The Pi Project: MLOps

PROJECT DOCUMENTATION

National Technical University of Athens, Greece
Computing Systems Laboratory

Academic Year 2024–2025

Akilas Antonis · Papadopoulou Elisavet · Raftopoulos Michalis · Theodosiou Kostas

Table of Contents

1 Project Overview

"What is this thing?"

1.1 Introduction

The Pi Project is an initiative of the Computing Systems Laboratory (CSLAB), that organizes groups of students to set up and manage clusters of Raspberry Pis. The students were divided into two teams, with each one owning a subset of the available pis. One team works on HPC related projects, simulating a computing cluster environment and testing on various benchmarks. The other team, us, works on Machine Learning applications that include multiple computational nodes: think distributed and federated learning, IoT and ML-on-edge, etc. The idea is that, having its own end goal, each team will develop and maintain useful infrastructure that could be used to run further relevant experiments.

For the first year of the project (2024-2025) we set up the infrastructure and worked on a relatively simple ML application: Jen the Chatbot. Jenny is our instance of a quantized version of the Llama Language Model, lightweight enough to run inference on a single Raspberry Pi. Each node of the system loads the model weights, and the user can begin a chat session with any of the nodes. A Kubernetes load balancer ensures that the session allocation is performed fairly among the pis.

Through this document we wish to introduce our work to the next generation of students, allowing them to continue our work. It can also serve as a guide for any third party that might want to recreate our system. We have organized this work in the following chapters, each answering a question:

1. **Project Overview** - "what is this thing?"
   We briefly describe what this project is about and how the system is structured.

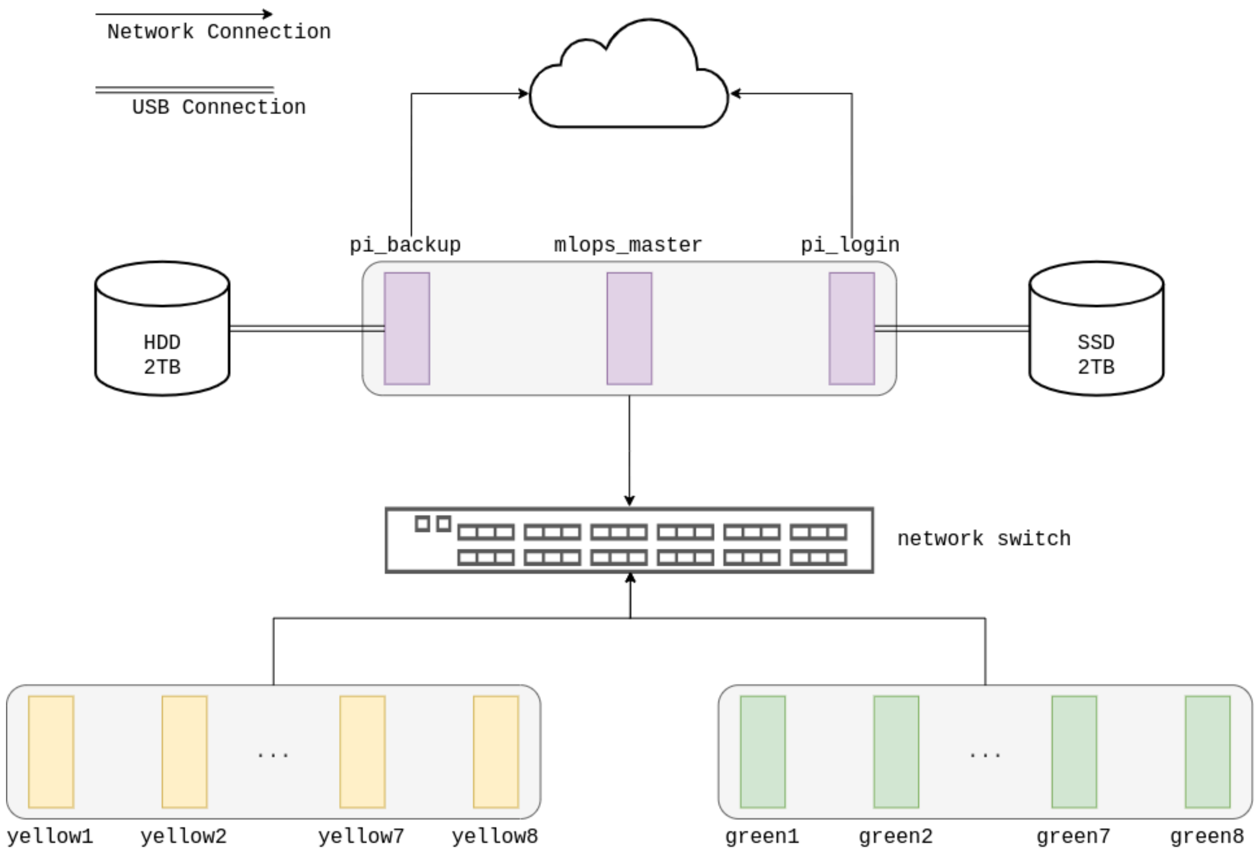
2. **System Setup** - *"How do you make it?"*
   We go through every step of our setup process. This could also serve as a tutorial for anyone wanting to reproduce our system.

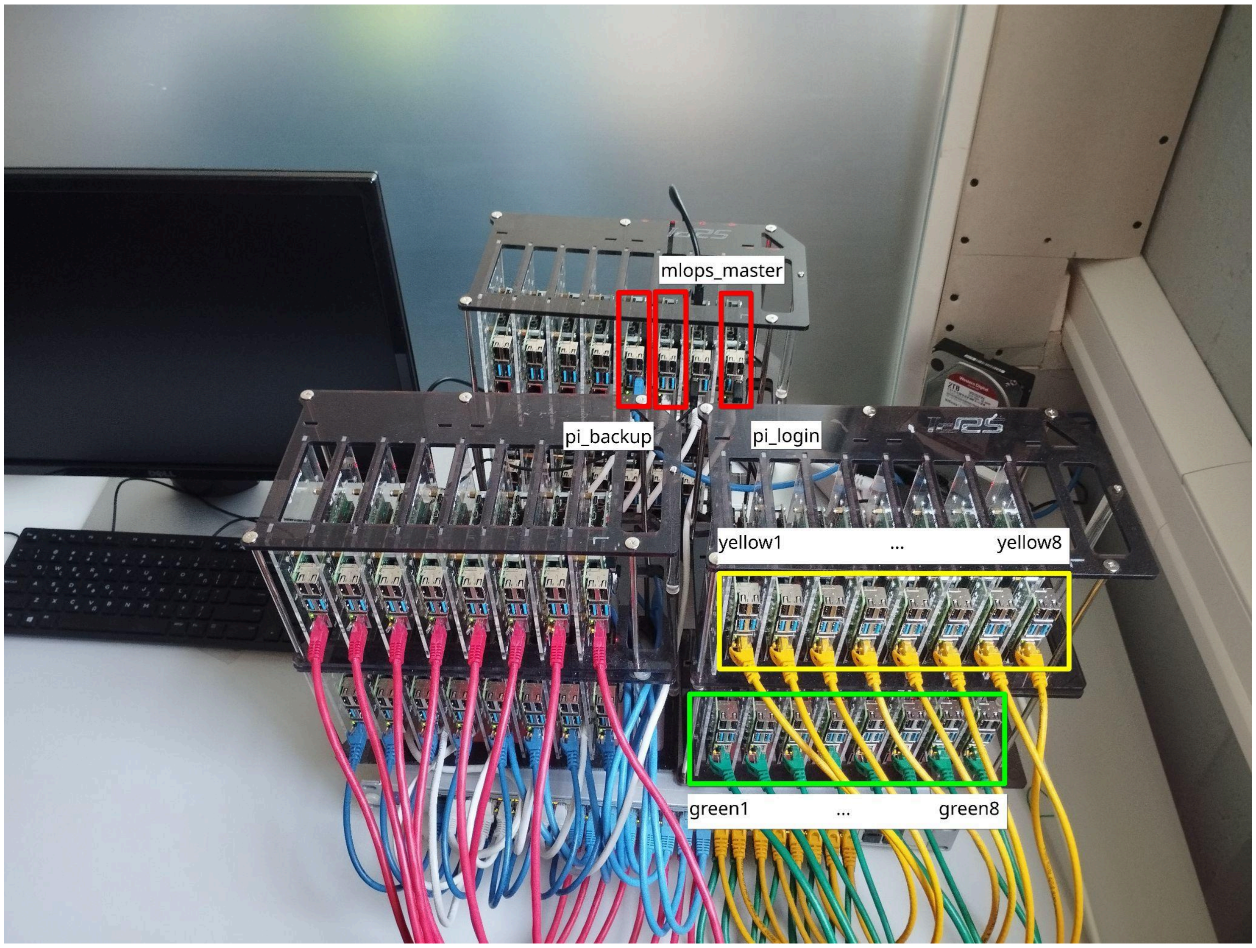
3. **System Functionalities** - *"How do you use it?"*
   You can see this chapter as a user guide. We describe the various processes used for system usage and maintenance.

