First Edition · July 2026

From Box to Cluster: **Building a Personal AI Supercomputer**

with NVIDIA DGX Spark Bundle — from bare hardware to a production inference cluster running large language models on Kubernetes

Mohinish Shaikh & **Sanwi Sarode**

8	256 GB	2×	7
CHAPTERS	COMBINED GPU MEMORY	GB10 BLACKWELL GPUS	STACK COMPONENTS

FRONT MATTER

Preface

Why We Wrote This Book

In early 2026, NVIDIA released the DGX Spark — a personal AI supercomputer the size of a Mac mini, packing a Grace Blackwell GB10 GPU with 128GB of unified memory into a single quiet desktop unit. For the first time, running frontier-scale AI workloads from a home lab or a startup office became genuinely practical.

We bought two of them.

What followed was weeks of hard-won lessons: first-boot quirks, ARM64 compatibility landmines, networking configurations that broke SSH, container images that silently ran on the wrong architecture, and tensor parallelism tuning that only worked after we understood the interplay between NCCL, Ray, and the k3s networking layer. None of this was fully documented in a single place.

This book is the documentation we wish we had when we started.

What This Book Covers

- **Hardware setup** — physical connections, first-boot wizard, static IP, SSH, Docker
- **System updates** — OS, CUDA 13.0, and driver updates via the DGX Dashboard
- **Kubernetes** — k3s, GPU Operator, Helm, namespace architecture
- **Distributed inference** — KubeRay, cross-node Ray cluster, ARM64 gotchas
- **vLLM** — tensor parallelism across both nodes, HuggingFace secrets
- **AIBrix** — AI gateway for routing, multi-tenancy, agent lifecycle
- **Monitoring** — Prometheus, Grafana, GPU metrics via DCGM Exporter

Who This Book Is For

This book assumes you are comfortable with a Linux terminal, have basic Kubernetes familiarity, and have purchased or are evaluating the NVIDIA DGX Spark Bundle. You do not need to be a DevOps engineer or CUDA expert — we explain every decision and command, including why certain approaches were tried and abandoned.

> **NOTE ON ARM64**
>
> The DGX Spark runs on ARM64 (Grace CPU). Many popular container images — including the standard Ray image — are x86 only and will silently fail on ARM64. We call these out explicitly throughout the book and always provide the correct ARM64-native alternative.

— Mohinish Shaikh & Sanwi Sarode, July 2026

FRONT MATTER

Table of Contents

Part I — Node Setup

Part II — Model Serving

Back Matter

SECTION	CONTENTS
Command Cheatsheet	Quick-reference kubectl, vLLM, port-forward, iptables commands
Troubleshooting	SSH loss, GPU visibility, ARM64 errors, model download failures
FAQ	Common questions about DGX Spark, models, and cluster setup
About the Authors	Mohinish Shaikh and Sanwi Sarode

CHAPTER 1 · PART I

Introduction: What You Are Building

The DGX Spark Bundle

The NVIDIA DGX Spark Bundle is two DGX Spark personal AI supercomputers sold together with everything needed to link them into a single distributed cluster.

RESOURCE	PER SPARK	COMBINED
GPU	1× NVIDIA GB10 Blackwell	2× GPUs
GPU Memory	128GB unified	256GB unified
CPU Cores	20 (ARM64)	40 cores
Interconnect	—	ConnectX-7 RDMA (QSFP)
OS	DGX OS (Ubuntu 24.04.4 LTS)	

