# How to use Data Science and ML for AWS DevOps

Gustavo Ribeiro Amigo

# Intro

Gustavo Ribeiro Amigo

- Senior SDE @ Amazon Prime Video

  https://www.linkedin.com/in/gustavoamigo/

  https://github.com/gustavoamigo

  https://twitter.com/gustavoamigo

# Agenda

How can we use Python and its tooling to fine tune your service to perfection?

Tools:

- Pandas for data manipulation- https://pandas.pydata.org/
- Jupyter for notebooking - https://jupyter.org/
- Locust for load testing - https://locust.io/
- boto3 for AWS client - https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

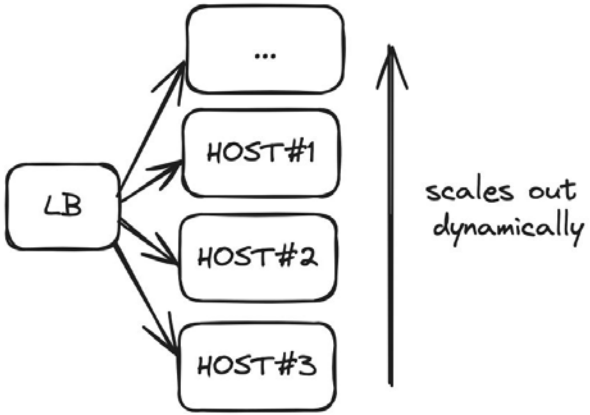
Target Audience

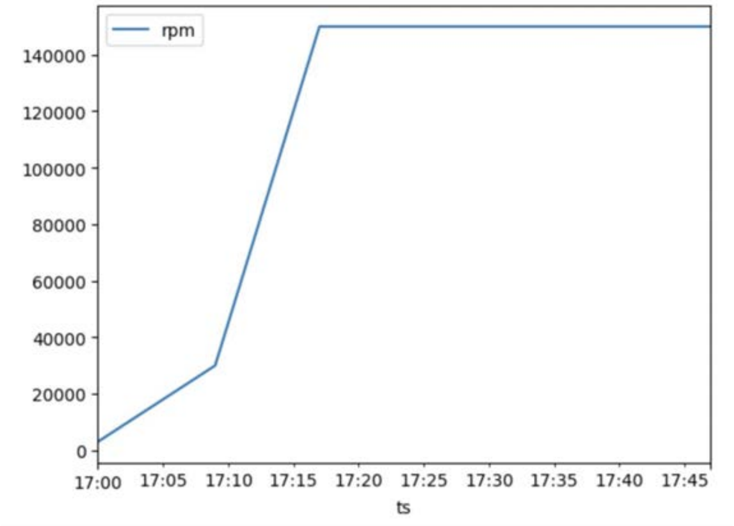
- Software Engineers / SRE / DevOps

# Scaling is Hard

- Testing is hard, specially in production
- Reasoning about scaling is hard
- Infrastructure is expansive

# Problem




How can you scale your application to this traffic shape?
Without problems and efficiently?

# Context

- It’s Python Application deployed on AWS Beanstalk
  - LB is Application Load Balancer
  - Application is deployed on EC2 boxes
  - Instance type t3.micro
- Scaling policy parameters
  - Scale up/down increment - Current: 1/-1
  - Upper threshold - Current: 25%
  - Down threshold - Current: 15%
  - Metric is average CPU Utilization

# Approach

- Load test one host
- Retrieve data from AWS to Pandas
- Determine one host capacity
- Run local experiments to find the best parameters
- Test parameters in production
- Analyse results

