Performance Modeling and Evaluation of Serverless AI Applications in Edge-Cloud

Presentation for the Research Project in Software Engineering II

Caspar Dietz, Marko Prascevic

Agenda

1. Research Goal
2. OpenFaas in Kubernetes Setup
3. Distributed Load Testing with JMeter Setup
4. Performance Evaluation
5. Results
6. Conclusion

Research Goal

Our Case Study: Face Blur

- Benchmark and evaluate the performance of AI applications running in FaaS
- We study the performance using an open-source face blur repository
- Given a request rate/load of the client, we want to measure the response time of the serverless application running on PoliCloud

OpenFaas in Kubernetes

Setup

Setup Overview

JMeter Coordinator

Possible Configurations:
- Request rate
- Number of users
- Ramp-up time

Results:

A. JMeter:
- Mean time of Users
- Median time of Users
- User Success rate

B. Python:
- Total elapsed time of the python execution
- Time to process face blur
- Mean Response Time HTTP
- Median Response Time HTTP
- Request Success rate

Create Users

JMeter Slave

.MP4

Cuts .mp4 file into .png frames

Converts .png frames into JSON

POST request

JSON

Server (1 Replica)

POST request to JSON

Retrieves .png from JSON

.PNG

Calls Face Blur Functions

Converts blurred images (.png) into JSON

JSON

Send blurred Images as Response to the Client

JSON

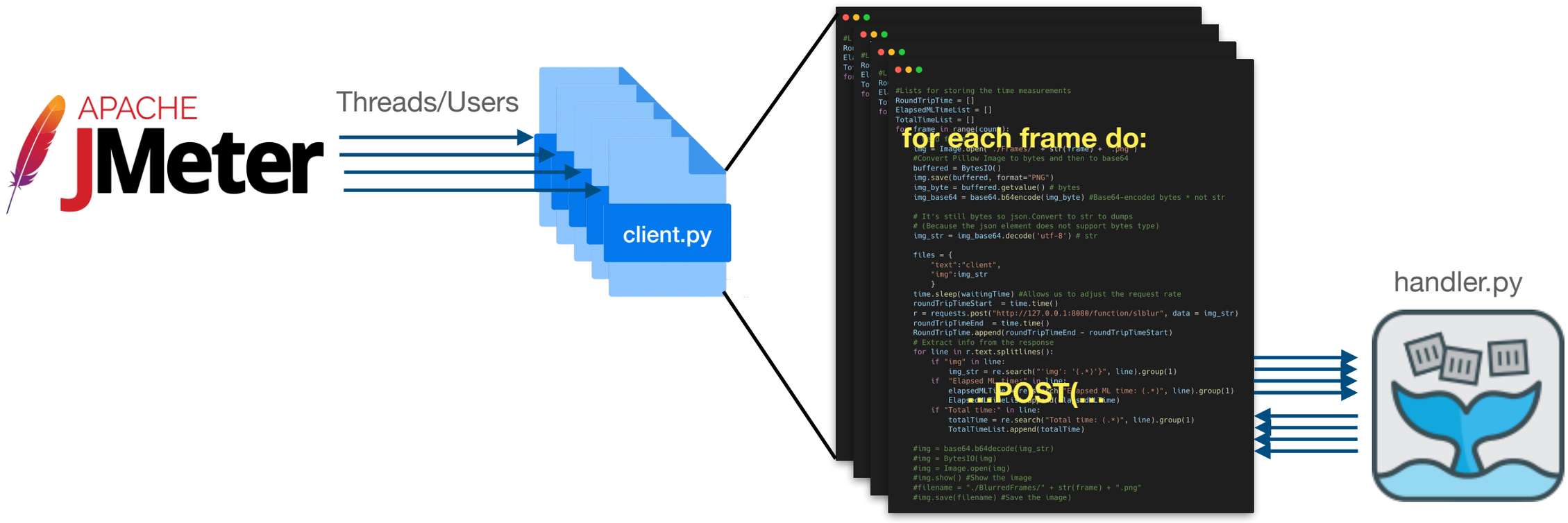
Retrieves .png from JSON

Reassembling the .mp4

Setup

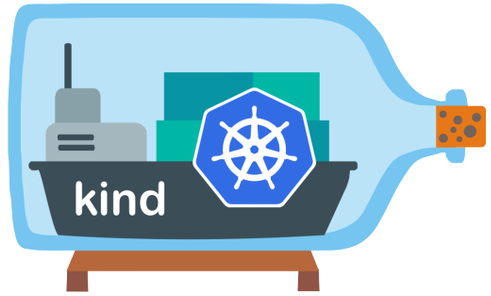
JMeter Simulates Users Running the Python Client

- JMeter OSProcess sampler runs threads on the client.py
- client.py cuts a given input video into frames and sends each frame (in a loop) to the server
- The server responds with the blurred frames form which the client can reconstruct the video



Setup

KinD: Local Kubernetes Cluster



Local registry on Port 5001

Local Kubernetes cluster we deploy into

```
#!/bin/sh
set -o errexit

# create registry container unless it already exists
reg_name='kind-registry'
reg_port='5001'
if [ "$(docker inspect -f '{{.State.Running}}' "${reg_name}" 2>/dev/null || true)" != 'true' ];
then
  docker run \
    -d --restart=always -p "127.0.0.1:${reg_port}:5000" --name "${reg_name}" \
    registry:2
fi

# create a cluster with the local registry enabled in containerd
cat <<EOF | kind create cluster --config=-
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
containerdConfigPatches:
- |-
  [plugins."io.containerd.grpc.v1.cri".registry.mirrors."localhost:${reg_port}"]
    endpoint = ["http://${reg_name}:5000"]
EOF

# connect the registry to the cluster network if not already connected
if [ "$(docker inspect -f='{{json .NetworkSettings.Networks.kind}}' "${reg_name}")" = 'null' ];
then
  docker network connect "kind" "${reg_name}"
fi

# Document the local registry
# https://github.com/kubernetes/enhancements/tree/master/keps/sig-cluster-lifecycle/generic/1755-communicating-a-local-registry
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: ConfigMap
metadata:
  name: local-registry-hosting
  namespace: kube-public
data:
  localRegistryHosting.v1: |
    host: "localhost:${reg_port}"
    help: "https://kind.sigs.k8s.io/docs/user/local-registry/"
EOF
```

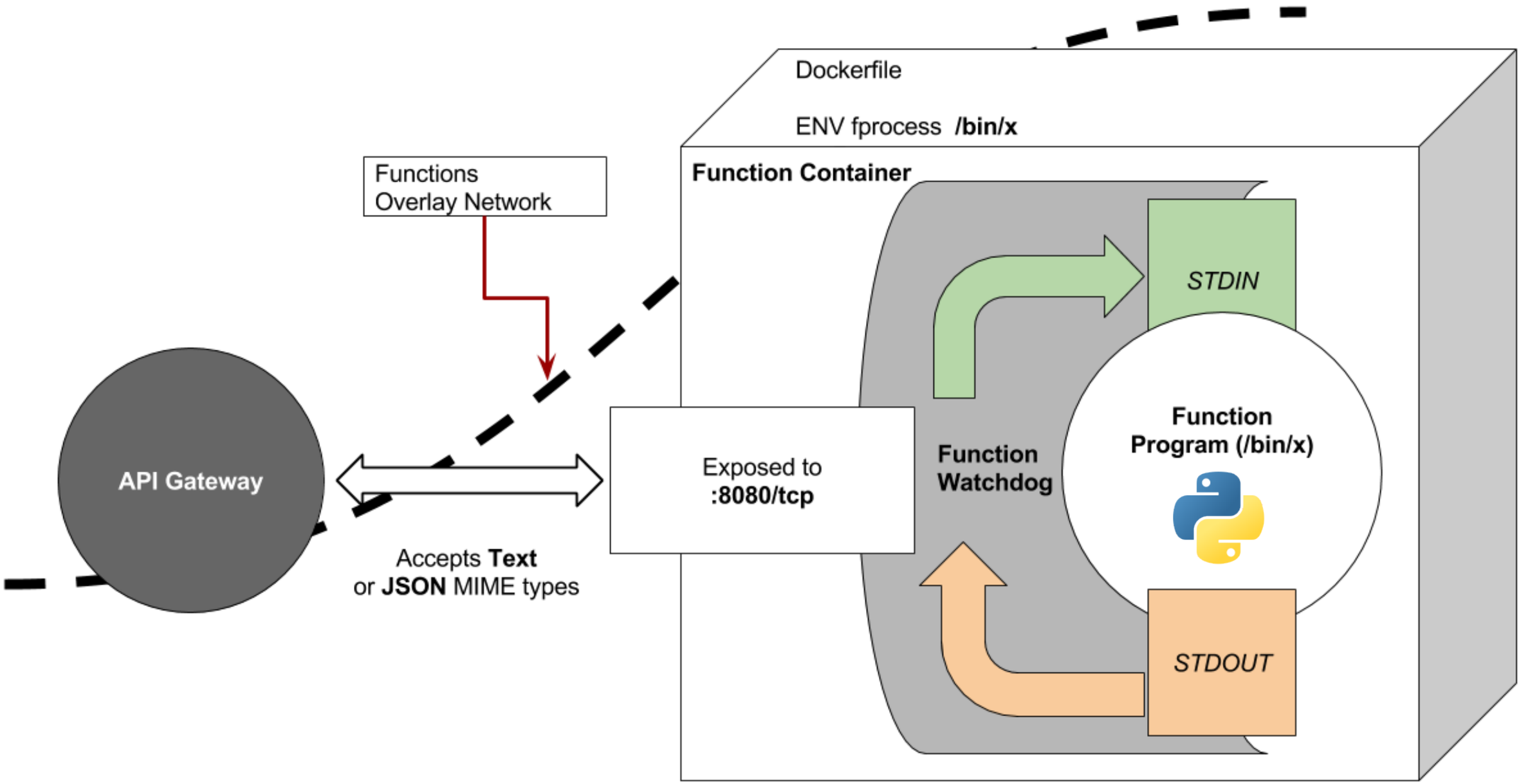
Setup

OpenFaas Watchdog

- The Function Watchdog is the entry-point allowing HTTP requests to be forwarded to the target process via STDIN
- The response is sent back to the caller by writing to STDOUT from your application
- Advantage: starts a new process encapsulating the appropriate function
  - Beneficial for long-running compute intensive tasks, as the concurrently running Python tasks do not need to share the same GIL (mutex/lock that allows only one thread to hold the control of the Python interpreter)
- Disadvantage: initiation of the Python3 interpreter with the user defined code can dominate the effective completion time of the function