Software Stack

DGX OS

Ubuntu 24.04.4 LTS with NVIDIA-optimized kernel and CUDA 13.0

`Kernel 6.17`

k3s

Lightweight Kubernetes — full API, minimal footprint, containerd runtime

`v1.35.5`

GPU Operator

Automatically deploys device plugin, DCGM exporter, container toolkit

`NVIDIA Helm`

KubeRay

Kubernetes operator managing the distributed Ray cluster lifecycle

`Ray 2.49.2`

🤖

vLLM

High-performance LLM inference — PagedAttention, tensor parallelism, OpenAI API

`0.10.1.1`

🔀

AIBrix

AI inference gateway — routing, multi-tenancy, agent lifecycle management

`v0.6.0`

Cluster Architecture

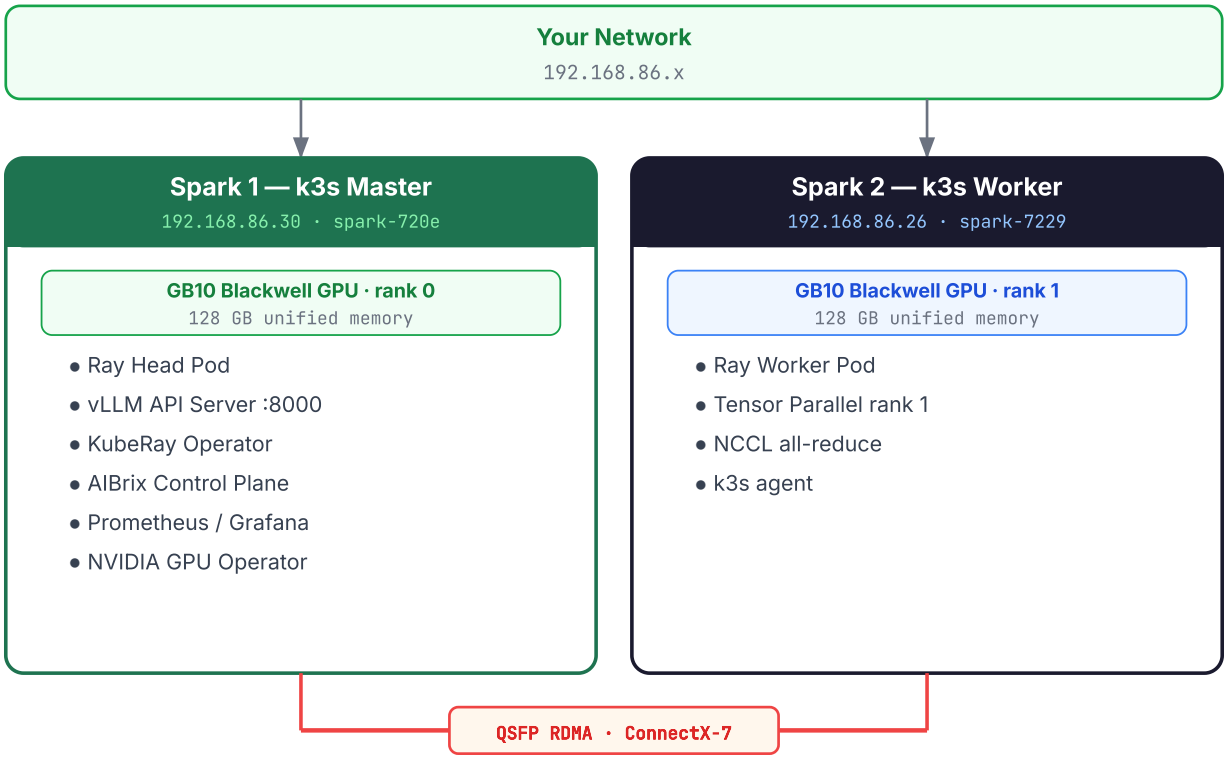

Figure 1.1 — DGX Spark Bundle: two-node cluster topology

Time Estimates

CHAPTER	ESTIMATED TIME
Hardware Setup	60–90 min
CUDA Updates	30–60 min
k3s + GPU Operator	20–30 min

CHAPTER	ESTIMATED TIME
KubeRay	15–20 min
vLLM (incl. model download)	30–60 min
AIBrix	10–15 min
Monitoring	20–30 min
Total	**3–5 hours**

CHAPTER 2 · PART I

Hardware Setup and First Boot

Hardware Specifications

COMPONENT	SPECIFICATION
GPU	NVIDIA GB10 Blackwell (Grace Blackwell Superchip)
GPU Memory	128GB unified (shared with CPU)
CPU	20-core ARM64 (Grace CPU)
Storage	NVMe SSD
Network	ConnectX-7 (QSFP, RDMA-capable)
OS	DGX OS (Ubuntu 24.04.4 LTS)
Kernel	6.17.0-1018-nvidia

Physical Setup

1. Connect both Sparks to power — they power on automatically when plugged in
2. Connect a monitor to **Spark 1 only** via HDMI
3. Connect a USB-C hub to Spark 1 and attach keyboard + mouse
4. Connect both Sparks to your router via Ethernet
5. Connect the QSFP cable between the ConnectX-7 ports on each unit

> **USB-C NOTE**
>
> The DGX Spark has USB-C ports only. A USB-C to USB-A hub is required for standard keyboards and mice.