# Load test one host

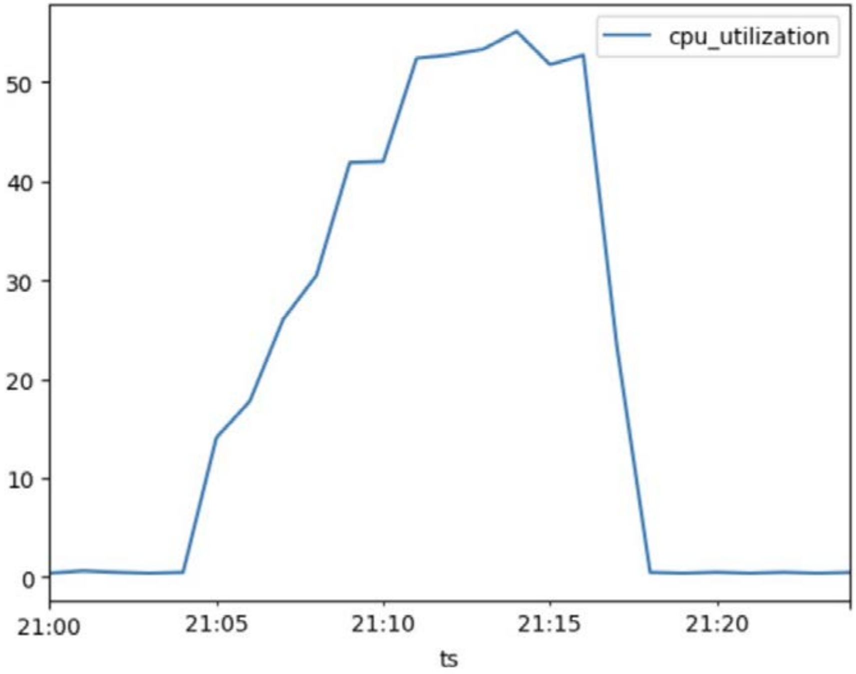

locust.io

# Retrieve data from AWS to pandas

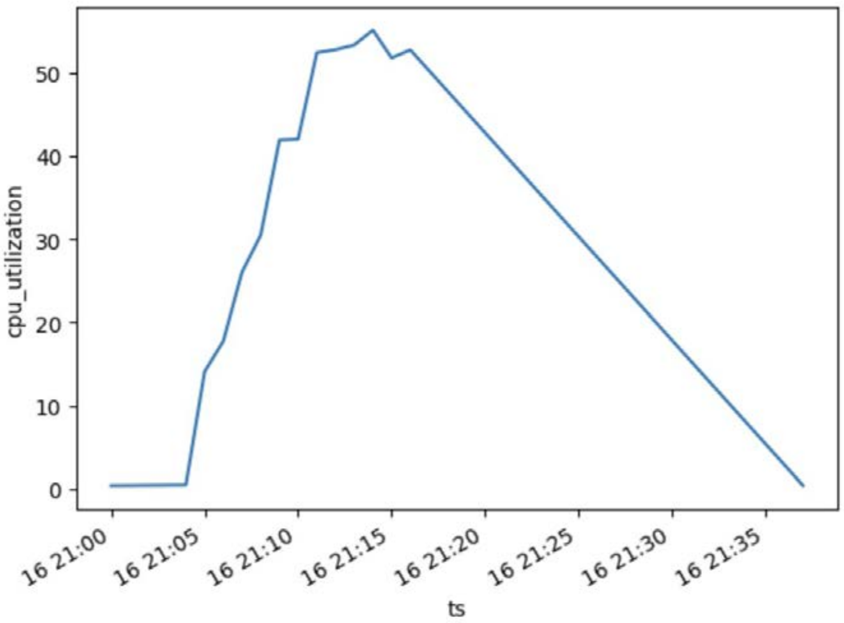
```
start_time = datetime.fromisoformat('2024-02-16 21:00:00+00:00')
end_time = datetime.fromisoformat('2024-02-16 21:25:00+00:00')

cpu_utilization_data = cloudwatch.get_metric_data(
    MetricDataQueries=[
        {
            'Id': 'm1',
            'MetricStat': {
                'Metric': {
                    'Namespace': 'AWS/EC2',
                    'MetricName': 'CPUUtilization',
                    'Dimensions': [
                        {
                            'Name': 'AutoScalingGroupName',
                            'Value': 'awseb-e-3kmqmtr6e2-stack-AWSEBAutoScalingGroup-CEAartlYOGnq'
                        }
                    ]
                },
                'Period': 60,
                'Stat': 'Average',
            },
            'ReturnData': True
        }
    ],
    StartTime=start_time,
    EndTime=end_time,
    )
cpu_utilization_data_result = cpu_utilization_data['MetricDataResults'][0]
cpu_df = pd.DataFrame.from_dict({'ts': cpu_utilization_data_result['Timestamps'], 'cpu_utilization':cpu_utilization_data_result['Values']})
cpu_df = cpu_df.set_index('ts')
cpu_df.plot()
```

