

Telecommunications systems
03 2025

Adaptive Load Balancer

A Software Defined Network project

Prudhvi Ponnada
Sai Siri Sree Venturi

Chapter 1

Adaptive Load Balancer

1.1 Aim

The aim of this project is to implement a containerized, load-balanced web service within the given Azure Virtual Machine (VM) using Docker. The system distributes incoming HTTP requests across multiple Flask-based backend servers using a Weighted Round Robin (WRR) algorithm. The setup ensures efficient traffic management without relying on external load balancers, keeping the architecture simple and self-contained.

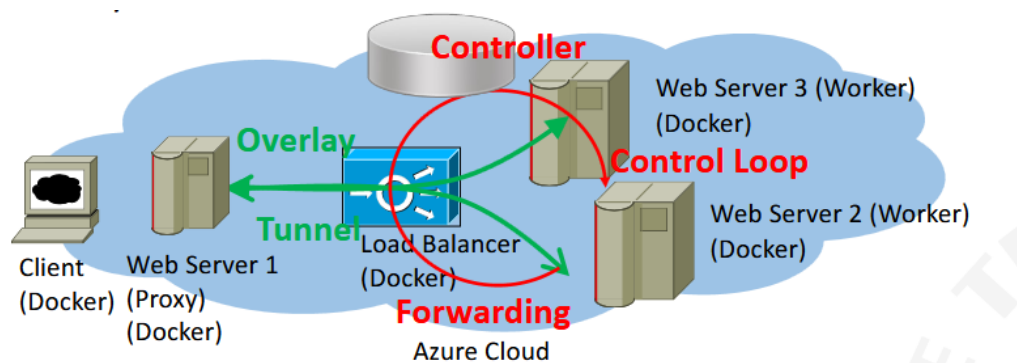
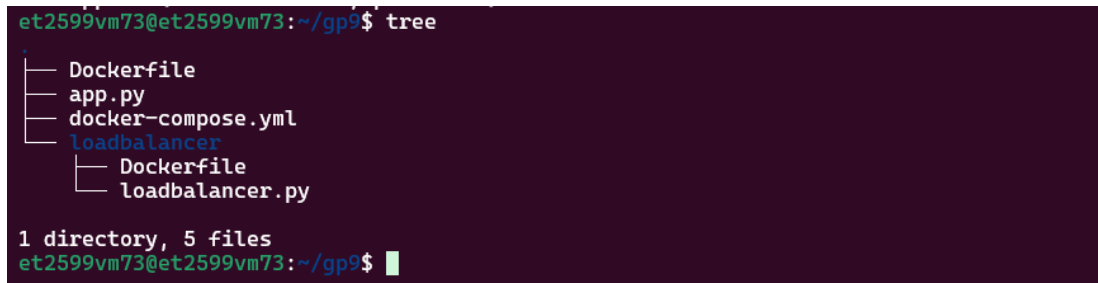


Figure 1.1: Network setup

1.2 Implemented solution

Over here we provide the outline / tree diagram of our implemented solution:



```
et2599vm73@et2599vm73:~/gp9$ tree
.
├── Dockerfile
├── app.py
├── docker-compose.yml
├── loadbalancer
│   ├── Dockerfile
│   └── loadbalancer.py
└── 1 directory, 5 files
et2599vm73@et2599vm73:~/gp9$
```

Figure 1.2: Network setup

The system is deployed as multiple Docker containers within a single Azure Virtual Machine. The core components are:

Web Servers (Flask-based Microservices)

- Three backend servers (server1, server2, server3) each run a same Flask application inside a container.
- These servers respond to HTTP requests and indicate which server processed the request.

