CDW Documentation

DGX Spark Monitoring Stack

DGX Spark Monitoring Stack

What We're Building

This guide sets up a **production-grade, end-to-end GPU/CPU observability stack** on an NVIDIA DGX Spark. By the end you will have:

- **Real-time terminal monitoring** of CPU cores, GPU utilization, power, temperature, and memory via a live TUI
- **A secured Prometheus metrics endpoint** on the Spark, protected by a Bearer token, scraping every 5 seconds
- **Grafana dashboards** running in Docker that visualize all metrics with live auto-refresh
- **Validated load testing** via a built-in synthetic CPU + GPU load generator

The full data path is:

```
nv-monitor (host) → Prometheus (Docker) → Grafana (Docker) → Your Mac browser (SSH tunnel)
```

Overview

Component	Role
nv-monitor	Reads CPU/GPU/memory metrics from the OS and NVML; exposes a token-authenticated Prometheus `/metrics` endpoint
Prometheus	Scrapes and stores time-series metrics from nv-monitor every 5 seconds
Grafana	Queries Prometheus and renders live dashboards in your browser
demo-load	Synthetic CPU + GPU load generator for validating the pipeline end-to-end

All Prometheus and Grafana services run in Docker containers, keeping the host clean and easy to remove. nv-monitor runs directly on the host to access hardware interfaces.

Step 1: SSH into the DGX Spark

From your Mac terminal, SSH into the Spark:

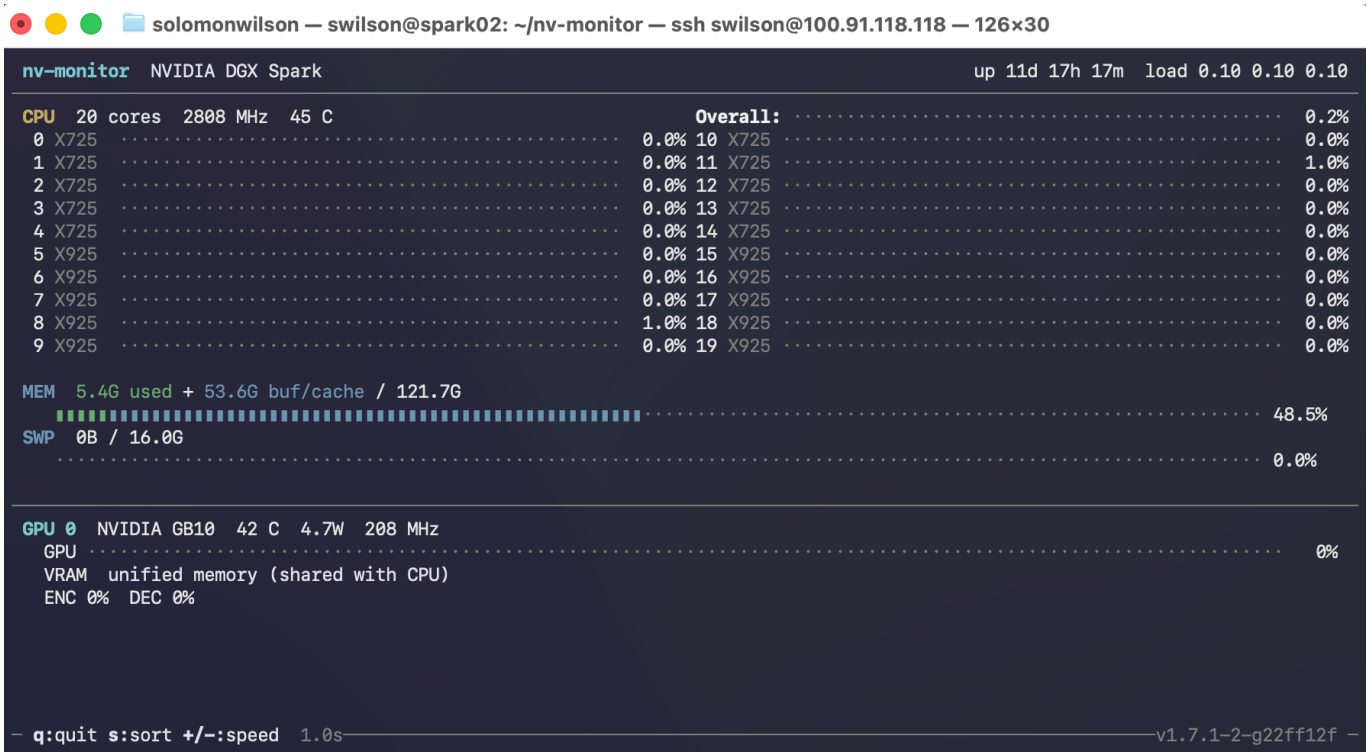
```
ssh YOUR_USERNAME@YOUR_SPARK_IP
```

All steps below are run on the Spark unless noted otherwise.

