

Assignment 2 - submitted by Prakriti Dave

A little explanation about my code:

Client side:

1. **PrakMain:** is where I call the function to execute all my phases and also start the thread that reads from a blocking queue all the metrics and writes to a csv file.
2. **MyRunClass:** is the class that implements runnable and executes each thread as per the iterations.
3. **MyMetrics:** is just a class that stores the metrics recorded for each request response, such as timestamp and latencies
4. **MyFirstClient:** is where all the requests are made to the server using webTarget and Client

Server:

1. **NewConnectionClass:** is the class where I configure my DBCP configuration using C3P0 library:

```
public class NewConnectionClass {  
    private static ComboPooledDataSource cpds = null;  
    static{  
        try{  
            cpds = new ComboPooledDataSource();  
            cpds.setDriverClass("com.mysql.jdbc.Driver");  
            cpds.setJdbcUrl("jdbc:mysql://prakschema.ctg53dqnivdw.us-west-2.rds.amazonaws.com:3306/prakschema");  
            cpds.setUser("prakriti");  
            cpds.setPassword("prakriti");  
            // Optional Settings  
            cpds.setMinPoolSize(50);  
            cpds.setInitialPoolSize(50);  
            cpds.setAcquireIncrement(25);  
            cpds.setMaxPoolSize(248);  
            cpds.setMaxStatementsPerConnection(8);  
            cpds.setStatementCacheNumDeferredCloseThreads(1);  
            //  
            //cpds.setMaxStatements(8);  
            //cpds.setNumHelperThreads(10);  
            //cpds.setTestConnectionOnCheckin(true);  
            //cpds.setTestConnectionOnCheckout(false);  
            //cpds.setMaxConnectionAge(28);  
  
            }catch (Exception e){  
                e.printStackTrace();  
            }  
    }  
  
    public static ComboPooledDataSource getDataSource() { return cpds; }  
}
```

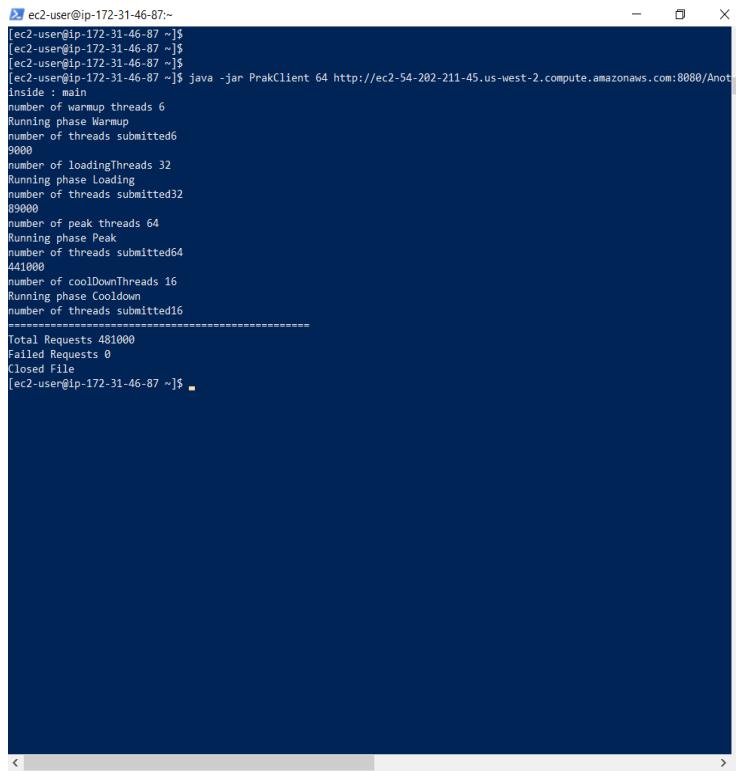
2. **UserDao:** creates a connection object interacts with the Database to perform relevant task.
3. **MyFirstApp:** class is where all the REST end points are defined.

Also, I would like to mention I used a python script to generate all the stats and plot the graph.

Step 1.

Results for a test run with the default client settings for threads/tests, 64 threads, 100 tests/phase

I ran my client on ec2 instance, primarily because I had network issues while running it locally for 256 threads and I would keep getting connection time out. There was quite an improvement on Wall Time After moving my client from local to ec2.



The screenshot shows a terminal window with the following content:

```
[ec2-user@ip-172-31-46-87 ~]$ java -jar PrakClient 64 http://ec2-54-202-211-45.us-west-2.compute.amazonaws.com:8080/Annot
Inside : main
number of warmup threads 6
Running phase Warmup
number of threads submitted6
9000
number of loadingThreads 32
Running phase Loading
number of threads submitted32
69000
number of peak threads 64
Running phase Peak
number of threads submitted64
441000
number of coolDownThreads 16
Running phase Cooldown
number of threads submitted16
=====
Total Requests 481000
Failed Requests 0
Closed File
[ec2-user@ip-172-31-46-87 ~]$
```

Stats:

In []: `# StartTimestamp = sorted(StartTimestamp)
EndTimestamp = sorted(EndTimestamp)

dt_base = min(StartTimestamp)
startTime = min(StartTimestamp)
endTime = max(EndTimestamp)

walltime = (endTime-startTime)/1000
throughput = int(length/walltime)

print("walltime: ", walltime, "seconds")
print("throughput: ", throughput)
print("median Latency: ", medianLatency, " milliseconds")
print("95th percentile latency: ", latency[index95], "milliseconds")
print("99th percentile latency: ", latency[index99], "milliseconds")`

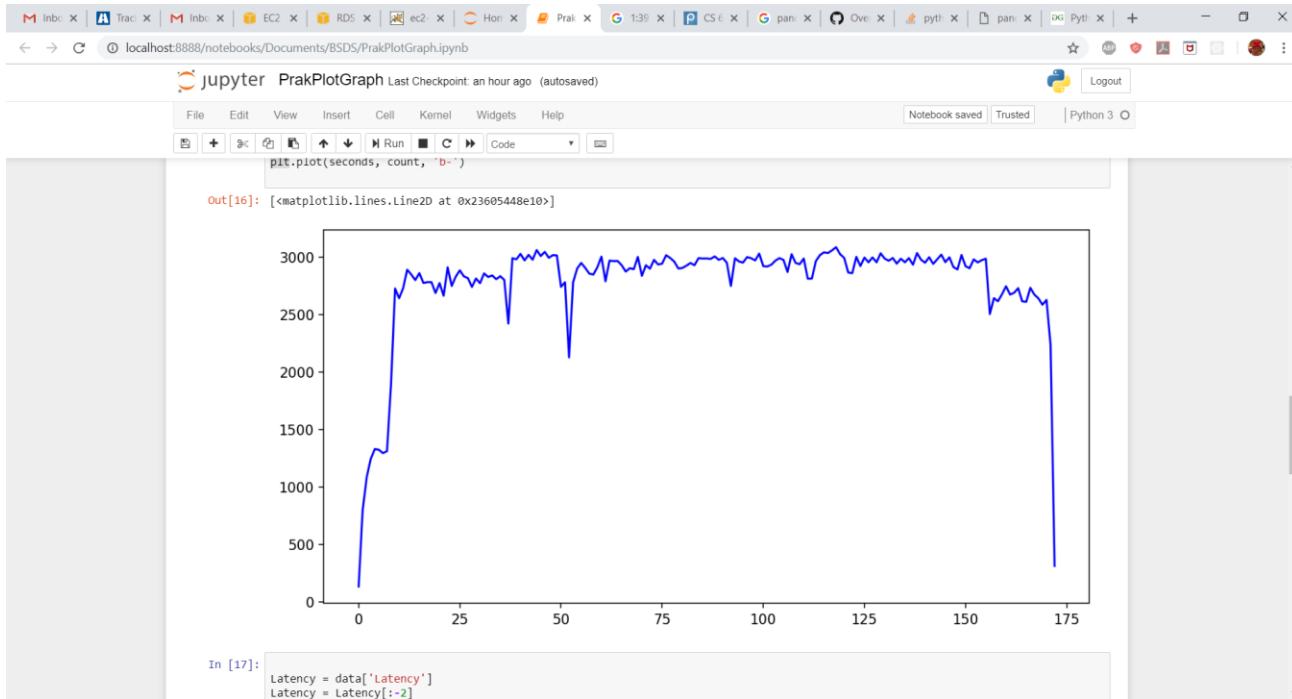
Out[]: `walltime: 172.606 seconds
throughput: 2786
median Latency: 17 milliseconds
95th percentile latency: 37.0 milliseconds
99th percentile latency: 52.0 milliseconds`

In [16]: `time_dict = {}

for each_ts in StartTimestamp:
 time_in_seconds = (each_ts - dt_base)//1000
 if(time_in_seconds in time_dict):
 time_dict[time_in_seconds] += 1
 else:
 time_dict[time_in_seconds] = 1

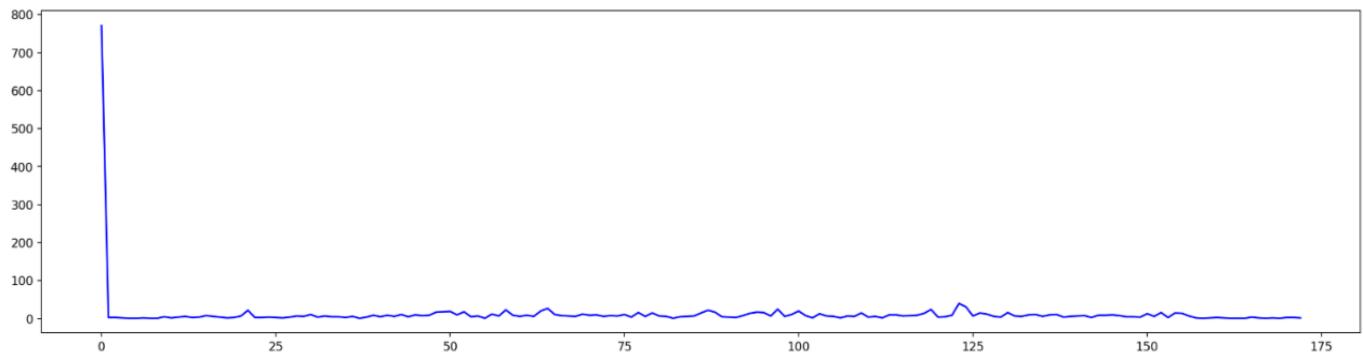
seconds = []
count = []`

Throughput



Latency

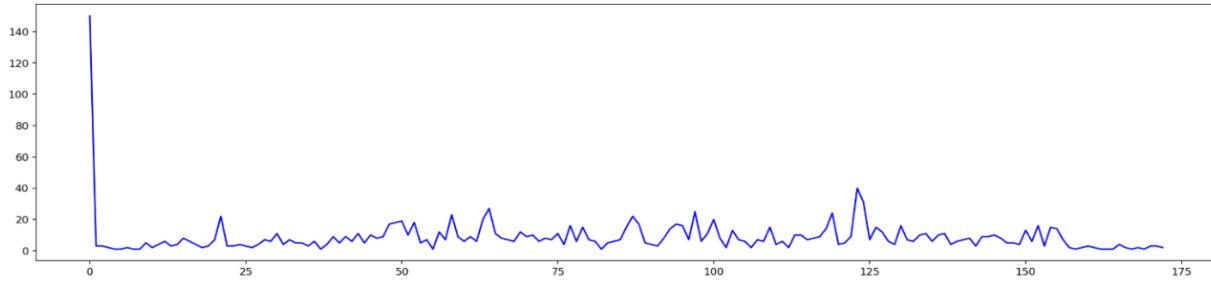
```
max latency: 792.0
min latency: 1.0
[<matplotlib.lines.Line2D at 0x23602245470>]
```



Latency normalized to max 150, for better visualization

```
fig = plt.figure(figsize=(20,5), dpi=160)
for i in range(len(seconds)):
    if avg_lat[i] > 150:
        avg_lat[i] = 150

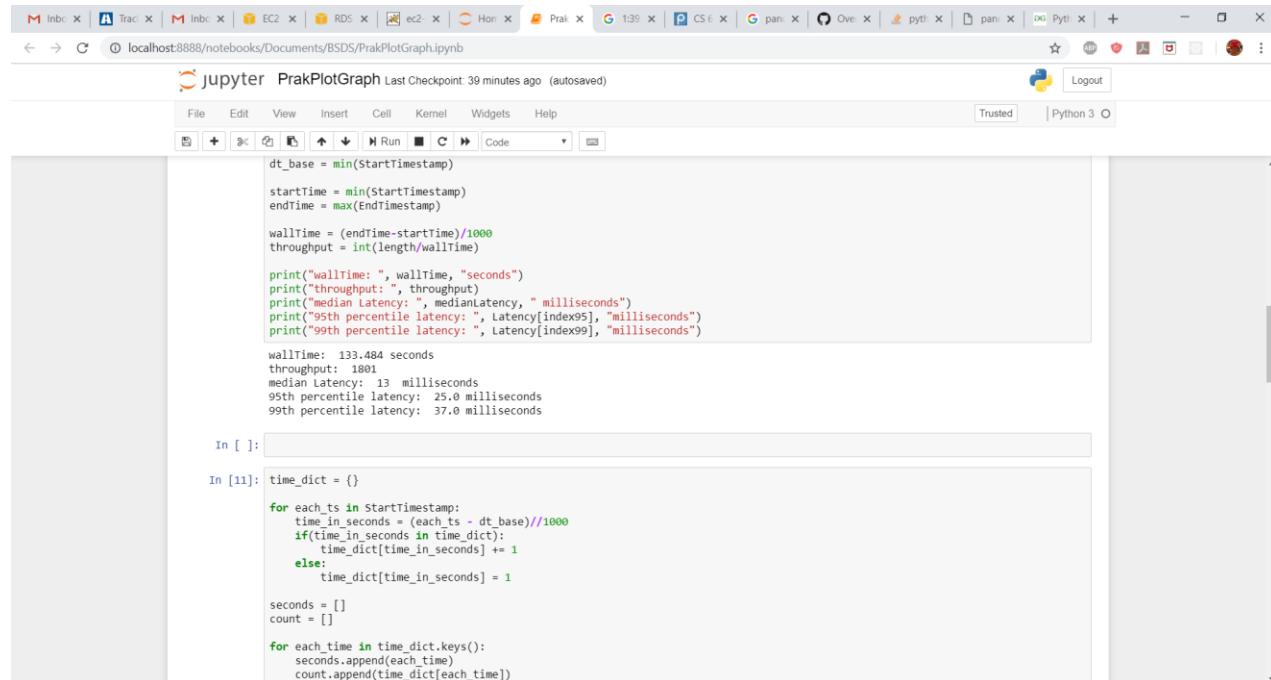
plt.plot(seconds, avg_lat, 'b-')
[<matplotlib.lines.Line2D at 0x236025e34a8>]
```