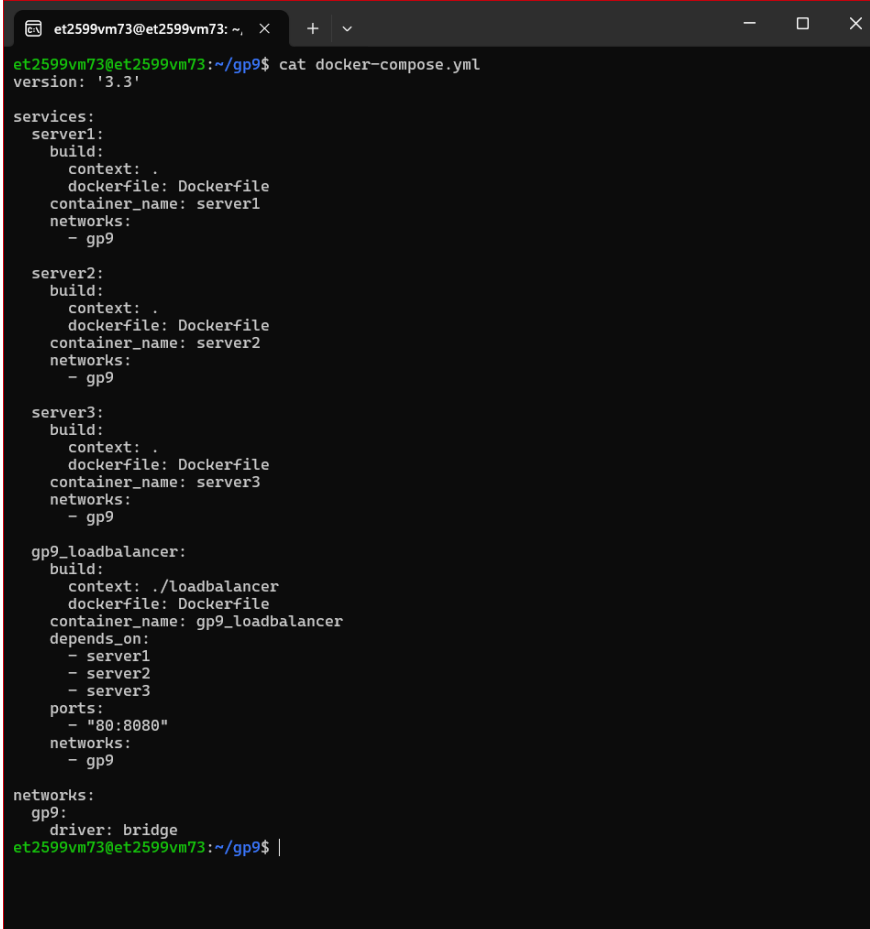
Load Balancer (Weighted Random Selection with Python Flask)

- Implements Weighted Random Selection (WRS) to distribute requests based on dynamic stress and preconfigured weights: Servers with higher weights handle more traffic.
- The load balancer ensures an efficient distribution.

Usage of docker compose

- docker-compose.yml is used to automate the deployment of all containers.
- This setup ensures easy scalability and maintainability.
- The docker-compose made it easy to set up the multiple containers in a network gp9 due to this we avoided using so many commands to create the Docker containers and attaching to the same network.

In the given Azure Virtual Machine (VM) setup, all services run as Docker containers and are managed using Docker-Compose.yml file shown in the figure 1.3. When the system starts with "docker-compose up --build", it initializes three Flask-based web servers, a load balancer. When a client accesses the service using the VM's IP address with port 80 (<http://20.93.27.209>) to the load balancer. The load balancer implements a Weighted Round Robin (WRR) algorithm, distributing incoming requests among the three Flask servers based on predefined weights, ensuring that servers with higher weights receive more traffic than others according to the stress implied on. The selected Flask server processes the request and returns a response indicating which server handled it on the webpage. Each time the webpage is refreshed, the request may be routed to a different server according to weight, causing the displayed server/container number to change. This setup efficiently distributes traffic across multiple backend servers making it a lightweight and self-contained solution within a single Azure VM.

A terminal window with a dark background and light green text. The window title is 'et2599vm73@et2599vm73: ~, X'. The prompt is 'et2599vm73@et2599vm73:~/gp9\$'. The command 'cat docker-compose.yml' has been executed, displaying the following YAML configuration:

```
version: '3.3'

services:
  server1:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: server1
    networks:
      - gp9

  server2:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: server2
    networks:
      - gp9

  server3:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: server3
    networks:
      - gp9

  gp9_loadbalancer:
    build:
      context: ./loadbalancer
      dockerfile: Dockerfile
    depends_on:
      - server1
      - server2
      - server3
    ports:
      - "80:8080"
    networks:
      - gp9

networks:
  gp9:
    driver: bridge
```

The prompt is now 'et2599vm73@et2599vm73:~/gp9\$ |'.

