 'quantity': 98}]  
-  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ CATALOG=192.168.20.14 ORDER=192.168.20.14 python3 front_end.py  
192.168.20.14  
<class 'str'>  
192.168.20.14  
<class 'str'>  
front_end server running on: 192.168.20.14:6060  
-
```

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



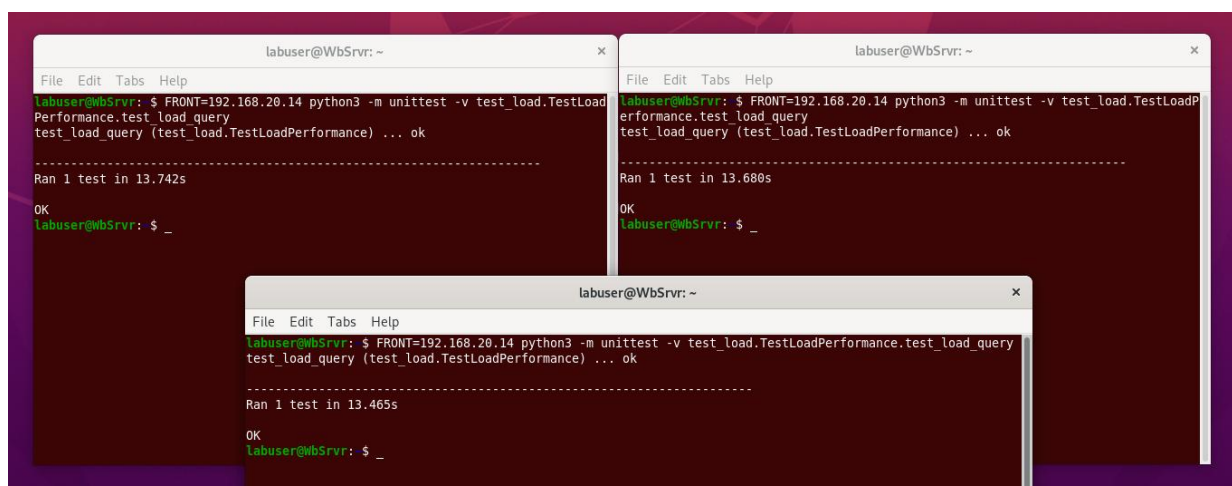
```
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=192.168.20.14 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
  
-----  
Ran 1 test in 7.767s  
  
OK  
labuser@WbSrvr: $ _
```

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



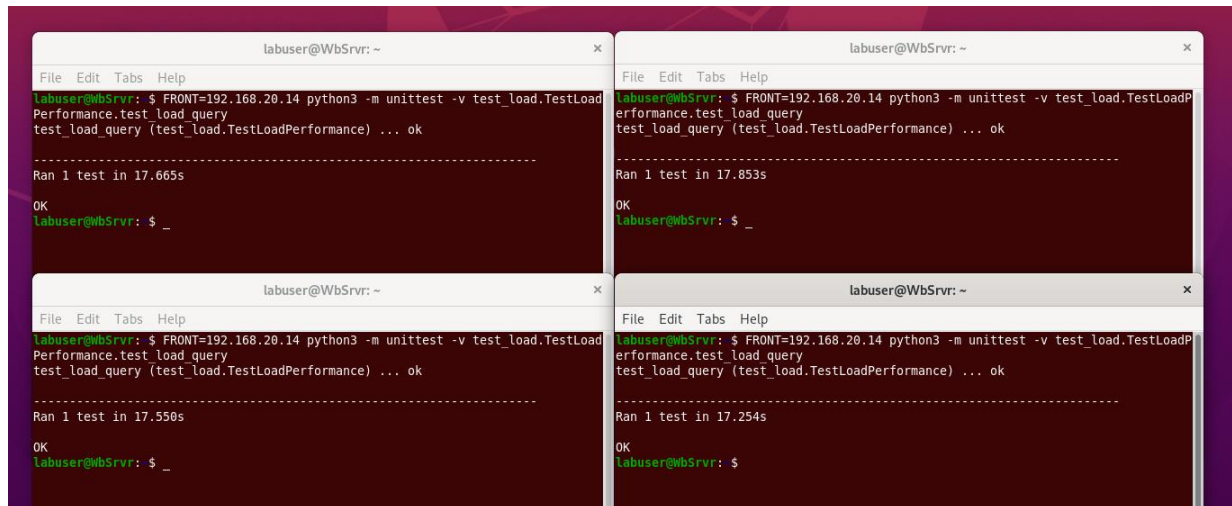
```
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=192.168.20.14 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
  