Step 3: Run for thread {32, 64, 128, 256}, all run are for 100 tests/phase

32 Threads

```
[ec2-user@ip-172-31-46-87:~]$ java -jar PrakClient 64 http://ec2-54-202-211-45.us-west-2.compute.amazonaws.com:8080/Annotator
Inside : main
number of warmup threads 6
Running phase Warmup
number of threads submitted6
9000
number of loadingThreads 32
Running phase Loading
number of threads submitted32
89000
number of peak threads 64
Running phase Peak
number of threads submitted64
441000
number of coolDownThreads 16
Running phase Cooldown
number of threads submitted16
=====
Total Requests 481000
Failed Requests 0
Closed File
[ec2-user@ip-172-31-46-87:~]$ java -jar PrakClient 32 http://ec2-54-202-211-45.us-west-2.compute.amazonaws.com:8080/Annotator
Inside : main
number of warmup threads 3
Running phase Warmup
number of threads submitted3
4500
number of loadingThreads 16
Running phase Loading
number of threads submitted16
44500
number of peak threads 32
Running phase Peak
number of threads submitted32
220500
number of coolDownThreads 8
Running phase Cooldown
number of threads submitted8
=====
Total Requests 240500
Failed Requests 0
Closed File
[ec2-user@ip-172-31-46-87:~]$
```



The screenshot shows a Jupyter Notebook interface with the title "jupyter PrakPlotGraph". The notebook has a single cell containing Python code. The code calculates throughput and various latency percentiles from a timestamp range. It then creates a histogram-like structure to analyze the distribution of times.

```
dt_base = min(StartTimeStamp)
startTime = min(StartTimeStamp)
endTime = max(EndTimeStamp)
walltime = (endTime-startTime)/1000
throughput = int(length/walltime)

print("walltime: ", walltime, "seconds")
print("throughput: ", throughput)
print("median latency: ", medianLatency, "milliseconds")
print("95th percentile latency: ", Latency[index95], "milliseconds")
print("99th percentile latency: ", Latency[index99], "milliseconds")

walltime: 133.484 seconds
throughput: 1801
median latency: 13 milliseconds
95th percentile latency: 25.0 milliseconds
99th percentile latency: 37.0 milliseconds
```

In []:

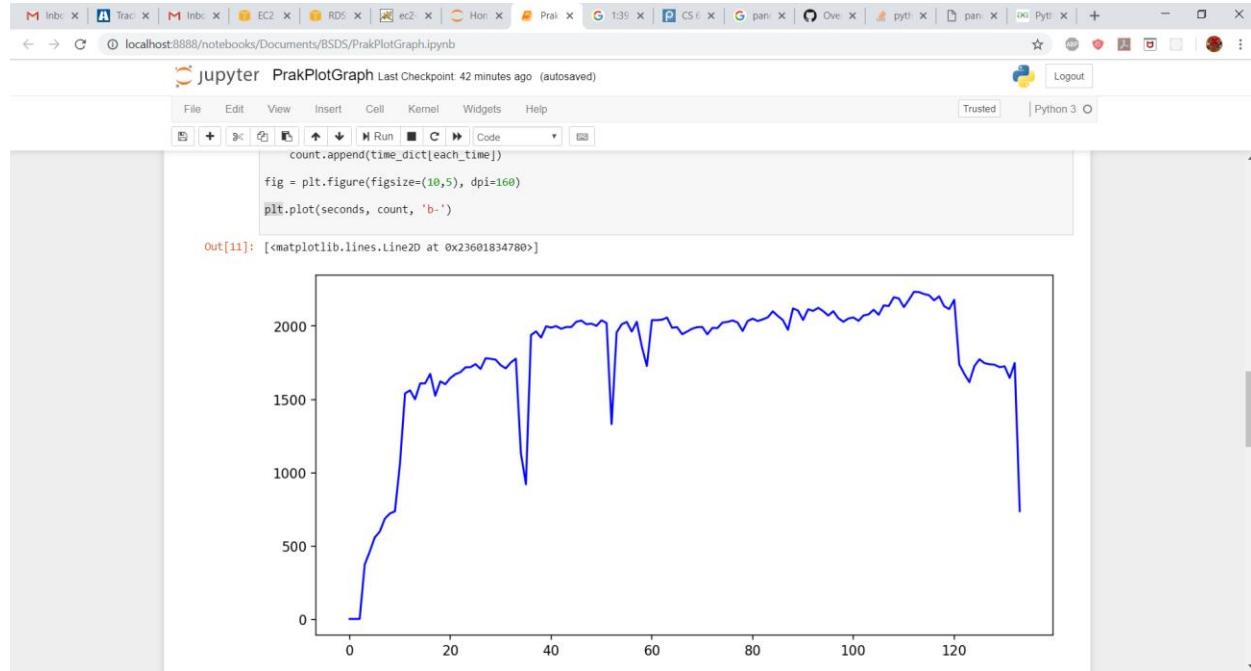
```
In [11]: time_dict = {}

for each_ts in StartTimeStamp:
    time_in_seconds = (each_ts - dt_base)//1000
    if time_in_seconds in time_dict:
        time_dict[time_in_seconds] += 1
    else:
        time_dict[time_in_seconds] = 1

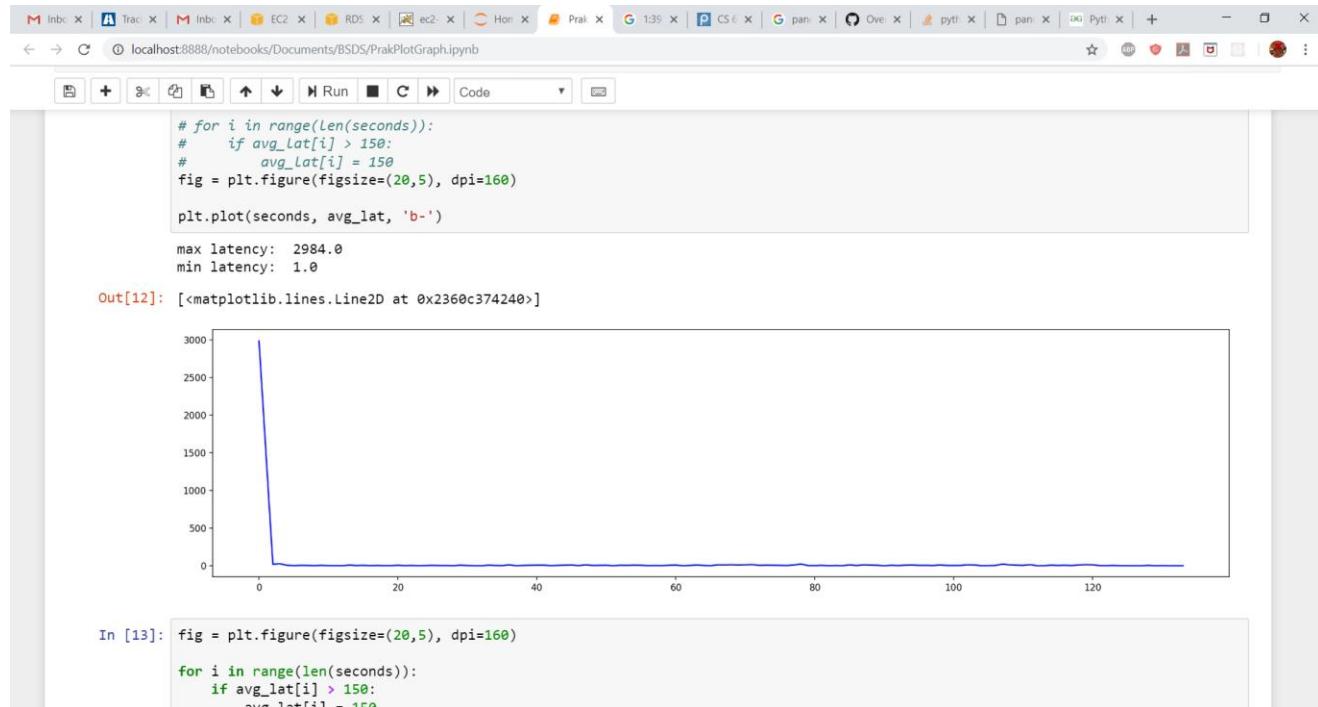
seconds = []
count = []

for each_time in time_dict.keys():
    seconds.append(each_time)
    count.append(time_dict[each_time])
```

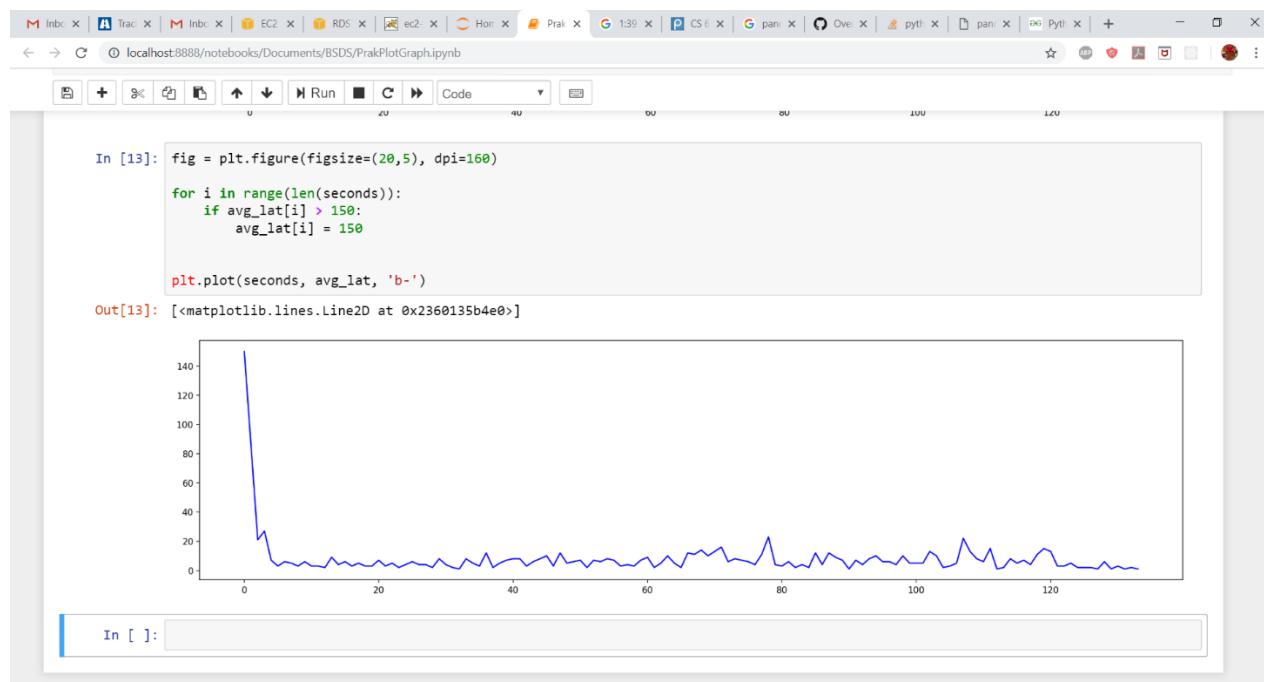
Throughput:



Latency



Latency normalized to max 150



64 Threads are same as for Step1

128 threads:

```
ec2-user@ip-172-31-46-87:~$ java -jar PrakClient 128 http://ec2-54-202-211-45.us-west-2.compute.amazonaws.com:8088/Ano
inside : main
number of warmup threads 12
Running phase Warmup
number of threads submitted12
18000
number of loadingThreads 64
Running phase Loading
number of threads submitted64
178000
number of peak threads 128
Running phase Peak
number of threads submitted128
882000
number of coolDownThreads 32
Running phase Cooldown
number of threads submitted32
0
-----
Total Requests 962000
Failed Requests 0
Closed File
[ec2-user@ip-172-31-46-87 ~]$
```

Jupyter PrakPlotGraph Last Checkpoint: an hour ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
startTime = min(StartTimeStamp)
endTime = max(EndTimeStamp)

wallTime = (endTime-startTime)/1000
throughput = int(length/wallTime)

print("wallTime: ", wallTime, "seconds")
print("throughput: ", throughput)
print("median Latency: ", medianLatency, "milliseconds")
print("95th percentile latency: ", Latency[index95], "milliseconds")
print("99th percentile latency: ", Latency[index99], "milliseconds")

walltime: 404.822 seconds
throughput: 2376
median Latency: 38 milliseconds
95th percentile latency: 90.0 milliseconds
99th percentile latency: 119.0 milliseconds
```

In []:

```
time_dict = {}

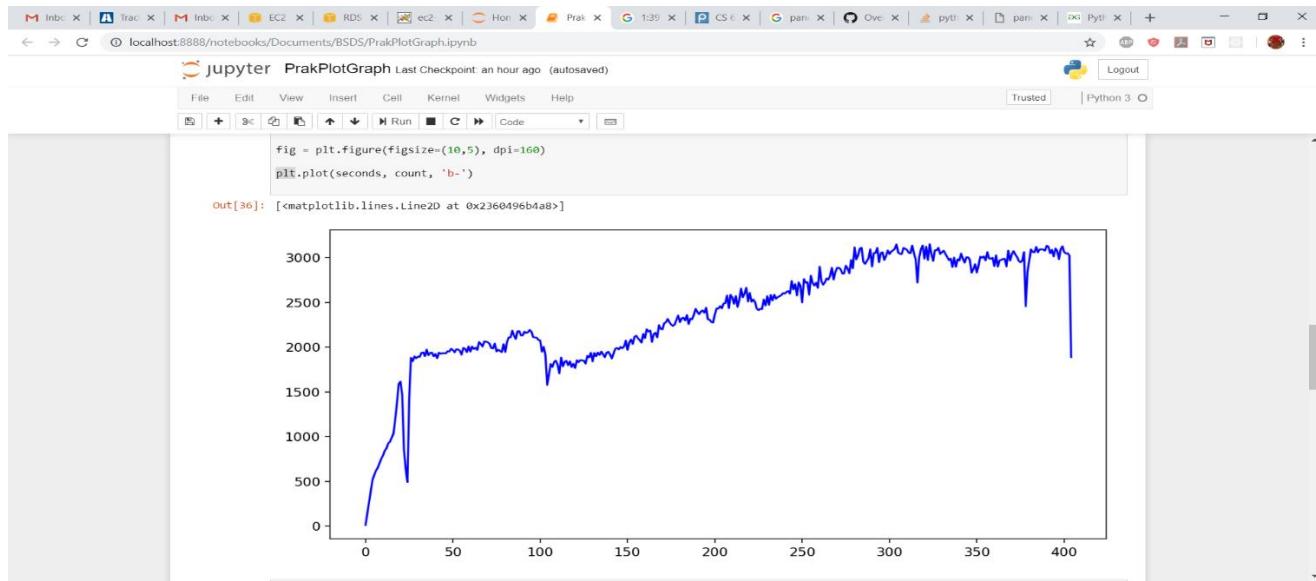
for each_ts in StartTimeStamp:
    time_in_seconds = (each_ts - dt_base)//1000
    if(time_in_seconds in time_dict):
        time_dict[time_in_seconds] += 1
    else:
        time_dict[time_in_seconds] = 1

seconds = []
count = []

for each_time in time_dict.keys():
    seconds.append(each_time)
    count.append(time_dict[each_time])

fig = plt.figure(figsize=(10,5), dpi=160)
```

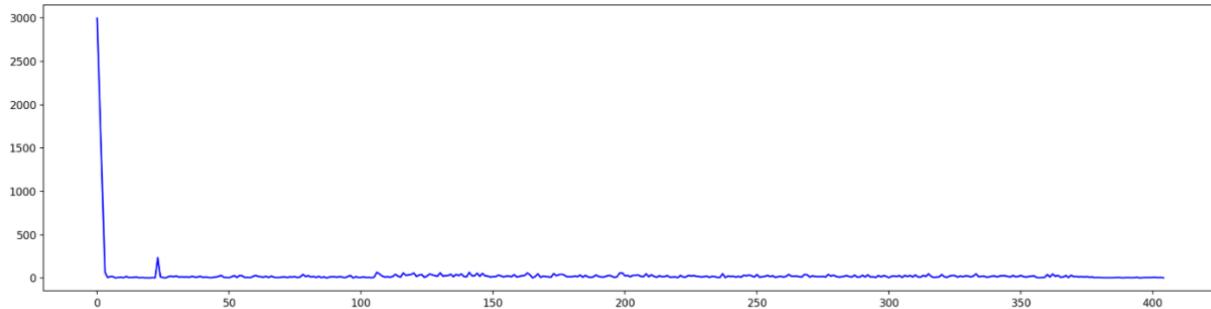
Throughput



Latency