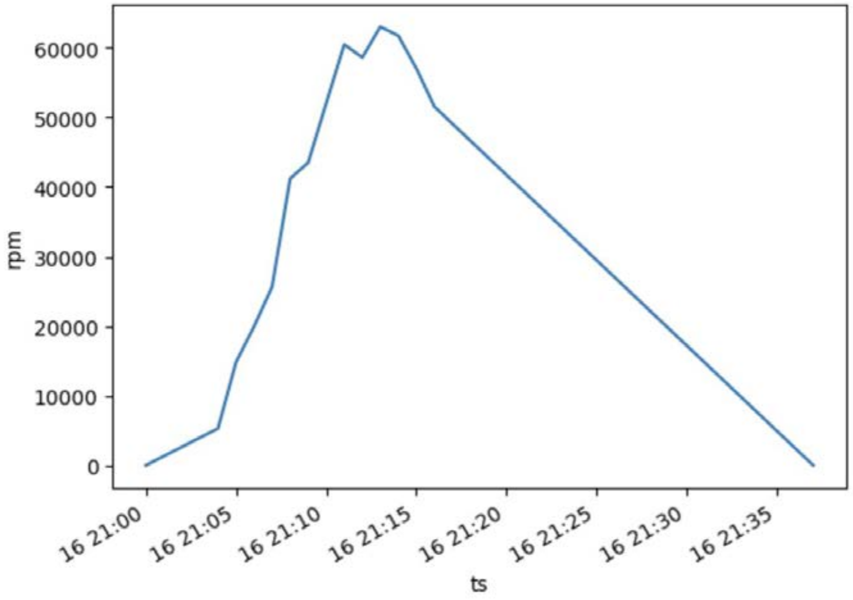
pro-tip:
save to csv

# Retrieve data from AWS to pandas

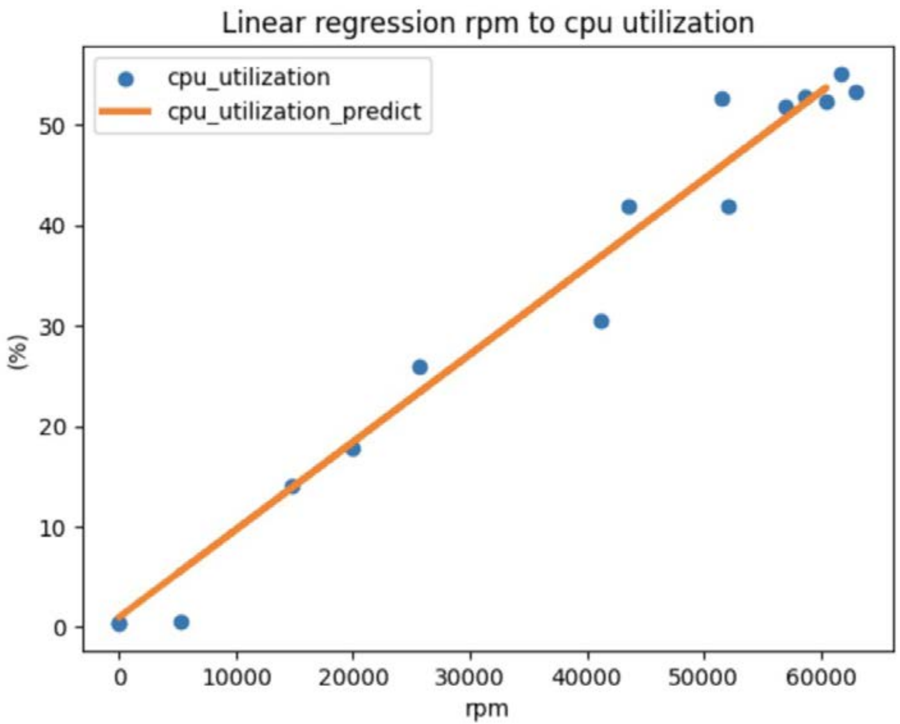


# Analyse one host performance



# We can do better



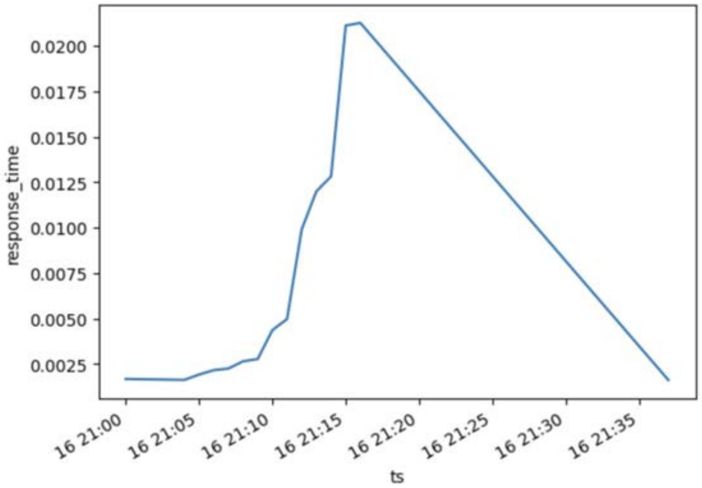

Takeaways:

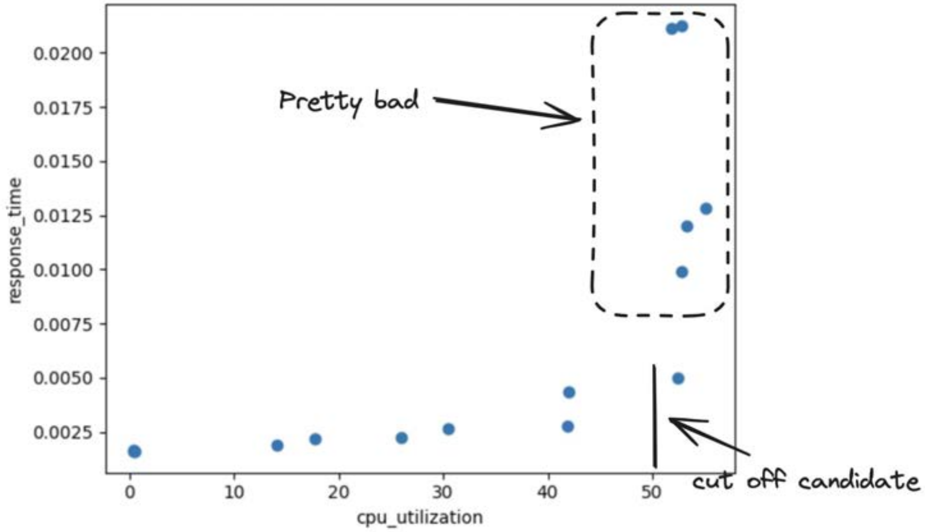
-> cpu rpm versus cpu is linear

-> cpu_avg_coef = 0.0008967

-> 40k RPM ~ 35% cpu utilization

# Another thing - latency




My application is really bad when cpu is over 50%

## How can we scale?

Without problems?

- CPU below 50%

Efficiently?

- Using AWS autoscaling - but how to configure it?
- Using most CPU at all time (ie maximize average CPU utilization) - how to find them?

# Approaches

- Test multiple parameters in production
  - Risky
  - It’s high effort
- Run local experimentations using Python ←

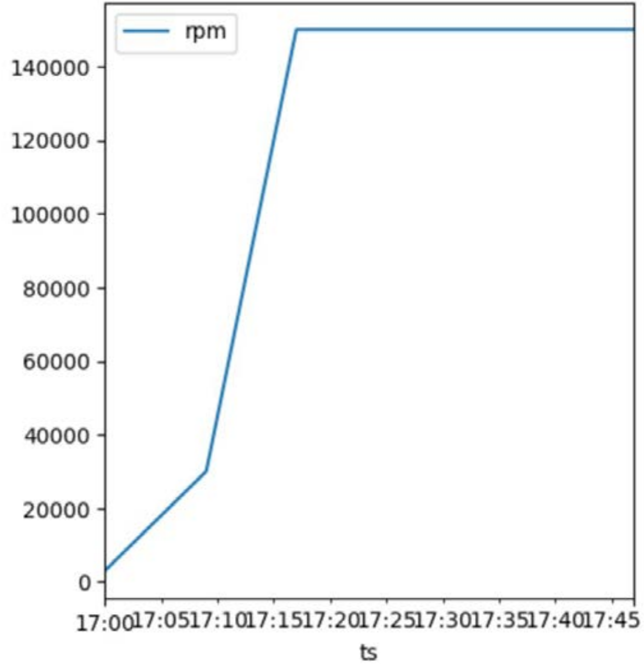
# Create a scaling simulator

```
def scaling_sim(df, cpu_utilization_to_rpm_coef,
                upper_threshold = 25,
                lower_threshold = 15,
                start_up_time = 4,
                scale_up_increment = 1,
                scale_down_increment = -1):
```