-----  
Ran 1 test in 9.333s  
  
OK  
labuser@WbSrvr: $ _  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=192.168.20.14 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
  
-----  
Ran 1 test in 9.067s  
  
OK  
labuser@WbSrvr: $ _
```

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

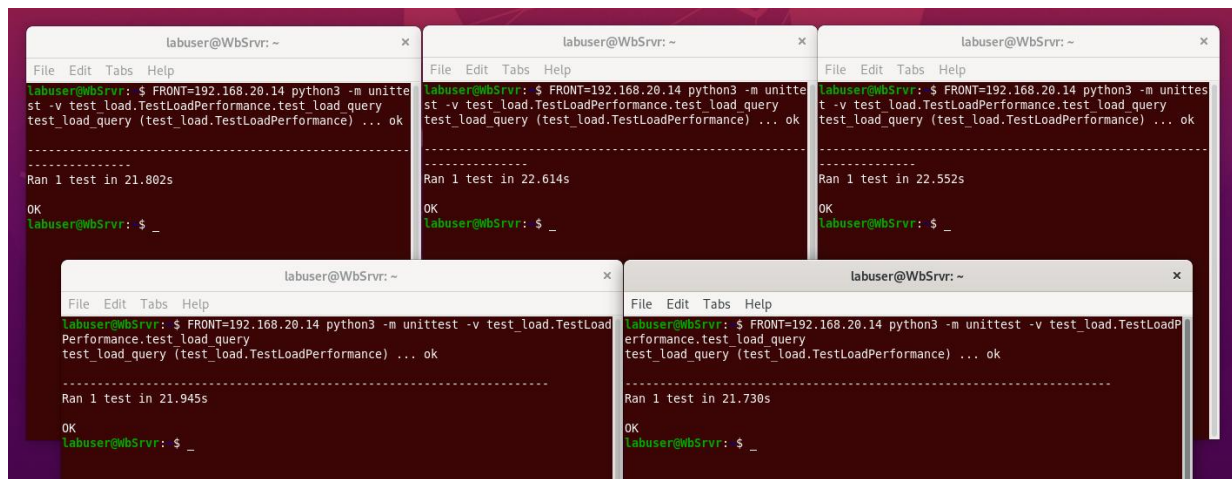


```
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=192.168.20.14 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
  
-----  
Ran 1 test in 13.742s  
  
OK  
labuser@WbSrvr: $ _  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=192.168.20.14 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
  
-----  
Ran 1 test in 13.680s  
  
OK  
labuser@WbSrvr: $ _  
  
labuser@WbSrvr: ~  
File Edit Tabs Help  
labuser@WbSrvr: $ FRONT=192.168.20.14 python3 -m unittest -v test_load.TestLoadPerformance.test_load_query  
test_load_query (test_load.TestLoadPerformance) ... ok  
  