Figure 1.3: Docker-compose.yml file

These files are necessary for the creating flask app in python language and the Dockerfile is instructions to the Docker container to install and run dependencies needed to be shown in figures 1.5 and 1.4 respectively. The Dependencies are installed as per the Dockerfile shown in figure 1.6.

```

GNU nano 4.8 Dockerfile
# Web Server Dockerfile Server 1 server 2 server 3
FROM python:3.9-slim
WORKDIR /app
COPY app.py .
# Install Flask and psutil
RUN pip install flask psutil
RUN apt-get update && apt-get install -y stress
CMD ["python", "app.py"]

```

Figure 1.4: Docker file for the web-servers

```

et2599vm73@et2599vm73:~/gp9$ cat app.py
import socket
from flask import Flask, jsonify
import os
import psutil

app = Flask(__name__)

@app.route("/")
def home():
    return f"Hi from: {socket.gethostname()} please Verify  
me with docker-compose ps output!"

@app.route("/metrics")
def metrics():
    stress = psutil.cpu_percent() + psutil.virtual_memory(
    ).percent
    return jsonify({
        "cpu": psutil.cpu_percent(),
        "memory": psutil.virtual_memory().percent,
        "stress": stress/200
    })

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
et2599vm73@et2599vm73:~/gp9$

```

Figure 1.5: app.py file for the web-server

```

et2599vm73@et2599vm73: ~, x + v - □ x
GNU nano 4.8 Dockerfile
# Use an official Python image
FROM python:3.9

# Set the working directory inside the container
WORKDIR /app

# Copy the load balancer script into the container
COPY loadbalancer.py /app/loadbalancer.py

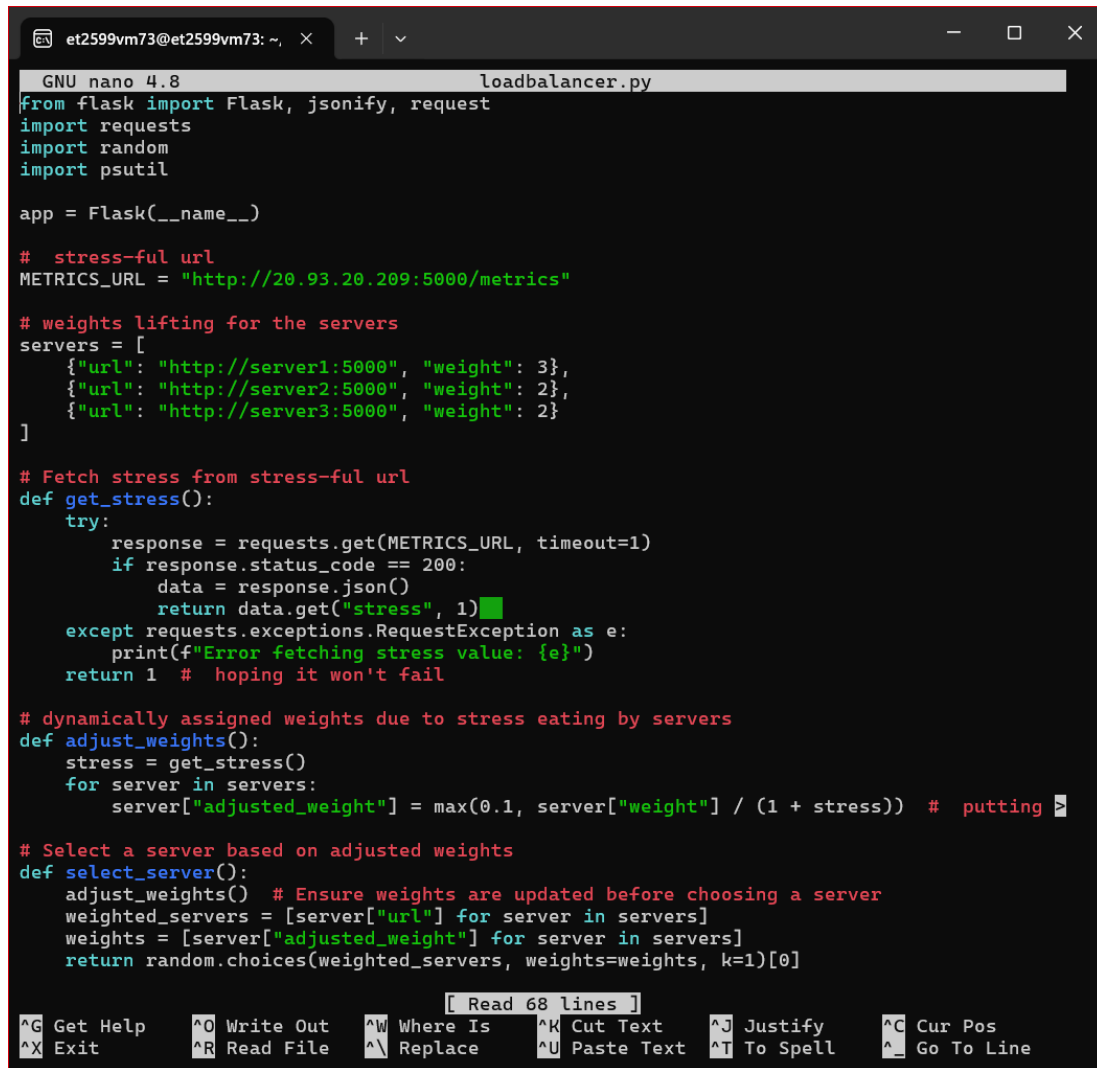
# Install dependencies
RUN pip install flask requests psutil

# Expose the required port
EXPOSE 8080

# Run the load balancer script
CMD ["python", "/app/loadbalancer.py"]

