Group - 9

DYNAMIC LOAD BALANCING WITH HORIZONTAL SCALING

Implementation with Codes
(Cloud Computing)

Problem Statement

- As web applications experience varying levels of traffic, ensuring smooth performance and minimizing server downtime and balancing load becomes crucial.

- In highly distributed systems, managing server load efficiently is a key challenge. When multiple servers handle user requests, some servers may become overloaded, while others may remain underutilized, managed by horizontal scaling.

- This leads to performance degradation, increased response times, and higher operational costs. Additionally, the failure of any individual server can cause service disruption unless proper fault tolerance is implemented.

Objectives

What we wish to achieve?

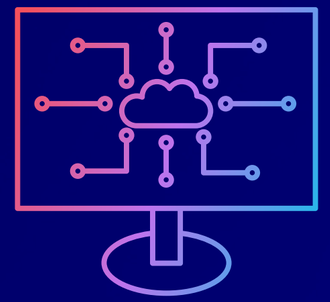
- **Efficient Traffic Distribution:** Implement a dynamic load balancing system to distribute traffic across multiple servers, ensuring optimal performance.
- **Horizontal Scaling:** Automatically scale server resources by adding or removing servers based on CPU usage to enhance resource utilization and system performance.

[Cont...]

- **Algorithm Integration:** Support multiple load balancing algorithms, including Round Robin, Least Connections, Least Response Time and Least resource utilization allowing for dynamic switching based on traffic patterns.

- **Continuous Monitoring:** Continuously monitor server health to ensure high availability, redirecting traffic to healthy servers in case of failures.

- **High Availability and Fault Tolerance:** Maintain system uptime and resilience through proactive traffic management and server monitoring.



Group - 9

Load Balancer

What is a Load Balancer?

- Load Balancer is a networking device or software application.
- Distributes incoming network traffic.
- Ensures high availability and reliability.
- Optimizes resource use and performance.
- Protects against server overloads.
- Can be hardware or software-based.

Group - 9

Load Balancing Algorithms

What are LB Algorithms?

- **LB algorithms define how incoming requests are distributed across multiple servers..**
- **The goal is to choose the best server for each client request.**

Purpose:

Ensure efficient distribution of traffic to prevent overloading servers and improve overall system performance.

Categories of LB Algorithms

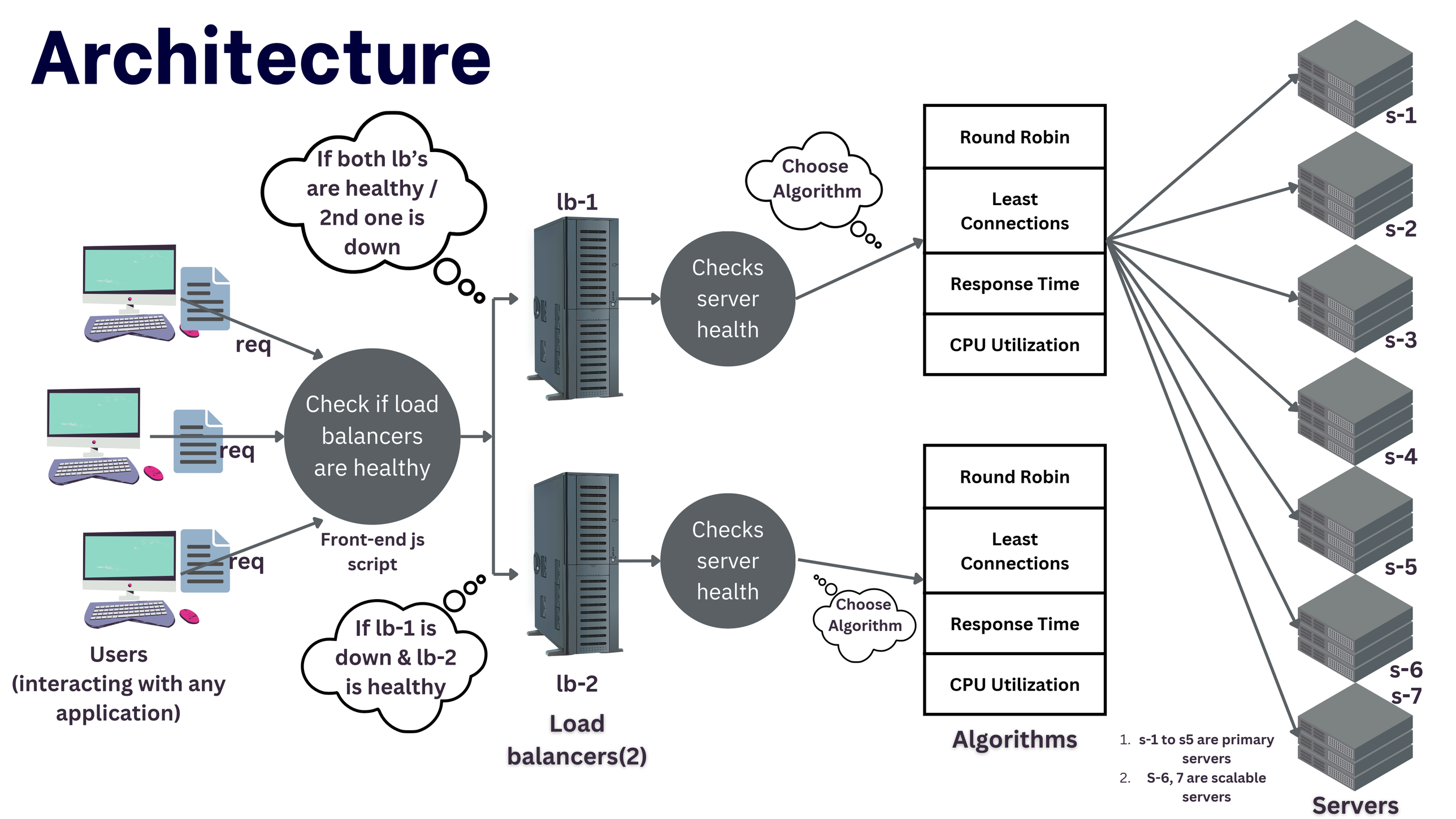
Static Algorithms:

- **Fixed rules, independent of server state.**
- **eg: Random, Round Robin, IP hash etc.,**

Dynamic Algorithms:

- **Adapt based on current server conditions**
- **e.g., response time, active connections etc.,**

Architecture



Group - 9

ALGORITHMS

Implementation of algorithms

We used several algorithms in this such as:

- Round Robin
- Least Connections
- Least Response time
- Least resource Utilization

Group - 9

Round Robin

Implementation of algorithms

Distributes requests sequentially to each server in a circular order, ensuring an even load across all servers.

Output:

"Request data at 2024-10-09T11:02:41.977Z", Server: 8001
"Request data at 2024-10-09T11:02:42.058Z", Server: 8002
"Request data at 2024-10-09T11:02:42.151Z", Server: 8003
"Request data at 2024-10-09T11:02:42.243Z", Server: 8004
"Request data at 2024-10-09T11:02:42.337Z", Server: 8005
"Request data at 2024-10-09T11:02:42.431Z", Server: 8001
"Request data at 2024-10-09T11:02:42.512Z", Server: 8002
"Request data at 2024-10-09T11:02:42.604Z", Server: 8003

Group - 9

Round Robin

Pros and Cons?

Pros

- Simple to implement.
- Distributes requests evenly.
- No server state tracking is required.

Cons

- Doesn't consider server load.
- May overload slower servers.
- May result in request latency if some servers are slower.

Group - 9

Round Robin

```
let roundRobinIndex = 0;
 case "Round Robin":
     selectedServer = servers[roundRobinIndex % servers.length];
     roundRobinIndex++;
     break;
```

- We have an array `servers` containing healthy servers.
- The selected server is chosen by index % len(servers) to avoid arrayOutOfBound error.
- We will select the server in a circular loop and then increment the index to server next request.

Group - 9

Least Connections:

Implementation of algorithms

Directs traffic to the server with the fewest active connections, optimizing resource utilization and enhancing response times.

Group - 9

Least Connections:

Pros and Cons?

Pros

- Directly accounts for active connections.
- Better for servers with varying capacities.
- Helps prevent server overload by considering active connections.

Cons

- Requires tracking connection counts.
- Overhead for managing connections.
- Doesn't account for the processing time of ongoing requests.

Group - 9

Least Connections:

```
const connectionsCount = {};
case "Least Connections":
    selectedServer = servers.reduce((prev, curr) => {
        connectionsCount[curr] = connectionsCount[curr] || 0;
        return connectionsCount[prev] < connectionsCount[curr] ? prev : curr;
    });
    connectionsCount[selectedServer]++;
    break;
```

- We track active connections for each server in the `connectionsCount` object, using the server's URL as the key. The server with the fewest active connections is selected for the next request.
- After a server is selected, its connection count is incremented and then decremented when req leaves.

Group - 9

Least Response Time

Implementation of algorithms

Routes requests to the server that has the lowest response time, improving overall performance and user experience.

Output:

```
[0] Request processed with data: Request data at 2024-10-09T19
:15:07.629Z, Server: 8005, Response Time: 1992ms
[1] Request processed with data: Request data at 2024-10-09T19
:15:07.533Z, Server: 8003, Response Time: 1493ms
[1] Request processed with data: Request data at 2024-10-09T19
:15:07.724Z, Server: 8003, Response Time: 1071ms
```

Group - 9

Least Response Time

Pros and Cons?