Setup

OpenFaas Watchdog



Setup

Communication with our Cluster

- We need to port-forward the OpenFaas gateway service to our localhost port

- Although the the cluster is deployed locally, internally Kubernetes manages it’s own IP addresses

- We are port forwarding to access the service inside the Kubernetes cluster

```
kubectl port-forward -n openfaas svc/gateway 8080:8080
```

Setup

The Client

Cut the video into individual .png frames and store them

Frame by frame: open image, encode it and send it

Encode the frames (string -> Bytes)

Adjust the request rate by waiting until next POST

POST request to OpenFaas gateway

Extract the infos from the handler’s response

```python
#Lists for storing the time measurements
RoundTripTime = []
ElapsedMLTimeList = []
TotalTimeList = []
for frame in range(count):
    # Read frame
    img = Image.open('./Frames/' + str(frame) + '.png')
    #Convert Pillow Image to bytes and then to base64
    buffered = BytesIO()
    img.save(buffered, format="PNG")
    img_byte = buffered.getvalue() # bytes
    img_base64 = base64.b64encode(img_byte) #Base64-encoded bytes * not str

    # It's still bytes so json.Convert to str to dumps
    # (Because the json element does not support bytes type)
    img_str = img_base64.decode('utf-8') # str

    files = {
        "text":"client",
        "img":img_str
        }
    time.sleep(waitingTime) #Allows us to adjust the request rate
    roundTripTimeStart  = time.time()
    r = requests.post("http://127.0.0.1:8080/function/slblur", data = img_str)
    roundTripTimeEnd  = time.time()
    RoundTripTime.append(roundTripTimeEnd - roundTripTimeStart)
    # Extract info from the response
    for line in r.text.splitlines():
        if "img" in line:
            img_str = re.search("'img': '(.*)'}", line).group(1)
        if  "Elapsed ML time:" in line:
            elapsedMLTime = re.search("Elapsed ML time: (.*)", line).group(1)
            ElapsedMLTimeList.append(elapsedMLTime)
        if "Total time:" in line:
            totalTime = re.search("Total time: (.*)", line).group(1)
            TotalTimeList.append(totalTime)

    #img = base64.b64decode(img_str)
    #img = BytesIO(img)
    #img = Image.open(img)
    #img.show() #Show the image
    #filename = "./BlurredFrames/" + str(frame) + ".png"
    #img.save(filename) #Save the image)
```

Setup

The OpenFaas Handler

Take timestamps

Decode the text data into bytes

Detect faces and apply blur to frame

Convert to bytes and encode. Then convert the data that was bytes to string

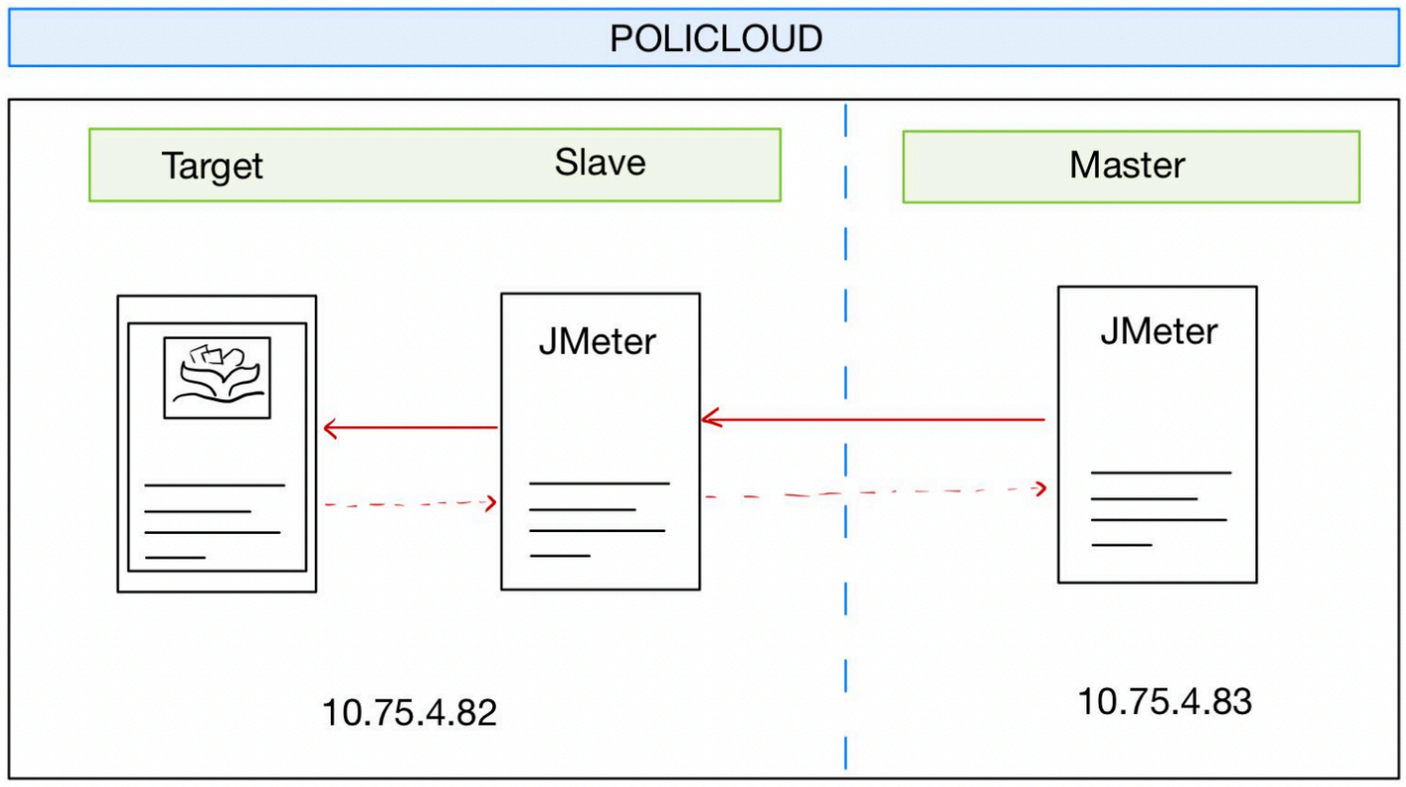
Return the JSON to the client

```python
def handle(req):
    total_start_time = time.time()
    img = base64.b64decode(req)
    img = BytesIO(img) # _io.Converted to be handled by BytesIO pillow
    img = Image.open(img)
    # create detection object
    detector = Detector('face.pb', name="detection")
    # real face detection
    image = np.array(img)
    faces = detector.detect_objects(image, threshold=0.4)
    # apply blurring
    image = blurBoxes(image, faces)
    """
    Prepare the image to be sent back to the client
    """
    #Convert Pillow Image to bytes and then to base64
    buffered = BytesIO()
    #img.save(buffered, format="PNG")
    img_byte = buffered.getvalue() # bytes
    img_base64 = base64.b64encode(img_byte) #Base64-encoded bytes * not str

    #It's still bytes so json.Convert to str to dumps
    #(Because the json element does not support bytes type)
    img_str = img_base64.decode('utf-8') # str

    response = {
        "text":str(total_end_time - total_start_time),
        "img":img_str
        }
    total_end_time = time.time()
    print("Total time: " + str(total_end_time - total_start_time))
    return response
```

Setup for the JMeter Distributed Load Testing

Distributed JMeter

Overview of the Setup



Terminology :

- Master node: the system running the the JMeter application (in the scope of this project in a non-GUI mode) and controls the tests by using different set-ups (configured by .jmx files)
- Slave node: the system running the jmeter-server, which takes commands from the master node generating the loads to the target system
- Target: system under test

Distributed JMeter

Distributed Load Testing

Pre-Requisites:

- Download and install Apache JMeter
- Same version of Apache JMeter on all the machines
- Both the Master and Slave machines should be at same subnet. RMI cannot communicate across subnets without a proxy; therefore, neither can JMeter without a proxy
    - The RMI provides remote communication between the java applications. JMeter uses Remote Method Invocation (RMI) as the remote communication mechanism
- Same version of Java on all machines

Distributed JMeter

Master Engine Configuration

1. Open `jmeter.properties` present in the bin directory on the Master machine
2. Set `server.rmi.ssl.disable=true`, since we don't want to use SSL for RMI
3. Set `server.rmi.port=8001`. This is the port that will be used to communicate with the server. This is a public port.
4. Set `client.rmi.localport=8001`. JMeter/RMI also requires a reverse connection in order to return sample results from the server to the client
5. `remote_hosts=X.X.X.X` (Slave Machine IP address)
6. Needed for capturing responses:
    1. jmeter.save.saveservices.response_data=true
    2. jmeter.save.saveservices.response_data.on_error=true
7. For not collecting responses in batches mode=Standard

Distributed JMeter

Slave Engine Configuration

1. Open `jmeter.properties` file present in the bin directory on the Slave machine

2. Set `server.rmi.ssl.disable=true`, since we don't want to use SSL for RMI

3. Set `server_port=8001`

4. Set `server.rmi.localport=8001`.

Distributed JMeter

Firewall Configuration

- On both the slave (server) side and the master side (client), it is needed to disable firewall rules. Since the machines use the UFW firewall settings, the command to allow communication through a certain port is “sudo ufw allow port_number”