4. **FOES (Frequently Occurring Errors& Solutions)** - *"How do you fix it?"*
   We provide a collection of debugging tips and well established errors that keeped us up at night.

1.2 Architecture

In the diagram below we briefly describe the architecture of our network. The gray boxes correspond to different racks, so the groups here correspond to the actual grouping performed in the physical system. You can notice that some Pis are labeled with the colors "yellow" and "green". These colors correspond to the colors of their Ethernet cables, so that they are easily distinguishable.





Every Pi is connected to a central switch (*Ubiquiti UniFi USW Pro 48 PoE Managed L2 PoE+*), so they can all see each other through their local IPs. Besides network connectivity, this switch provides the power of the Pis, through their Ethernet cables (Power over Ethernet - PoE). None of the Pis have any kind of local persistent storage, besides the Login Pi and the Backup Pi. The OS of each device is stored in the 2TB SSD and all the nodes (besides Login and Backup) boot over a Network FileSystem (NFS). We will describe this process in detail in later chapters. We will now briefly describe the roles of each node:

pi_login: The master of the entire system. It's called the Login Pi / Login Node / pi_login because it is the device that we ssh into to control our network. Along with the Backup Pi, it is the only one with direct internet access and it's responsible for every vital mechanism in the system. Through the Login Node, every other Pi (except Backup) accesses the SSD and boots over NFS. Also, every other node can connect to the internet through pi_login. It should be mentioned that pi_login is the only device that is shared with the HPC team. For that reason you should be extra careful when playing with its components.

pi_backup: The Backup Pi / Backup Node / pi_backup is the only machine in the system that has its own SD card and does not boot over nfs from pi_login. It is also the only machine that can directly access the internet, without the interference of pi_login. It is designed that way, so that when pi_login dies (this happens more often than we would like to admit), we can still access the backup node to reboot and troubleshoot the system. To maximize our hardware utilization, we gave pi_backup the additional role of managing an external HDD, where we store various utility data, such as container images for our applications.

mlops_master: While pi_login is responsible for the core mechanisms of the system (NFS boot, internet access, etc), mlops_master is the orchestrator at the application level. It contains scripts that automate processes and monitor the workers, and take the role of the server node in our kubernetes cluster. We try to put as much of the system overhead as possible in mlops_master, to prevent interference with the HPC team in the login node.

yellowX/greenX: These are the worker nodes of the system. They boot over NFS and can be accessed remotely through pi_login and mlops_master.

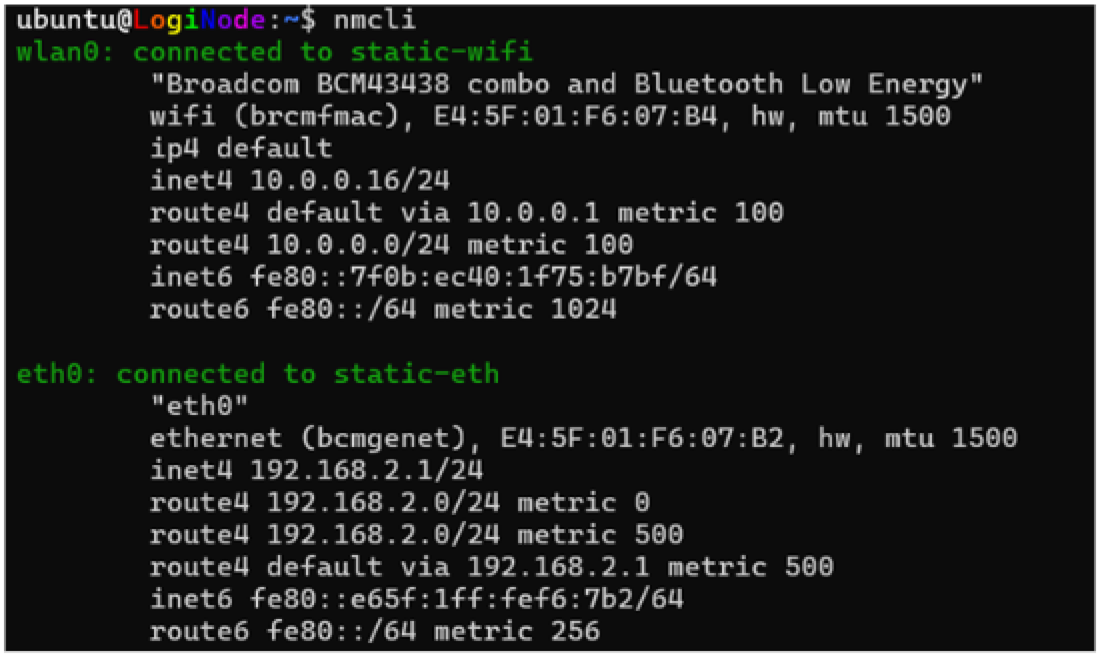
2 System Setup

“How do you make it?”

2.1 The Login Node

We begin the System Setup by provisioning the cluster’s Login Node – the Raspberry Pi that serves as the entry point to the cluster. This node is a Raspberry Pi with serial number **f154e171**, equipped with a direct-attached SSD, onto which we installed the **Ubuntu 24.04.2 LTS** image. Initial bring-up was performed via local console (monitor and keyboard) to complete first-boot tasks, after which the node joined the site WLAN through wlan0.

Networking on the Login Node is managed by Linux’s Network Manager with a static IP assigned to wlan0, providing stable, discoverable address for downstream components. This foundation – hardware, operating system and reliable network identity – establishes the fixed point the rest of the cluster will depend on in later stages.


```
ubuntu@LogiNode:~$ nmcli
wlan0: connected to static-wifi
        "Broadcom BCM43438 combo and Bluetooth Low Energy"
        wifi (brcmfmac), E4:5F:01:F6:07:B4, hw, mtu 1500
        ip4 default
        inet4 10.0.0.16/24
        route4 default via 10.0.0.1 metric 100
        route4 10.0.0.0/24 metric 100
        inet6 fe80::7f0b:ec40:1f75:b7bf/64
        route6 fe80::/64 metric 1024

eth0: connected to static-eth
        "eth0"
        ethernet (bcmgenet), E4:5F:01:F6:07:B2, hw, mtu 1500
        inet4 192.168.2.1/24
        route4 192.168.2.0/24 metric 0
        route4 192.168.2.0/24 metric 500
        route4 default via 192.168.2.1 metric 500
        inet6 fe80::e65f:1ff:fef6:7b2/64
        route6 fe80::/64 metric 256
```

Via eth0 interface (the Ethernet port) the Login Node Pi connects to the **Ubiquiti UniFi USW Pro 48 PoE Managed L2 PoE+** switch. This switch powers all cluster Pis (PoE) and provides network connectivity among them. Other than PoE, the switch isn't used for any additional functionality; it operates purely at the Data Link Layer (Layer 2) level as a simple L2 switch.

2.2 Netboot Setup

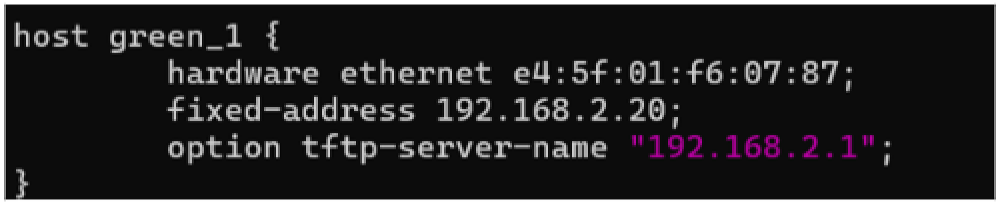
The remaining Pis do not have permanently attached storage; they must abstain a network share and boot over the network. This is referred to as "NFS boot" or "Netboot".