```

Figure 1.6: Docker file to load balancer



```

GNU nano 4.8                                loadbalancer.py
from flask import Flask, jsonify, request
import requests
import random
import psutil

app = Flask(__name__)

# stress-ful url
METRICS_URL = "http://20.93.20.209:5000/metrics"

# weights lifting for the servers
servers = [
    {"url": "http://server1:5000", "weight": 3},
    {"url": "http://server2:5000", "weight": 2},
    {"url": "http://server3:5000", "weight": 2}
]

# Fetch stress from stress-ful url
def get_stress():
    try:
        response = requests.get(METRICS_URL, timeout=1)
        if response.status_code == 200:
            data = response.json()
            return data.get("stress", 1)
    except requests.exceptions.RequestException as e:
        print(f"Error fetching stress value: {e}")
    return 1 # hoping it won't fail

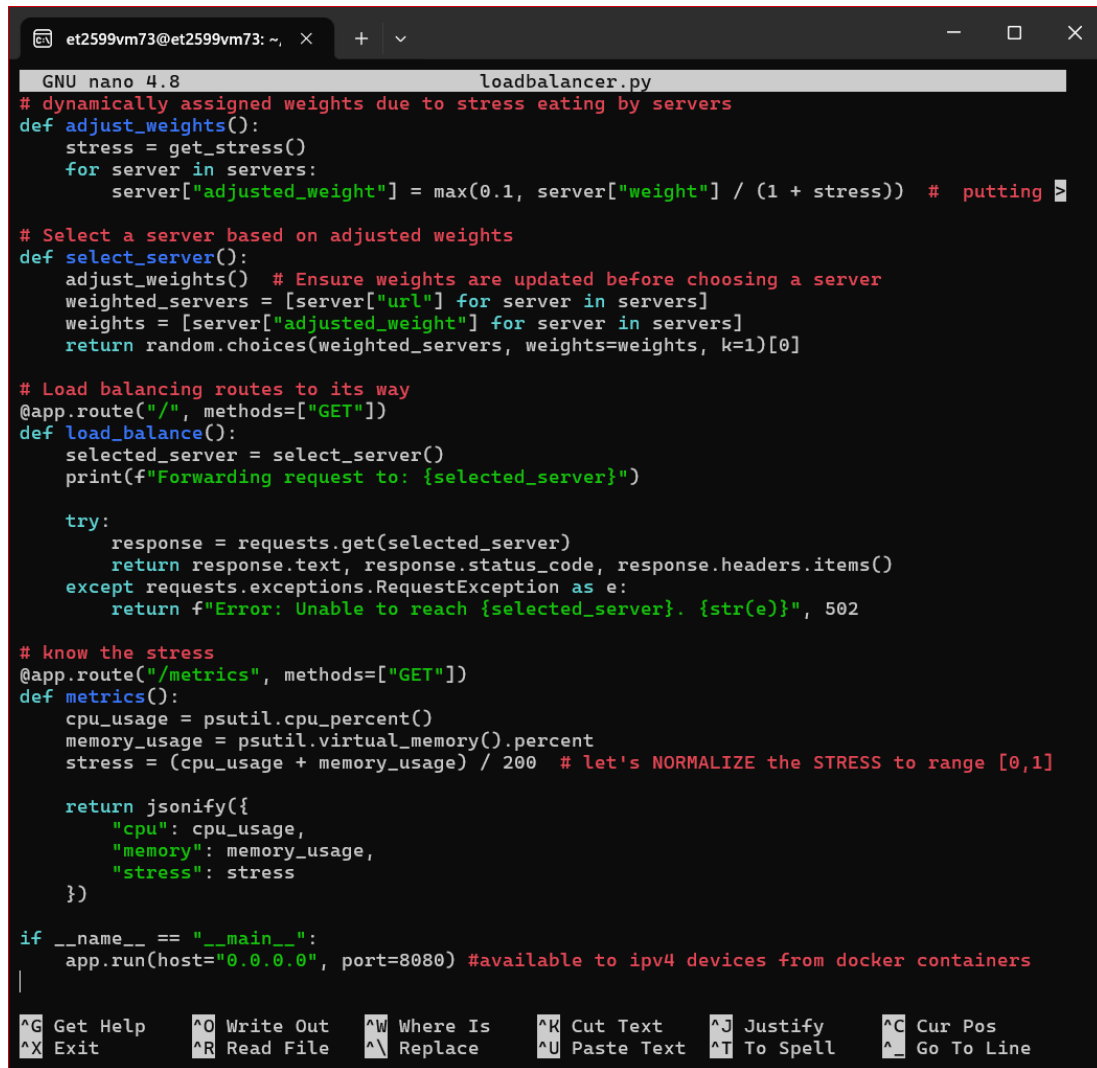
# dynamically assigned weights due to stress eating by servers
def adjust_weights():
    stress = get_stress()
    for server in servers:
        server["adjusted_weight"] = max(0.1, server["weight"] / (1 + stress)) # putting

# Select a server based on adjusted weights
def select_server():
    adjust_weights() # Ensure weights are updated before choosing a server
    weighted_servers = [server["url"] for server in servers]
    weights = [server["adjusted_weight"] for server in servers]
    return random.choices(weighted_servers, weights=weights, k=1)[0]

[ Read 68 lines ]
^G Get Help      ^O Write Out    ^W Where Is    ^K Cut Text    ^J Justify    ^C Cur Pos
^X Exit          ^R Read File    ^_ Replace      ^U Paste Text  ^T To Spell   ^_ Go To Line