- If client.rmi.local port is non-zero, it will be used as the base for local port numbers for the client engine. At the moment JMeter will open up to three ports beginning with the port defined in. So to avoid any traffic blockage by firewall, we need to open port range 8001–8003

- On the slave machine it is needed to also enable communication over the 8001 port

Once all these steps are accomplished, start the communication

Distributed JMeter

Starting up the execution

- On the server side it is necessary to start JMeter server. First navigate to the bin folder in the apache-jmeter folder. Afterwards execute the command:
    - ./jmeter-server -Jmode=standard
- After the command jmeter-server will become available and visible.
- On the client side (master machine) navigate to bin folder and execute the following command:
    - sh jmeter.sh -n -t path_to_jmx_files -l csv_file_to_be_saved -e -o path_to_html_folder -R remote_master_ip
- Parameter meanings: -n is to start jmeter in non-gui mode, -t is to say that the imx file follows, -l the path to the civ file follows, -e -o are the parameters needed to specify that the jmeter results should create a html page with the JMeter dashboard, -R specifies the remote jmeter that we are trying to reach

Performance Evaluation

Experimental Setup

What we Control:

- **JMeter**:
  - Number of users/threads
  - Ramp-up time
- **Python**
  - Video source/amount of frames in the client

What we Measure:

- **JMeter**:
  - Median response time of users
  - Mean response time of users
  - User success rate
- **Python**
  - RoundtripTime/ResponseTime: time elapsed between sending and receiving the results on the client side
  - TotalHandlerTime: Total elapsed time of the python execution in the handler
  - MLTime: Time to process face blur in the handler

Experimental Setup

The VM Running the Serverless Function

- Dual Core Intel Xeon(R) Silver 4114 CPU @ 2.2GHz
- 16GiB DIMM RAM
- 20GiB EXT4 Volume

bin — dietz@ardagna-vm-3: ~ — ssh dietz@10.75.4.82 — 122×41

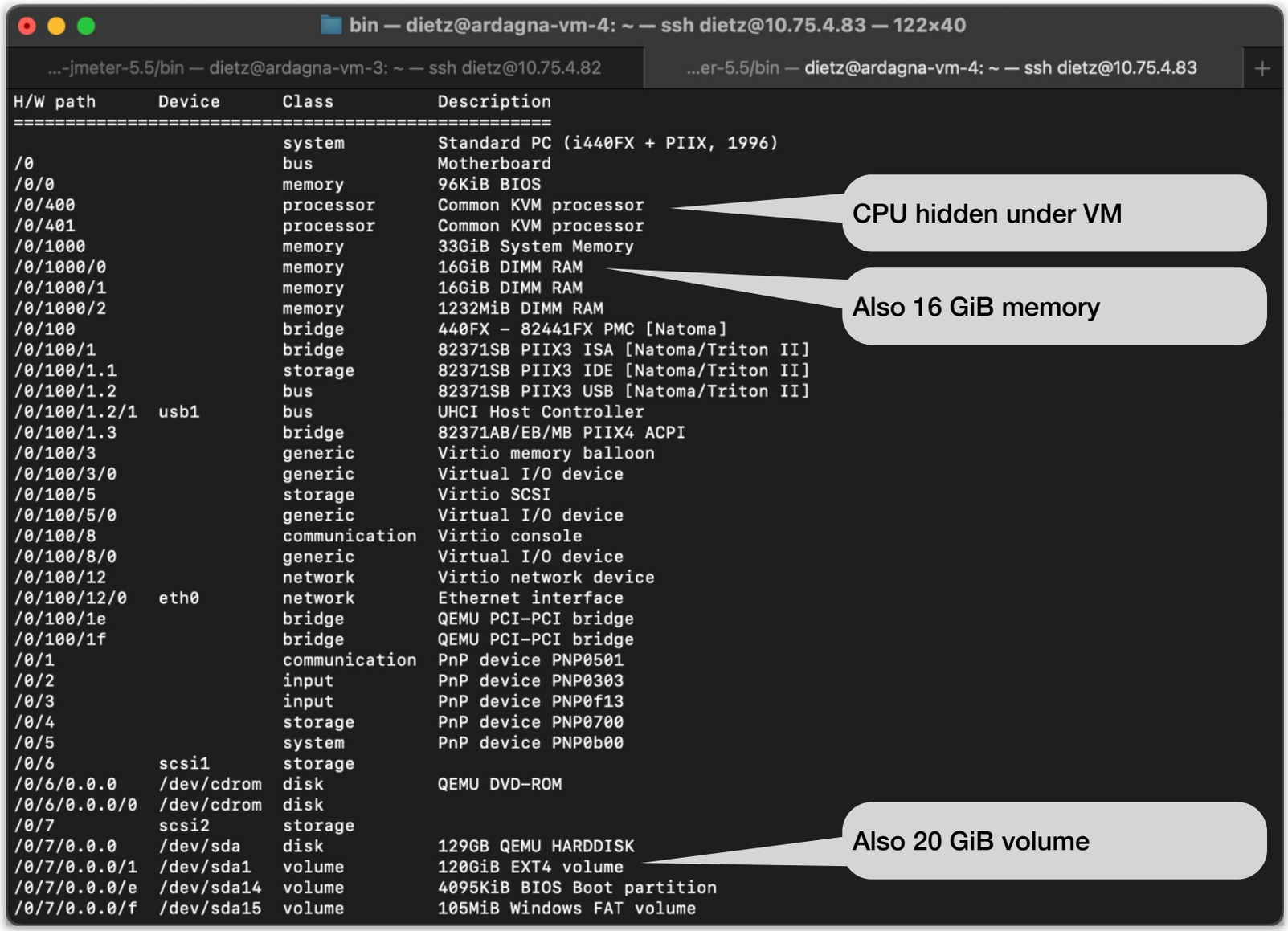
...-jmeter-5.5/bin — dietz@ardagna-vm-3: ~ — ssh dietz@10.75.4.82 | ...er-5.5/bin — dietz@ardagna-vm-4: ~ — ssh dietz@10.75.4.83

```
H/W path      Device      Class          Description
=====================================================
                          system         Standard PC (i440FX + PIIX, 1996)
/0                        bus            Motherboard
/0/0                      memory         96KiB BIOS
/0/400                    processor      Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz
/0/401                    processor      Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz
/0/1000                   memory         16GiB System Memory
/0/1000/0                 memory         16GiB DIMM RAM
/0/100                    bridge         440FX - 82441FX PMC [Natoma]
/0/100/1                  bridge         82371SB PIIX3 ISA [Natoma/Triton II]
/0/100/1.1                storage        82371SB PIIX3 IDE [Natoma/Triton II]
/0/100/1.2                bus            82371SB PIIX3 USB [Natoma/Triton II]
/0/100/1.2/1  usb1        bus            UHCI Host Controller
/0/100/1.3                bridge         82371AB/EB/MB PIIX4 ACPI
/0/100/3                  generic        Virtio memory balloon
/0/100/3/0                generic        Virtual I/O device
/0/100/5                  storage        Virtio SCSI
/0/100/5/0                generic        Virtual I/O device
/0/100/8                  communication  Virtio console
/0/100/8/0                generic        Virtual I/O device
/0/100/12                 network        Virtio network device
/0/100/12/0   eth0        network        Ethernet interface
/0/100/1e                 bridge         QEMU PCI-PCI bridge
/0/100/1f                 bridge         QEMU PCI-PCI bridge
/0/1                      communication  PnP device PNP0501
/0/2                      input          PnP device PNP0303
/0/3                      input          PnP device PNP0f13
/0/4                      storage        PnP device PNP0700
/0/5                      system         PnP device PNP0b00
/0/6          scsi1       storage
/0/6/0.0.0    /dev/cdrom  disk           QEMU DVD-ROM
/0/6/0.0.0/0  /dev/cdrom  disk
/0/7          scsi2       storage
/0/7/0.0.0    /dev/sda    disk           21GB QEMU HARDDISK
/0/7/0.0.0/1  /dev/sda1   volume         20GiB EXT4 volume
/0/7/0.0.0/e  /dev/sda14  volume         4095KiB BIOS Boot partition
/0/7/0.0.0/f  /dev/sda15  volume         105MiB Windows FAT volume
/1            veth771d7d3 network        Ethernet interface
/2            vethc7a77a5 network        Ethernet interface
/3            veth5bb8bd5 network        Ethernet interface
```

Experimental Setup

The VM Running the JMeter Master

- Apache JMeter version 5.3 to generate users that send HTTP requests invoking the function
- Set up on the same local network in order to minimise latency



Experimental Setup

JMeter Configuration

- Load configuration and initial request rate were configured in .jmx files
- Thread group:
    - Defines the number of threads (users)
    - Defines ramp-up time which presents the time that JMeter has to start up the entered amount of threads
- OS Process Sampler:
    - Added the path to the python directory and the command to be executed by the thread (i.e. client.py X). X represents the number of seconds the thread would sleep before sending a POST request
- Listeners were added to collect data
- In the scope of this project a “View result tree” listener was added to the thread group with the option of saving results to a xml file, which would contain messages from the client and the server

Experimental Setup

Python Handler

- Handler itself is written using only a buffer => We never write the image to the disk => Memory intensive but not disk intensive
- Tensorflow has to load a 22MB model (face.pb) => Disk intensive
- Base64 encoded data increases the file size more than 25%. This not only increases your bandwidth bill, but also increases the download time

Experimental Setup

Python Client

- With JMeter we configure an initial request rate: 0,25 t/s
- Since the POST is synchronous, this request rate will not be constant over the run of the threads. We therefore wanted another mean by which we can vary the response rate
- To further test the performance of our function, we measured the response time and success rate with varying amounts of sleep before the POST request
- The hypotheses is that this provides more time for the first POST requests to successfully go trough. This is due to the fact that the requests get spread and do not congest the handler all at once