Pros

- Improves user experience by minimizing latency.
- Adapts well to varying server performance.
- Efficiently utilizes faster servers.

Cons

- Requires continuous monitoring of response times, adding overhead.
- Its performance is dependent on how good the response time estimates are.

Group - 9

Least Response Time

```
const startTime = Date.now();
const response = await post(`${selectedServer}/processRequest`, {
    data: req.body.data,
    algorithm,
});
const endTime = Date.now();
const elapsedTime = endTime - startTime;
responseTime[selectedServer] = elapsedTime;
```

- Response time calculated as ` response recieve time - request start time`
- Uses a `responseTime` object to store each server's response time.
- Uses `reduce` to find the server with the lowest response time.
- Sends the request to the selected server.
- Records the processing time in `responseTime` for future comparisons.

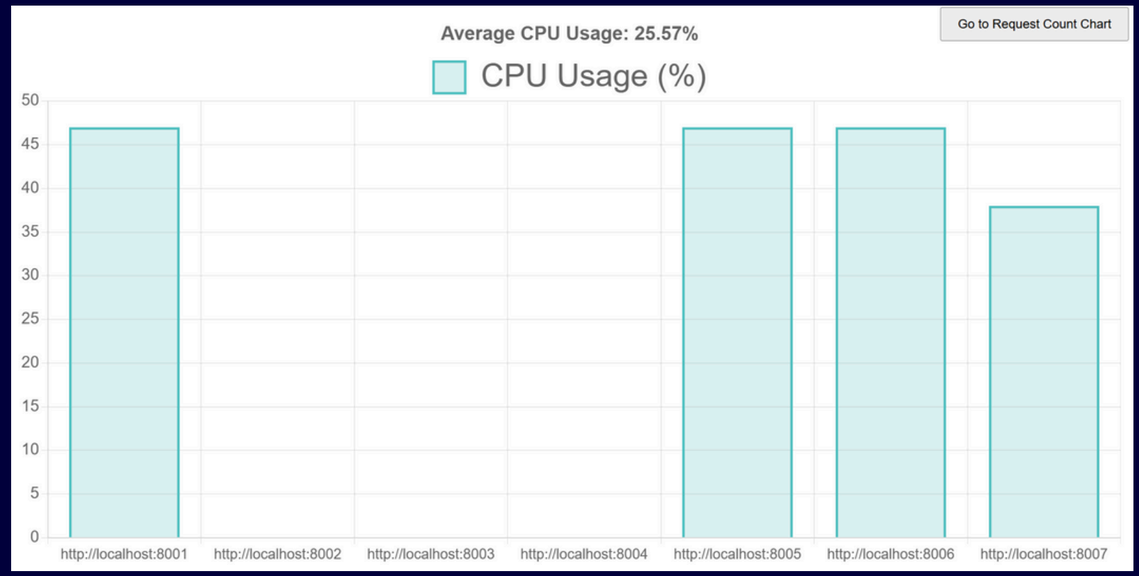
Group - 9

Least resource Utilization

Implementation of algorithms

Use metrics like CPU usage, connections count, response time to calculate server score and select the most suitable server based on current workload conditions.

Output:



Group - 9

Least resource Utilization

Pros and Cons?

Pros

- Balances load based on server performance.
- Efficient resource usage.
- Adapts to changes in server workload in real time.

Cons

- Requires real-time CPU monitoring
- More complex to implement and manage.
- May favor servers that are slower but have lower CPU utilization at the moment.

Group - 9

Code:

```
case "Dynamic":
    try {
        const serverData = await Promise.all(
            servers.map(async (curr) => {
                const cpuUsageResponse = await get(`${curr}/cpuUsage`);
                const cpuUsage = cpuUsageResponse.data.usage;
                connectionsCount[curr] = connectionsCount[curr] || 0;
                responseTime[curr] = responseTime[curr] || 0;
                const score = cpuUsage / 100 + connectionsCount[curr] / 10 + responseTime[curr] / 1000;
                return { server: curr, score };
            })
        );
        const selected = serverData.reduce((prev, curr) => (curr.score < prev.score ? curr : prev));
        selectedServer = selected.server;
```

Group - 9

CPU Utilization calculation

- Calculates CPU utilization percentage.
- Retrieves CPU stats using _cpus().
- Tracks totalDiff (total CPU time) and idleDiff (idle time).
- Compares current and previous CPU times.

$$\text{CPU Usage} = 100 - \left(\frac{\text{idleDiff}}{\text{totalDiff}} \times 100 \right)$$

- Returns: CPU utilization percentage (0% = idle, 100% = fully utilized).