* link to the code in the e

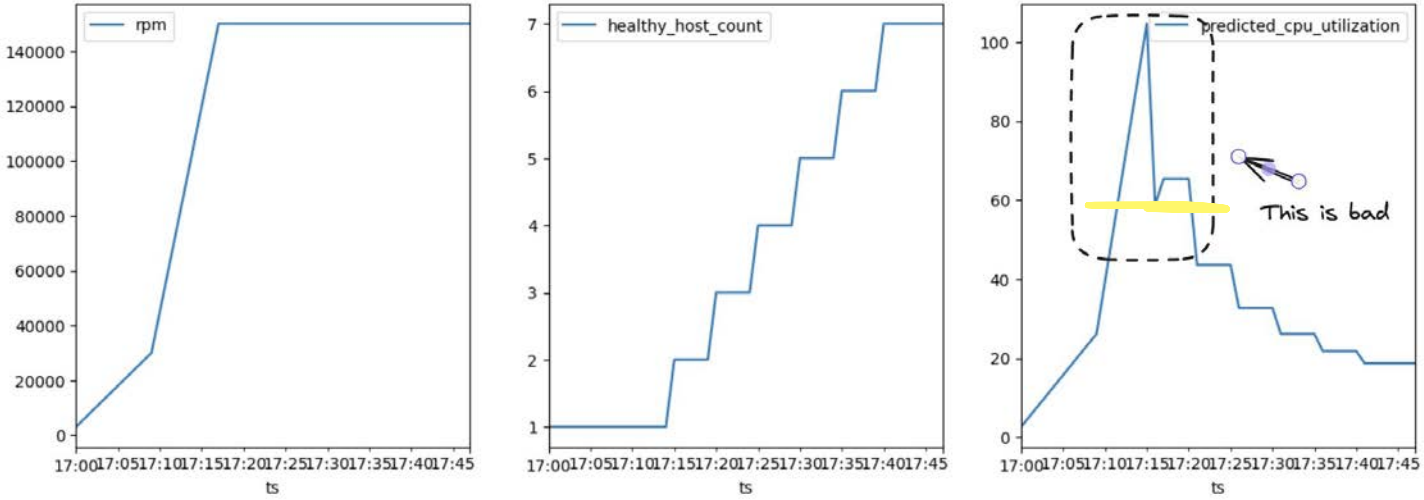
# Create a traffic shape generator

```
stages = [
    {'duration': 10, 'rpm': 30000},
    {'duration': 8, 'rpm': 150000},
    {'duration': 30, 'rpm': 150000}
]

load_shape_df = load_shape(stages)
```



# And run a first local experiment

```
load_shape_df = load_shape(stages)
result_df = scaling_sim(load_shape_df, cpu_avg_coef)
```



# What does it means?

If our simulator is accurate enough, the parameters below won’t scale as CPU utilization will breach our 50% limit