-----  
Ran 1 test in 13.465s  
  
OK  
labuser@WbSrvr: $ _
```

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



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



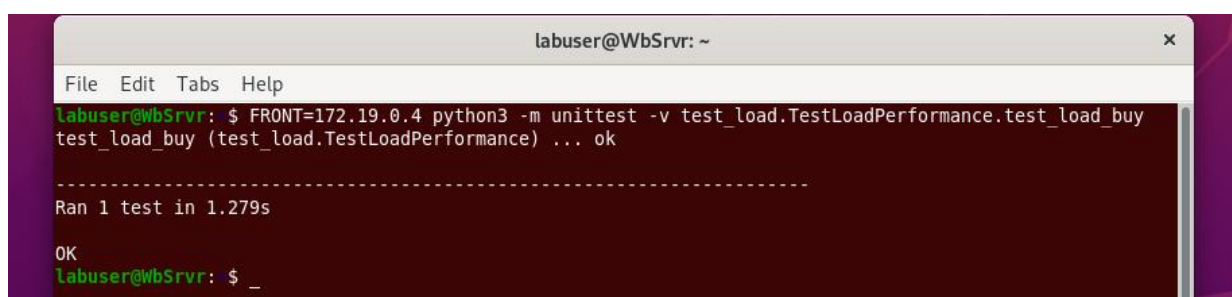
We have analyzed the average latency of each request in different scenarios in evaluation document. Please check out the details there.

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**.

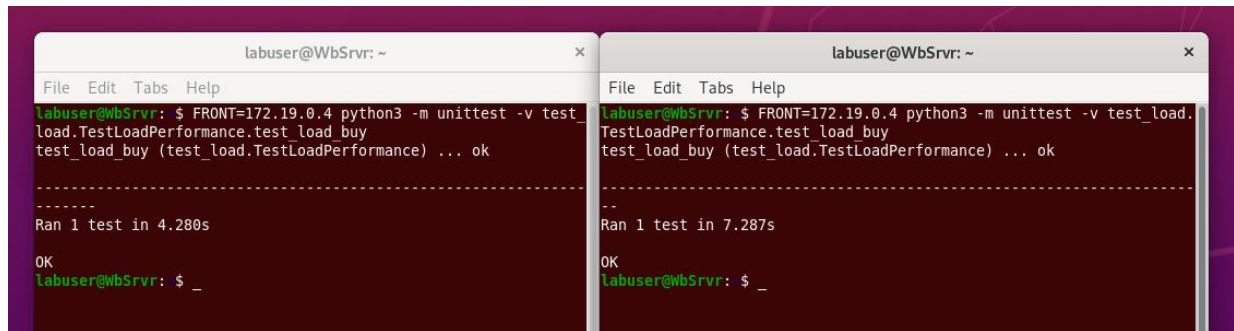