```
max latency: 3002.0
min latency: 1.0
```

```
[<matplotlib.lines.Line2D at 0x2360cb3e240>]
```



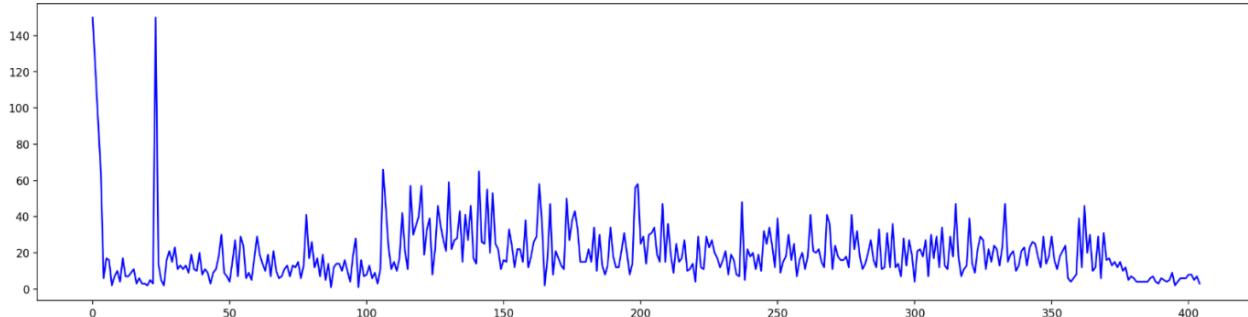
Latency normalized to max 150

```
fig = plt.figure(figsize=(20,5), dpi=160)

for i in range(len(seconds)):
    if avg_lat[i] > 150:
        avg_lat[i] = 150

plt.plot(seconds, avg_lat, 'b-')

[<matplotlib.lines.Line2D at 0x236049dfdd8>]
```



256 threads:

```
[ec2-user@ip-172-31-46-87 ~]$ clear
[ec2-user@ip-172-31-46-87 ~]$ [ec2-user@ip-172-31-46-87 ~]$ [ec2-user@ip-172-31-46-87 ~]$ java -jar PrakClient 256 http://ec2-54-202-211-45.us-west-2.compute.amazonaws.com:8080/Ano
inside : main
number of warmup threads 25
Running phase Warmup
number of threads submitted25
37500
number of loadingThreads 128
Running phase Loading
number of threads submitted128
357500
number of peak threads 256
Running phase Peak
number of threads submitted256
1765500
number of coolDownThreads 64
Running phase Cooldown
number of threads submitted64
-----
Total Requests 1925500
Failed Requests 0
Closed File
[ec2-user@ip-172-31-46-87 ~]$
```

The screenshot shows a Jupyter Notebook interface with the following details:

- Top Bar:** Shows various open tabs including Inbox, Trac, Inbox, EC2, RDS, ec2, Prak, CS 6, pan, Ove, pyti, pan, Pyti.
- Toolbar:** Includes standard notebook controls like File, Edit, View, Insert, Cell, Kernel, Widgets, Help, and a Trusted button.
- Code Cell:** Contains the following Python code:

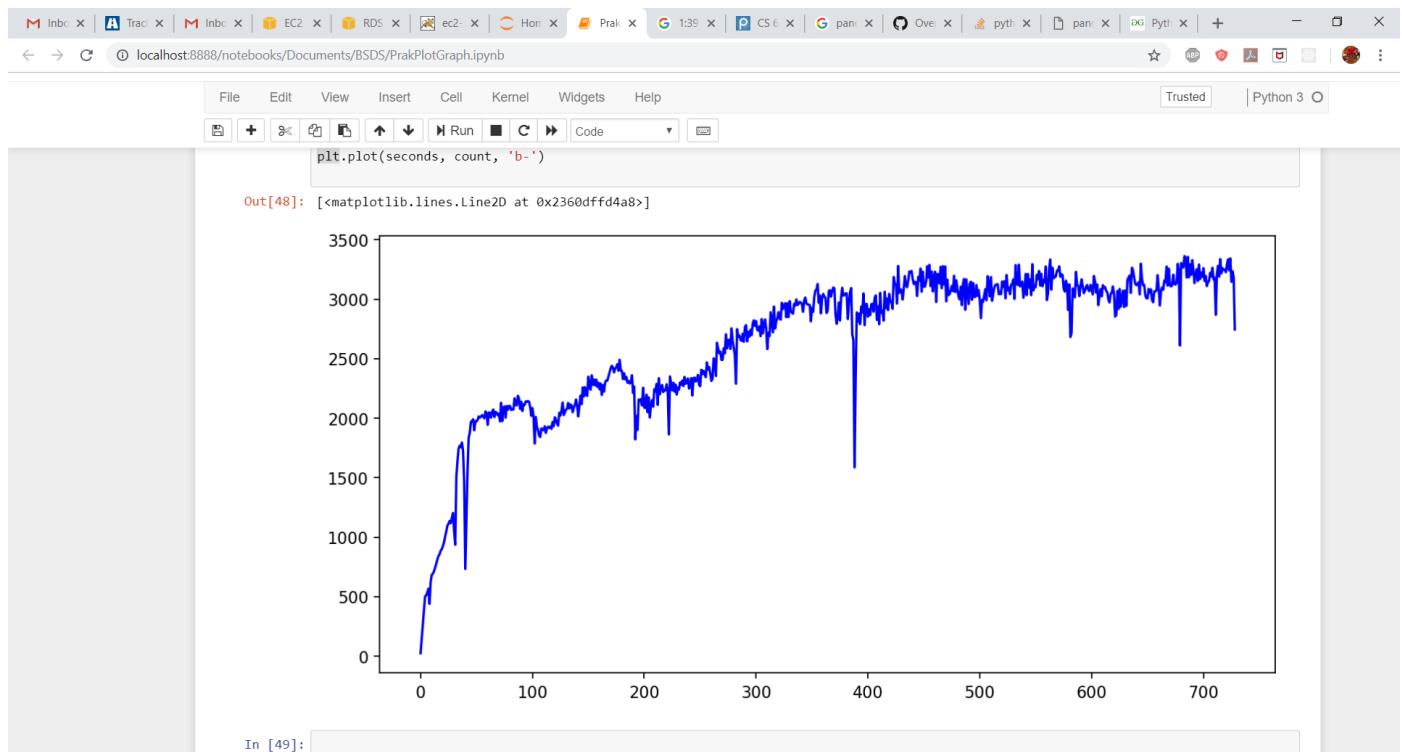
```
dt_base = min(StartTimeStamp)
startTime = min(StartTimeStamp)
endTime = max(EndTimeStamp)

wallTime = (endTime-startTime)/1000
throughput = int(length/walltime)

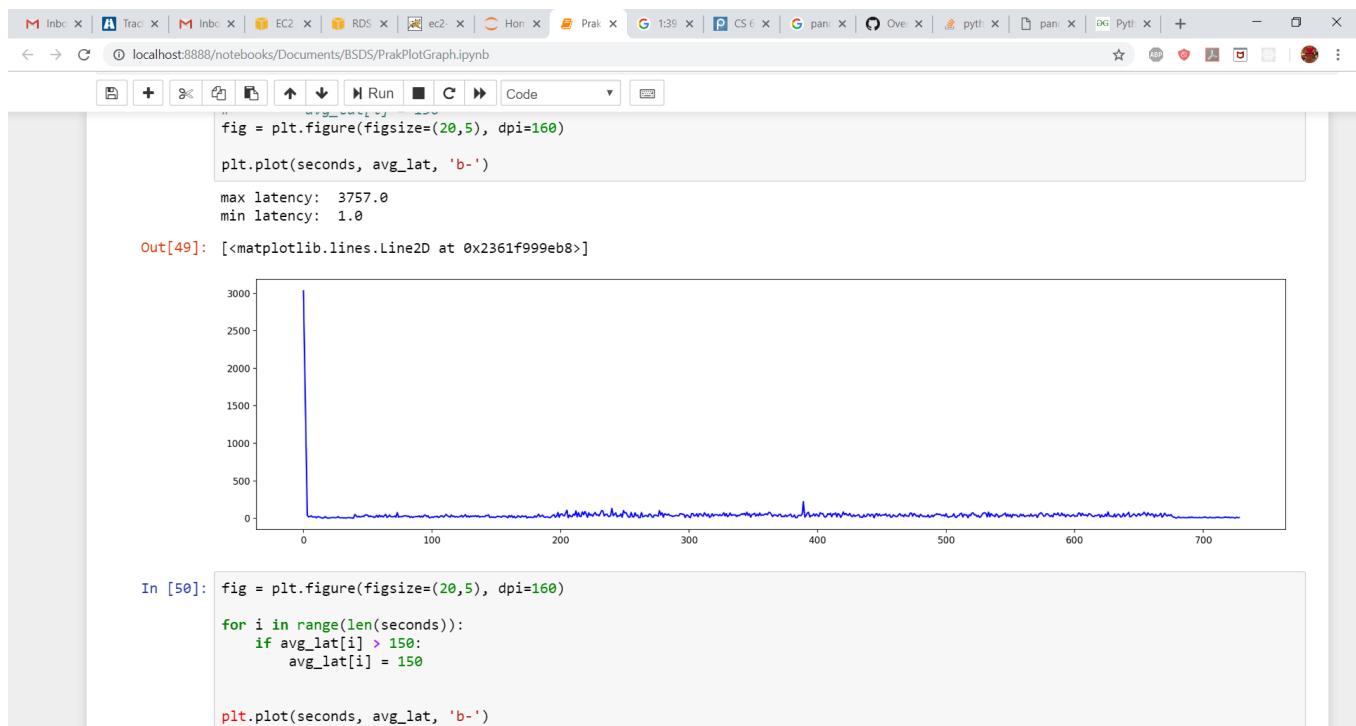
print("wallTime: ", wallTime, "seconds")
print("throughput: ", throughput)
print("median Latency: ", medianLatency, " milliseconds")
print("95th percentile latency: ", Latency[index95], "milliseconds")
print("99th percentile latency: ", Latency[index99], "milliseconds")