First Boot Setup Wizard

1. Select keyboard layout
2. Accept the license agreement
3. Create your user account (username: `moonlit` used in this guide)
4. Connect to network (Ethernet recommended)

5. System downloads and installs complete software image automatically

> **WARNING**
>
> Do **not** power off during the software download step. The download cannot be resumed if interrupted. Allow 15–45 minutes.

Static IP Assignment

Configure static IPs on both Sparks before any Kubernetes setup. Dynamic IPs will break the cluster between sessions.

Spark 1 (master): `192.168.86.30` | **Spark 2 (worker):** `192.168.86.26`

On **Spark 1**:

```
sudo nmcli con mod "$(nmcli -g NAME con show --active | head -1)" \
  ipv4.addresses 192.168.86.30/24 \
  ipv4.gateway 192.168.86.1 \
  ipv4.dns 8.8.8.8 \
  ipv4.method manual
sudo nmcli con up "$(nmcli -g NAME con show --active | head -1)"
```

On **Spark 2**:

```
sudo nmcli con mod "$(nmcli -g NAME con show --active | head -1)" \
  ipv4.addresses 192.168.86.26/24 \
  ipv4.gateway 192.168.86.1 \
  ipv4.dns 8.8.8.8 \
  ipv4.method manual
sudo nmcli con up "$(nmcli -g NAME con show --active | head -1)"
```

Persistent iptables Rules

Add these on **both Sparks** before k3s installation to prevent SSH from being blocked after k3s restarts:

```
sudo iptables -I INPUT -s 192.168.86.0/24 -j ACCEPT
sudo iptables -I FORWARD -s 192.168.86.0/24 -j ACCEPT
sudo apt install iptables-persistent -y
sudo netfilter-persistent save
```

SSH Setup on Spark 2

```
# On Spark 2 — switch from socket-activated to persistent SSH
sudo systemctl stop ssh.socket
sudo systemctl disable ssh.socket
sudo systemctl enable ssh
sudo systemctl start ssh
```

Enable password auth in `/etc/ssh/sshd_config`: set `PasswordAuthentication yes`, then `sudo systemctl restart ssh`.

```
# From Spark 1 — verify connection
ssh moonlit@192.168.86.26
```

Docker Configuration

Run on **both Sparks** — configures cgroup v2 mode and NVIDIA runtime:

```
sudo python3 -c "
import json, os
path = '/etc/docker/daemon.json'
d = json.load(open(path)) if os.path.exists(path) else {}
d['default-cgroupns-mode'] = 'host'
d['default-runtime'] = 'nvidia'
d['runtimes'] = {'nvidia': {'path': 'nvidia-container-runtime', 'args': []}}
json.dump(d, open(path, 'w'), indent=2)
"
sudo systemctl restart docker
sudo usermod -aG docker $USER
newgrp docker
```

CHAPTER 3 · PART I

CUDA and System Updates

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel computing platform and programming model that gives software direct access to GPU cores. DGX Spark ships with a specific CUDA toolkit version — keeping it current ensures compatibility with the latest model-serving frameworks like vLLM and PyTorch, and unlocks performance improvements tied to newer driver releases.

Method 1 — DGX Dashboard (Recommended)

1. Find `dgx-spark-dashboard.desktop` on the desktop
2. Right-click → **Allow Launching** (first time only)
3. Double-click to open, or navigate to `http://localhost`
4. Log in → click **"Update Now"**
5. Allow system to reboot, then check for updates again — repeat until no more updates appear

> **NORMAL BEHAVIOR**
>
> Seeing "Update Available" multiple times is expected. Some updates only become visible after previous ones are installed. Keep clicking until the dashboard shows no pending updates.