Group - 9

CPU Utilization Code:

```
function getCPUUsage() {
  const cpus = _cpus();
  let totalDiff = 0;
  let idleDiff = 0;

  cpus.forEach((cpu, index) => {
    const { user, nice, sys, idle, irq } = cpu.times;
    const last = lastCpuTimes[index];

    const total = user + nice + sys + idle + irq;
    const lastTotal = last.user + last.nice + last.sys + last.idle + last.irq;

    totalDiff += total - lastTotal;
    idleDiff += idle - last.idle;

    lastCpuTimes[index] = cpu.times;
  });

  const cpuUsage = totalDiff === 0 ? 0 : 100 - Math.floor((idleDiff / totalDiff) * 100);
  return cpuUsage;
}
```

Group - 9

Routes

Load Balancer

- GET /health
- POST /sendRequest
- GET /metrics

Application Server

- GET /health
- GET /cpuUsage
- POST /processRequest
- GET /requestsPerSecond
- GET /activeConnections

Fault Tolerance

- **Health Checks:** Monitor server status regularly.
- **Graceful Degradation:** Maintain partial functionality during failures.
- **Fallback Mechanism:** Reroute to healthy servers if needed.
- **Error Handling:** Provide clear error responses.
- **Logging:** Track errors for quick fixes.
- **Dynamic Scaling:** Adjust server count based on load.

Load balancer Health check

```
let healthyLBs = [];
let currentLBIndex = 0;

async function checkLoadBalancerHealth() {
    const newHealthyLBs = [];

    await Promise.all(loadBalancers.map(async (lb) => {

        try {
            const response = await fetch(`${lb.url}/health`);
            if (response.ok)
            {
                newHealthyLBs.push(lb)
            }else {
                console.log(`Load Balancer ${lb.id} is down:`)
            }
        } catch (error) {
            console.log(`Load Balancer ${lb.id} is down:`, error);
        }

    }));

    healthyLBs = newHealthyLBs;
    document.getElementById('sendRequestButton').disabled = healthyLBs.length === 0;

    if(healthyLBs.length == 0){
        window.alert("Load balancers are down")
    }
}
```

```
const loadBalancers = [
    { id: 'load-balancer-1', url: 'http://localhost:4001' },
    { id: 'load-balancer-2', url: 'http://localhost:4002' },
];

let healthyLBs = [];
let currentLBIndex = 0;
```

```
function startHealthCheck() {
    checkLoadBalancerHealth();
    setInterval(checkLoadBalancerHealth, 600000);
}

function getNextLoadBalancer() {
    if (healthyLBs.length === 0)
        throw new Error('No healthy load balancers available.');
    const lb = healthyLBs[currentLBIndex];
    currentLBIndex = (currentLBIndex + 1) % healthyLBs.length;
    return lb;
}
```

Horizontal Scaling

```
const monitorCPUUsage = async () => {
    const serverUsages = await Promise.all(
        primaryServers.map(async (server) => {
            try {
                const usageResponse = await get(`${server}/cpuUsage`);
                return { url: server, usage: usageResponse.data.usage };
            } catch (error) {
                console.error(`Error fetching CPU usage for ${server}: ${error.message}`);
                return { url: server, usage: 0 };
            }
        })
    );

    serverUsages.forEach(({ url, usage }) => {

        if (usage > HIGH_USAGE_THRESHOLD) {
            console.log(`High usage detected on ${url}. Scaling up.`);
            scaleUpServer();
        }

        else if (usage < LOW_USAGE_THRESHOLD) {
            console.log(`Low usage detected on ${url}. Scaling down.`);
            scaleDownServer(url);
        }
    });
};
```

Threshold values:

```
const HIGH_USAGE_THRESHOLD = 75;
const LOW_USAGE_THRESHOLD = 25;
const MIN_PRIMARY_SERVERS = 4;
```

```
const scaleUpServer = () => {
    const scalingServer = scalingServers.pop();
    if (scalingServer) {
        primaryServers.push(scalingServer);
        console.log(`Scaled up: Moved ${scalingServer} to primary servers.`);
    } else {
        console.log("No available scaling servers to add.");
    }
};
const scaleDownServer = (serverUrl) => {
    const index = primaryServers.indexOf(serverUrl);
    if (index > -1 && primaryServers.length > MIN_PRIMARY_SERVERS) {
        primaryServers.splice(index, 1);
        scalingServers.push(serverUrl);
        console.log(`Scaled down: Moved ${serverUrl} to scaling servers.`);
    } else {
        console.log(`Cannot scale down ${serverUrl}, minimum primary servers limit r
    }
};
```

Group - 9

Thank You...