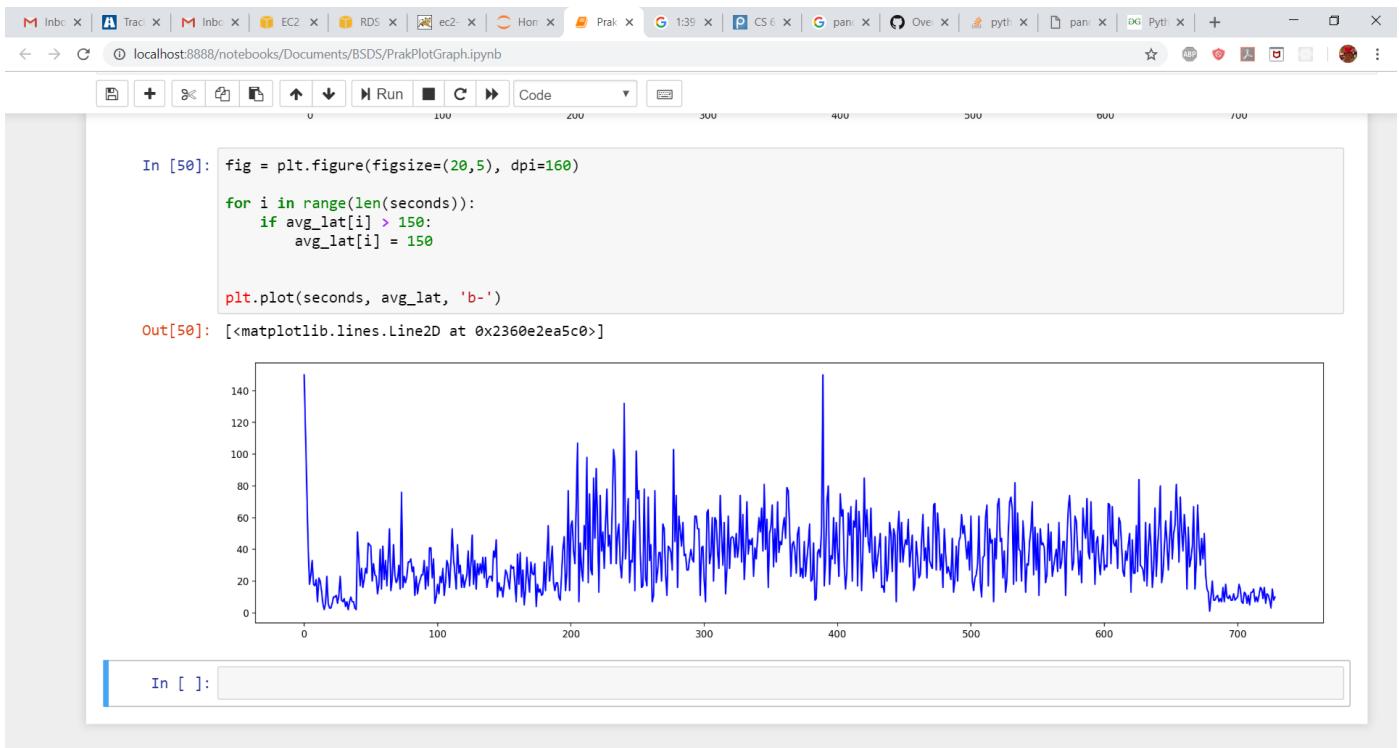
wallTime: 728.965 seconds
throughput: 2641
median Latency: 73 milliseconds
95th percentile latency: 149.0 milliseconds
99th percentile latency: 197.0 milliseconds
```
- Output:** Displays the execution results of the code cell.
- Input Cell:** Labeled "In []:" with some partially visible code.

Throughput



Latency:





Step 4:

For this step I created an **AWS Load Balancer, Auto Scaling group and a Launch Configuration**

Load Balancer Configuration

The screenshot shows the AWS Management Console for the EC2 service, specifically the Load Balancers section. A search bar at the top right shows the filter "bsdsLoadBalancer". Below it, a table displays the configuration for the load balancer "bsdsLoadBalancer". The table includes columns for Name, ARN, DNS name, Scheme, Type, Availability Zones, Creation time, Hosted zone, State, VPC ID, IP address type, and AWS WAF Web ACL. The "DNS name" is listed as "bsdsLoadBalancer-664473791.us-west-2.elb.amazonaws.com (A Record)". The "Scheme" is "internet-facing" and the "Type" is "application". The "Availability Zones" are "subnet-959533f2 - us-west-2a, subnet-d711099e - us-west-2b". The "Creation time" is "November 24, 2018 at 9:42:04 PM UTC-8". The "Hosted zone" is "Z1H1FL5HABSF5". The "State" is "active". The "VPC ID" is "vpc-2b3fb04c". The "IP address type" is "ipv4".

Name	ARN	DNS name	Scheme	Type	Availability Zones	Creation time	Hosted zone	State	VPC ID	IP address type	AWS WAF Web ACL
bsdsLoadBalancer	arn:aws:elasticloadbalancing:us-west-2:532748787565:loadbalancer/app/bsdsLoadBalancer/3e4d95fe26601fb	bsdsLoadBalancer-664473791.us-west-2.elb.amazonaws.com (A Record)	internet-facing	application	subnet-959533f2 - us-west-2a, subnet-d711099e - us-west-2b	November 24, 2018 at 9:42:04 PM UTC-8	Z1H1FL5HABSF5	active	vpc-2b3fb04c	ipv4	

Auto Scaling Group:

I set the min and max instances to 1 & 5 respectively.

The screenshot shows the AWS Management Console interface for the Auto Scaling service. On the left, there's a navigation sidebar with links like EC2 Dashboard, Instances, Images, and Network & Security. The main content area displays a table of Auto Scaling groups. One group is selected, showing its detailed configuration. The group name is 'newAutoScalingGroup' and it uses a launch configuration named 'finalLaunchConfig'. The launch configuration specifies a desired capacity of 2, a minimum of 1, and a maximum of 5. It is configured to use the 'us-west-2a' and 'us-west-2b' availability zones, with a default cooldown of 60 seconds. Other settings include a target group named 'finalTargetGroup' and an EC2 health check type with a grace period of 60 seconds.

I had set the **Scaling Policies** such that if average CPU usage, for 1min was more than 40 percent, the load balancer would spawn another instance,

And if average CPU usage goes below 30 percent for a min, it would delete an instance.

The screenshot shows the 'Scaling Policies' section for the 'newAutoScalingGroup'. It displays two policies: 'Decrease Group Size' and 'Increase Group Size'. The 'Decrease Group Size' policy is triggered by a metric alarm 'breaches the alarm threshold: CPUUtilization < 30 for 60 seconds' for the metric dimensions 'AutoScalingGroupName = newAutoScalingGroup'. The action is to 'Remove 1 Instances when 30 >= CPUUtilization > -infinity'. The 'Increase Group Size' policy is triggered by a metric alarm 'breaches the alarm threshold: CPUUtilization >= 30 for 60 seconds' for the metric dimensions 'AutoScalingGroupName = newAutoScalingGroup'. The action is to 'Add 1 instances when 30 <= CPUUtilization < +infinity'. Both policies use a step scaling policy type.

Launch Configuration

Screenshot of the AWS Management Console showing the Launch Configuration page.

The left sidebar shows the navigation menu with the "Launch Configurations" option selected under "Auto Scaling".

The main content area displays a list of launch configurations:

Name	AMI ID	Instance Type	Creation Time
finalLaunchCo...	ami-038180e0982ed6378	t2.micro	December 2, 2018 at 11:20:39 P...
newLaunchCo...	ami-0057d2ef4e2540454	t2.micro	November 24, 2018 at 10:58:57 ...

Details for the selected launch configuration ("finalLaunchConfig") are shown:

AMI ID	AMI ID	Instance Type
ami-038180e0982ed6378	ami-038180e0982ed6378	t2.micro
IAM Instance Profile	Kernel ID	
Key Name: bdsFirstKeyPair	Monitoring: false	
EBS Optimized	Security Groups	
false	sg-0f2ce8a77576950cd	
Spot Price	Creation Time	
	Sun Dec 02 23:20:39 GMT-800 2018	
RAM Disk ID	Block Devices	
	/dev/xvda	
User data	IP Address Type	
-	Only assign a public IP address to instances launched in the default VPC and subnet. (default)	

Feedback and Language options are at the bottom left, and copyright and legal links are at the bottom right.

Therads {32, 64, 128, 256} with Load Balancer

32 threads with Load Balancer

```
ec2-user@ip-172-31-46-87:~$ Running phase Warmup
number of thread submitted3
500
number of loadingThreads 16
Running phase Loading
number of threads submitted16
44500
number of peak threads 32
Running phase Peak
number of threads submitted32
220500
number of coolDownThreads 8
Running phase Cooldown
number of threads submitted8
-----
Total Requests 240500
Failed Requests 0
Closed File
[ec2-user@ip-172-31-46-87 ~]$
```

Jupyter PrakPlotGraph Last Checkpoint: 11 hours ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
# StartTimestamp = sorted(StartTimeStamp)
# EndTimestamp = sorted(EndTimeStamp)

dt_base = min(StartTimeStamp)
startTime = min(StartTimeStamp)
endTime = max(EndTimeStamp)

walltime = (endTime-startTime)/1000
throughput = int(length/walltime)

print("wallTime: ", walltime, "seconds")
print("throughput: ", throughput)
print("median Latency: ", medianLatency, "milliseconds")
print("95th percentile latency: ", Latency[index95], "milliseconds")
print("99th percentile latency: ", Latency[index99], "milliseconds")

walltime: 132.41 seconds
throughput: 1816
median Latency: 12 milliseconds
95th percentile latency: 25.0 milliseconds
99th percentile latency: 36.0 milliseconds
```

In []:

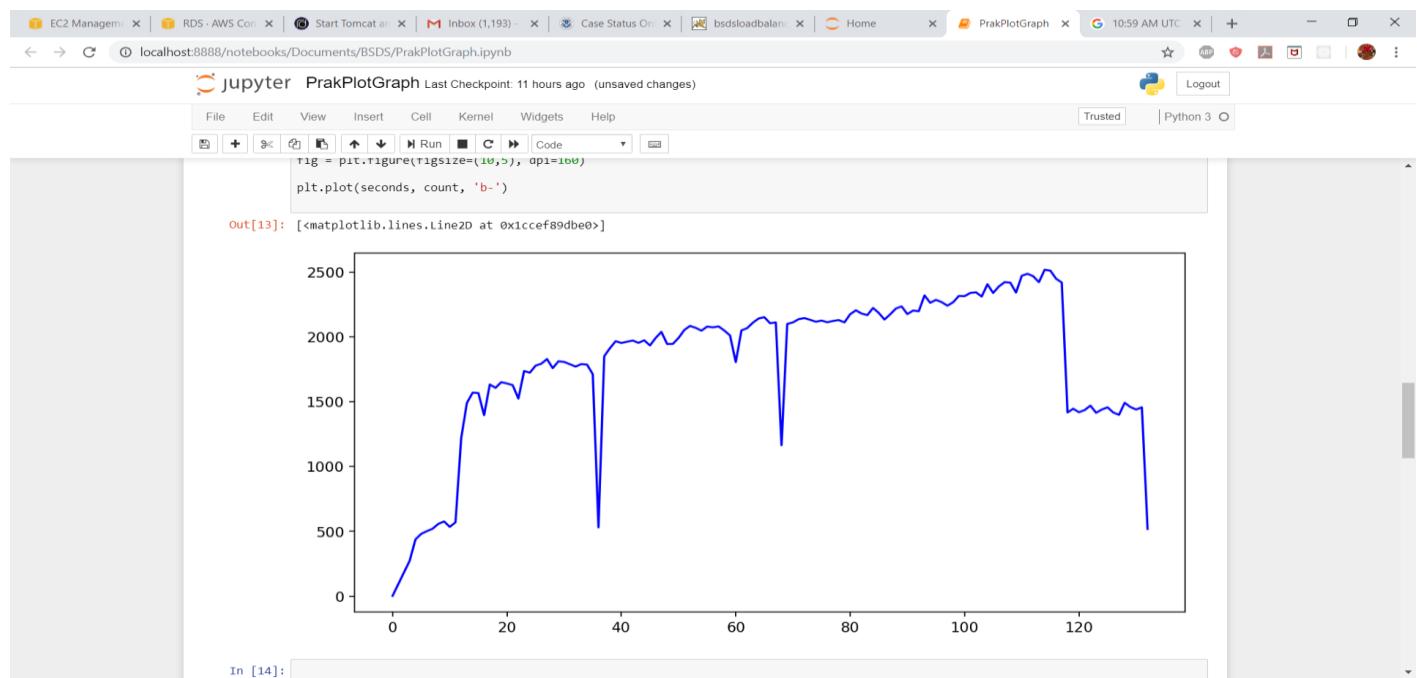
```
In [13]: time_dict = {}

for each_ts in StartTimestamp:
    time_in_seconds = (each_ts - dt_base)//1000
    if(time_in_seconds in time_dict):
        time_dict[time_in_seconds] += 1
    else:
        time_dict[time_in_seconds] = 1

seconds = []
count = []

for each_time in time_dict:
    seconds.append(each_time)
    count.append(time_dict[each_time])
```

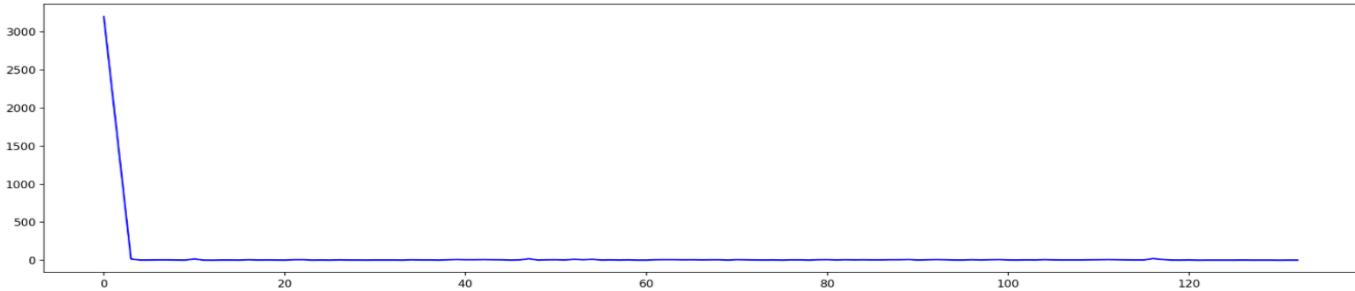
Throughput:



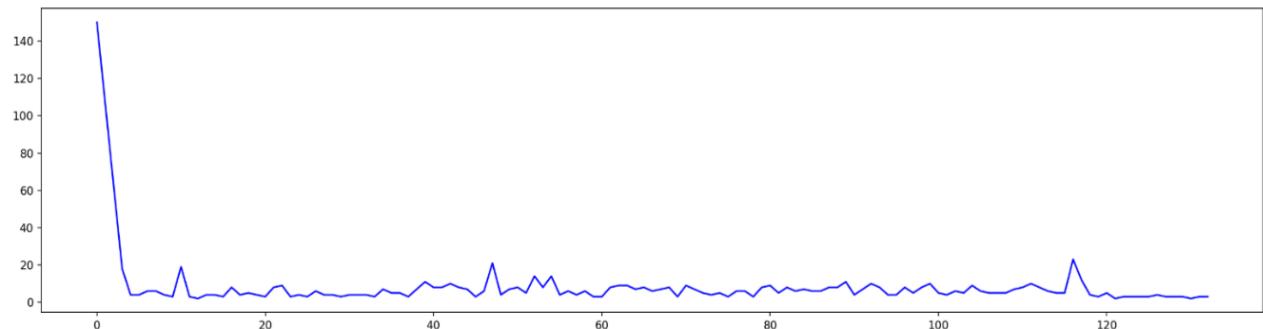
Latency:

```
max latency: 3199.0
min latency: 2.0
```

```
[<matplotlib.lines.Line2D at 0x1cceeaa7eba8>]
```



```
Out[15]: [<matplotlib.lines.Line2D at 0x1ccefd7fdd8>]
```



For 32 threads, only 1 instance was used

The screenshot shows the AWS Auto Scaling console. On the left, the navigation menu includes EC2 Dashboard, Events, Tags, Reports, Limits, Instances, Launch Templates, Spot Requests, Reserved Instances, Dedicated Hosts, Scheduled Instances, Capacity Reservations, Images, AMIs, Bundle Tasks, Elastic Block Store, Volumes, Snapshots, Lifecycle Manager, Network & Security, Security Groups, Elastic IPs, and Placement Groups. The main area displays the 'Create Auto Scaling group' section with a table for 'Auto Scaling Group: newAutoScalingGroup'. The table shows one instance with the following details:

Name	Launch Configuration /	Instances	Desired	Min	Max	Availability Zones	Default Cooldown	Health Check Grace
newAutoScalin...	finalLaunchConfig	1	1	1	5	us-west-2a, us-west-2b	60	60

Below this, the 'Instances' tab is selected in the 'Auto Scaling Group: newAutoScalingGroup' navigation bar. It shows a single instance table:

Instance ID	Lifecycle	Launch Configuration / Template	Availability Zone	Health Status	Protected from
i-0987e3a607b59f32a	InService	final.launchConfig	us-west-2b	Healthy	

CPU utilization

The screenshot shows the AWS CloudWatch Metrics console. The left sidebar includes EC2 Dashboard, Events, Tags, Reports, Limits, Instances, Launch Templates, Spot Requests, Reserved Instances, Dedicated Hosts, Scheduled Instances, Capacity Reservations, Images, AMIs, Bundle Tasks, Elastic Block Store, Volumes, Snapshots, Lifecycle Manager, and Network & Security. The main area displays the 'CloudWatch Monitoring Details' for an instance. A modal window titled 'CloudWatch Monitoring Details' shows a line graph of 'CPU Utilization (Percent)' over time. The graph has two y-axes: 'CPU Utilization (Percent)' from 0 to 35 and 'CPU Utilization (Percent)' from 0 to 40. The x-axis shows dates from 12/4 to 12/4 at 04:50, 05:00, 05:10, 05:20, 05:25, 05:30, 05:35, 05:40, and 05:45. The data series shows a sharp spike from near zero to approximately 32% at 05:35. The graph interface includes controls for 'Statistic: Average', 'Time Range: Last Hour', and 'Period: 5 Minutes'.