Method 2 — Terminal

Run on both Sparks (directly or via SSH for Spark 2):

```
sudo apt update
sudo apt dist-upgrade -y
sudo fwupdmgr refresh
sudo fwupdmgr upgrade
sudo reboot
```

Expected Versions After Update

COMPONENT	VERSION
OS	Ubuntu 24.04.4 LTS
Kernel	6.17.0-1018-nvidia
CUDA Driver	580.159.03
CUDA Runtime	13.0

```
# Verify
nvidia-smi --query-gpu=driver_version --format=csv,noheader
# Expected: 580.159.03

nvcc --version
# Expected: release 13.0
```

CHAPTER 4 · PART I

Kubernetes Cluster with k3s

k3s is a lightweight, production-grade Kubernetes distribution by Rancher (SUSE). It packages the full Kubernetes control plane into a single binary under 100 MB, stripping components unnecessary for edge and small-cluster deployments. k3s is ideal for DGX

Spark Bundle: you get complete Kubernetes API compatibility with minimal system overhead, leaving the bulk of CPU and memory free for GPU workloads.

Install k3s on Spark 1 (Master)

```
curl -sfL https://get.k3s.io | \
  INSTALL_K3S_EXEC="--write-kubeconfig-mode 644 --disable=traefik" \
  sh -
```

Configure kubectl

```
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml
echo "export KUBECONFIG=/etc/rancher/k3s/k3s.yaml" >> ~/.bashrc
```

Get the Join Token

```
sudo cat /var/lib/rancher/k3s/server/node-token
```

Join Spark 2 as Worker

```
curl -sfL https://get.k3s.io | \
  K3S_URL=https://192.168.86.30:6443 \
  K3S_TOKEN=<TOKEN_FROM_SPARK1> \
  sh -
```

Assign Worker Role Label

```
kubectl label node spark-7229 node-role.kubernetes.io/worker=worker
```

Verify Cluster

```
kubectl get nodes
# Expected:
# NAME         STATUS   ROLES                  VERSION
# spark-720e   Ready    control-plane,master   v1.35.5+k3s1
# spark-7229   Ready    worker                 v1.35.5+k3s1
```

Install Helm

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 |
bash
helm version
```

NVIDIA GPU Operator

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia
helm repo update

helm install gpu-operator nvidia/gpu-operator \
  --namespace gpu-operator \
  --create-namespace
```

```
# Verify both GPUs visible
kubectl get nodes -o json | grep '"nvidia.com/gpu":'
# Expected: "nvidia.com/gpu": "1"  (one per node)

kubectl get nodes -o json | grep "nvidia.com/gpu.family"
# Expected: "nvidia.com/gpu.family": "blackwell"
```

Create Namespaces

```
kubectl create namespace core-services
kubectl create namespace monitoring
```

CHAPTER 5 · PART II

KubeRay for Distributed AI

KubeRay is the Kubernetes operator for Ray, an open-source framework for distributed Python workloads. It manages the lifecycle of RayClusters — head nodes, worker nodes, and autoscaling — as native Kubernetes resources. On DGX Spark Bundle, KubeRay bridges both nodes into a single Ray cluster so vLLM can split a large model across both GB10 GPUs using tensor parallelism.

CRITICAL — ARM64 ARCHITECTURE WARNING

The standard Ray image (`rayproject/ray`) is **x86-only** and will fail on DGX Spark with "exec format error". Always use:

```
nvcr.io/nvidia/vllm:25.09-py3
```

This image is ARM64-native with Ray 2.49.2, vLLM 0.10.1.1, and CUDA 13.0 built in.

Install KubeRay Operator

```
helm repo add kuberay https://ray-project.github.io/kuberay-helm/
helm repo update

helm install kuberay-operator kuberay/kuberay-operator \
  --namespace kuberay-system \
  --create-namespace \
  --set nodeSelector."kubernetes\\.io/hostname"=spark-720e
```

```
kubectl get pods -n kuberay-system
# kuberay-operator pod should show Running
```

NGC Registry Login

```
# On both Sparks
docker login nvcr.io
# Username: $oauthtoken
# Password: <YOUR_NGC_API_KEY>
```

HuggingFace Token Secret

```
kubectl create secret generic hf-token \
  --from-literal=token=$HF_TOKEN \
  -n core-services
```

Validate Cross-Node Networking

```
kubectl run test-spark1 \
  --image=busybox \
  --overrides='{"spec":{"nodeSelector":{"kubernetes.io/hostname":"spark-
720e"}}}' \
  --command -- sleep 3600

kubectl run test-spark2 \
  --image=busybox \
  --overrides='{"spec":{"nodeSelector":{"kubernetes.io/hostname":"spark-
7229"}}}' \
  --command -- sleep 3600

kubectl get pods -o wide  # get SPARK2_POD_IP
kubectl exec test-spark1 -- ping -c 4 <SPARK2_POD_IP>
kubectl delete pod test-spark1 test-spark2
```