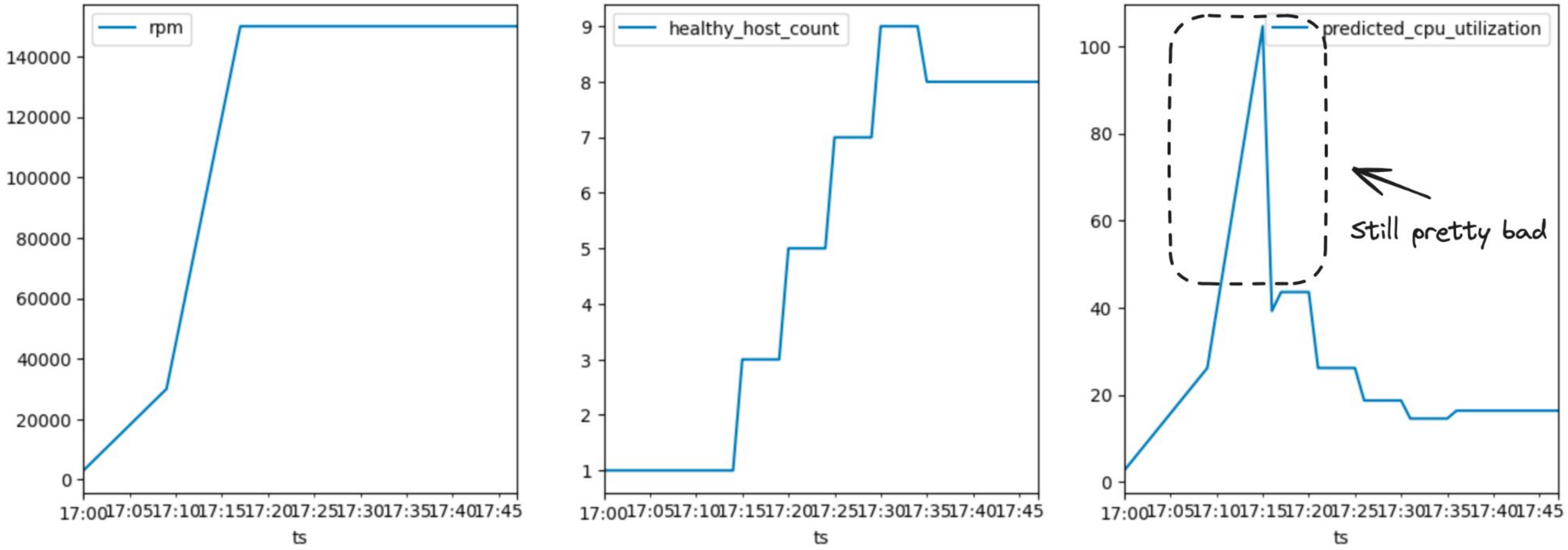
upper_threshold = 25
scale_up_increment = 1

* *lower_threshold* and *scale_down_increment* won’t matter as they only impact scaling in/down your application.

# First attempt

Change *scale_up_increment* to 2

```
result_df = scaling_sim(load_shape_df, cpu_avg_coef, scale_up_increment=2)
```



# Find the best parameters

```
simulations = []
for scale_up_increment_var in range(1, 10):
    for upper_threshold_var in range(11, 35):
        result_sim_df = scaling_sim(load_shape_df, cpu_avg_coef,
                                    scale_up_increment = scale_up_increment_var,
                                    upper_threshold = upper_threshold_var ,
                                    lower_threshold = round(upper_threshold_var * 0.7))
        max_cpu_in_simulation = result_sim_df['predicted_cpu_utilization'].max()
        if(max_cpu_in_simulation < 50): # only consider simulations where cpu was below 50 percent
            simulations += [{
                'scale_up_increment': scale_up_increment_var,
                'upper_threshold': upper_threshold_var,
                'lower_threshold':   round(upper_threshold_var * 0.7),
                'average_cpu_utilization': result_sim_df['predicted_cpu_utilization'].mean(),
                'max_cpu_utilization': max_cpu_in_simulation
            }]
simulations_df = pd.DataFrame.from_records(simulations)
```