Experimental Setup

OpenFaas

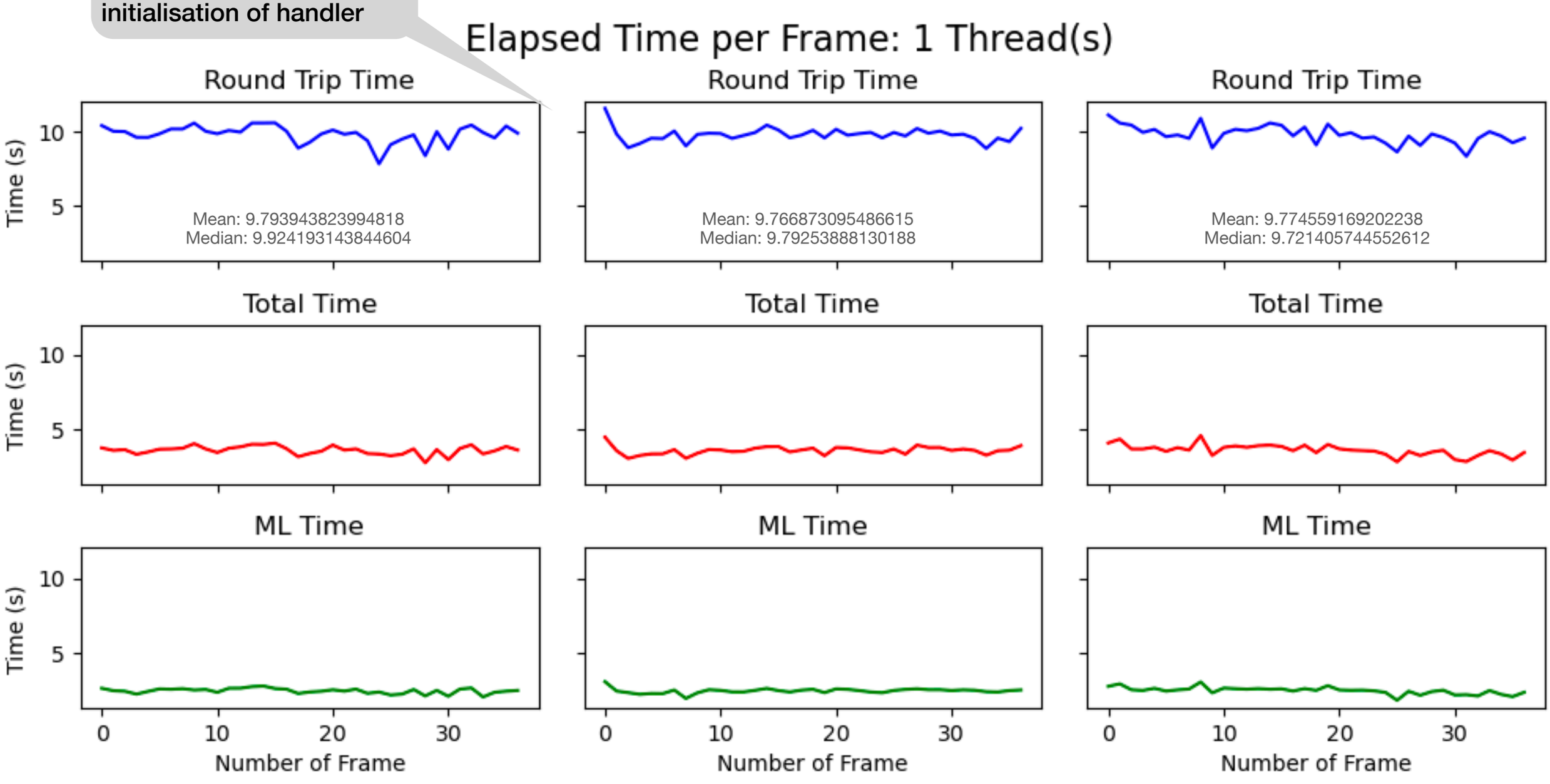
- We used the standard timeout configuration of OpenFaas: read_timeout: “25s”, write_timeout: “25s” and exec_timeout: “25s”
- This configuration is recommended by OpenFaas and is justified by our use case, i.e. blurring a frame should be possible within 25s