Deploy RayCluster

```
kubectl apply -f - <<'EOF'
apiVersion: ray.io/v1
kind: RayCluster
metadata:
  name: vllm-cluster
  namespace: core-services
spec:
  rayVersion: '2.49.2'
  headGroupSpec:
    rayStartParams:
      dashboard-host: '0.0.0.0'
      num-gpus: '1'
    template:
      spec:
        nodeSelector:
          kubernetes.io/hostname: spark-720e
```

```
        containers:
        - name: ray-head
          image: nvcr.io/nvidia/vllm:25.09-py3
          command: ["/bin/bash", "-c"]
          args:
          - |
            ray start --head \
              --dashboard-host=0.0.0.0 \
              --num-gpus=1 \
              --block &
            sleep 30
            python3 -m vllm.entrypoints.openai.api_server \
              --model Qwen/Qwen2.5-7B-Instruct \
              --tensor-parallel-size 2 \
              --distributed-executor-backend ray \
              --host 0.0.0.0 \
              --port 8000 \
              --gpu-memory-utilization 0.85 \
              --max-num-seqs 4
          env:
          - name: HF_TOKEN
            valueFrom:
              secretKeyRef:
                name: hf-token
                key: token
          resources:
            limits:
              nvidia.com/gpu: "1"
              memory: "100Gi"
  workerGroupSpecs:
  - replicas: 1
    groupName: worker-group
    rayStartParams:
      num-gpus: '1'
    template:
      spec:
        nodeSelector:
          kubernetes.io/hostname: spark-7229
        containers:
        - name: ray-worker
          image: nvcr.io/nvidia/vllm:25.09-py3
```

```
          command: ["/bin/bash", "-c"]
          args:
          - |
            ray start \
              --address=vllm-cluster-head-svc.core-
services.svc.cluster.local:6379 \
              --num-gpus=1 \
              --block
          resources:
            limits:
              nvidia.com/gpu: "1"
              memory: "100Gi"
EOF
```

Verify Ray Cluster

```
HEAD=$(kubectl get pods -n core-services -l ray.io/node-type=head -o name |
head -1)
kubectl exec -n core-services $HEAD -- python3 -c \
  "import ray; ray.init(); print(ray.cluster_resources())"
# Expected: {'GPU': 2.0, 'CPU': 40.0, 'accelerator_type:GB10': 2.0}
```

CHAPTER 6 · PART II

vLLM Inference Engine

vLLM is a high-throughput inference engine for large language models, built around PagedAttention — a memory management technique that eliminates KV-cache fragmentation and dramatically increases GPU utilization. It exposes an OpenAI-compatible REST API and supports tensor parallelism across multiple GPUs, making it the model-serving engine of choice on the DGX Spark Bundle cluster.

Startup Time Expectations

SCENARIO	TIME
First startup (14GB model download)	~25 minutes
Subsequent startups (cached)	3–5 minutes

Key vLLM Configuration Flags

FLAG	VALUE	PURPOSE
`--tensor-parallel-size`	2	One rank per GPU/node
`--distributed-executor-backend`	ray	Use Ray for distribution
`--gpu-memory-utilization`	0.85	85% GPU for KV cache
`--max-num-seqs`	4	Max concurrent sequences

Monitor Startup

```
kubectl logs -n core-services \
  -l ray.io/node-type=head \
  --follow
# Wait for: "Application startup complete."
```

Verification

```
HEAD=$(kubectl get pods -n core-services -l ray.io/node-type=head -o name |
head -1)

# List models
```

```
kubectl exec -n core-services $HEAD -- \
  curl -s http://localhost:8000/v1/models

# Test chat completion
kubectl exec -n core-services $HEAD -- \
  curl -s http://localhost:8000/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "Qwen/Qwen2.5-7B-Instruct",
    "messages": [{"role": "user", "content": "What is tensor parallelism?"}],
    "max_tokens": 100
  }'
```