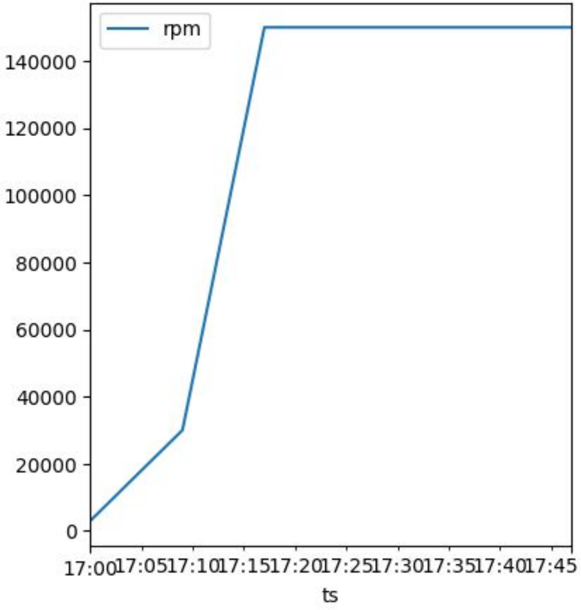
# And the winner is

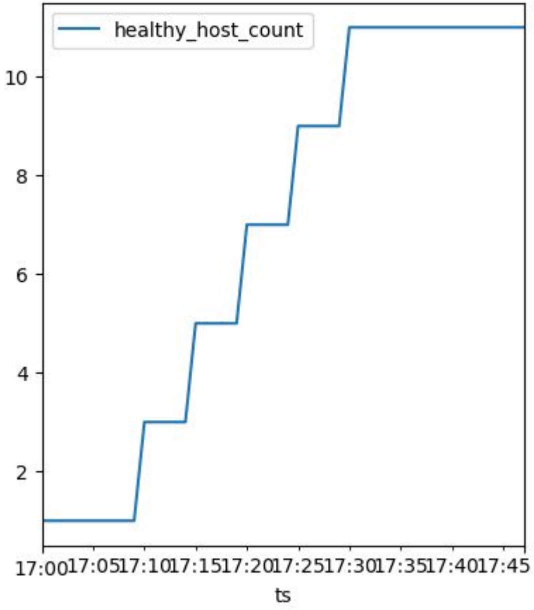
```
best_param = simulations_df.sort_values(by=['average_cpu_utilization', 'upper_threshold', 'scale_up_increment'],
best_param
```

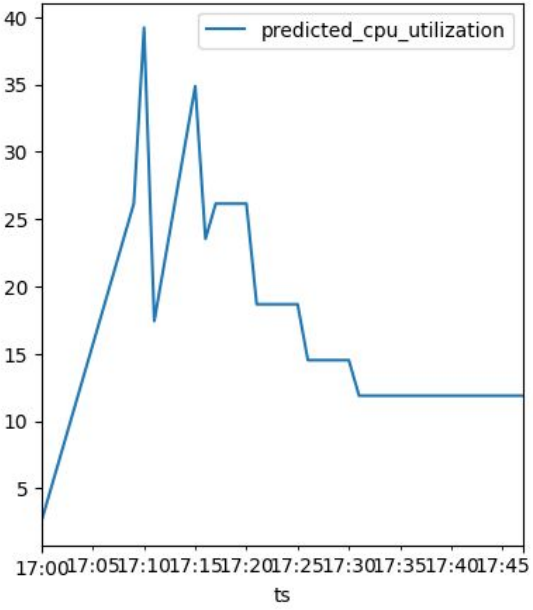
✓ 0.0s

| | scale_up_increment | upper_threshold | lower_threshold | average_cpu_utilization | max_cpu_utilization |
|---|---|---|---|---|---|
| 4 | 2 | 15 | 10 | 16.876191 | 39.226077 |