Step 2: Install Build Dependencies

```
sudo apt install build-essential libncurses-dev -y
```

- **build-essential:** gcc, make, and standard C libraries
- **libncurses-dev:** required for the terminal UI (ncursesw wide character support)



If already installed, apt will report “already the newest version” and exit cleanly — that is fine.

Step 3: Clone and Build nv-monitor

```
cd ~
git clone https://github.com/wentbackward/nv-monitor
cd nv-monitor
make
```

If the repo already exists from a previous run, git will print “destination path 'nv-monitor' already exists” and make will print “Nothing to be done for 'all'” — both are fine, the binary is already built.

Verify it works by launching the interactive TUI:

```
./nv-monitor
```

Press **q** to quit.

What the TUI shows

- **CPU section:** per-core usage bars, ARM core type labels (X925 = performance, X725 = efficiency)
- **GPU section:** utilization, temperature, power draw, clock speed
- **Memory section:** used, buf/cache, swap
- **VRAM:** shows “unified memory (shared with CPU)” on GB10 — this is expected, nvmlDeviceGetMemoryInfo returns NOT_SUPPORTED on the Grace-Blackwell unified memory architecture
- **History chart:** rolling 20-sample graph of CPU (green) and GPU (cyan)

Step 4: Run nv-monitor as a Prometheus Exporter

Start nv-monitor in headless mode with a Bearer token:

```
cd ~/nv-monitor
./nv-monitor -n -p 9101 -t YOUR_SECRET_TOKEN &
```

Replace `YOUR_SECRET_TOKEN` with a cryptographically secure token. Generate one with:

```
openssl rand -hex 32
```

Example output: `a3f8c21d9e4b76f0123456789abcdef0a1b2c3d4e5f6789012345678abcdef01`

You will use this same token in three places — keep it consistent:

1. The `-t` flag when starting nv-monitor (this step)
2. The `credentials:` field in `prometheus.yml` (Step 5)
3. Any `curl` commands used to verify the endpoint manually

If you ever rotate the token, update all three locations and restart both nv-monitor and Prometheus.

Flags explained

- **-n:** headless mode — no TUI, runs silently in the background
- **-p 9101:** expose Prometheus metrics endpoint on port 9101
- **-t YOUR_SECRET_TOKEN:** require this Bearer token on every HTTP request
- **&:** run in background so the terminal stays free

On startup it prints:

```
Prometheus metrics at http://0.0.0.0:9101/metrics
Running headless (Ctrl+C to stop)
```

Verify it is working:

```
curl -s -H "Authorization: Bearer YOUR_SECRET_TOKEN" localhost:9101/metrics
| head -10
```

You should see output starting with `# HELP nv_build_info`.

Available nv-monitor metrics

- `nv_cpu_usage_percent` — per-core CPU usage
- `nv_cpu_temperature_celsius` — CPU temperature
- `nv_gpu_utilization_percent` — GPU utilization
- `nv_gpu_power_watts` — GPU power draw in watts
- `nv_gpu_temperature_celsius` — GPU temperature
- `nv_memory_used_bytes` — RAM used in bytes
- `nv_load_average` — system load average (1m, 5m, 15m)
- `nv_uptime_seconds` — system uptime

Step 5: Create the Prometheus Configuration

```
mkdir ~/monitoring
```

```
cat > ~/monitoring/prometheus.yml << 'EOF'
global:
  scrape_interval: 5s
scrape_configs:
  - job_name: 'nv-monitor'
    authorization:
      credentials: 'YOUR_SECRET_TOKEN'
    static_configs:
      - targets: ['172.17.0.1:9101']
EOF
```

Replace `YOUR_SECRET_TOKEN` with the same token you used in Step 4.

Why 172.17.0.1 and not localhost?

- Docker containers have their own network namespace
- `localhost` inside a container refers to the container itself, not the host machine
- `172.17.0.1` is the Docker bridge gateway — the IP that containers use to reach the host
- Verify the gateway IP on your system: `docker network inspect bridge | grep`

Gateway

Step 6: Start Prometheus and Grafana in Docker

```
docker run -d \
  --name prometheus \
  -p 9090:9090 \
  -v ~/monitoring/prometheus.yml:/etc/prometheus/prometheus.yml \
  prom/prometheus
```

```
docker run -d \
  --name grafana \
  -p 3000:3000 \
  grafana/grafana
```

Connect both containers to a shared Docker network so Grafana can reach Prometheus by name:

```
docker network create monitoring
docker network connect monitoring prometheus
docker network connect monitoring grafana
```

Verify both are healthy:

```
docker ps
curl -s localhost:9090/-/healthy
curl -s localhost:3000/api/health
```

Expected responses:

- `Prometheus Server is Healthy.`
- `{“database”:“ok”,…}`

Step 7: Allow Docker Bridge to Reach nv-monitor

Docker containers live in the `172.17.x.x` subnet. The host firewall must allow them to reach port 9101.