```

Figure 1.7: Weighted random selection in load balancer application



```

GNU nano 4.8                                loadbalancer.py
# dynamically assigned weights due to stress eating by servers
def adjust_weights():
    stress = get_stress()
    for server in servers:
        server["adjusted_weight"] = max(0.1, server["weight"] / (1 + stress)) # putting

# Select a server based on adjusted weights
def select_server():
    adjust_weights() # Ensure weights are updated before choosing a server
    weighted_servers = [server["url"] for server in servers]
    weights = [server["adjusted_weight"] for server in servers]
    return random.choices(weighted_servers, weights=weights, k=1)[0]

# Load balancing routes to its way
@app.route("/", methods=["GET"])
def load_balance():
    selected_server = select_server()
    print(f"Forwarding request to: {selected_server}")

    try:
        response = requests.get(selected_server)
        return response.text, response.status_code, response.headers.items()
    except requests.exceptions.RequestException as e:
        return f"Error: Unable to reach {selected_server}. {str(e)}", 502

# know the stress
@app.route("/metrics", methods=["GET"])
def metrics():
    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent
    stress = (cpu_usage + memory_usage) / 200 # let's NORMALIZE the STRESS to range [0,1]

    return jsonify({
        "cpu": cpu_usage,
        "memory": memory_usage,
        "stress": stress
    })

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080) #available to ipv4 devices from docker containers