Models You Can Serve

MODEL	HUGGINGFACE ID	NOTES
Qwen2.5-7B (default)	`Qwen/Qwen2.5-7B-Instruct`	Fast, fits easily
Qwen2.5-72B	`Qwen/Qwen2.5-72B-Instruct`	Needs `--max-model-len`
Llama 3.3 70B	`meta-llama/Llama-3.3-70B-Instruct`	Needs HF access request
Mistral 7B	`mistralai/Mistral-7B-Instruct-v0.3`	Good alternative

CHAPTER 7 · PART II

AIBrix AI Gateway

AIBrix is an open-source AI inference gateway from the vLLM project. It sits in front of one or more vLLM endpoints and adds request routing, per-namespace quota enforcement, agent lifecycle management, and GPU utilization-aware scheduling — turning a raw vLLM API server into a multi-tenant inference platform.

What AIBrix Provides

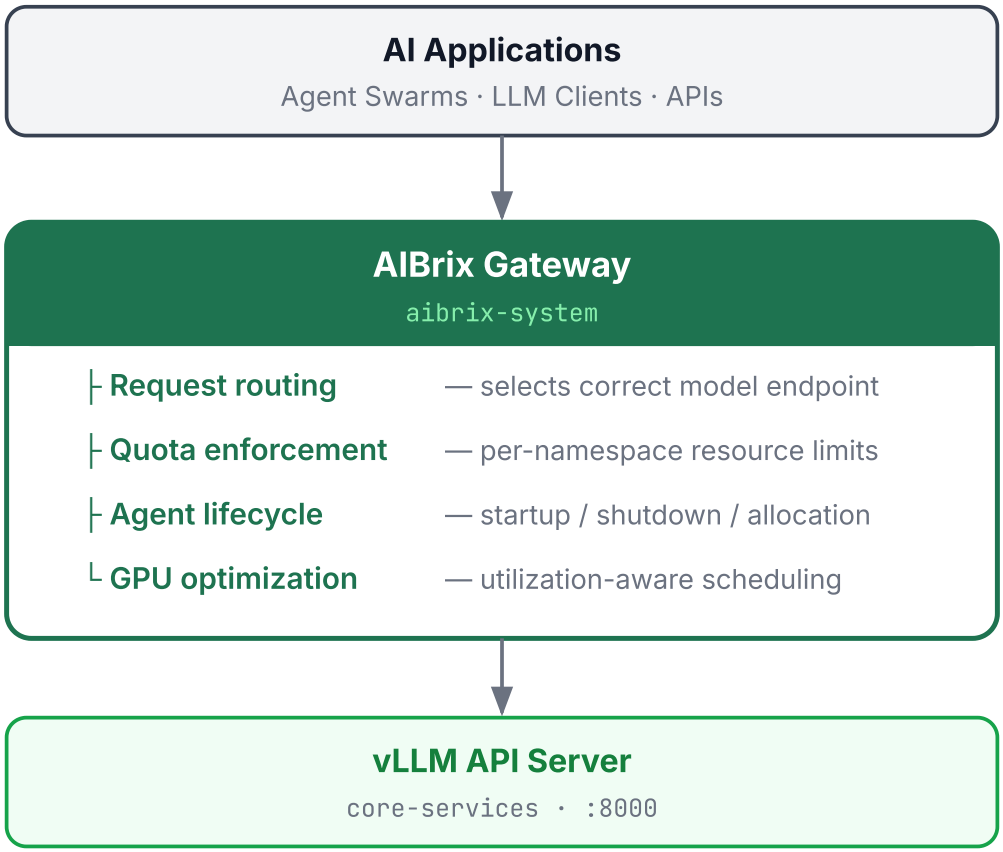

Figure 7.1 — AIBrix Gateway: request flow

Step 1 — Install Dependencies

```
kubectl apply \
  -f https://github.com/vllm-project/aibrix/releases/download/v0.6.0/aibrix-dependency-v0.6.0.yaml \
  --server-side \
  --force-conflicts
```

Step 2 — Install AIBrix Core

```
kubectl apply \
  -f https://github.com/vllm-project/aibrix/releases/download/v0.6.0/aibrix-core-v0.6.0.yaml
```

Verify Components

```
kubectl get pods -n aibrix-system
# All 6 pods should show Running:
# aibrix-controller-manager
# aibrix-gateway-plugins
# aibrix-gpu-optimizer
# aibrix-kuberay-operator
# aibrix-metadata-service
# aibrix-redis-master
```