64 threads with a load balancer

```
[ec2-user@ip-172-31-46-87:~] $ java -jar PrakClient 64 http://bsdsloadbalancer-664473791.us-west-2.elb.amazonaws.com:8080
inside : main
number of warmup threads 6
Running phase Warmup
number of threads submitted6
9000
number of loadingThreads 32
Running phase Loading
number of threads submitted32
89000
number of peak threads 64
Running phase Peak
number of threads submitted64
441000
number of coolDownThreads 16
Running phase Cooldown
number of threads submitted16
-----
Total Requests 481000
Failed Requests 0
Closed File
[ec2-user@ip-172-31-46-87 ~]$
```

Jupyter PrakPlotGraph Last Checkpoint: 11 hours ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
startTIme = min(StartTime)
endTime = max(EndTime)

wallTime = (endTime-startTime)/1000
throughput = int(length/wallTime)

print("wallTime: ", wallTime, "seconds")
print("throughput: ", throughput)
print("median Latency: ", medianLatency, "milliseconds")
print("95th percentile latency: ", latency[index95], "milliseconds")
print("99th percentile latency: ", latency[index99], "milliseconds")

walltime: 217.197 seconds
throughput: 2214
median Latency: 21 milliseconds
95th percentile latency: 44.0 milliseconds
99th percentile latency: 62.0 milliseconds
```

In []:

```
time_dict = {}

for each_ts in StartTimestamp:
    time_in_seconds = (each_ts - dt_base)//1000
    if(time_in_seconds in time_dict):
        time_dict[time_in_seconds] += 1
    else:
        time_dict[time_in_seconds] = 1

seconds = []
count = []

for each_time in time_dict.keys():
    seconds.append(each_time)
    count.append(time_dict[each_time])

fig = plt.figure(figsize=(10,5), dpi=160)
```

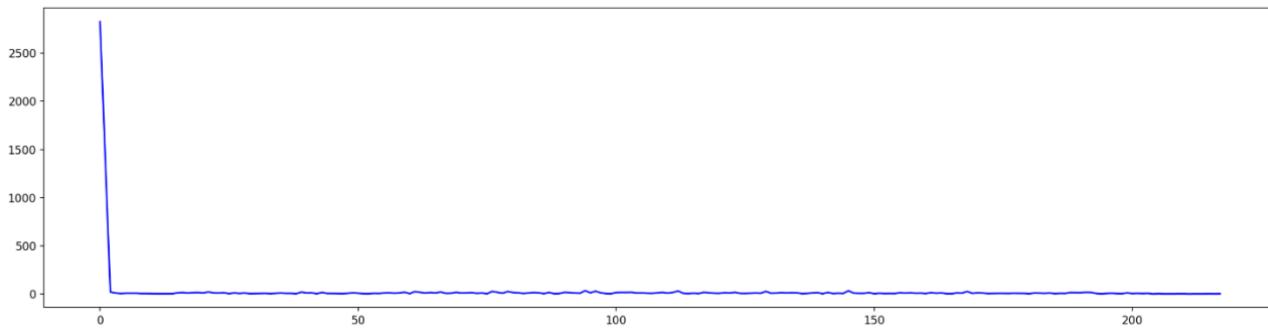
Throughput



Latency

```
max latency: 2827.0
min latency: 1.0

[<matplotlib.lines.Line2D at 0x1cc80039a90>]
```

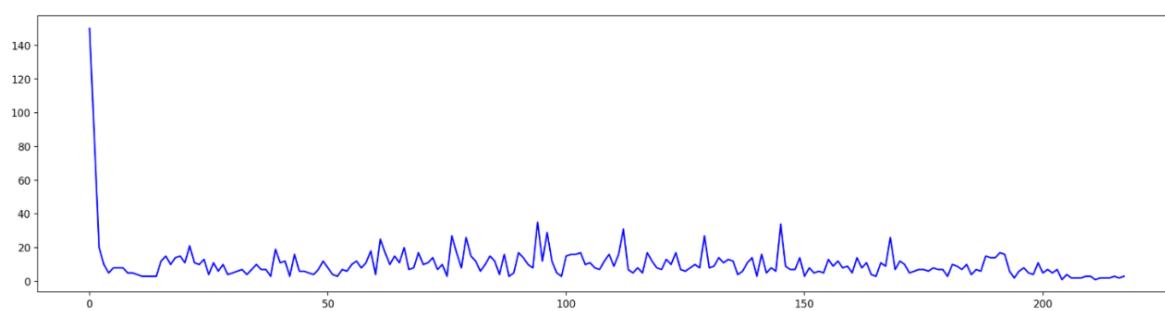


```
In [20]: fig = plt.figure(figsize=(20,5), dpi=160)

for i in range(len(seconds)):
    if avg_lat[i] > 150:
        avg_lat[i] = 150

plt.plot(seconds, avg_lat, 'b-')

Out[20]: [<matplotlib.lines.Line2D at 0x1cc8019ad68>]
```



For 64 threads, only 1 CPU was used

The screenshot shows the AWS Auto Scaling console. On the left, the navigation menu includes EC2 Dashboard, Events, Tags, Reports, Limits, Instances (selected), Launch Templates, Spot Requests, Reserved Instances, Dedicated Hosts, Scheduled Instances, Capacity Reservations, Images (AMIs, Bundle Tasks), Elastic Block Store (Volumes, Snapshots, Lifecycle Manager), and Network & Security (Security Groups, Elastic IPs, Placement Groups). The main content area shows an Auto Scaling group named "newAutoScalingGroup" with one instance running. The instance details show it has an ID of "i-010ad1f1362c98145", is in "InService" state, and is associated with the "finalLaunchConfig". The CPU utilization for this instance is shown as 0%.

CPU instances:

The screenshot shows the AWS EC2 Instances console. The navigation menu is identical to the previous screen. A modal window titled "CloudWatch Monitoring Details" is open, displaying a line graph of CPU Utilization (Percent) over time. The graph shows three data points for the instance "i-0987e3a607b59f32a": at 04:55 it is at ~58%, drops to 0% between 05:25 and 05:30, peaks at ~32% around 05:35, dips to ~12% around 05:40, and then rises sharply to ~52% by 05:45. The "Close" button is visible at the bottom right of the modal.

128 threads with Load Balancer:

```
[ec2-user@ip-172-31-46-87:~]$ java -jar PrakClient 128 http://bsdsloadbalancer-664473791.us-west-2.elb.amazonaws.com:808
inside : main
number of warmup threads 12
Running phase Warmup
number of threads submitted12
18000
number of loadingThreads 64
Running phase Loading
number of threads submitted64
178000
number of peak threads 128
Running phase Peak
number of threads submitted128
182000
number of coolDownThreads 32
Running phase Cooldown
number of threads submitted32
-----
Total Requests 962000
Failed Requests 0
Closed File
[ec2-user@ip-172-31-46-87 ~]$
```

Jupyter PrakPlotGraph Last Checkpoint: 13 hours ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
print("wallTime: ", wallTime, "seconds")
print("throughput: ", throughput)
print("median Latency: ", medianLatency, " milliseconds")
print("95th percentile latency: ", Latency[index95], "milliseconds")
print("99th percentile latency: ", Latency[index99], "milliseconds")

wallTime: 426.355 seconds
throughput: 2256
median Latency: 41 milliseconds
95th percentile latency: 94.0 milliseconds
99th percentile latency: 125.0 milliseconds
```

In []:

```
In [28]: time_dict = {}

for each_ts in StartTimestamp:
    time_in_seconds = (each_ts - dt_base)//1000
    if time_in_seconds in time_dict:
        time_dict[time_in_seconds] += 1
    else:
        time_dict[time_in_seconds] = 1

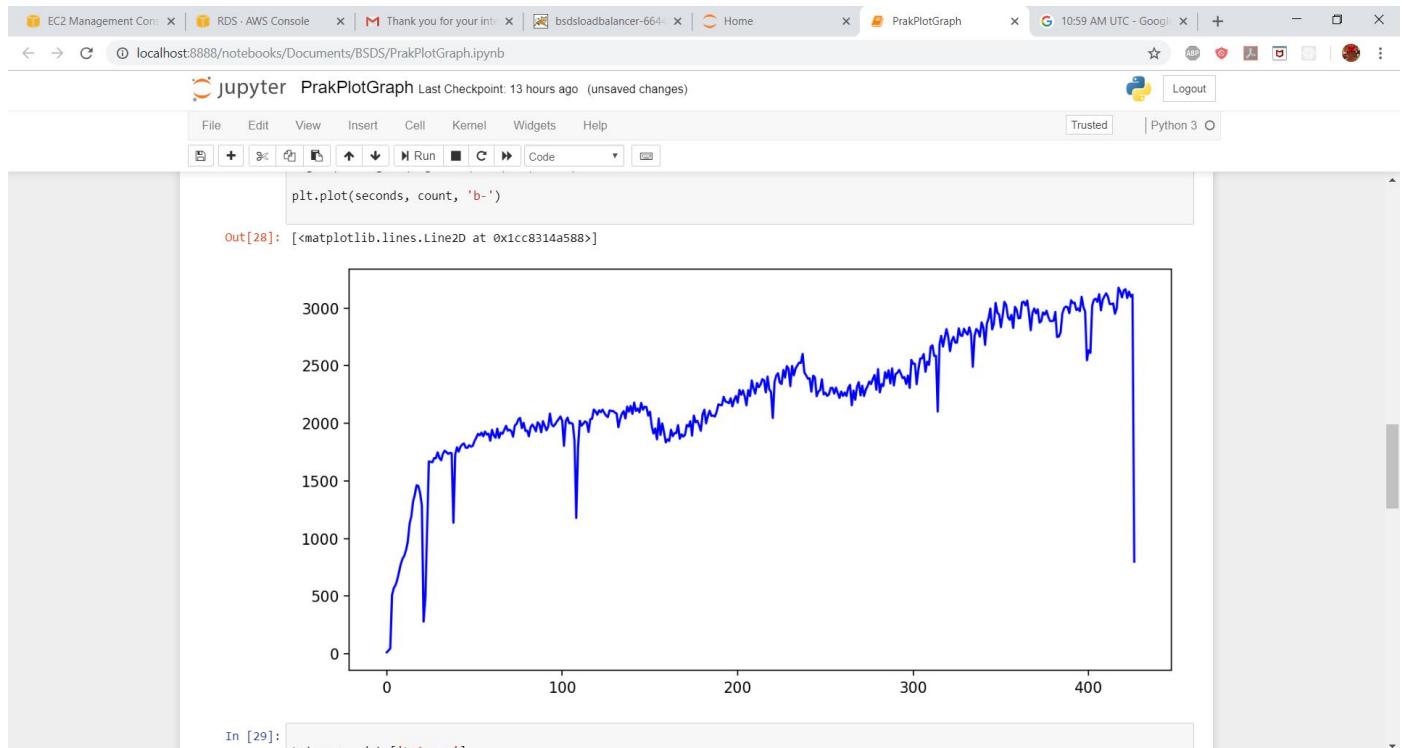
seconds = []
count = []

for each_time in time_dict.keys():
    seconds.append(each_time)
    count.append(time_dict[each_time])

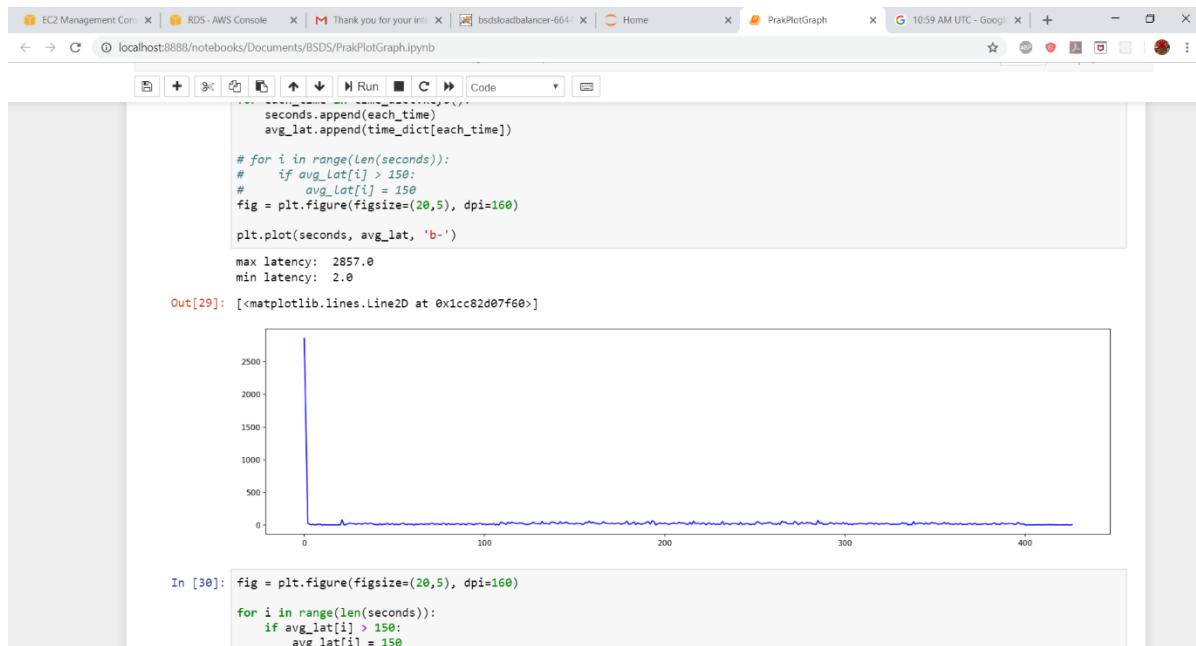
fig = plt.figure(figsize=(10,5), dpi=160)
plt.plot(seconds, count, 'b-')
```

Out[28]: [<matplotlib.lines.Line2D at 0x1cc8314a588>]

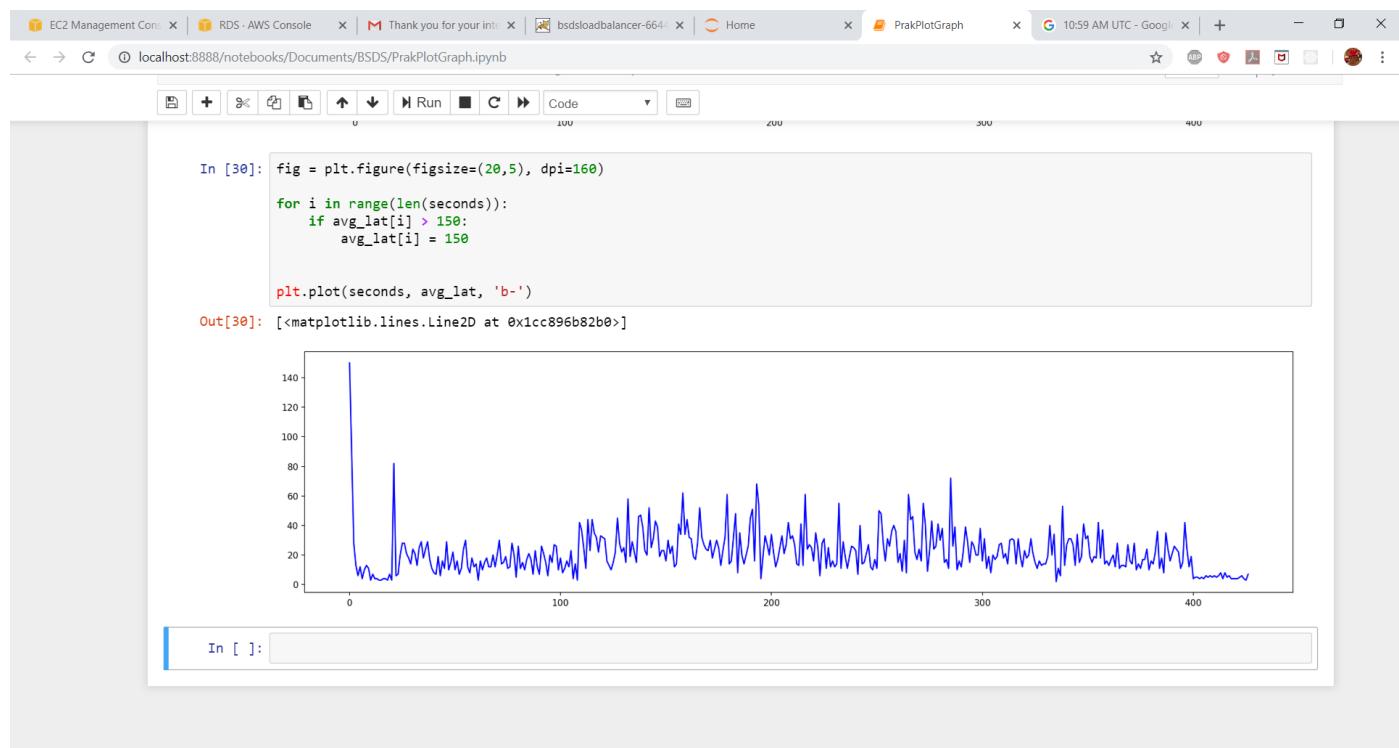
Throughput:



Latency plot:



Latencies averaged to 150



For 128 threads, 2 instances were spawned

The screenshot shows the AWS Auto Scaling Groups page. The left sidebar includes links for EC2 Dashboard, Events, Tags, Reports, Limits, Instances, Launch Templates, Spot Requests, Reserved Instances, Dedicated Hosts, Scheduled Instances, Capacity, Reservations, Images, AMIs, Bundle Tasks, Elastic Block Store, Volumes, Snapshots, Lifecycle Manager, Network & Security, Security Groups, Elastic IPs, and Placement Groups. The main content area shows the 'Create Auto Scaling group' button and a table for the 'newAutoScalingGroup' launch configuration. The table details include Name: newAutoScalingGroup, Launch Configuration: finalLaunchConfig, Instances: 2, Desired: 1, Min: 1, Max: 5, Availability Zones: us-west-2a, us-west-2b, Default Cooldown: 60, and Health Check Grace Period: 60. Below this, the 'Instances' tab for the 'newAutoScalingGroup' is selected, showing two healthy instances: i-0ae02e49b46614be8 and i-0b618a37f3e1c956a, both in the InService lifecycle state and running in the us-west-2a and us-west-2b availability zones respectively.

Name	Launch Configuration	Instances	Desired	Min	Max	Availability Zones	Default Cooldown	Health Check Grace Period
newAutoScalingGroup	finalLaunchConfig	2	1	1	5	us-west-2a, us-west-2b	60	60

Auto Scaling Group: newAutoScalingGroup						
Details	Activity History	Scaling Policies	Instances	Monitoring	Notifications	Tags
Actions						
Filter: Any Health Status Any Lifecycle State						
Instance ID	Lifecycle	Launch Configuration / Template	Availability Zone	Health Status	Protected from	
i-0ae02e49b46614be8	InService	finalLaunchConfig	us-west-2a	Healthy		
i-0b618a37f3e1c956a	InService	finalLaunchConfig	us-west-2b	Healthy		

256 Threads with Load Balancer:

```
[ec2-user@ip-172-31-46-87 ~]$ java -jar PrakClient 256 http://bsdsloadbalancer-664473791.us-west-2.elb.amazonaws.com:8080/AnotherProjectWar/rest/myfirstapp 1 100000 100
inside : main
number of warmup threads 25
Running phase Warmup
number of threads submitted25
37500
number of loadingThreads 128
Running phase Loading
number of threads submitted128
357500
number of peak threads 256
Running phase Peak
number of threads submitted256
1765500
number of coolDownThreads 64
Running phase Cooldown
number of threads submitted64
357500
-----
Total Requests 1923500
Failed Requests 0
Closed File
[ec2-user@ip-172-31-46-87 ~]$ Connection reset by 34.216.54.197 port 22
PS C:\Users\prakriti\Downloads> .
```

Jupyter PrakPlotGraph Last Checkpoint: 21 hours ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
dt_base = min(startTime)
startTime = min(StartTime)
endTime = max(EndTime)
wallTime = (endTime-startTime)/1000
throughput = int(length/wallTime)

print("wallTime: ", wallTime, "seconds")
print("throughput: ", throughput)
print("median Latency: ", medianLatency, "milliseconds")
print("95th percentile latency: ", latency[index95], "milliseconds")
print("99th percentile latency: ", latency[index99], "milliseconds")

walltime: 426.355 seconds
throughput: 2256
median Latency: 41 milliseconds
95th percentile latency: 94.0 milliseconds
99th percentile latency: 125.0 milliseconds
```

In []:

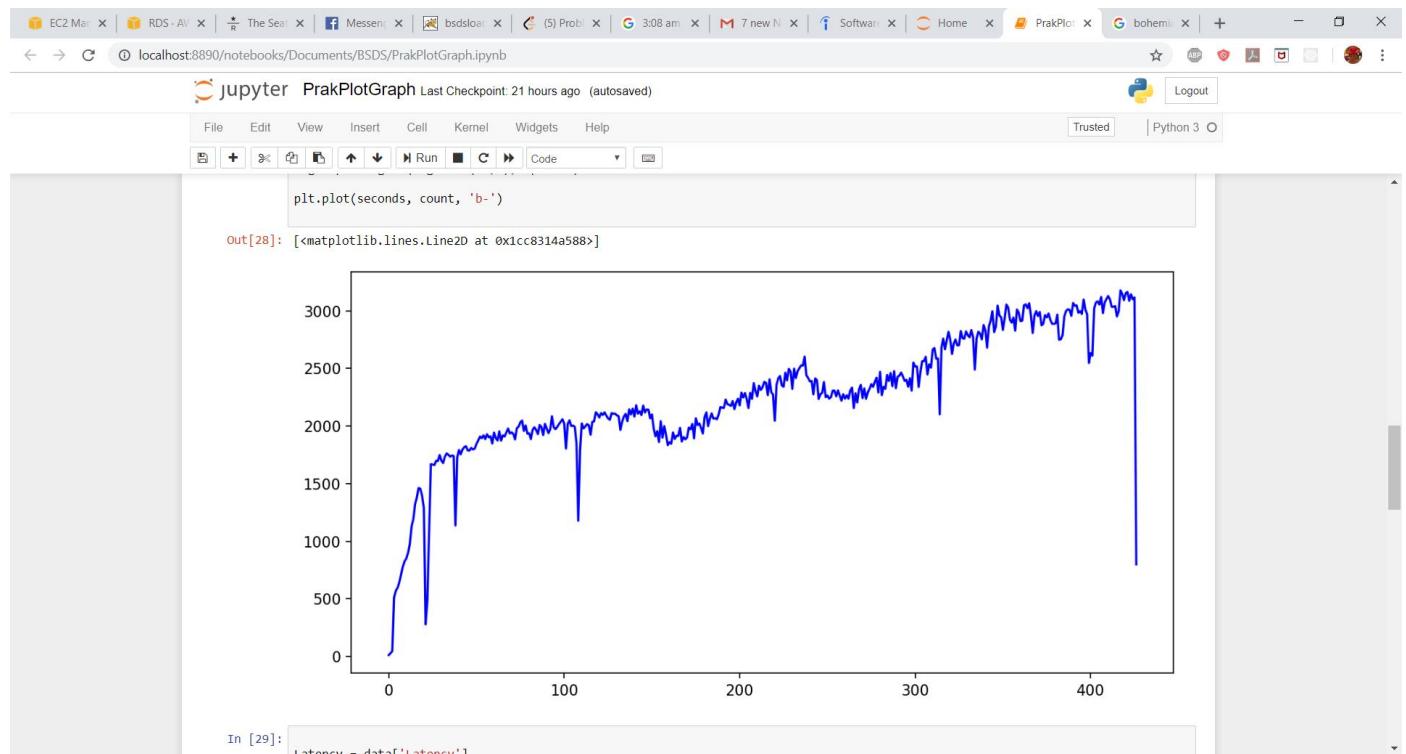
```
In [28]: time_dict = {}

for each_ts in StartTimestamp:
    time_in_seconds = (each_ts - dt_base)//1000
    if(time_in_seconds in time_dict):
        time_dict[time_in_seconds] += 1
    else:
        time_dict[time_in_seconds] = 1

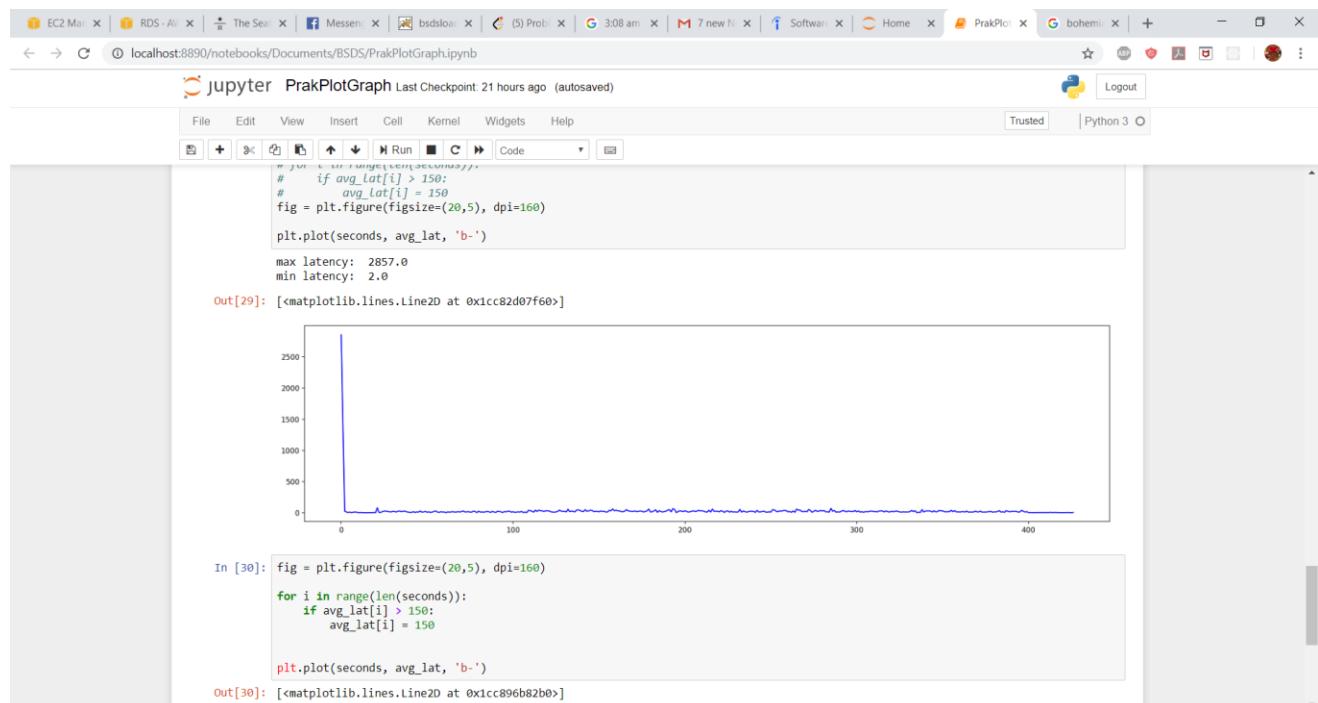
seconds = []
count = []