Note: The DGX Spark does not have UFW installed. Use iptables directly:

```
sudo iptables -I INPUT -s 172.17.0.0/16 -p tcp --dport 9101 -j ACCEPT
```

This is the critical rule that allows Prometheus (running in Docker) to scrape nv-monitor (running on

the host).

Verify the rule was added:

```
sudo iptables -L INPUT -n -v | grep 9101
```

You should see a line showing packets accepted from `172.17.0.0/16` on port `9101`.

Troubleshooting: Permission denied on iptables

If you see `Permission denied` or `Operation not permitted`:

Check sudo access:

```
sudo -l
```

Look for `iptables` in the allowed commands list. If it is not there, contact your sysadmin.

Check if iptables is available:

```
which iptables
iptables --version
```

If the binary is missing: `sudo apt install iptables -y`

Verify the Docker bridge gateway IP: The default Docker bridge is `172.17.0.0/16`, but your system may differ:

```
docker network inspect bridge | grep Gateway
```

Replace `172.17.0.0/16` in the iptables command with the actual subnet if it differs.

Note on SUDO POLICY VIOLATION broadcast messages

The Spark has a sysadmin audit policy that broadcasts a message to all terminals when sudo is used. The command still executes — this is just a notification to the admin team. It is not an error.

Step 8: Access UIs from Your Mac via SSH Tunnel

SSH port forwarding is the recommended way to access the Grafana and Prometheus UIs from your Mac. It is simpler and more secure than opening firewall ports, and works over Tailscale.

On your **Mac**, open a **new local terminal** (not an SSH session to the Spark — the prompt must show your Mac hostname):

```
ssh -L 9090:localhost:9090 -L 3000:localhost:3000
YOUR_USERNAME@YOUR_SPARK_IP
```

Keep this terminal open. Then open in your Mac browser:

- **Prometheus:** http://localhost:9090/targets
- **Grafana:** http://localhost:3000

Common mistake — running the tunnel from inside the Spark

If you run the SSH tunnel command from a terminal that is already SSH'd into the Spark, it will SSH back to itself and fail with “Address already in use” — because ports 9090 and 3000 are already bound by the Docker containers on the Spark. Always run the tunnel from a Mac local terminal.

Why SSH tunneling?

- Works over Tailscale without needing to open additional firewall ports
- Traffic is encrypted by default
- Easy to disconnect by closing the terminal

Step 9: Verify Prometheus is Scraping

Open `http://localhost:9090/targets` in your browser.

You should see the **nv-monitor** job listed with:

- State: **UP** (green)
- Scrape duration: under 10ms (typically ~2ms)

If the state shows DOWN, see the Troubleshooting section.

Step 10: Configure Grafana

Open `http://localhost:3000` in your browser.

- Login: **admin** / **admin**
- Set a new password when prompted

Add Prometheus as a data source

1. Click **Connections** in the left sidebar
2. Click **Data sources**
3. Click **Add data source**
4. Select **Prometheus**
5. Set URL to: `http://prometheus:9090`
6. Click **Save & test**
7. You should see: **Successfully queried the Prometheus API**

Why ''http://prometheus:9090'' works

Both containers are on the same Docker network (`monitoring`). Docker provides DNS resolution between containers on the same network, so `prometheus` resolves to the Prometheus container's IP automatically. Using `localhost:9090` here would not work — it would refer to the Grafana container itself.

Step 11: Build the Dashboard

All steps below are performed in the **Grafana web UI** at `http://localhost:3000` in your Mac's browser (via the SSH tunnel opened in Step 8).

1. Click **Dashboards → New → New dashboard**
2. Click **+ Add visualization**
3. Add each panel below one at a time
4. For each panel: select the metric in the Builder tab, set the title in the right panel options, confirm the visualization type, then click **Back to dashboard**

Dashboard panels

- **CPU Usage %** — metric: `nv_cpu_usage_percent` — type: Time series
- **CPU Temperature** — metric: `nv_cpu_temperature_celsius` — type: Time series
- **GPU Utilization %** — metric: `nv_gpu_utilization_percent` — type: Time series
- **GPU Power (W)** — metric: `nv_gpu_power_watts` — type: Time series
- **GPU Temperature** — metric: `nv_gpu_temperature_celsius` — type: Time series
- **Memory Used** — metric: `nv_memory_used_bytes` — type: Gauge — unit: bytes (SI)

Save the dashboard. Set auto-refresh to **10s** using the dropdown next to the Refresh button.