2.2.1 DHCP Server

The first step in that process is setting up a DHCP server.

The DHCP server is the critical component that assigns IP addresses to the worker Pis[1] and advertises the TFTP server location, so each node can fetch the required boot files for network boot. In our setup, we use `isc-dhcp-server`, configured via `/etc/dhcp/dhcp.conf`. We chose static leases for all nodes, to ensure that every machine always receives the same, predictable address.

[1] Strictly speaking, the worker Pis are the ones labeled as yellowX/greenX. However, we often refer to "workers" as all the nodes that are booting of NFS, so yellowX/greenX along with the mlops_master node



```
host green_1 {
        hardware ethernet e4:5f:01:f6:07:87;
        fixed-address 192.168.2.20;
        option tftp-server-name "192.168.2.1";
}
```

Hostnames for the Pis follow a simple convention, based on the color of their Ethernet cables:

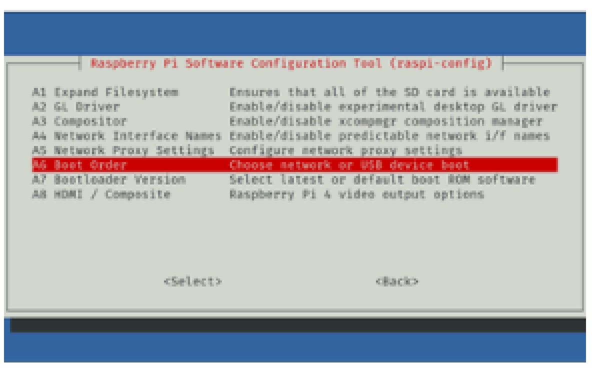
Hostname	IP Address
`Login Node`	`192.168.2.1`
`MLops_master`	`192.168.2.2`
`Green 1-8`	`192.168.2.20-27`
`Yellow 1-8`	`192.168.2.28-35`

To make each Raspberry Pi netboot-ready, we first flash a Raspberry Pi OS image to an SD card and boot each Pi from that card. This intermediate step is needed because Raspberry Pis don't have a traditional BIOS; you must update the device's bootloader settings to enable network boot.

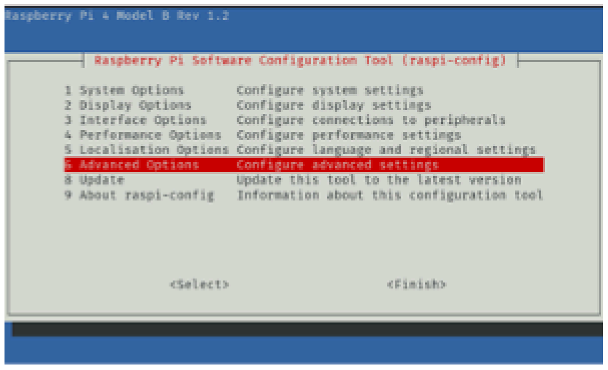
On each Pi, we then ran:

```
$ sudo raspi-config
```

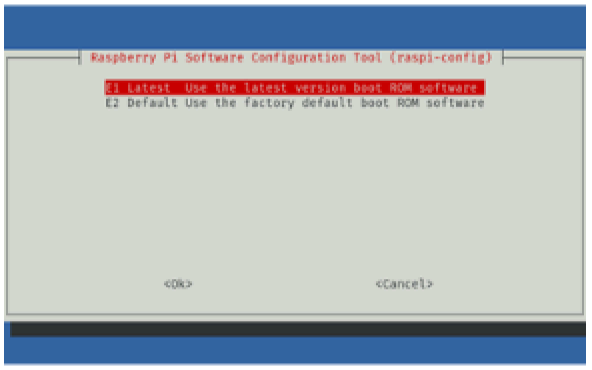
Next, we opened the Bootloader menu, and enabled network boot (PXE/NFS boot) as shown below. We saved the changes and rebooted to apply the new boot order. After confirming the setting had taken effect, we could remove the SD card and proceed with network booting.



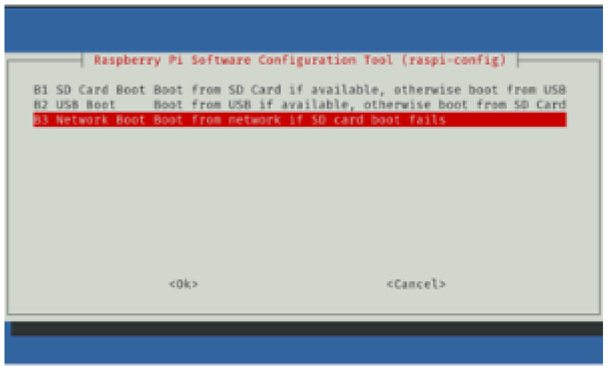
```
Raspberry Pi Software Configuration Tool (raspi-config)

A1 Expand Filesystem        Ensures that all of the SD card is available
A2 GL Driver                Enable/disable experimental desktop GL driver
A3 Compositor               Enable/disable xcompmgr composition manager
A4 Network Interface Names  Enable/disable predictable network i/f names
A5 Network Proxy Settings   Configure network proxy settings
A6 Boot Order               Choose network or USB device boot
A7 Bootloader Version       Select latest or default boot ROM software
A8 HDMI / Composite         Raspberry Pi 4 video output options

          <Select>                    <Back>
```



```
Raspberry Pi 4 Model B Rev 1.2

Raspberry Pi Software Configuration Tool (raspi-config)

1 System Options       Configure system settings
2 Display Options      Configure display settings
3 Interface Options    Configure connections to peripherals
4 Performance Options  Configure performance settings
5 Localisation Options Configure language and regional settings
6 Advanced Options     Configure advanced settings
8 Update               Update this tool to the latest version
9 About raspi-config   Information about this configuration tool

          <Select>                    <Finish>
```



```
Raspberry Pi Software Configuration Tool (raspi-config)

E1 Latest  Use the latest version boot ROM software
E2 Default Use the factory default boot ROM software

          <Ok>                    <Cancel>
```



```
Raspberry Pi Software Configuration Tool (raspi-config)

B1 SD Card Boot Boot from SD Card if available, otherwise boot from USB
B2 USB Boot     Boot from USB if available, otherwise boot from SD Card
B3 Network Boot Boot from network if SD card boot fails

          <Ok>                    <Cancel>
```

Additionally, Raspberry Pis can bypass DHCP and attempt to netboot from a hard-coded IP. WE encountered this and fixed it by editing the EEPROM configuration, via running:

```
$ sudo -E rpi-eeprom-config --edit
```

and removing any lines that define TFTP_IP and CLIENT_IP.

After reboot, the EEPROM is refreshed and the Pi is ready for the Netboot. We repeated this procedure on every Pi, one by one. For later steps, we also recorded each device's serial number with:

```
$ cat /proc/cpuinfo | grep Serial
```

2.2.2 TFTP Server

Once the Pis were netboot-enabled and receiving IPs from DHCP, we set up the TFTP server. We use tftpd-ha, configured via `/etc/default/tftpd-hpa`.

The TFTP root directory is where Pis request their boot files (such as firmware, kernel, intramfs). We need to make sure that the directory exists, is readable by the TFTP daemon, and contains expected Raspberry Pi boot files.

2.2.3 NFS Root FIlesystem

After a Pi fetches the kernel and boot configuration from TFTP, it needs a root filesystem to mount and complete boot. We installed and configured nfs-kernel-server, and defined exports in:

`/etc/exports`

Both the root and boot directories should be exported, so the clients can mount them during boot. We then started the services on the Login Node:

```
$ sudo systemctl start isc-dhcp-server
$ sudo systemctl start tftpd-hpa
$ sudo systemctl start nfs-kernel-server
```

At this point, the Login Node's DHCP, TFTP, and NFS services are up.

2.2.4 Build the Netboot OS Image

We converted our OS image (.img) into the directory structure required for netboot. We used kpartx to map the image to device nodes under /dev/mapper.

- `/dev/mapper/loop2p1` → **Partition 1** (Boot)
- `/dev/mapper/loop2p2` → **Partition 2** (Root filesystem)

We copied both partitions into each PI's NFS root, e.g.: `$NFS/green1/`. We then updated the client's etc/fstab to include the network mount(s), and adjusted cmdline.txt to point the kernel to the NFS root.