```

[^]G Get Help [^]O Write Out [^]W Where Is [^]K Cut Text [^]J Justify [^]C Cur Pos
[^]X Exit [^]R Read File [^]\ Replace [^]U Paste Text [^]T To Spell [^]_ Go To Line

Figure 1.8: Weighted random selection in load balancer application

1.3 Key Functionalities of load balancer

Fetching System Stress

The full python code is shown in the figure 1.7 and 1.8. The load balancer queries an external metrics endpoint:

- URL: `http://20.93.20.209/metrics`
- Retrieves CPU and memory usage.
- Normalizes the **stress** value between 0 and 1.

Adjusting Server Weights

Each server has an initial weight, and its adjusted weight is computed as:

$$\text{adjusted_weight} = \frac{\text{original_weight}}{1 + \text{stress}} \quad (1.1)$$

Higher stress results in a lower adjusted weight, reducing traffic to overloaded servers.

Server Selection Using Weighted Random Choice

The server is chosen based on updated weights using the `random.choices()` function. This ensures that:

- Higher weighted servers receive more traffic.
- Overloaded servers receive fewer requests.

Handling Incoming Requests

The load balancer:

1. Fetches the latest **stress value**.
2. Adjusts **server weights**.
3. Selects a backend server probabilistically.
4. Forwards the request to the selected server.
5. Returns the response or a **502 error** if the backend is unreachable.

1.4 Example Request Flow

Low Stress Scenario

- CPU = 20%, Memory = 30%
- Stress = $(20 + 30) / 200 = 0.25$

Server	Weight	Adjusted Weight
Server1	3	2.4
Server2	2	1.6
Server3	2	1.6

High Stress Scenario

- CPU = 80%, Memory = 70%
- Stress = $(80 + 70) / 200 = 0.75$

Server	Weight	Adjusted Weight
Server1	3	1.71
Server2	2	1.14
Server3	2	1.14

1.5 Working status

After building the Docker containers by using the docker-compose.yml with the command "docker-compose up --build". It will build four containers named as gp9 loadbalancer, server1 , server2 and server3. Upon execution of command "curl" for the vm-ip address in port 80 (HTTP) i.e 20.93.20.209 and by appending "metrics" will be shown with the stress on CPU, memory and the percentage average of it as shown in the image 1.9. It can be reached from any IPv4 devices to the webpage we designed as shown in the figure 1.10.