# Which actually looks better

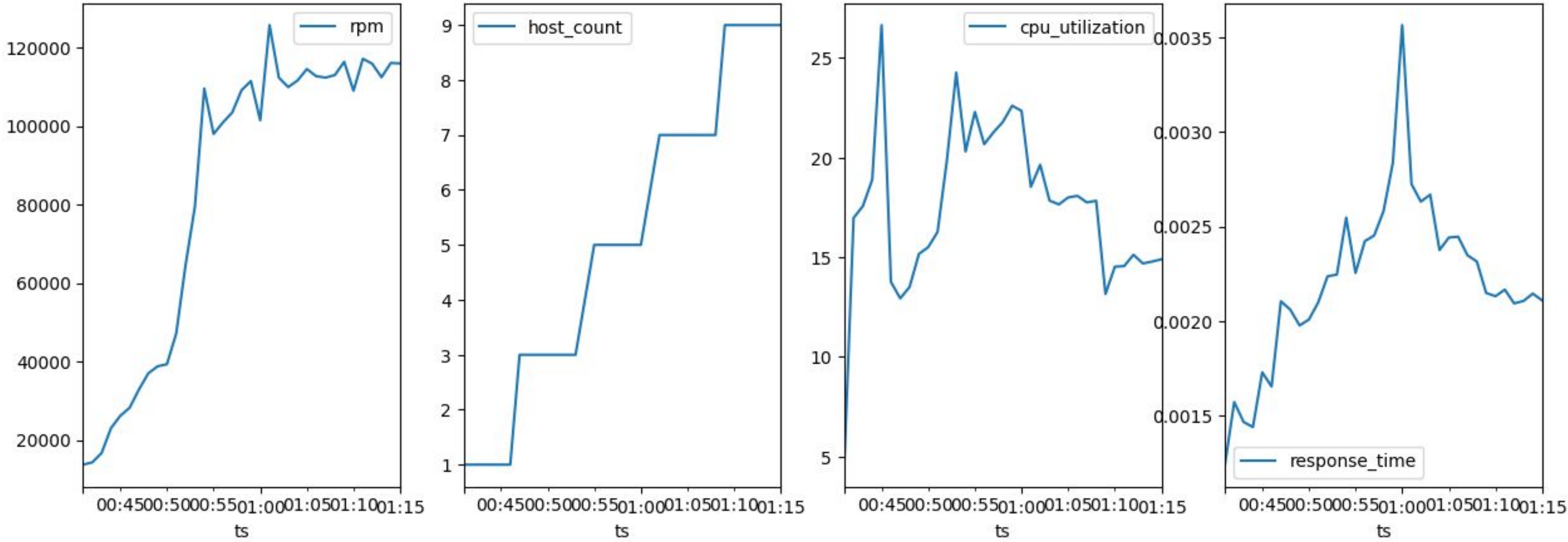


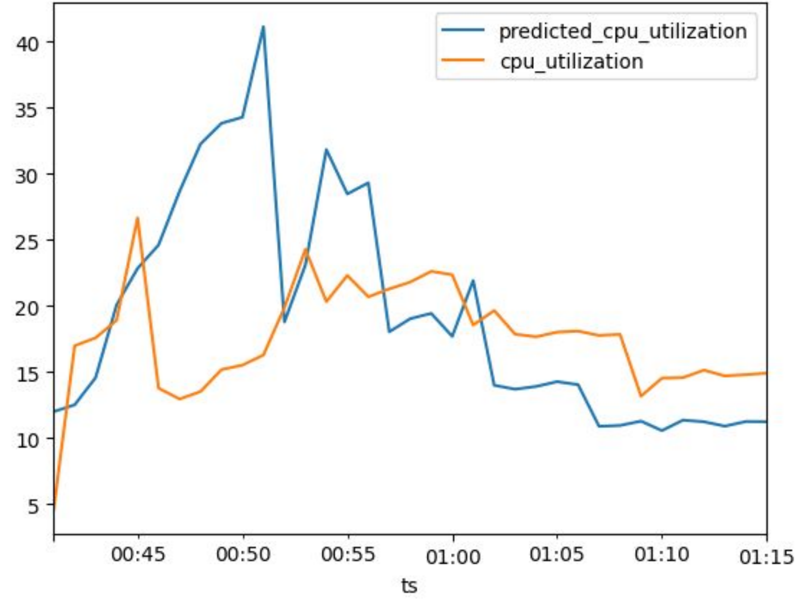



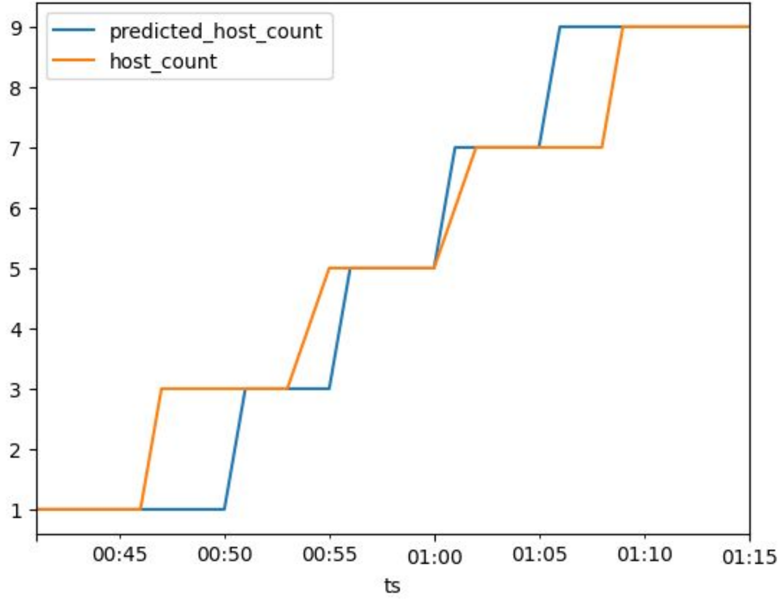
# Testing new parameters in production

# Testing new parameters in production



# Simulation versus Real life



# Conclusion

- Changing upper_threshold to 15 and scale_up_increment to 2 worked to scale our application **efficiently** and **without problems** as per our load test.

# Code

## Code created for presentation here

https://github.com/gustavoamigo/conf42-python-24

## Tools used:

- Pandas for data manipulation- https://pandas.pydata.org/
- Jupyter for notebooking - https://jupyter.org/
- Locust for load testing - https://locust.io/
- boto3 for AWS client - https://boto3.amazonaws.com/v1/documentation/api/latest/index.html