(e.g. `root=/dev/nfs nfsroot=<login-node-ip>:/srv/nfs/green1, vers=... ip=dhcp`).

We automated this process with a Bash script, namely `pi_boot.sh`.

For the final mapping step, the Raspberry Pi boot firmware looks in the TFTP root for a file named after the device's serial number. We created that file and made it a symlink to the Pi's own boot directory. With everything correctly wired, the Pis netbooted on the next restart.

2.2.5 Passwordless Access to Clients

To avoid passwords and simplify operations, we set up SSH key-based access. To connect, we copied the Login Node's public key into each Pi's `~/.ssh/authorized_keys`. Because the home directories are available via NFS, we could write the key directly into the user's home on the server side and then SSH in. This process needs to happen after the first successful client boot, when the home/ubuntu directory has been created.

2.3 Internet Access

The only PIs connected directly to the local WLAN are the login and backup node. All the other PIs therefore need a way to connect to the internet so updates, and other crucial components can be downloaded. This issue can be fixed by setting up the Login Node PI as a NAT gateway that routes the traffic to and from the nodes using NAT tables. The process of doing this is very straight forward:

First, we enable IP forwarding on the login Node by adding the line:

```
net.ipv4.ip_forward = 1
```

Inside the file `/etc/sysctl.conf`.

Then. run:

```
$ sudo sysctl -p
```

to apply the change.

Next, we need to set up a masquerade rule that changes the IPs of the requests so they can reach the gateway of the WLAN. This is done using the following IP tables rule:

```
$ sudo iptables -t nat -A POSTROUTING -o wlan0 -j MASQUERADE
```

If we want the above command to be persistent across restarts, we can place it inside the file: `/etc/rc.local`

To see all applied changes, use:

```
$ sudo iptables -t nat -L -n -v
```

2.4 vmlinuz update patch

A known bug in Ubuntu based nfs systems is the change of the kernel file permissions after apt package updates.

Specifically, after an update the `/boot/vmlinuz` file permissions are changed to 600 instead of 644.

In order to resolve this issue we added a script in the */etc/kernel/postinst.d* directory which is executed automatically after every kernel update. The name of the script is `zzz-chmod-vmlinuz` so its executed last (based on alphabetical order) and its content is a simple `chmod 0644 /boot/vmlinuz-*` command.

2.5 Pi-Backup

Because Login node's place in the system was fundamental, its crash out blocked the access to the entire Cluster, forcing us to attend the lab to reboot it. The need for automation and safety led us to the setting up of another, second in hierarchy node, namely Pi-Backup.

Pi-Backup is booting from an SD card, saving the unnecessary trouble of cable connectivity etc. It's been set up with Raspberry Pi OS, and is accessible from outside the Cluster, at its own IP address.

Additionally, Pi-Login crashing means having the DHCP server stop, and thus losing connectivity to switch. In order to tackle this issue, we installed a DHCP server inside the Pi-Backup with mirror configuration to the one in Login.

The SD card boot proved to be much more reliable than the SSD and migrating the login node's OS to another SD card sounds like a good idea. However due to the scale of the existing data this process is not straight forward.

2.6 The UniFi frontend

The controller for the Unifi switch allows for simple management of the cluster through a web interface, and most importantly allows remote restarts of nodes by POE control. The controller is easily installed as a docker container using:

```
$ sudo docker run -d \
  --name unifi \
  --restart unless-stopped \
  -v /srv/unifi:/unifi \
  -e TZ='Europe/Athens' \
  -p 3478:3478/udp -p 8080:8080 -p 8443:8443 \
  jacobalberty/unifi:latest
```

Once the container is running we can access the front-end at the port 8443 of the Loginode.

Tunneling the port through ssh allows the controller to be accessible on your local device, using: `-L 8443:localhost:8443` when connecting using ssh.

2.7 Fancy Names

The username of every worker node is *Ubuntu* for consistency. However, this makes recognizing the nodes difficult and ultimately causes confusion.

That's why we edited the PS1 line in every node to change the displayed name. We also changed its colour for a prettier working environment.

This change was automated with the *`fancy_name.sh`* script.

2.8 Kubernetes Setup

At this stage, our system is pretty much set up and ready to start accommodating software looking for metal to run on. To manage this software running on distributed hardware, we chose to utilize Kubernetes orchestration, specifically K3s - a lightweight fork of Kubernetes.

The K3s server will be mlops_master and K3s agents will run on the worker nodes. Before we install K3s itself, we need to install *fuse-overlayfs* on every machine:

In mlops_master and all workers:

```
$ sudo apt install fuse-overlayfs
```

During the container runtimes, K3s utilizes overlayfs (overlay filesystems), a linux kernel utility used by many popular filesystems (such as ext4). Although this utility achieves lightweight, writeable filesystems, it is not supported by NFS. *fuze-overlayfs* helps patch this issue by implementing overlayfs logic in the userspace.

For the K3s installation we followed the instructions from K3s Documentation. The only change is that we need to specify the use of fuse-overlayfs. For the K3s-agent installation we also need to provide the server token. We also chose to set node names for the workers that match the corresponding pi names (yellowX/greenX).

In mlops_master:

```
$ curl -sfL https://get.k3s.io | sh -s - server
--snapshotter=fuse-overlayfs
```

In worker nodes:

```
$ curl -sfL https://get.k3s.io | K3S_NODE_NAME=<PI_NAME>
K3S_URL=https://192.168.2.2:6443
K3S_TOKEN=<TOKEN>INSTALL_K3S_EXEC="--snapshotter=fuse-overlayfs" sh
-
```

After some time, K3s is activated and operates as a systemd service (`k3s.service` for mlops_master, `k3s-agent.service` for workers).

Lastly, we would like the Master Node to perform solely orchestration work and not receive container workload. We can do that with the following command:

In mlops_master:

```
$ sudo kubectl taint nodes ubuntu
node-role.kubernetes.io/control-plane:NoSchedule
```

2.9 Jen the Chatbot

The final step of our System's Setup process is the creation and deployment of a containerized Flask web application, intended to expose Large Language Model (LLM) functionality via HTTP endpoints.

The app consists of a small HTTP API that wraps an LLM. The text is sent in a JSON request; the service asks a model to generate a reply; it returns that reply as JSON. The repository is a straightforward Flask/Docker skeleton: an `app/` package with the routes and initialization, a `run.py` entrypoint, a `config.py` that reads settings from environment variables, a `requirements.txt`, and a `Dockerfile` so we can package everything the same way on every machine. In short, it's a tiny, portable API whose only job is to turn a prompt into a model response and hand it back quickly and predictably.

To run this on Raspberry Pis, we first turned the source code into a container image that matches their architecture. We built an `ARM64` image from the provided `Dockerfile`, using the python Flask library. The image runs as a non-root user and uses a read-only root filesystem; these choices make the container safer by default without complicating the app. Once built, the image was made available to the cluster— by exporting it to a tarball, copying it to the `MLops_master` host over SSH, and importing it into k3s's container runtime. From that point on, every node can start the same bits, with the same dependencies, in the same way.

The second piece of the puzzle is the model data. Rather than copying large model files to every Pi, we keep a single authoritative copy on the backup node's HDD under `/mnt/nfs_share/models`. That directory is exported over NFS to the cluster on a read-only basis. Read-only matters: it prevents a buggy pod from corrupting the model, and it keeps the update story simple—when you want to refresh the model, you do it once on the backup node and all pods see the new files the next time they read them.

Kubernetes ties this storage into the application with native building blocks. We declare a **PersistentVolume** that points to the NFS export on the backup node, and a **PersistentVolumeClaim** in the application namespace that binds to that volume with "read-only many" access. The Deployment for the Flask app then mounts that claim inside each pod at `/models`. The application reads its configuration from environment variables—things like `MODEL_DIR=/models`, the listening port—so the same container image can serve different environments without code changes. Liveness and readiness probes point at `/healthz`, so rollouts happen safely: Kubernetes only sends traffic to a pod after it proves it's ready.