Register a Model (ModelAdapter)

```
kubectl apply -f - <<'EOF'
apiVersion: model.aibrix.ai/v1alpha1
kind: ModelAdapter
metadata:
  name: qwen-7b
  namespace: <your-app-namespace>
spec:
  modelName: Qwen/Qwen2.5-7B-Instruct
  replicas: 1
  podSelector:
    matchLabels:
      ray.io/node-type: head
EOF
```

CHAPTER 8 · PART II

Cluster Overview and Monitoring

Prometheus is an open-source metrics collection and alerting system that scrapes time-series data from targets such as the NVIDIA GPU Operator and Kubernetes node exporters. Grafana is the visualization layer that queries Prometheus and renders real-time dashboards. Together they form the standard Kubernetes observability stack, deployed here via the `kube-prometheus-stack` Helm chart.

Install Monitoring Stack

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

helm install monitoring prometheus-community/kube-prometheus-stack \
  --namespace monitoring
```

Accessing Grafana

```
# Get admin password
kubectl --namespace monitoring get secrets monitoring-grafana \
  -o jsonpath="{.data.admin-password}" | base64 -d; echo

# Port-forward
kubectl --namespace monitoring port-forward \
  $(kubectl --namespace monitoring get pod \
    -l "app.kubernetes.io/name=grafana" -o name) 3000

# Open: http://localhost:3000  Login: admin / <password above>
```

GPU Metrics

METRIC	DESCRIPTION
`DCGM_FI_DEV_GPU_UTIL`	GPU compute utilization (%)
`DCGM_FI_DEV_FB_FREE`	Free GPU frame buffer (MiB)
`DCGM_FI_DEV_FB_USED`	Used GPU frame buffer (MiB)

METRIC	DESCRIPTION
`DCGM_FI_DEV_POWER_USAGE`	GPU power draw (Watts)
`DCGM_FI_DEV_GPU_TEMP`	GPU temperature (°C)

Full Cluster Status

LAYER	COMPONENT	STATUS
Hardware	2× DGX Spark (GB10, 128GB each)	Connected via QSFP
OS	DGX OS — Ubuntu 24.04.4 LTS	CUDA 13.0
Kubernetes	k3s v1.35.5 — 2 nodes Ready	Flannel CNI
GPU	NVIDIA GPU Operator	2 GPUs visible
Distributed	KubeRay + Ray 2.49.2	Head + Worker across nodes
Inference	vLLM 0.10.1.1	Serving on port 8000
Gateway	AIBrix v0.6.0	Routing + multi-tenancy
Monitoring	Prometheus + Grafana	GPU metrics via DCGM

BACK MATTER

Quick Reference — Command Cheatsheet

Cluster Health

```
kubectl get nodes
kubectl get pods -A | grep -v Running | grep -v Completed
kubectl top nodes
kubectl top pods -A --sort-by=memory
```

vLLM Testing

```
HEAD=$(kubectl get pods -n core-services -l ray.io/node-type=head -o name |
head -1)
kubectl exec -n core-services $HEAD -- curl -s http://localhost:8000/v1/models
kubectl logs -n core-services -l ray.io/node-type=head --tail=20
```

Port Forwards

```
# Grafana on :3000
kubectl -n monitoring port-forward svc/monitoring-grafana 3000:80

# Ray Dashboard on :8265
HEAD=$(kubectl get pods -n core-services -l ray.io/node-type=head -o name |
head -1)
kubectl port-forward -n core-services $HEAD 8265:8265
```

iptables Recovery