for each_time in time_dict.keys():
    seconds.append(each_time)
    count.append(time_dict[each_time])
```

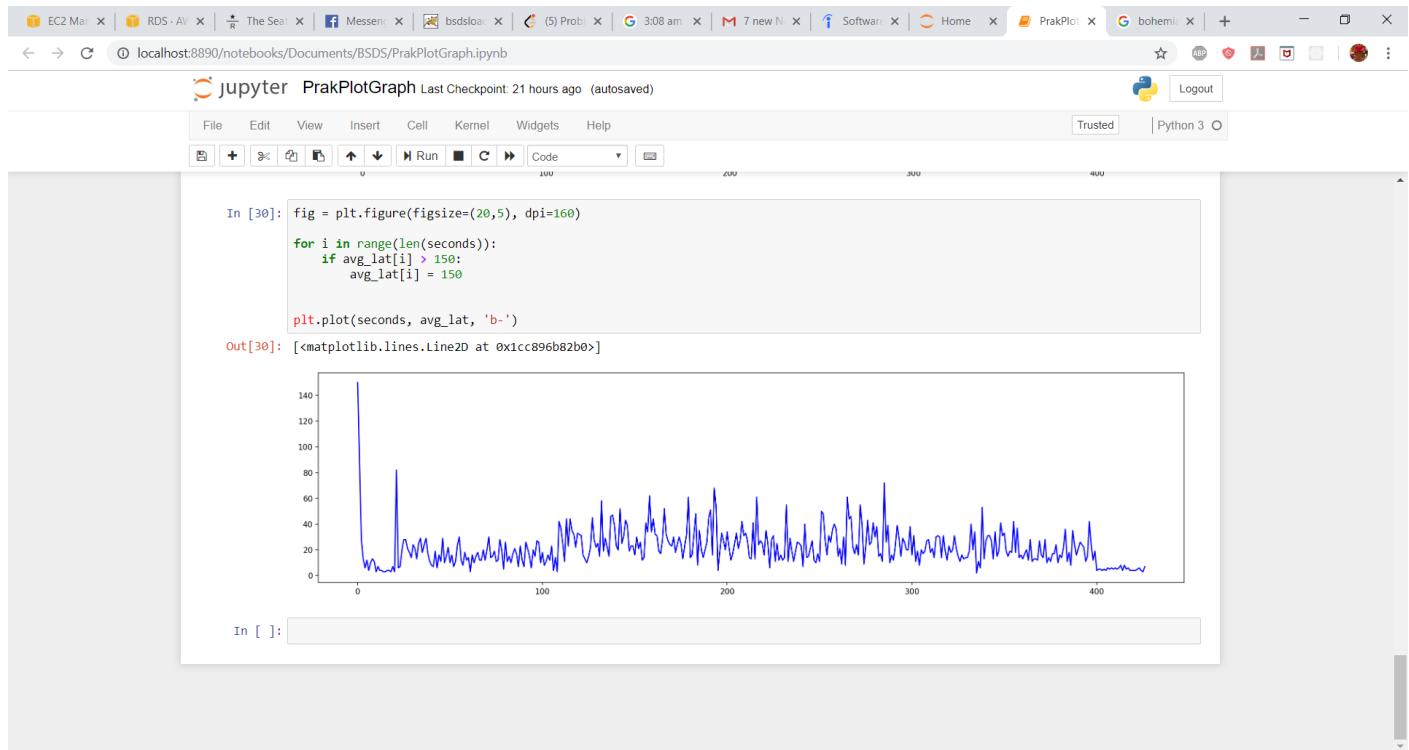
Throughput:



Latency:

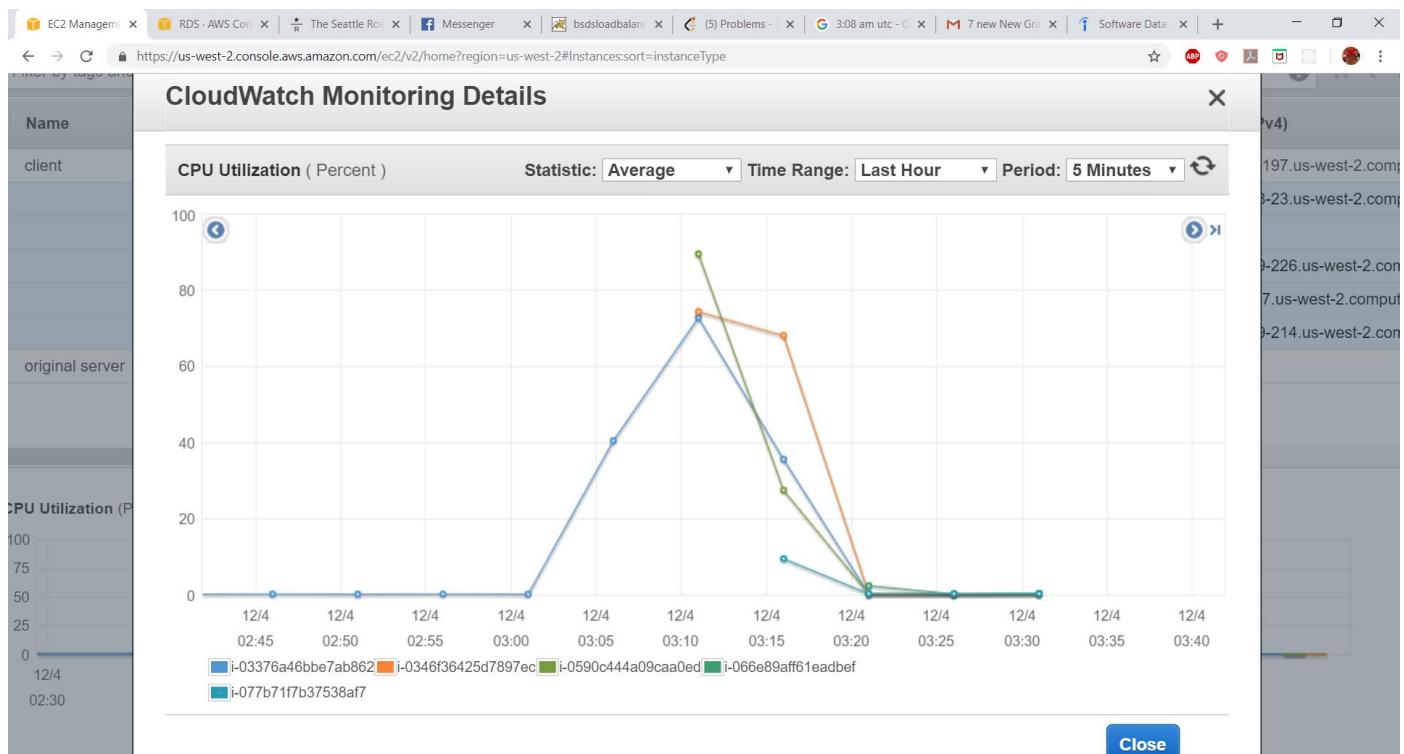


Latency averaged to 150



For 256 threads, all 5 instances were spawned, 4 were running during the peak phase, and at the end of peak 5th instance was spawned

CPU utilization of 5 instances



Instances Launched at Peak Phase:

The screenshot shows the AWS CloudWatch Metrics console. The top navigation bar includes tabs for EC2 Management, RDS - AWS Com, The Seattle Roll, Messenger, bsdloadbalanc, (S) Problems, 3:08 am utc, 7 new New Grid, Software Data, and more. The user is signed in as prakruti dave from Oregon.

The main interface displays a table for the 'newAutoScalingGroup' with the following details:

Name	Launch Configuration /	Instances	Desired	Min	Max	Availability Zones	Default Cooldown	Health Check Grac...
newAutoScalin...	finalLaunchConfig	5	5	1	5	us-west-2a, us-west-2b	60	60

Below this, the 'Instances' tab is selected in the 'Auto Scaling Group: newAutoScalingGroup' navigation bar. It shows five instances listed:

Instance ID	Lifecycle	Launch Configuration / Template	Availability Zone	Health Status	Protected from
i-03376a46bbe7ab862	InService	finalLaunchConfig	us-west-2b	Healthy	
i-0346f36425d7897ec	InService	finalLaunchConfig	us-west-2a	Healthy	
i-0590c444a09caa0ed	InService	finalLaunchConfig	us-west-2b	Healthy	
i-066e89aff61eadbef	InService	finalLaunchConfig	us-west-2a	Healthy	
i-077b71f7b37538af7	InService	finalLaunchConfig	us-west-2a	Healthy	

At the bottom of the page, there are links for Feedback, English (US), Copyright notice (© 2008-2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.), Privacy Policy, and Terms of Use.

Step 5: Load testing

Load testing with 350 threads:

The terminal window shows the execution of a Java application named PrakClient. The command used is:

```
java -jar PrakClient 350 http://bsdloadbalancer-664473791.us-west-2.elb.amazonaws.com:808
```

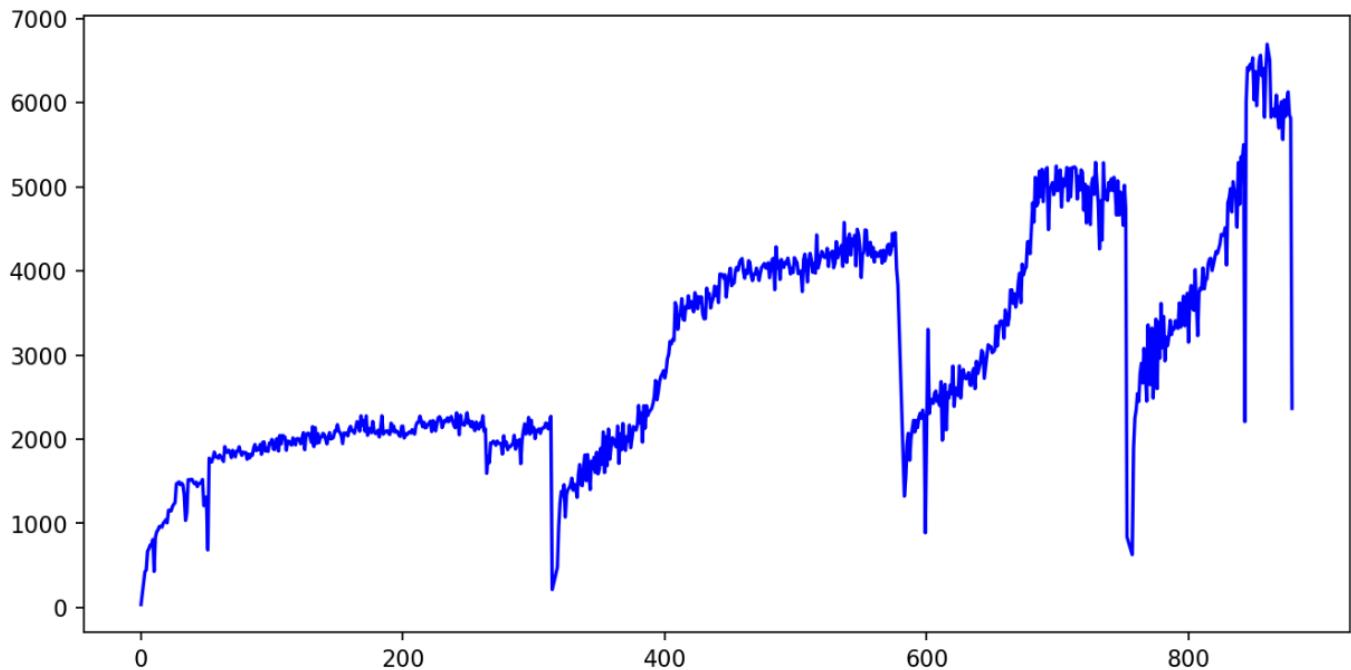
The output of the application shows the following performance metrics:

```
inside : main
number of warmup threads 35
Running phase Warmup
number of threads submitted35
52500
number of loadingThreads 175
Running phase Loading
number of threads submitted175
490000
number of peak threads 350
Running phase Peak
number of threads submitted350
2406988
number of coolDownThreads 87
Running phase Cooldown
number of threads submitted87
=====
Total Requests 2624488
Failed Requests 0
Closed File
[ec2-user@ip-172-31-46-87 ~]$
```

```
print("wallTime: ", wallTime, "seconds")
print("throughput: ", throughput)
print("median Latency: ", medianLatency, " milliseconds")
print("95th percentile latency: ", Latency[index95], "milliseconds")
print("99th percentile latency: ", Latency[index99], "milliseconds")
```

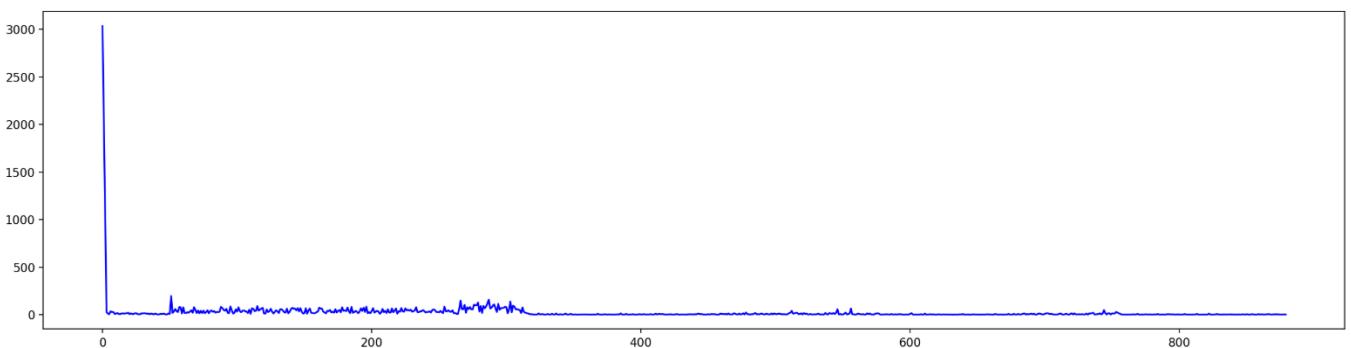
```
wallTime: 879.672 seconds
throughput: 2983
median Latency: 24 milliseconds
95th percentile latency: 321.0 milliseconds
99th percentile latency: 573.0 milliseconds
```

Throughput:



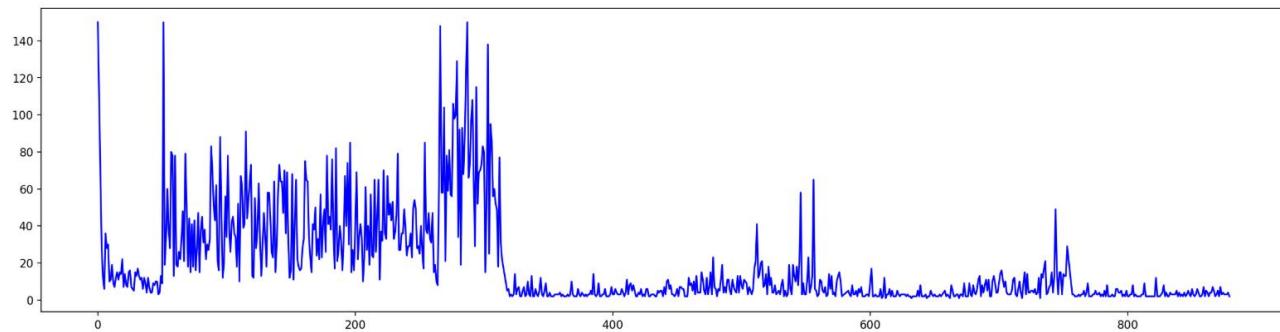
Latency:

```
max latency: 5902.0
min latency: 1.0
[<matplotlib.lines.Line2D at 0x1fa28cd0630>]
```

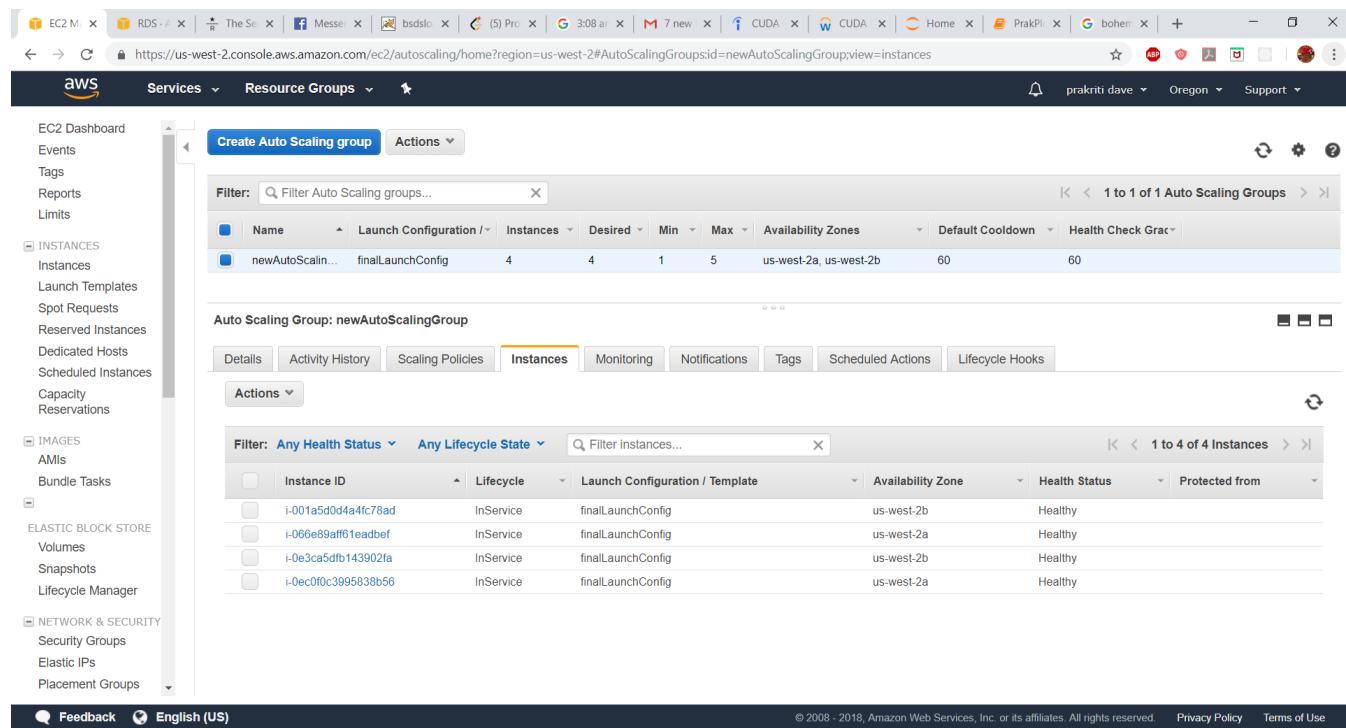


Latency averaged to 150:

Out[5]: [`<matplotlib.lines.Line2D at 0x1fa1499c940>`]



All 5 instances were spawned for 350 threads, during the Peak Phase

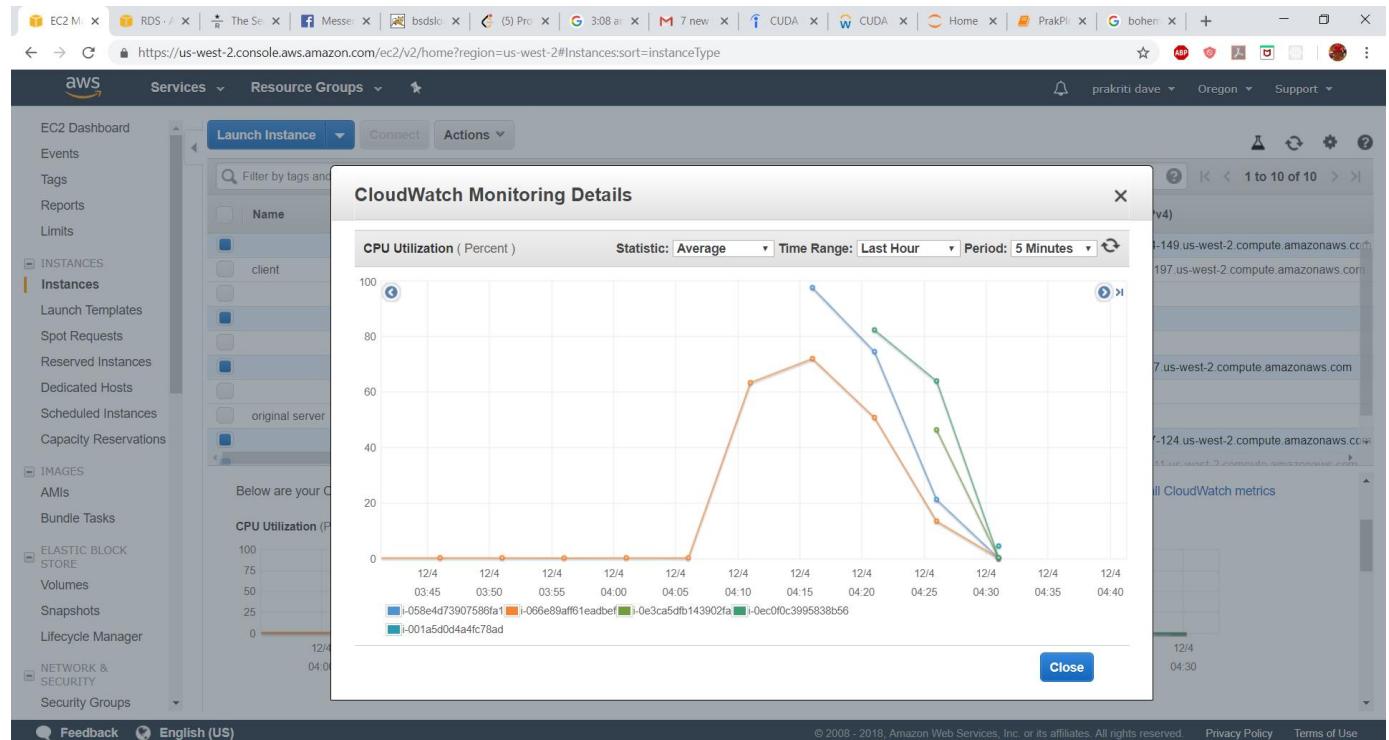


The screenshot shows the AWS CloudWatch Metrics console with a line graph titled "Average Latency". The x-axis represents time from 0 to 800, and the y-axis represents latency from 0 to 140. The graph shows a highly fluctuating blue line with several sharp peaks reaching up to 140, indicating high variability in latency during the peak phase.

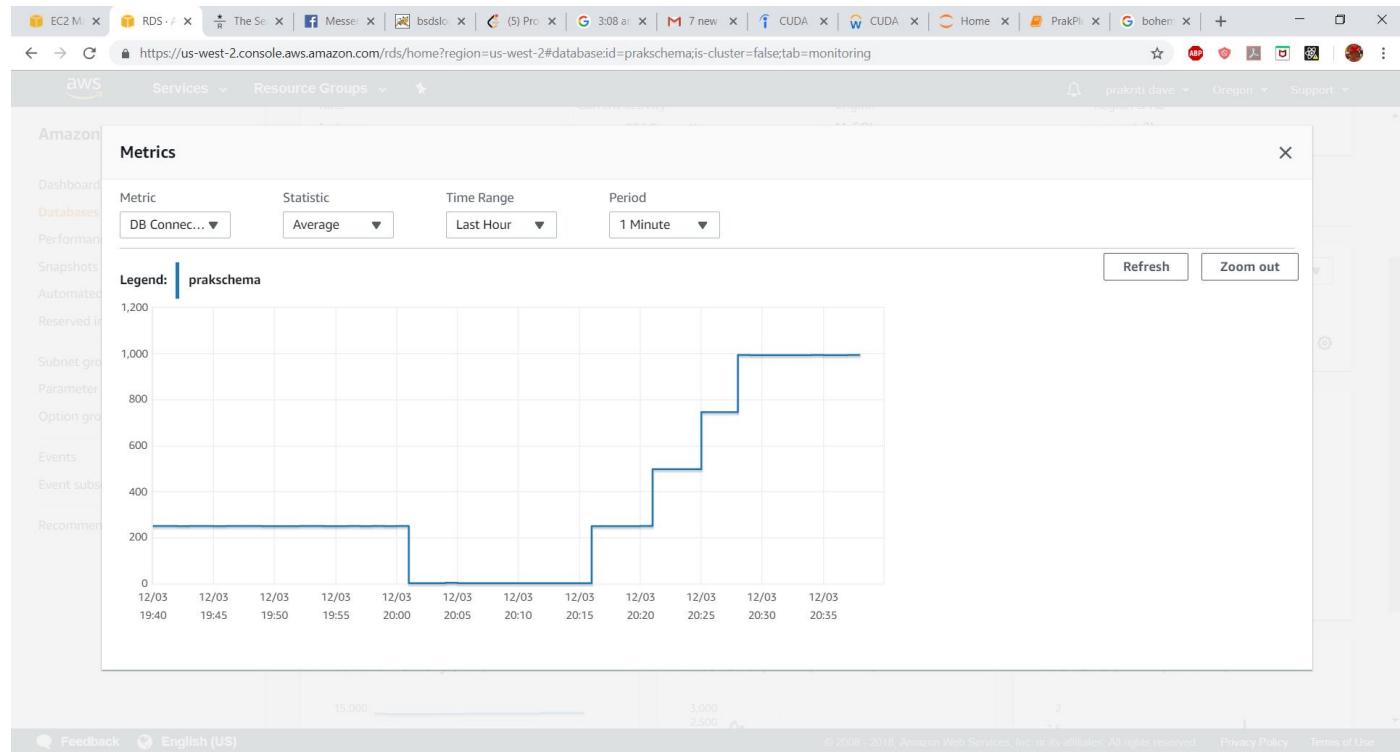
Below the graph, the text "All 5 instances were spawned for 350 threads, during the Peak Phase" is displayed.

The AWS CloudWatch Metrics console interface includes a sidebar with navigation links for EC2 Dashboard, Events, Tags, Reports, Limits, Instances, Launch Templates, Spot Requests, Reserved Instances, Dedicated Hosts, Scheduled Instances, Capacity Reservations, Images, AMIs, and Bundle Tasks. The main content area shows the metrics for the "newAutoScalingGroup" instance group, with a table of 4 instances, all of which are healthy and in the InService lifecycle state.

CPU utilization :



RDS connections:



450 Threads, 100 iterations:

For 450 threads, all my 5 instances are spawned during peak phase. However my peak phase is still running and 4 of my instances have terminated so there seems to be a bottleneck from client to server, since the RDS size is t2.xlarge and it can handle 1400 connections. So after a while the throughput is pretty less even during the peak phase.

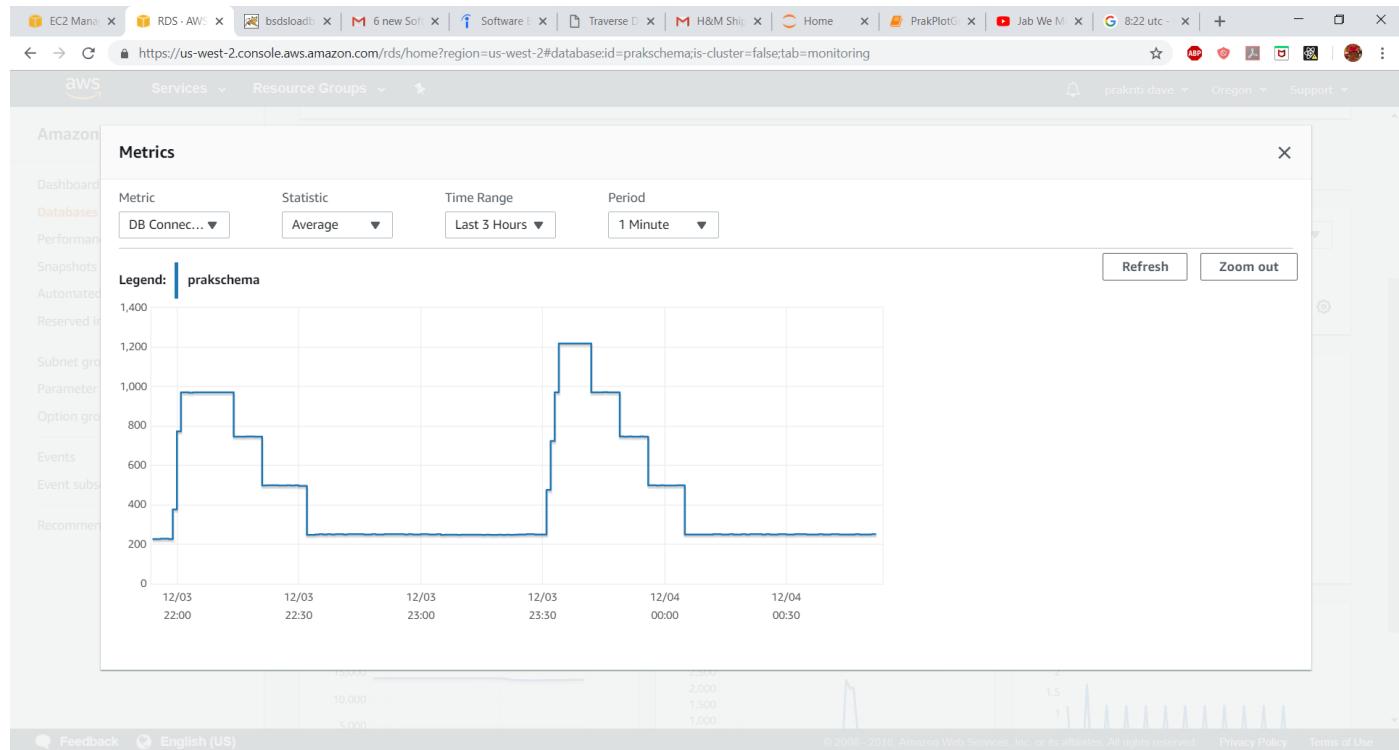
```
ec2-user@ip-172-31-46-87:~  
PS C:\Users\prakrkit1\Downloads> PS C:\Users\prakrkit1\Downloads> ssh -i "bsdsFirstKeyPair.pem" ec2-user@ec2-34-216-54-197.us-west-2.compute.amazonaws.com  
Last login: Tue Dec 4 06:37:33 2018 from 65.122.177.243  
[ec2-user@ip-172-31-46-87 ~]  
[ec2-user@ip-172-31-46-87 ~]$ [ec2-user@ip-172-31-46-87 ~]$ [ec2-user@ip-172-31-46-87 ~]$ [ec2-user@ip-172-31-46-87 ~]$ clear  
[ec2-user@ip-172-31-46-87 ~]$ java -jar PrakClient 450 http://bsdsloadbalancer-664473791.us-west-2.elb.amazonaws.com:8080/AnotherProjectWar/rest/myfirstapp 1 100000 100  
Running phase Warmup  
number of warmup threads 45  
Running phase Warmup  
number of threads submitted45  
67500  
Number of loadingThreads 225  
Running phase Loading  
number of threads submitted225  
630000  
Number of peak threads 450  
Running phase Peak  
number of threads submitted450
```

The screenshot shows the AWS EC2 Management console with the URL <https://us-west-2.console.aws.amazon.com/ec2/v2/home?region=us-west-2#instances:sort=instanceType>. The left sidebar includes links for Load Balancing, Auto Scaling, Systems Manager Services, and Systems Manager Shared Resources. The main content area displays a table of instances:

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)
client	i-007a9242f54a62009	t2.micro	us-west-2b	running	2/2 checks ...	None	ec2-34-216-54-197.us-west-2.compute.amazonaws.com
	i-012acdd83734b4a50	t2.micro	us-west-2b	running	2/2 checks ...	None	ec2-34-216-10-134.us-west-2.compute.amazonaws.com
	i-02e8a04fc0a8392da	t2.micro	us-west-2a	running	2/2 checks ...	None	ec2-34-222-145-216.us-west-2.compute.amazonaws.com
	i-03b74894fc49f73d2	t2.micro	us-west-2a	running	2/2 checks ...	None	ec2-54-190-18-70.us-west-2.compute.amazonaws.com
	i-09783c0a946997944	t2.micro	us-west-2a	running	2/2 checks ...	None	ec2-34-223-223-22.us-west-2.compute.amazonaws.com
	i-0be481ac1980cf5f1	t2.micro	us-west-2b	running	2/2 checks ...	None	ec2-54-185-165-40.us-west-2.compute.amazonaws.com
original server	i-0d91506078cb31616	t2.micro	us-west-2b	stopped			

At the bottom, there are links for Feedback, English (US), Copyright notice (2008-2018), Privacy Policy, and Terms of Use.

DB connections:



Points of observation/Lessons learned the hard way:

1. For almost a week I was stuck with 256 threads, I kept getting the error either **Connection Time out or Bind error exception, Address not available**. This happened with the Load Balancer and I was not able to figure out why. This was the reason I moved my client to ec2 as well. I later realized I could get away with just one webTarget object but I was creating a new one for each request, which was one of the main reasons why my resources were getting exhausted at the network side. I also modified a couple of my settings on the DBCP pool configuration and Tomcat server. I changed my RDS instance size to t2.medium and then later to t2.xlarge for better performance with load balancer.
2. Before I moved my client to ec2 for 256 threads , I had also executed my client on local, both with and without load balancer upto 128 threads, and surprisingly the throughput graph looks better for them. 😊

I have uploaded my results for client run on local as well, in my github repo.