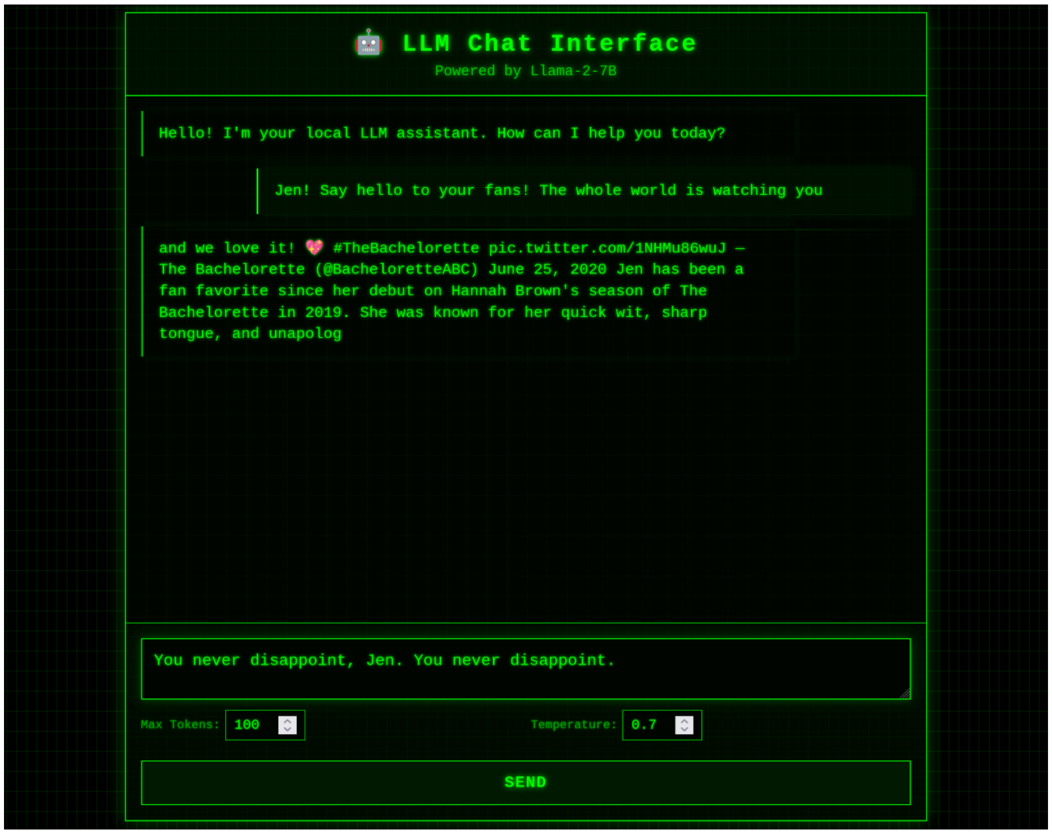
Networking is the last piece. Inside the cluster we expose the pods with a Service, and for clients on your LAN we use MetalLB to assign a stable LoadBalancer IP. Requests travel to that IP and are balanced to whichever replica is healthy. At runtime the sequence is simple.

1. The scheduler places a pod on an `ARM64` Pi;
2. kubelet mounts the NFS-backed PVC into the container at `/models`;
3. `gunicorn` starts the Flask app, which reads its settings, loads or wires up the model from that directory, and begins responding.
4. The readiness probe succeeds and the Service starts sending it traffic.

Each additional replica goes through the same boot: same image, same configuration, and the same shared model folder. If a node fails, the Deployment recreates the pod elsewhere, and the new instance mounts the very same model from the SSD without any special per-node steps.

This design fits the Pis well. The container image keeps the runtime lean and identical across diverse nodes; the shared NFS mount avoids duplicated gigabytes and manual synchronization; probes, non-root, and a read-only root filesystem keep the service robust; and the Service/LoadBalancer layering gives you a stable, easy-to-reach endpoint.

When you ship a new version, you build the ARM64 image, point the Deployment at the new tag, and let the rolling update replace old pods with new ones once they pass `/healthz`. If something goes wrong, you roll back and you're back to a known-good state in moments. In the end, "Jen the Chatbot" is just a neat, containerized Flask API, but the way it's wired—shared storage for the model, clean configuration, and Kubernetes primitives doing the heavy lifting—makes it dependable on a modest, heterogeneous Raspberry Pi cluster.



```
Hello! I'm your local LLM assistant. How can I help you today?

Jen! Say hello to your fans! The whole world is watching you

and we love it! 💖 #TheBachelorette pic.twitter.com/1NHMu86wuJ —
The Bachelorette (@BacheloretteABC) June 25, 2020 Jen has been a
fan favorite since her debut on Hannah Brown's season of The
Bachelorette in 2019. She was known for her quick wit, sharp
tongue, and unapolog

You never disappoint, Jen. You never disappoint.

Max Tokens: 100          Temperature: 0.7
```

2.10 Expose K3s Service

You can see the various deployed services and their exposed ports with:

```
$ kubectl get svc
```

We simply want to redirect our target port to pi_login, so we can later redirect it to our pc.

Now, there are many ways to do this. The most efficient one is probably to hit the Cluster IP: `192.168.2.21:5000`

You might notice that this is the IP of green2, a seemingly random node[2]. This doesn't mean that green2 is responsible for handling all external requests. In fact, when hitting the Cluster IP you are probably not hitting green2 at all! That's because this is actually a

[2] And, to be completely honest, we don't remember how we ended up using this as the Cluster IP

virtual IP. For more context, check the MetalLB Documentation. You can validate the IP's functionality with curl:

From any node:

```
$ curl 192.168.2.21:5000
```

Another way to access your service is from mlops_master's localhost:30633. This is a bit less efficient, since mlops_master itself redirects data from the Cluster IP. However, since in that case we know which device is going to be serving us, it is very easy to redirect its port , using ssh tunneling:

From your PC:

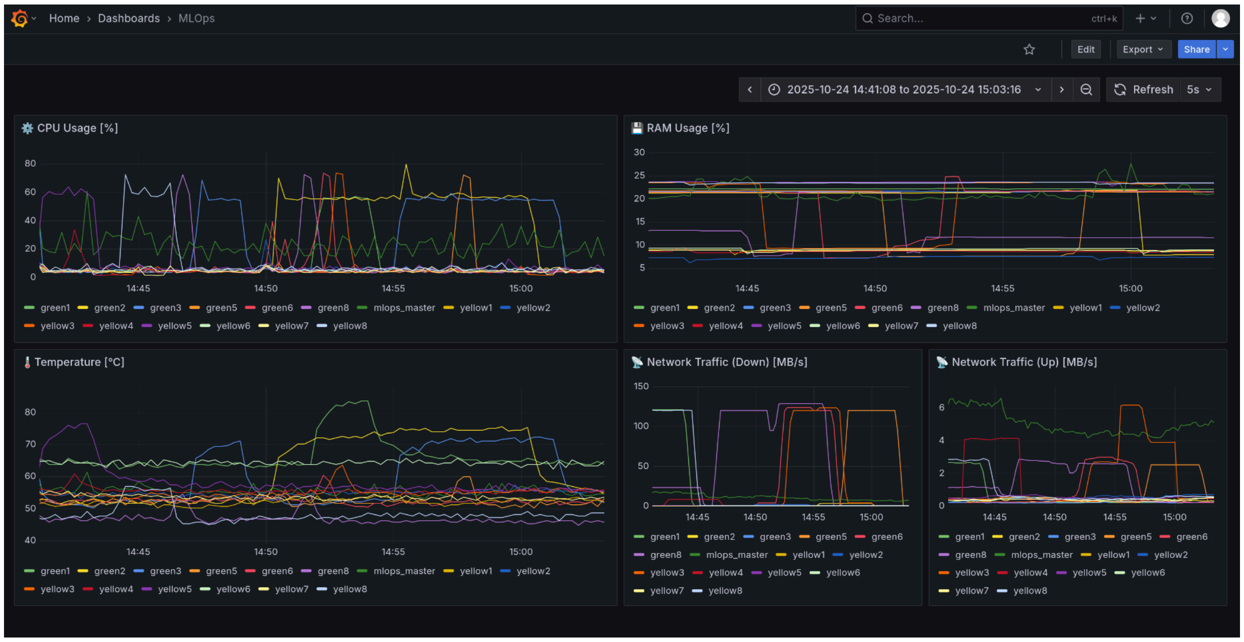
```
$ ssh -L 6001:localhost:6001 -p 4444 ubuntu@147.102.3.83
```

From the Login Node:

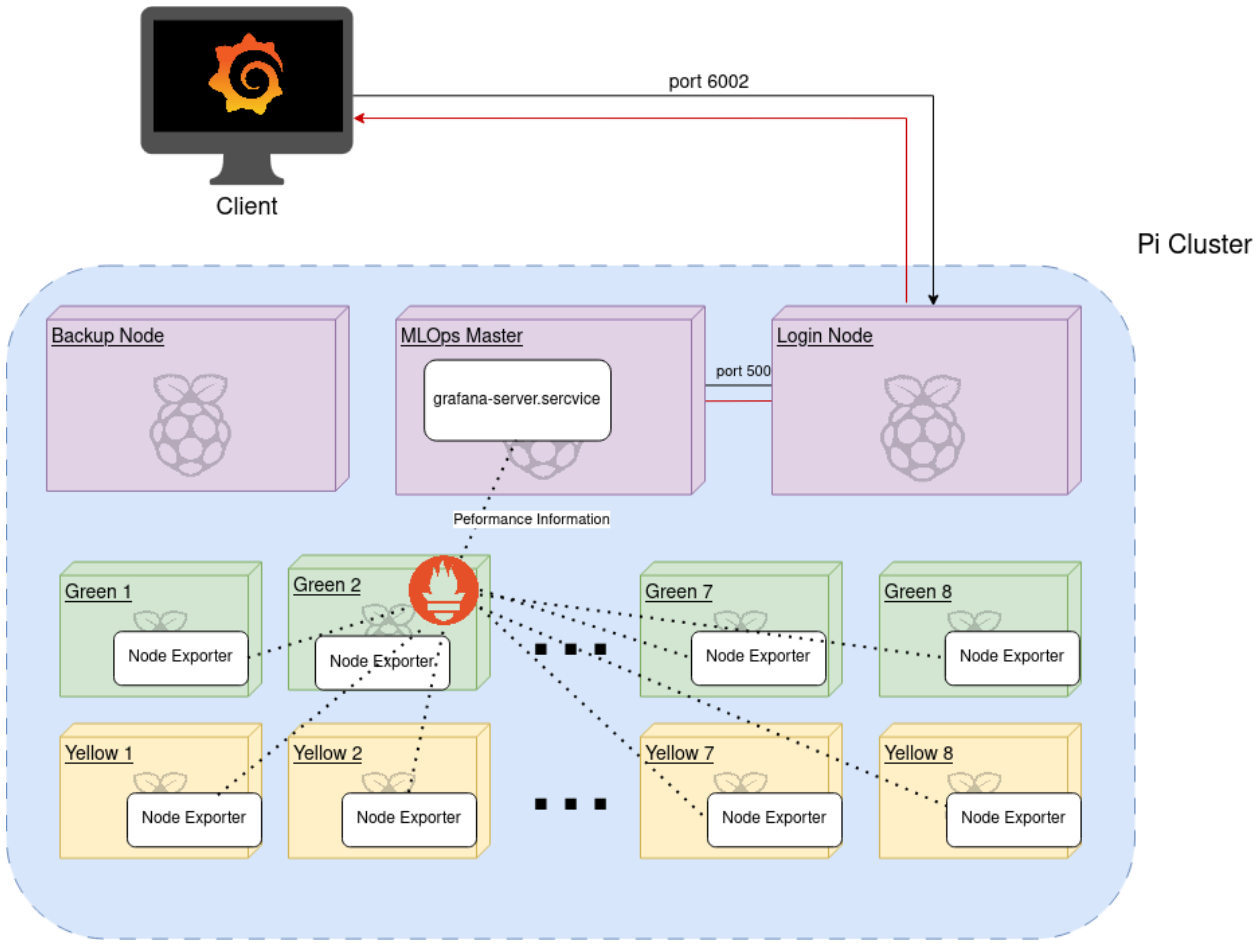
```
$ ssh -L 6001:localhost:30633 mlops_master
```

Now, if you go to your browser and visit `http://localhost:6001/`, Jen will appear before your eyes!

2.11 Prometheus/Grafana



These services will allow us to neatly monitor the state of our system, in terms of metrics such as cpu usage, temperature, etc. Prometheus is responsible for collecting and storing any such data from the cluster nodes. It then exposes this data to the specified port. Grafana then reads this data and plots them in a frontend, which in turn is exposed to a specified port. This process is better visualized in the following picture:



We initially set these services up as k3s pods, using the k3s package manager, helm:

```
$ helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
$ helm repo add grafana https://grafana.github.io/helm-charts
$ helm repo update


$ kubectl create namespace monitoring

$ helm install prometheus prometheus-community/prometheus -n
monitoring

$ helm install grafana grafana/grafana -n monitoring
```

This worked perfectly. Until it didn't. Everything broke down.

One issue was that k3s would allocate the prometheus and grafana servers to random nodes in the system, instead of mlops_master. The result was that when we were running our benchmarks and the node responsible for prometheus or grafana died, we would lose our entire monitoring system. Additionally, we could not store the metrics persistently, unless we used another nfs directory in pi_backup's HDD.

These issues could be solved within k3s, but the already slow nature of k3s-over-nfs made this process incredibly painstaking. So we opted for another option: we kept the already working prometheus node exporters as k3s pods, and installed the prometheus and grafana servers as systemd services in the mlops_master node.

2.11.1 Prometheus Server:

First, we create a new user. This is generally advised for security reasons:

```
$ sudo useradd --no-create-home --shell /bin/false prometheus
```

We then create some essential directories:

```
$ sudo mkdir /etc/prometheus /var/lib/prometheus
$ sudo chown prometheus:prometheus /etc/prometheus
/var/lib/prometheus
```

Then, we download the Prometheus binaries, and move them to the correct locations:

```
$ wget
https://github.com/prometheus/prometheus/releases/download/v3.4.1/p
ro$ metheus-3.4.1.linux-arm64.tar.gz
$ tar xvf prometheus-3.4.1.linux-arm64.tar.gz

$ cd prometheus-3.4.1.linux-arm64
$ sudo cp prometheus promtool /usr/local/bin/
$ sudo cp -r consoles console_libraries /etc/prometheus/
$ sudo cp prometheus.yml /etc/prometheus/
$ sudo chown -R prometheus:prometheus /etc/prometheus
```

Now, we want to set Prometheus to run as a systemd service. In the `/etc/systemd/system/prometheus.service` file, paste the following:

```
[Unit]
Description=Prometheus Monitoring for HPC Cluster
Wants=network-online.target
After=network-online.target

[Service]
User=prometheus
Group=prometheus
Type=simple
ExecStart=/usr/local/bin/prometheus \
    --config.file /etc/prometheus/prometheus.yml \
    --storage.tsdb.path=/var/lib/prometheus/ \
    --web.console.templates=/etc/prometheus/consoles \
    --web.console.libraries=/etc/prometheus/console_libraries

[Install]
WantedBy=multi-user.target
```

Before starting the service, we want to configure it, so that it can scrape the node exporter data. Copy and paste the following into `/etc/prometheus/prometheus.yml`:

```
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to every 15
seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The
default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
        - targets:
          # - alertmanager:9093

# Load rules once and periodically evaluate them according to the
global 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
```

```
# Here it's Prometheus itself.

scrape_configs:
  - job_name: "prometheus"
    static_configs:
      - targets: ["localhost:9090"]
        labels:
          app: "prometheus"

  - job_name: "mlops_nodes"
    static_configs:
      - targets: ['localhost:9100']
        labels:
          node: 'mlops_master'
      - targets: ['192.168.2.20:9100']
        labels:
          node: 'green1'
      - targets: ['192.168.2.21:9100']
        labels:
          node: 'green2'
      - targets: ['192.168.2.22:9100']
        labels:
          node: 'green3'
      - targets: ['192.168.2.23:9100']
        labels:
          node: 'green4'
      - targets: ['192.168.2.24:9100']
        labels:
          node: 'green5'
      - targets: ['192.168.2.25:9100']
        labels:
          node: 'green6'
      - targets: ['192.168.2.26:9100']
        labels:
          node: 'green7'
      - targets: ['192.168.2.27:9100']
        labels:
          node: 'green8'
      - targets: ['192.168.2.28:9100']
        labels:
          node: 'yellow1'
      - targets: ['192.168.2.29:9100']
        labels:
          node: 'yellow2'
      - targets: ['192.168.2.30:9100']
        labels:
          node: 'yellow3'
      - targets: ['192.168.2.31:9100']
        labels:
          node: 'yellow4'
      - targets: ['192.168.2.32:9100']
        labels:
          node: 'yellow5'
      - targets: ['192.168.2.33:9100']
        labels:
          node: 'yellow6'
      - targets: ['192.168.2.34:9100']
        labels:
          node: 'yellow7'
      - targets: ['192.168.2.35:9100']
        labels:
          node: 'yellow8'
```

With this config, we instruct Prometheus to collect the node exported data from ports 9100, which are used by default. We additionally give labels to each node, which we are going to use later in the Grafana stage. We also tell Prometheus to expose the collected data to localhost:9090

At last, start and enable the newly created service:

```
$ sudo systemctl daemon-reload
$ sudo systemctl start prometheus
$ sudo systemctl enable prometheus
```

2.11.2 Grafana Server:

For the Grafana server, we followed the process described in the documentation.