Results

Results

Round Trip Time, Total Time in Handler & ML Time in Handler

High response time for first frame because of initialisation of handler



0s Sleep: SR = 1

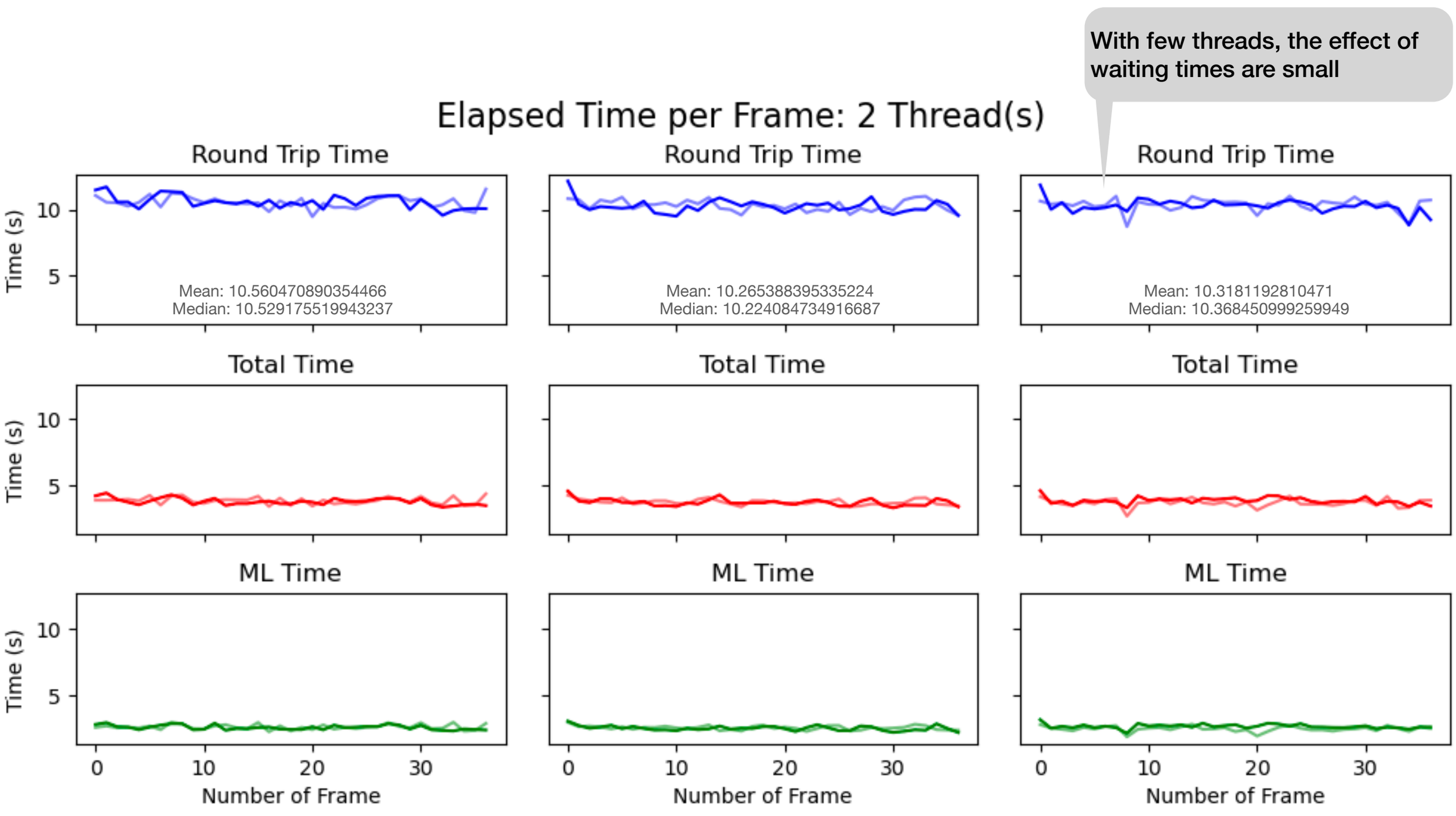
1s Sleep: SR = 1

5s Sleep: SR = 1

Results

Round Trip Time, Total Time in Handler & ML Time in Handler

With few threads, the effect of waiting times are small



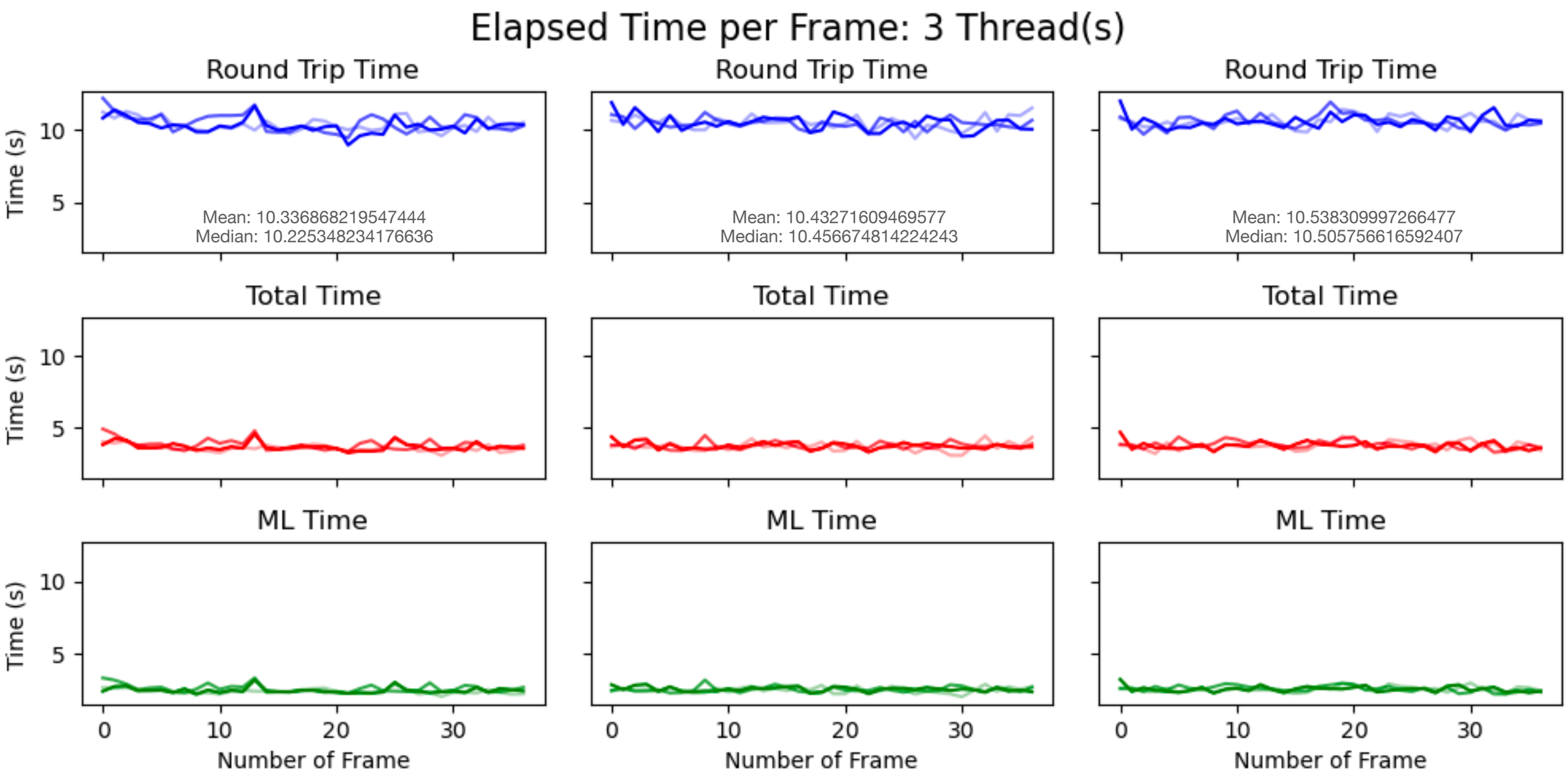
0s Sleep: SR = 1

1s Sleep: SR = 1

5s Sleep: SR = 1

Results

Round Trip Time, Total Time in Handler & ML Time in Handler



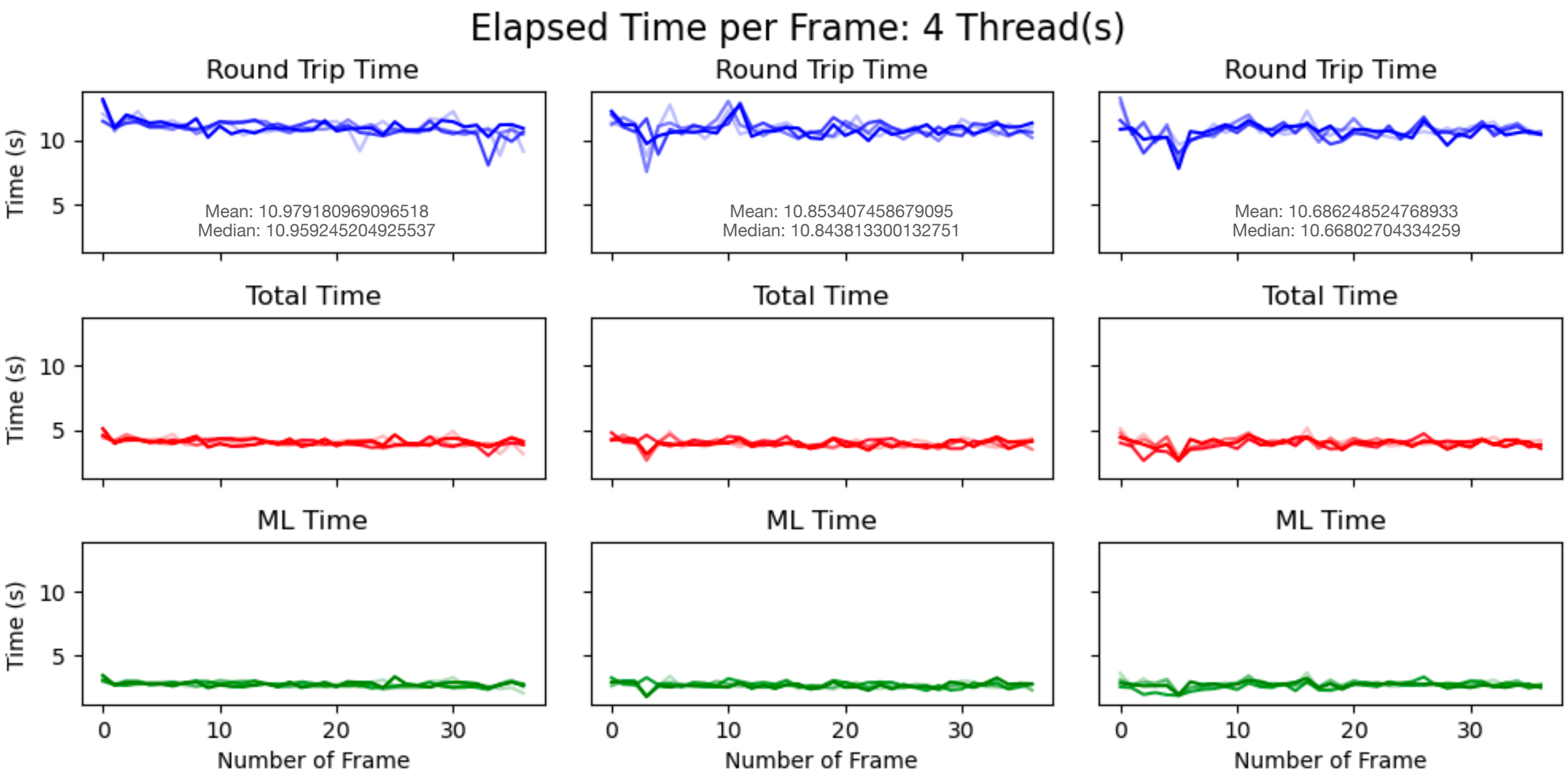