1.6 Conclusion

This load balancer dynamically adjusts traffic distribution based on real-time system stress. By prioritizing less-stressed servers, it improves reliability and prevents server overload. The implementation is efficient, scalable, and suitable for cloud-based deployments.

```

et2599vm73@et2599vm73: ~$ ssh -i et2599vm73_key.pem et2599vm73@20.93.20.209
Last login: Thu Mar 13 20:33:59 2025 from 80.78.288.97
et2599vm73@et2599vm73: $ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
20589127b438   gp9_gp9_loadbalancer  "python /app/loadbal..." 17 seconds ago Up 17 seconds 0.0.0.0:80->8080/tcp, [::]:80->8080/tcp gp9_loadbalancer
c13075bb2b9f   gp9_server3        "python app.py"           11 hours ago  Up 2 hours                               server3
f70665e26c7d   gp9_server1        "python app.py"           11 hours ago  Up 2 hours                               server1
9d0065968cef   gp9_server2        "python app.py"           11 hours ago  Up 2 hours                               server2
et2599vm73@et2599vm73: $ curl localhost:8080
curl: (7) Failed to connect to localhost port 8080: Connection refused
et2599vm73@et2599vm73: $ curl localhost:80
Hi from: c13075bb2b9f please Verify me with docker-compose ps output!et2599vm73@et2599vm73: $ curl localhost:80
Hi from: c13075bb2b9f please Verify me with docker-compose ps output!et2599vm73@et2599vm73: $ curl localhost:80
Hi from: f70665e26c7d please Verify me with docker-compose ps output!et2599vm73@et2599vm73: $ curl localhost:80
Hi from: f70665e26c7d please Verify me with docker-compose ps output!et2599vm73@et2599vm73: $ curl localhost:80
Hi from: c13075bb2b9f please Verify me with docker-compose ps output!et2599vm73@et2599vm73: $ curl localhost:80
Hi from: c13075bb2b9f please Verify me with docker-compose ps output!et2599vm73@et2599vm73: $ curl localhost:80
Hi from: f70665e26c7d please Verify me with docker-compose ps output!et2599vm73@et2599vm73: $ curl localhost:80
Hi from: f70665e26c7d please Verify me with docker-compose ps output!et2599vm73@et2599vm73: $ curl localhost:80
Hi from: 9d0065968cef please Verify me with docker-compose ps output!et2599vm73@et2599vm73: $ curl localhost:80
Hi from: c13075bb2b9f please Verify me with docker-compose ps output!et2599vm73@et2599vm73: $ curl localhost:80/metrics
{"cpu":0.8,"memory":27.7,"stress":0.13499999999999998}
et2599vm73@et2599vm73: $ curl localhost:80/metrics
{"cpu":3.7,"memory":27.7,"stress":0.157}
et2599vm73@et2599vm73: $ curl localhost:80/metrics
{"cpu":5.8,"memory":27.7,"stress":0.1675}
et2599vm73@et2599vm73: $ curl localhost:80/metrics
{"cpu":4.7,"memory":27.7,"stress":0.162}
et2599vm73@et2599vm73: $

```

Figure 1.9: Final output in terminal of VM

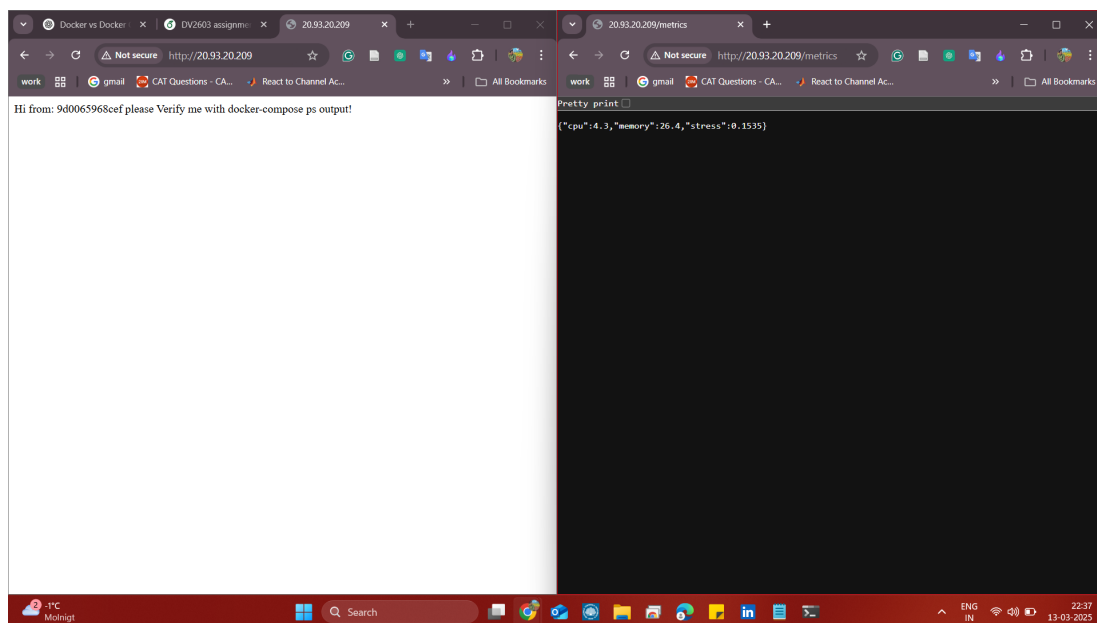


Figure 1.10: Final output in my PC