After installation, we need to access its GUI. By default, Grafana is exposed at port 3000. So, we redirect it to pi_login's 6002, which we then direct to our own localhost:6002:

On your pc:

```
$ ssh -L 6002:localhost:6002 -p 4444 ubuntu@147.102.3.83
```

On the Login Node:

```
$ ssh -L 6002:localhost:3000 mlops_master
```

You should now be able to see the grafana frontend by visiting localhost:6002 from your browser. We set up the credentials:

user: admin
password: k3sucks

Now, we need to connect Grafana to Prometheus. Go to:
Connections → Data sources → Prometheus
and set `http://localhost:9090` as the Prometheus server url.

You can now start querying the Prometheus/Grafana system, using the GraphQL language. You may experiment with explore data → Open in Explore Window. Below are the queries we used:

CPU usage:

```
100 - (avg by(node)
(irate(node_cpu_seconds_total{mode="idle"}[5m])) * 100)
```

RAM usage:

```
avg by(node) ((node_memory_MemTotal_bytes -
node_memory_MemAvailable_bytes) / node_memory_MemTotal_bytes * 100)
```

Temperature:

```
avg by(node) (node_hwmon_temp_celsius)
```

Network Traffic (Down):

```
sum by(node) (
  (rate(node_network_receive_bytes_total[5m]) * 8)
) / 1000000
```

Network Traffic (UP):

```
sum by(node) (
    (rate(node_network_transmit_bytes_total[5m]) * 8)
) / 1000000
```

In general, what you want to do with Grafana is create a dashboard and display the results of these queries in panels. You can easily do this by navigating the GUI.

3 System Functionalities

"How do you use it?"

3.1 Remote Access

Of course, you may access the login node locally, by connecting the lab monitor and keyboard. Remote access is done through ssh. To add yourself to the login node's authorized keys, you need to:

On your local machine:

1. Check if you have the `~/.ssh/id_ed25519.pub` file (or another similar encryption). If you don' t, simply run the `ssh-keygen` command.
2. Open the file and copy its content. That's your public key.

On the remote machine (login node):

1. Paste your public key on a new line in the `~/.ssh/authorized_keys` file.

Simple as that. You're pretty much set up. Now, to access the remote machine, run from your pc:

```
$ ssh ubuntu@147.102.3.83 -p 4444
```

Optionally, for easier access you may add the following lines to `~/.ssh/config`:

```
Host pi-login
     HostName 147.102.3.83
     Port 4444
     User ubuntu
     IdentityFile /home/mr/.ssh/id_ed25519
```

Now, you can connect to the login node simply by running:

```
$ ssh pi-login
```

You can access the backup node in a similar way, simply by replacing the user with 'kalisperas' and port with '2222'.

For the rest of the system nodes, we have set up ssh aliases in pi_login and mlops_master. You can see them in `~/.ssh/config`. They are essentially the node names (eg. "`$ssh green4`")

3.2 Access Jen the Chatbot

The Jen cluster deployment is constantly exposed at pi_login's `localhost:6001`. To access the deployment as a user, you forward this port into your own localhost.

From your PC:

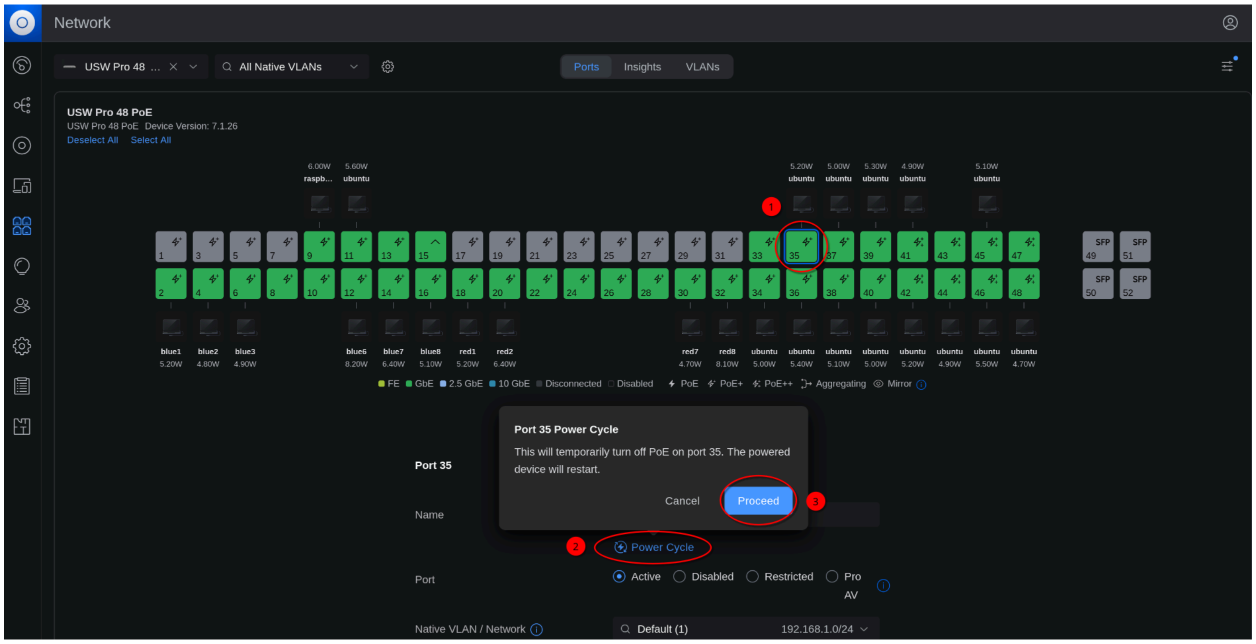
```
$ ssh -L 6001:localhost:6001 -p 4444 ubuntu@147.102.3.83
```

Now, in your browser, visit the address `localhost:6001` and say hi to Jen!

3.3 Rebooting Nodes

Well, this may become the most frequently used functionality. We' ve been doing this *a lot*, so we decided to make our lives easier by adding some automation in the process.

One way to restart the system nodes is through the UniFi frontend, as seen in the picture below. However, this method has two major flaws. Firstly, there is no way to restart multiple nodes at once, making the process of restarting the entire system quite painstaking. Secondly, and most crucially, if the login node dies for some reason (which happens quite often), the only way to perform a restart is by physically removing and replugging the ethernet cables. That is the main reason why we added the backup node in the first place.



Port 35 Power Cycle

This will temporarily turn off PoE on port 35. The powered device will restart.

Cancel Proceed

The backup node can connect to the UniFi service by ssh-ing into the switch machine. Utilizing that, we wrote a script, `~/power_cycle.sh`, that can perform node rebooting even when the login node is offline. The script takes only one argument:

```
$ ./power_cycle.sh <PI_NAME>
```

where `<PI_NAME>` can be:

- `yellowX` or `greenX`, where X=1,2,...,8 *(restarts the corresponding node)*
- `login` or `backup` or `mlops_master` *(restarts the corresponding node)*
- `yellow` or `green` *(restarts the entire yellow/green row)*
- `all` *(restarts all nodes, except the backup node itself)*

This way, we can restart remotely any node in the system. Additionally, this allows us to easily restart sections of nodes, or the entire network.