0s Sleep: SR = 1

1s Sleep: SR = 1

5s Sleep: SR = 1

Results

Round Trip Time, Total Time in Handler & ML Time in Handler



0s Sleep: SR = 1

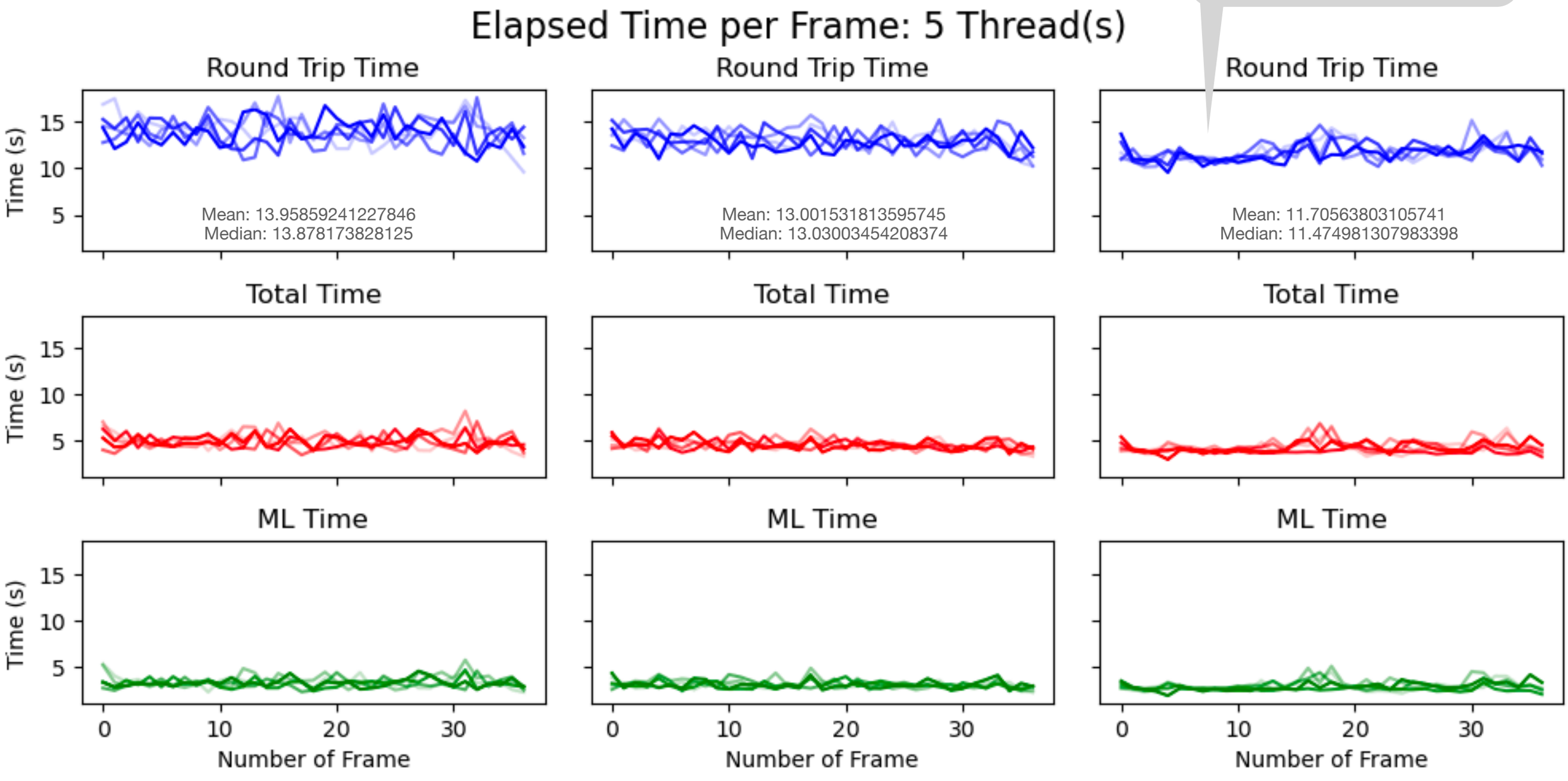
1s Sleep: SR = 1

5s Sleep: SR = 1

Results

Round Trip Time, Total Time in Handler & ML Time in Handler

Waiting time reduces the variance and mean in response time



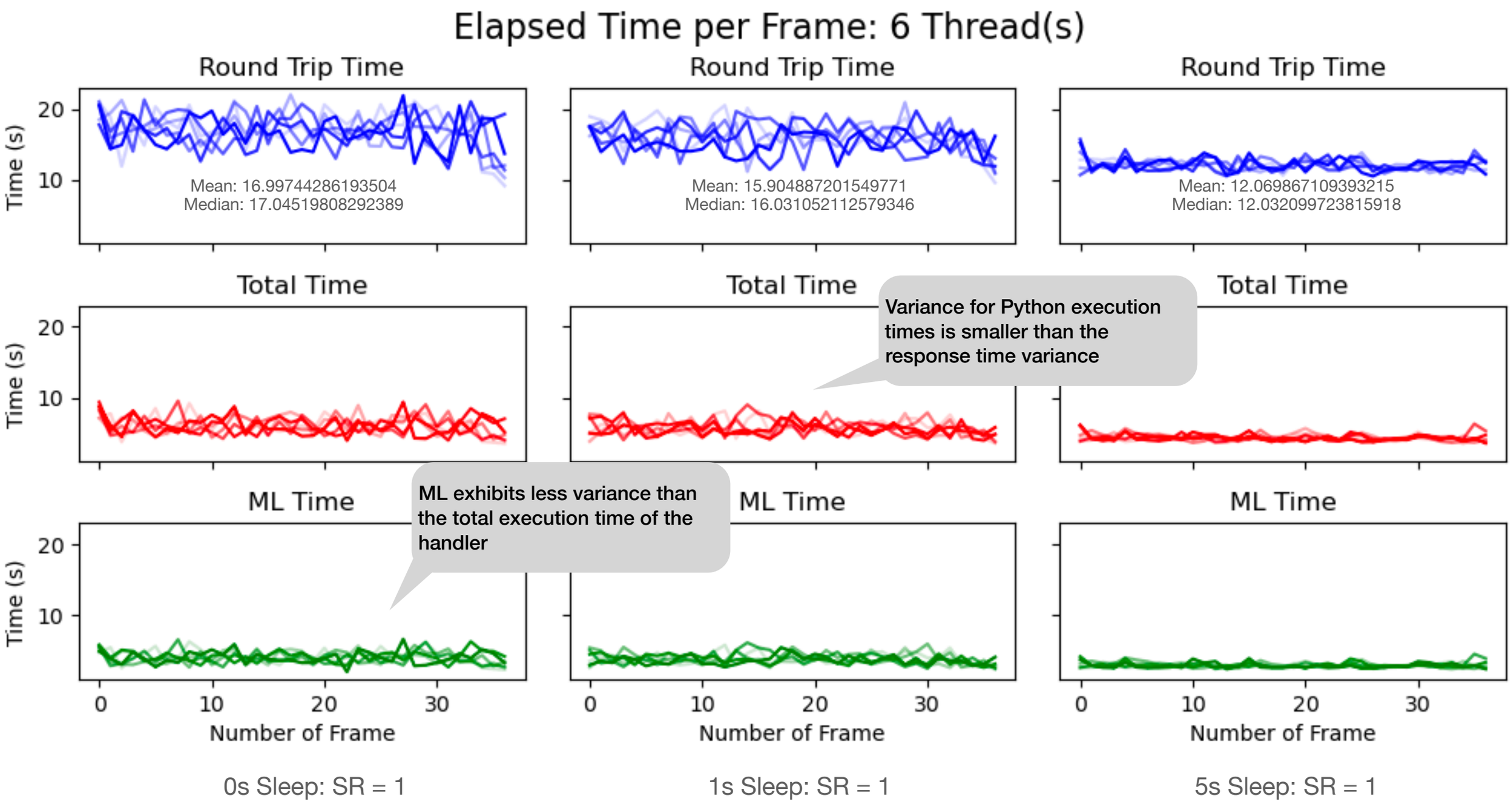
0s Sleep: SR = 1

1s Sleep: SR = 1

5s Sleep: SR = 1

Results

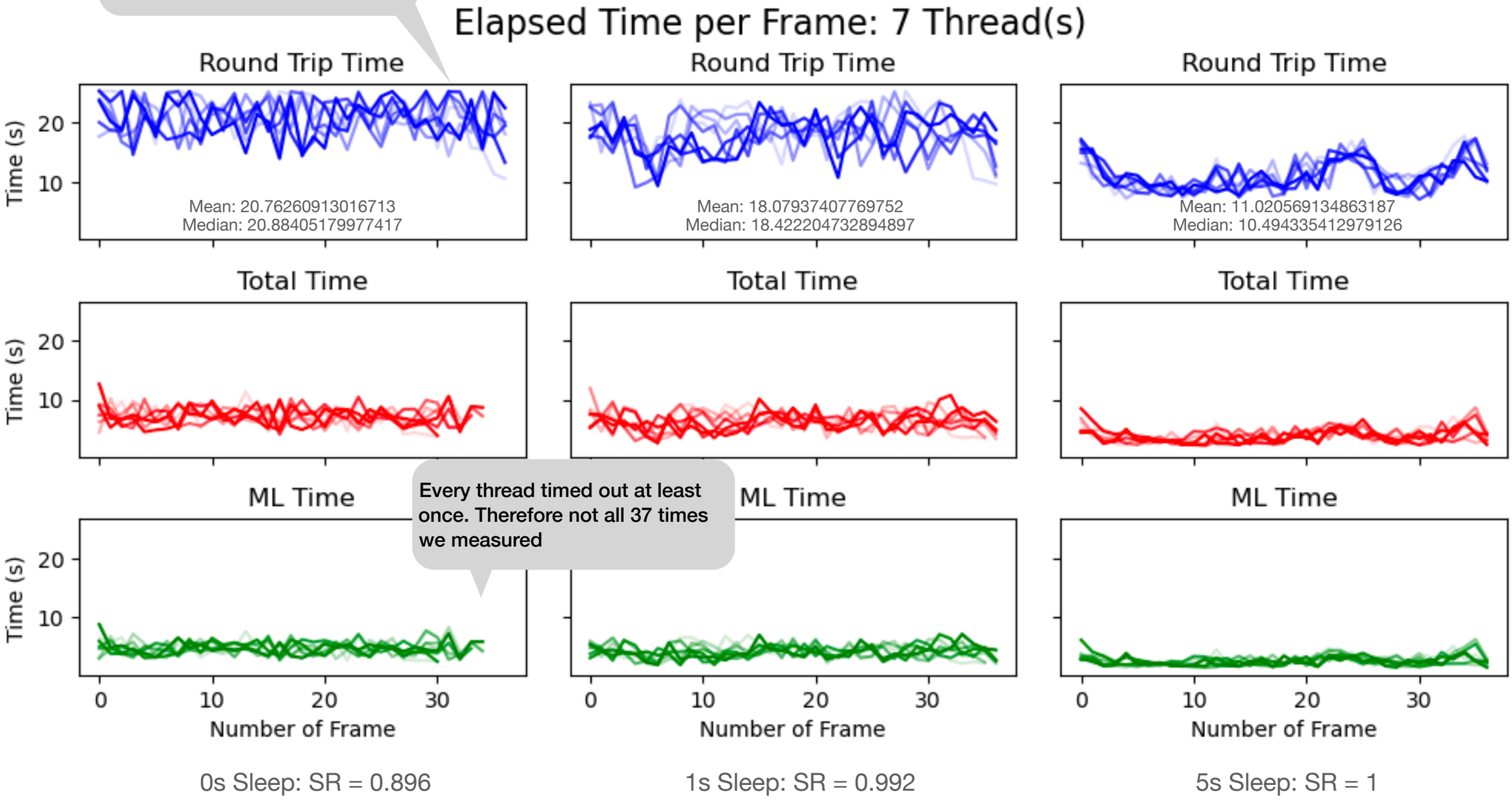
Round Trip Time, Total Time in Handler & ML Time in Handler