Important: select the correct data source when adding panels

When adding each panel, confirm the Data source dropdown shows the Prometheus data source you configured (not the default placeholder). If a panel shows “No data”, check this first.

Panel shows No data

1. Change the time range to **Last 5 minutes** and click **Run queries**
2. If still no data, click **Code** in the query editor and type the metric name directly, then run queries
3. The GPU utilization panel will show a flat 0% line at idle — that is correct, not missing data

Step 12: Load Test with demo-load

`demo-load` is included in the nv-monitor repo and already built by `make` in Step 3.

```
cd ~/nv-monitor
./demo-load --gpu
```

Expected output:

```
Starting CPU load on 20 cores (sinusoidal, phased)
Starting GPU load on 1 GPU (sinusoidal)
Will stop in 5m 0s (Ctrl+C to stop early)
GPU 0: calibrating... done
GPU 0: load active
```

This generates sinusoidal CPU and GPU load simultaneously for 5 minutes. Watch the Grafana dashboard — you should see all panels spike within a few seconds:

- GPU Power: rises from ~4.5W idle to ~12W under load
- CPU Usage %: cores hitting 80–100%
- GPU Utilization: rises from 0%
- CPU Temperature: climbs from ~45°C to ~70°C+

Press **Ctrl+C** to stop early, or wait 5 minutes for it to finish automatically.

Reconnecting After Reboot or Disconnect

nv-monitor and Docker containers do not auto-restart. To bring everything back:

On the Spark:

```
cd ~/nv-monitor
./nv-monitor -n -p 9101 -t YOUR_SECRET_TOKEN &
docker start prometheus grafana
```

On your Mac (new local terminal):

```
ssh -L 9090:localhost:9090 -L 3000:localhost:3000
YOUR_USERNAME@YOUR_SPARK_IP
```

Then open http://localhost:3000.

Tearing Everything Down

```
docker stop prometheus grafana
docker rm prometheus grafana
docker network rm monitoring
pkill nv-monitor
rm -rf ~/monitoring
```

Troubleshooting

nv-monitor binary does not exist after git clone

A file or directory named `nv-monitor` already existed in the home directory before cloning.

```
rm -rf ~/nv-monitor
git clone https://github.com/wentbackward/nv-monitor
cd nv-monitor
make
```

Prometheus target shows DOWN — context deadline exceeded

Apply both fixes:

Fix 1 — Use the correct target IP in `prometheus.yml`. The target must be the Docker bridge gateway, not localhost:

```
targets: ['172.17.0.1:9101']
```

Find the correct gateway IP with: `docker network inspect bridge | grep Gateway`

Then restart Prometheus: `docker restart prometheus`

Fix 2 — Allow Docker bridge through the firewall:

```
sudo iptables -I INPUT -s 172.17.0.0/16 -p tcp --dport 9101 -j ACCEPT
```

UFW command not found

The DGX Spark does not have UFW installed. Use iptables directly (see Step 7).

SSH tunnel fails with "Address already in use"

You ran the tunnel command from inside an existing SSH session to the Spark. The Spark already has Docker containers binding ports 9090 and 3000. Open a new terminal on your Mac (prompt must show your Mac hostname, not the Spark) and run the tunnel from there.

Grafana cannot connect to Prometheus — "lookup prometheus: no such host"

The containers are not on the same Docker network. Run:

```
docker network create monitoring
docker network connect monitoring prometheus
docker network connect monitoring grafana
```

Then set the Grafana data source URL to `http://prometheus:9090`.

Browser shows ERR_CONNECTION_RESET for port 9090 or 3000

Docker's iptables rules can bypass UFW, making direct browser access unreliable. Use SSH tunneling instead (see Step 8).

Grafana panel shows No data

```
1. Check the Data source dropdown — must point to your configured Prometheus
data source
2. Change time range to **Last 5 minutes** and click **Run queries**
3. Switch to **Code** mode and type the metric name directly
4. GPU utilization showing 0% at idle is correct — not an error
```

Memory Used shows a raw number like 4003753984

No unit is set on the panel. Edit the panel → Standard options → Unit → select **bytes (SI)**.

SUDO POLICY VIOLATION broadcast messages

This is a sysadmin audit policy. The command still executes — the broadcast is just a notification to the admin team. It is not an error.

AI Home