```
sudo iptables -F && sudo iptables -X
sudo iptables -P INPUT ACCEPT
sudo iptables -P FORWARD ACCEPT
sudo iptables -P OUTPUT ACCEPT
sudo netfilter-persistent save
```

BACK MATTER

Troubleshooting Index

SYMPTOM	LIKELY CAUSE	FIX
SSH to Spark 2 times out	iptables blocked after k3s restart	Flush iptables — see Ch 2
DNS not resolving	systemd-resolved cache stale	`sudo resolvectl flush-caches`
GPU not visible in k8s	GPU Operator pods not Running	Check `kubectl get pods -n gpu-operator`
Ray shows GPU: 1 not 2	Worker pod not connected	Check worker logs; verify DNS address
"exec format error"	x86 image on ARM64	Use `nvcr.io/nvidia/vllm:25.09-py3`
Model download fails	HF token missing/invalid	Check `kubectl get secret hf-token -n core-services`
AIBrix CrashLoopBackOff	Dependencies not installed first	Apply `aibrix-dependency` before `aibrix-core`
NGC image pull fails	containerd lacks NGC credentials	Create `docker-registry` Kubernetes secret — see Ch 5

BACK MATTER

Frequently Asked Questions

What is the NVIDIA DGX Spark Bundle?

The NVIDIA DGX Spark Bundle is two DGX Spark personal AI supercomputers sold together. Each unit has a Grace Blackwell GB10 GPU with 128GB of unified memory and a 20-core ARM64 CPU. Combined, the bundle provides 256GB of total GPU memory and 40 CPU cores, connected via a high-speed QSFP RDMA cable using ConnectX-7.

Which container image should I use for Ray on DGX Spark?

Use `nvcr.io/nvidia/vllm:25.09-py3` — not the standard `rayproject/ray` image, which is x86-only and will fail with "exec format error" on DGX Spark's ARM64 architecture. The NVIDIA vLLM image is ARM64-native and includes Ray 2.49.2, vLLM 0.10.1.1, and CUDA 13.0.

How long does vLLM take to start on DGX Spark?

First startup takes approximately 25 minutes: ~20 minutes to download the 7B model weights from HuggingFace, then ~5 minutes for model loading and tensor parallel initialization. Subsequent startups with a cached model take 3–5 minutes.

What models can I run on two DGX Spark units?

With 256GB of combined unified memory, you can run Qwen2.5-7B-Instruct, Qwen2.5-72B-Instruct, Llama 3.3 70B Instruct, Qwen3-235B-A22B-NVFP4 (quantized), Nemotron-

3-Super-120B, and Llama 405B in quantized form. vLLM distributes model weights across both GPUs using tensor parallelism with `--tensor-parallel-size 2`.

How do I fix SSH being blocked after k3s restarts?

k3s modifies iptables rules on restart. Flush them with:

```
sudo iptables -F && sudo iptables -X
sudo iptables -P INPUT ACCEPT
sudo iptables -P FORWARD ACCEPT
sudo iptables -P OUTPUT ACCEPT
```

To prevent this permanently, add `sudo iptables -I INPUT -s 192.168.86.0/24 -j ACCEPT` before k3s installation and save with `sudo netfilter-persistent save`.

What is AIBrix and why use it with vLLM?

AIBrix is an open-source AI inference gateway that provides request routing, multi-tenancy with namespace isolation, agent lifecycle management, and GPU utilization-aware scheduling. It allows multiple applications to safely share one vLLM instance with per-namespace quota enforcement and without exposing vLLM's internal service address to each application.

Why k3s instead of full Kubernetes?

k3s provides the full Kubernetes API in a single lightweight binary, eliminating the operational complexity of managing separate control-plane components and etcd. For a 2-node personal cluster, the tradeoff is that Flannel CNI (k3s default) does not support RDMA/RoCE, so NCCL falls back to TCP for cross-GPU communication. A future upgrade to RKE2 + Cilium will enable full RDMA support.

BACK MATTER

About the Authors

Mohinish Shaikh

Mohinish Shaikh is an AI infrastructure engineer and researcher focused on building high-performance AI systems on commodity and specialized hardware. He has worked across distributed systems, model serving infrastructure, and AI-native application design. His work on the DGX Spark Bundle project explores running production AI workloads on personal supercomputer hardware.

GitHub: github.com/mohnishbasha · LinkedIn: linkedin.com/in/mohinishbasha

Sanwi Sarode

Sanwi Sarode is an AI systems engineer specializing in distributed inference, Kubernetes-native AI infrastructure, and model optimization. Her work spans GPU cluster management, tensor parallelism tuning, and multi-tenant AI platform design. She co-authored this guide based on hands-on experience building and operating the DGX Spark cluster documented in these pages.

GitHub: github.com/sanwisarode · LinkedIn: linkedin.com/in/sanwi-sarode-785b0b282

Project Repository

Full configuration, automation scripts, and updates to this guide:

```
https://github.com/mohinishbasha/dgx-spark-bundle
```

License