Results

Round Trip Time, Total Time in Handler & ML Time in Handler

POST requests timed out because of OpenFaas default timeout configuration



Every thread timed out at least once. Therefore not all 37 times we measured

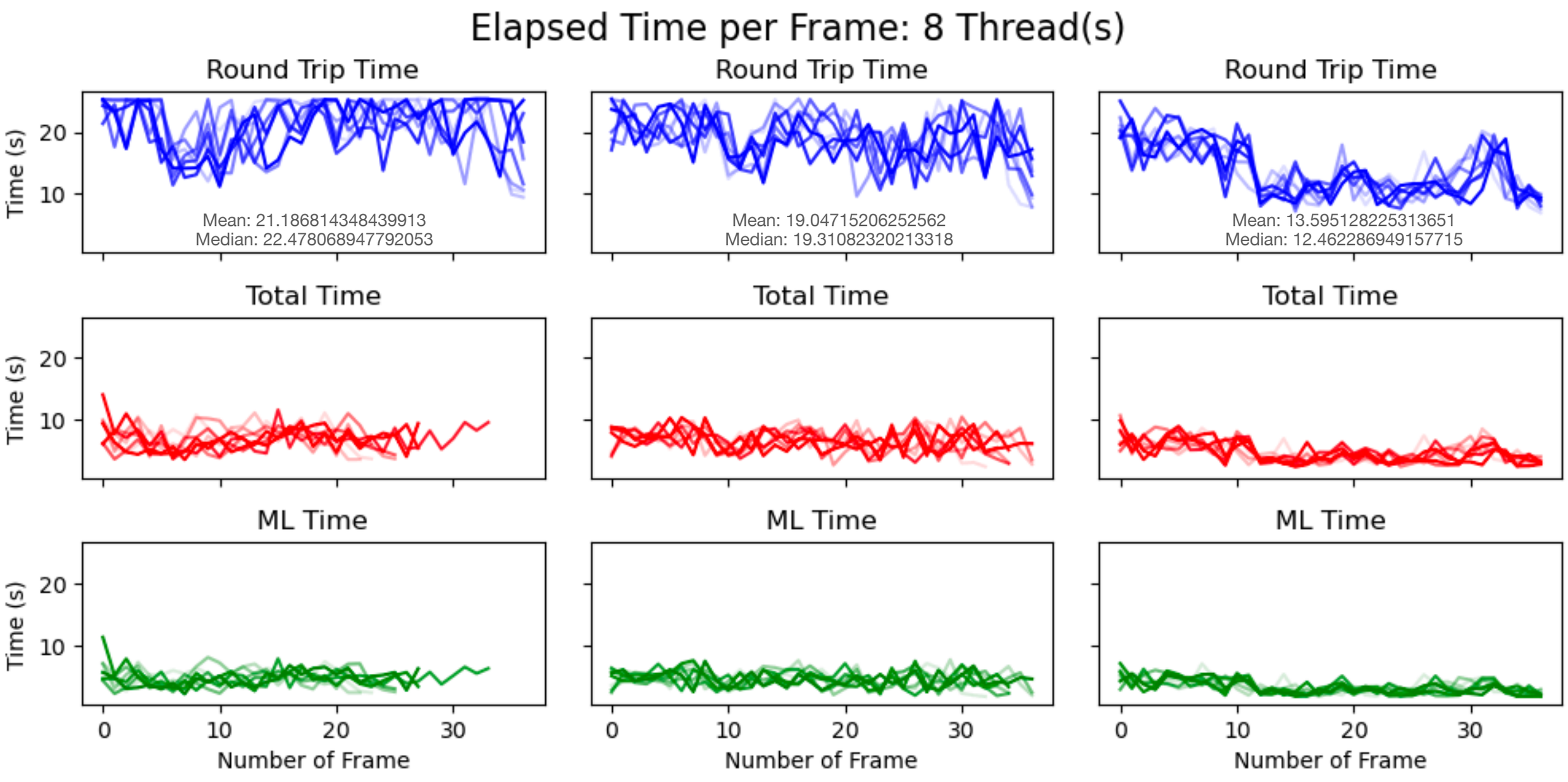
0s Sleep: SR = 0.896

1s Sleep: SR = 0.992

5s Sleep: SR = 1

Results

Round Trip Time, Total Time in Handler & ML Time in Handler



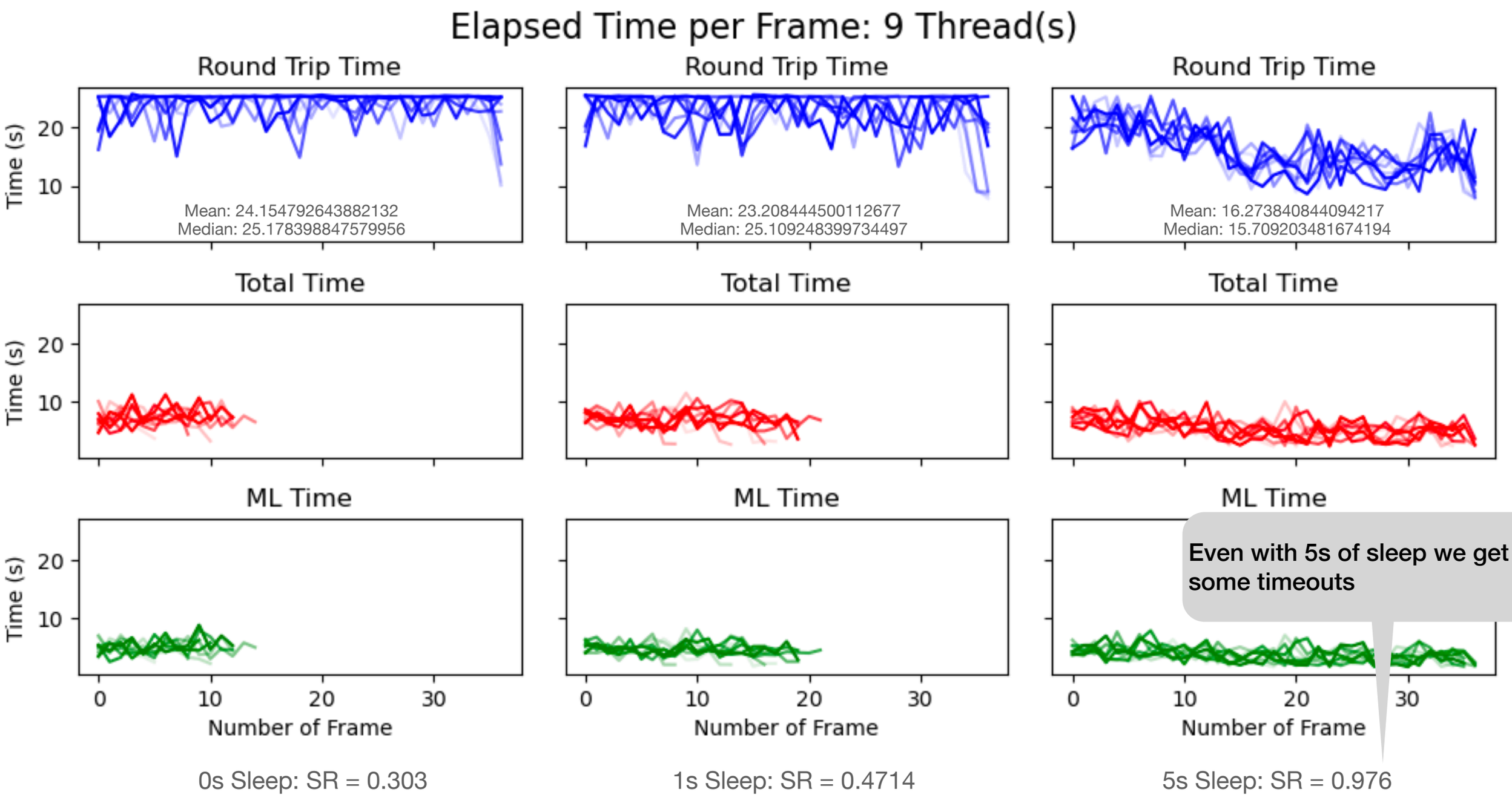
0s Sleep: SR = 0.720

1s Sleep: SR = 0.956

5s Sleep: SR = 1

Results

Round Trip Time, Total Time in Handler & ML Time in Handler



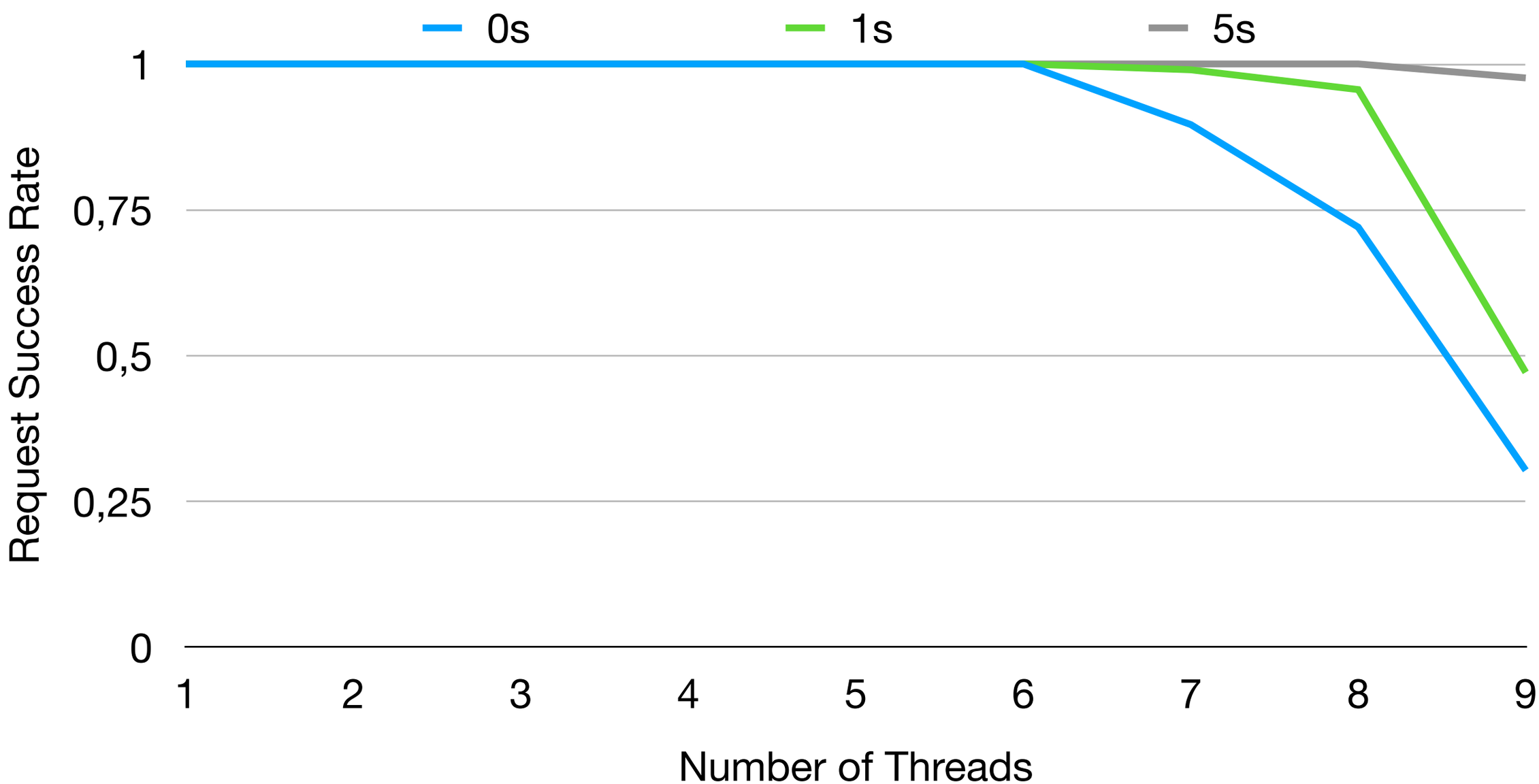
0s Sleep: SR = 0.303

1s Sleep: SR = 0.4714

5s Sleep: SR = 0.976

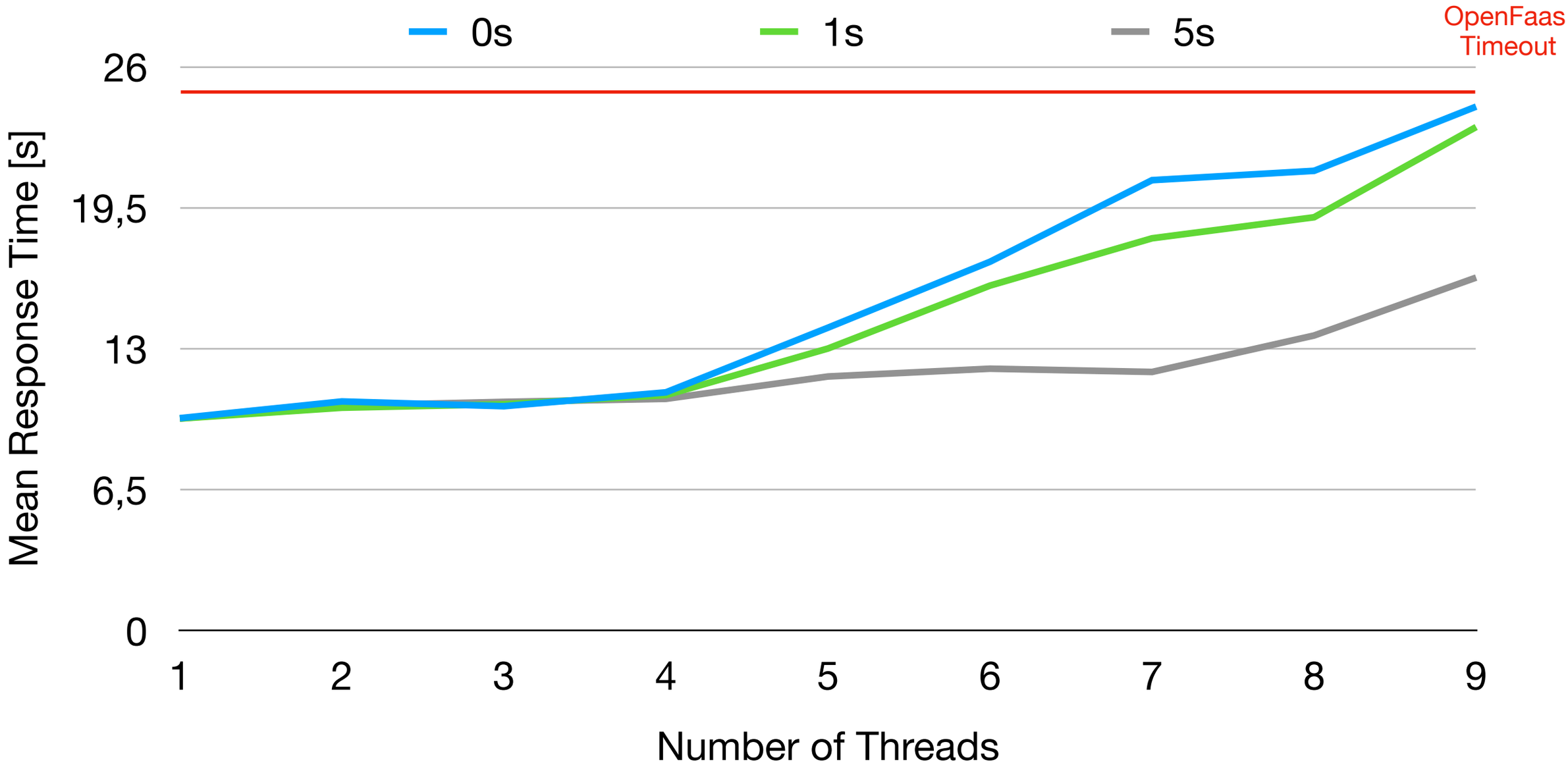
Results

Success Rate over Number of Threads/Sleep Time



Results

Mean Response Time over Number of Threads/Sleep Time



Conclusion

Considerations about the Results

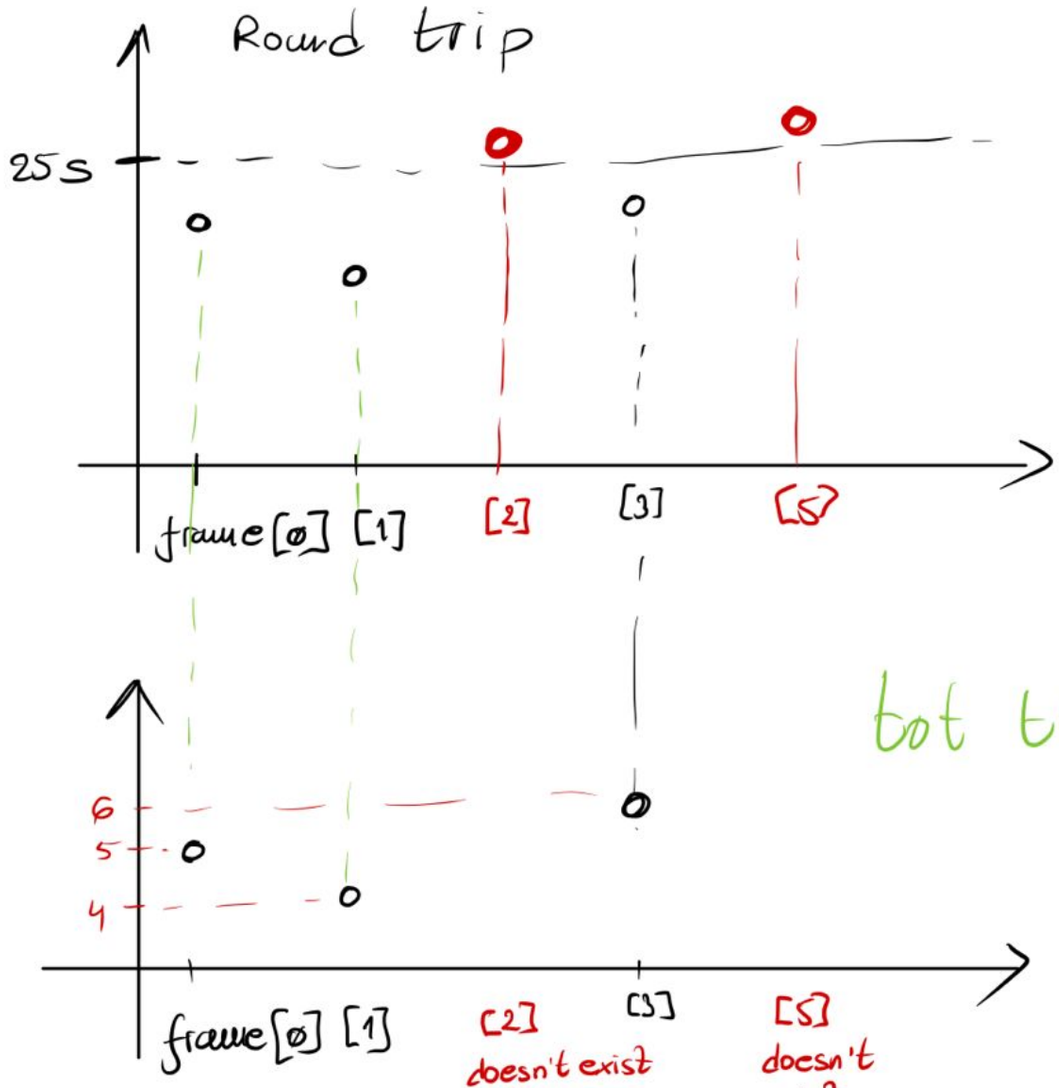
What we found out

- Initiation of the Python3 interpreter with our ML/TF code dominates the effective completion time of the function => High response time for first frame