After restarting the system, worker nodes might have an issue booting over NFS. If this occurs, you can try restarting the NFS service of pi_login:

In the Login Node:

```
$ sudo systemctl restart nfs-server
```

3.4 Adding and Removing Nodes

3.4.1 Adding a New Node

We have automated this process with some bash scripts. First, you need to add the new Pi's attributes to the `~/MLops_stuff/pi_info.txt` file. Namely, you add a new line with the following information:

```
<PI_NAME>     <PI_MAC>     <PI_SERIAL>     <PI_LOCAL_IP>
```

Then, on the Login Node:

```
$ cd ~/MLops_stuff
$ sudo ./pi_boot.sh <PI_NAME>
-- restart worker --
$ ./worker_ssh.sh <PI_NAME>
```

Where `PI_NAME` is the name of the new node (similar to yellowX/greenX)

`pi_boot.sh` copies the template boot and root directories into the `/mnt/netboot_common/nfs/`**`<PI_NAME>`** directory, and updates the NFS-relevant files with the new Pi's info.

`worker_ssh.sh` simply copies pi_login's public ssh key into the new node's authorized keys, to enable ssh access.

We have now set up the netboot of the new node. Now, we want to set up various utilities to bring it up-to-date with the rest of the cluster. This is now the responsibility of the MLOps Master Node, which automates this process with Ansible scripts. Ansible is an automation tool that works on a higher level than bash scripts. Technically, bash would be enough for this process, but we wanted to start incorporating a more advanced tool that might save a lot of time in later automations.

First, you need to update mlops_master's `~/pi_info.txt` file. Also, you need to add the new Pi's info into `~/Ansible/inventory.ini`

In the MLOps Master Node:

```
$ ./worker_ssh.sh <pi_name>
$ cd ~/Ansible
$ ansible-playbook -i inventory.ini <PI_NAME> worker_setup.yml
```

`worker_setup.yml` is an Ansible Playbook that performs the rest of the setup. Keep in mind that this script is under development and might result in errors. In that case, you can find playbooks in the ~/Ansible directory that perform separate steps of the setup process, and run the ones needed. Otherwise, you could just understand what the script is trying to do and do it manually :).

3.4.2 Removing a Node

Simply delete its root from the nfs directory:

On the Login Node:

```
$ sudo rm -rf /mnt/netboot_common/nfs/<PI_NAME>
```

3.5 Access Grafana

This process is already described in Section 2, but we also mention it here. First you need to redirect pi_login's port 6002 to your localhost.

On your pc:

```
$ ssh -L 6002:localhost:6002 -p 4444 ubuntu@147.102.3.83
```

Then, if not done already, redirect mlops_master's 3000 to pi_login's 6002:

On the Login Node:

```
$ ssh -L 6002:localhost:3000 mlops_master
```

You should now be able to see the grafana frontend by visiting localhost:6002 from your browser. For the credentials, use:

user: admin
password: k3sucks

You can see our dashboard at Dashboards → MLOps

4 FOES (Frequently Occurring Errors & Solutions)

"How do you fix it?"

4.1 Some Debugging Tips

Here we mention some general debugging methods and tests that we have found ourselves using extensively.

4.1.1 Check whether the worker nodes are alive and responding

This is one the first checks performed when something goes wrong. You can easily test all the worker nodes by running our `pssh-echo` alias from the mlops_master node. This alias runs the following command:

```
$ parallel-ssh -h ~/.hosts.txt -l ubuntu -i echo "Hello world"
```

which sshes into each of the machines in hosts.txt and executes a simple echo command. You can also run this command from the login node, with the hosts.txt being in the ~/MLops_stuff directory.

4.1.2 Ping

You can check the connectivity of the Pis by pinging one-another. For example:

From the Login Node:
```
$ ping 192.168.2.23
```
(pings green4)

4.2 Workers Cannot Netboot

Of course, there are many factors that could lead to this problem. Thankfully, the most frequent cause seems to be an issue with the NFS-related services. from the login node, try restarting:

- The NFS server service:
  `$ sudo systemctl restart nfs-server.service`
- The DHCP service:
  `$ sudo systemctl restart isc-dhcp-server.service`
- The TFTP service:
  `$ sudo systemctl restart tftpd-hpa.service`

4.3 Cannot SSH to Login Node

```
kex_exchange_identification: read: Connection reset by peer
Connection reset by 147.102.3.83 port 4444
```

This has to be the most frequent FOE in our project, as it occurs on a weekly basis. To solve it, simply restart the system through the backup node.

There are many theories as to why this occurs. It probably has to do with the external SSD, as at this state pi_login outputs read-only filesystem errors. A possible solution would be to put pi_login's boot partition in an SD card and boot it from there. We see this as a future task for our team.

4.4 Workers Sudo Issues

```
Stderr: sudo: unable to resolve host ubuntu: Temporary failure in name resolution
```

This has an easy fix. In the problematic node:

```
$ hostname
$ sudo vim /etc/hosts
-- add line "127.0.1.1 <hostname_result>" --
```

4.5 Incorrect timestamps

Besides incorrectly displaying time, this error can cause many issues in the k3s node communication, as the authentication process seems to utilize node timestamps. To fix it, run the following commands in the problematic node:

```
$sudo timedatectl set-timezone Europe/Athens
$sudo timedatectl set-ntp true
```

`set-ntp true` enables automatic time sync.

If the problem insists, try pinging the ntp server, most likely `ntp.ubuntu.com`. If you get no response, it is probably a network issue.

4.6 K3s Timeouts

```
Timeout: request did not complete within requested timeout - context
deadline exceeded
```

Another sworn FOE. We are still not sure about the underlying factor of this issue, but it might have to do with a k3s setting that assumes that a node has an error after a delayed response, while the pis are just slow to respond due to their limited specs. Whenever we stumble upon this error, we just repeat the command over and over again until it finally works, often passing the parameter `--request-timeout=5m` (*"the definition of insanity is doing the same thing yet expecting different results."* ~Albert Einstein).

This might have to do with another observation: k3s tends to constantly kill and recreate existing nodes. We believe that after timeout errors k3s assumes the nodes are dead and restarts them.

Besides the Pis' limited specs, there is another suspect: NFS. K3s stores persistent data, such as node and pod info, in a SQLite database in mlops_master. Since mlops_master boots over NFS, all those read/writes are performed over the network and thus face significant slowdowns. This seems to be the most logical theory, and could be solved if the Master Node specifically booted from a local SD card.

Appendix A: Node Info Table

This table summarized useful information about all our system nodes:

Name	Serial Number	MAC Address	IP	Switch Port
pi_login	f154e171	e4:5f:01:f6:07:b2	192.168.2.1	15
pi_backup	6d30a4c2	d8:3a:dd:0b:01:17	192.168.2.3	9
mlops_master	2490e89e	d8:3a:dd:0b:01:2c	192.168.2.2	11
green1	85b6d0ea	e4:5f:01:f6:07:87	192.168.2.20	34
green2	0783b082	e4:5f:01:f6:08:02	192.168.2.21	36
green3	c72aafcf	e4:5f:01:f6:07:c6	192.168.2.22	38
green4	7cb48d88	e4:5f:01:f6:07:8d	192.168.2.23	40
green5	43b5285d	e4:5f:01:f5:f1:5d	192.168.2.24	46
green6	7adb8fa7	e4:5f:01:f6:07:e4	192.168.2.25	42
green7	e38b3110	e4:5f:01:f5:f1:ea	192.168.2.26	44
green8	26e8a399	e4:5f:01:f5:f1:87	192.168.2.27	48
yellow1	800fbfc4	e4:5f:01:f5:f2:05	192.168.2.28	33
yellow2	4c9e9b20	e4:5f:01:f5:f2:11	192.168.2.29	35
yellow3	c0a3cab4	e4:5f:01:f5:f1:ae	192.168.2.30	37
yellow4	d31716f4	e4:5f:01:f5:f0:dc	192.168.2.31	39
yellow5	ac1e75dd	e4:5f:01:f5:f1:a5	192.168.2.32	41
yellow6	d510fe23	e4:5f:01:f5:f2:38	192.168.2.33	43
yellow7	69f990f7	e4:5f:01:f5:f1:96	192.168.2.34	45
yellow8	0349cde2	e4:5f:01:f5:f1:d2	192.168.2.35	47

All nodes have “ubuntu” as a username and password, except for pi_backup, where the username is “kalisperas” and the password is “patata”.

pi_login external ip:port	147.102.3.83:4444
pi_backup external ip:port	147.102.3.83:2222