Here we vary the number of clients from 1 to 5 and measure the total latency as the load goes up. For each client terminal, we type the following command:

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

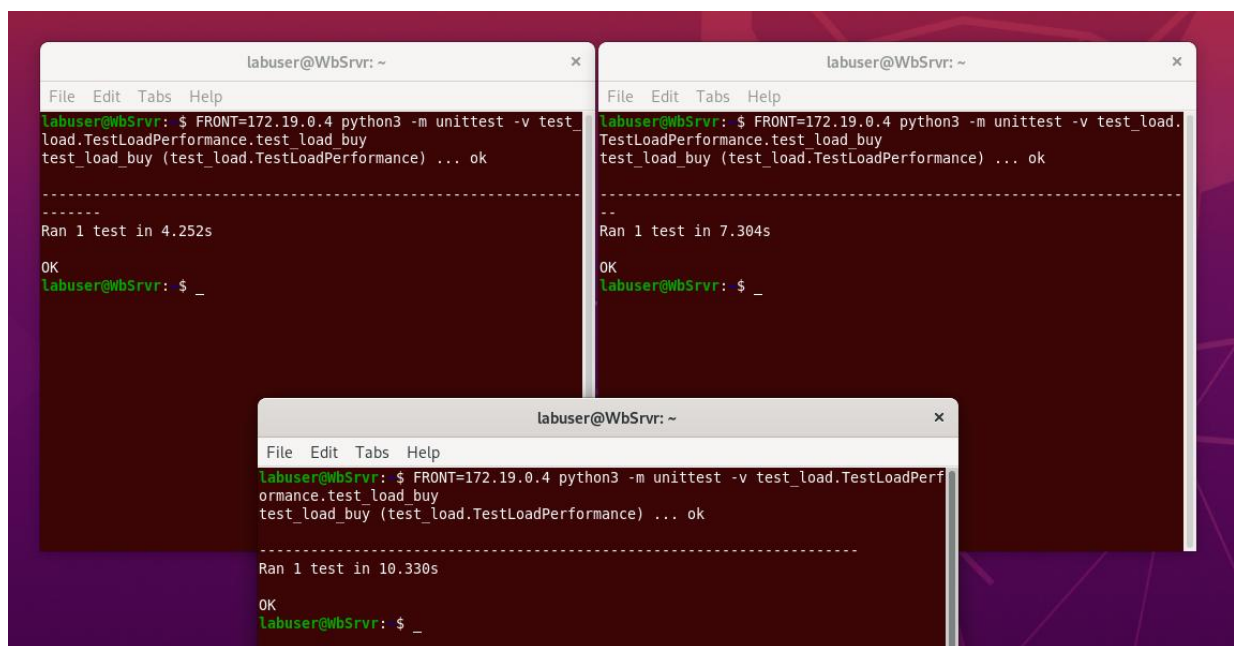
1 Client connected screenshot: terminal shows total latency of 100 Buy calls



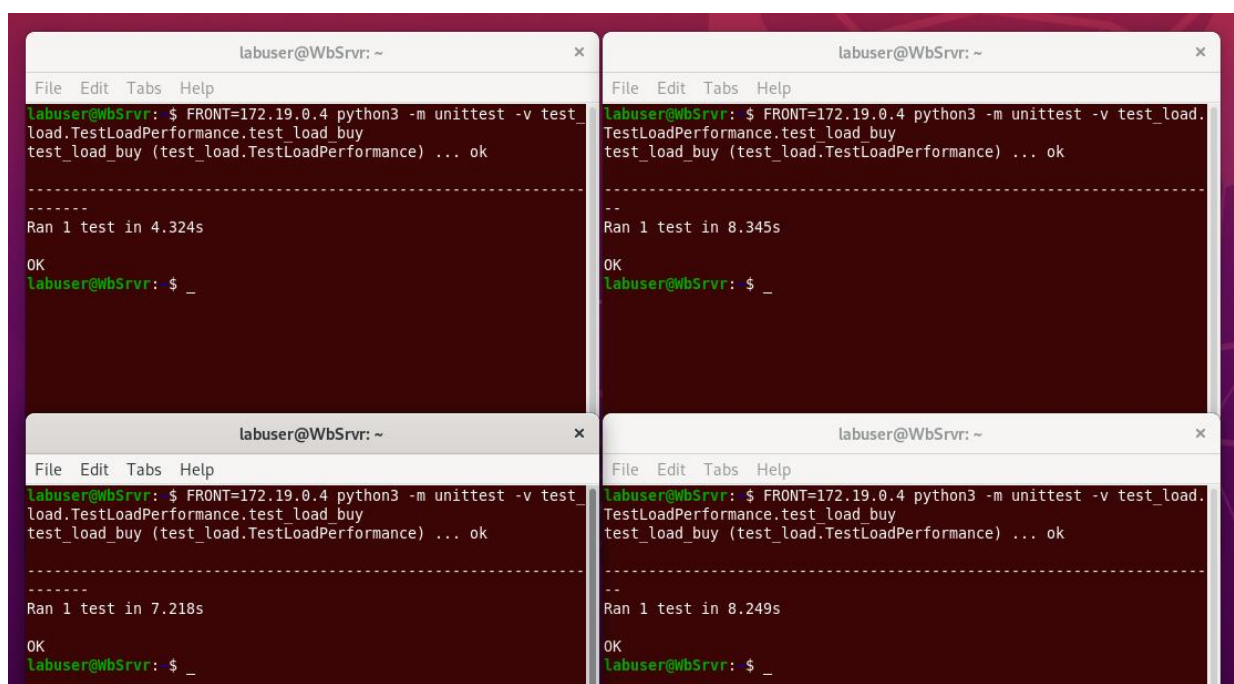
2 Client connected screenshot: terminal shows total latency of 100 Buy calls



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