- Sleeping generally reduces the variance and mean of the response time
  - The effect gets more prominent with more concurrent threads
    - Of course, this will increase the overall execution time of the threads/users up until a certain number of threads
- The time taken for the ML/TF code exhibits a comparatively small variance compared to the overall time of the handler execution
- Running the setup with one replica and the OpenFaas default timeouts, saturates the maximum accepted request rate of the serverless function
- Increasing the sleep time for small number of threads increases the mean and the median response time of users, however for larger amount of users the sleep time enhances the performance

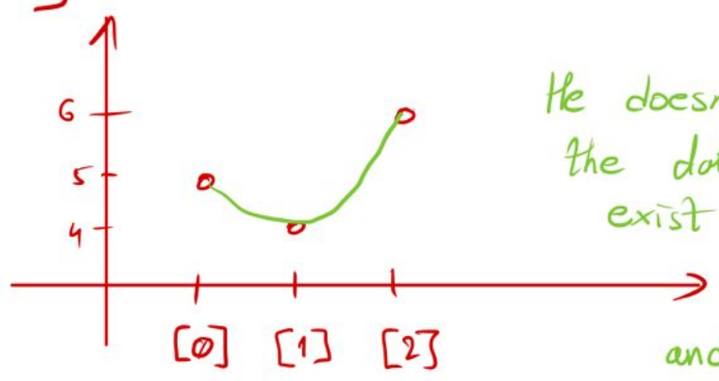
Appendix

https://github.com/CasparDietz/Serverless-AI

Our Github Repository



how its interpreted ?



He doesn't collect the data that doesn't exist just makes a smaller array and becaus he was missing [2] he moved [3] to [2]