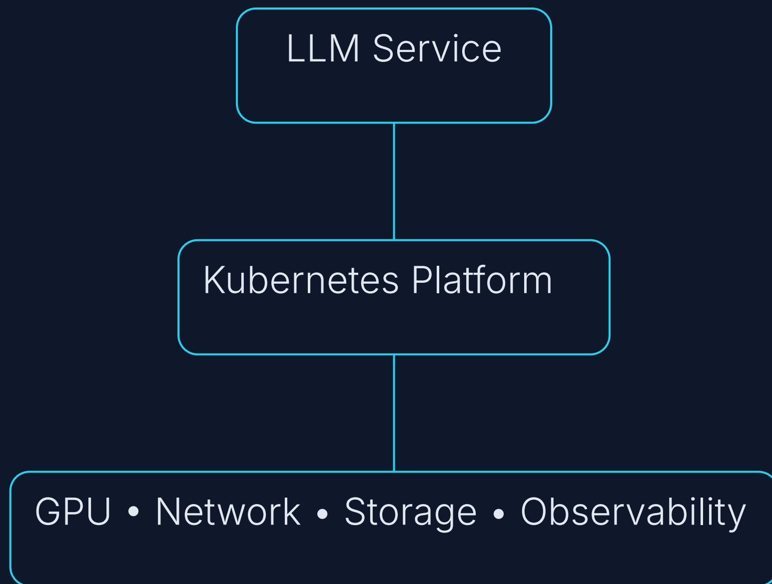
Building Reliable Infrastructure for LLM Workloads on Kubernetes

LLM reliability is not only about model quality.

It also depends on the platform under the model.

Networking, scheduling, failover, and observability before production



Sergey Speranskiy

Senior Infrastructure Platform Engineer

Why LLM workloads are different

Traditional Web Service

- CPU-based
- Short requests
- Easy horizontal scaling
- Restart is usually cheap
- Latency is more predictable

LLM Service

- GPU-based
- Long requests
- Limited GPU capacity
- Restart can be expensive
- Latency depends on tokens, queue, model size

What breaks when moving to production

Experiment

- Single node
- Manual deployment
- Default networking
- Best-effort capacity
- Basic logs

Production

- Multiple failure domains
- Automated rollout
- Observable network path
- Planned requests and limits
- Metrics, alerts, SLOs

Reference architecture

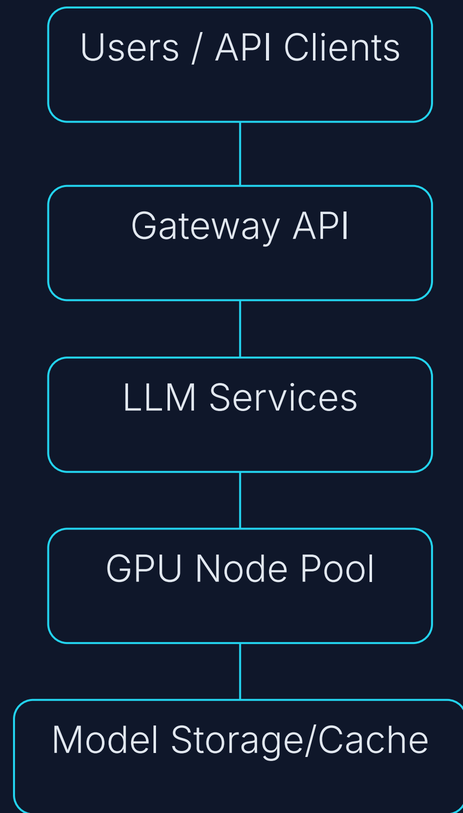
A reliable LLM platform is more than an inference container.

Core path:

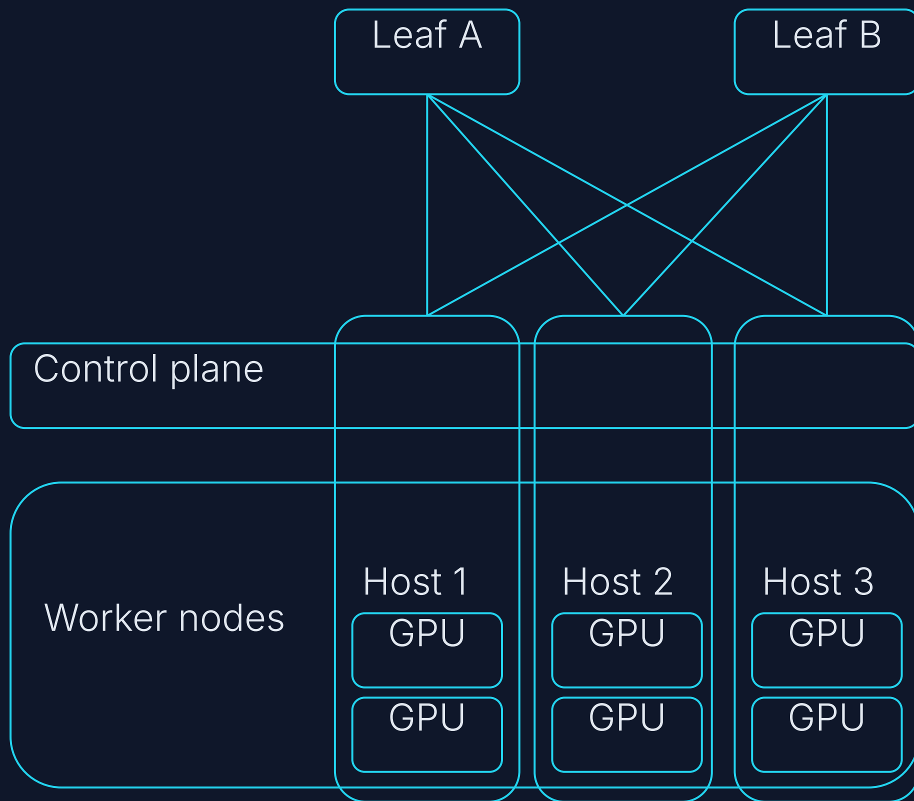
Users → Gateway API → LLM Services → GPU Nodes → Model Storage

Platform services:

Secrets • Observability • CI/CD • Control Plane



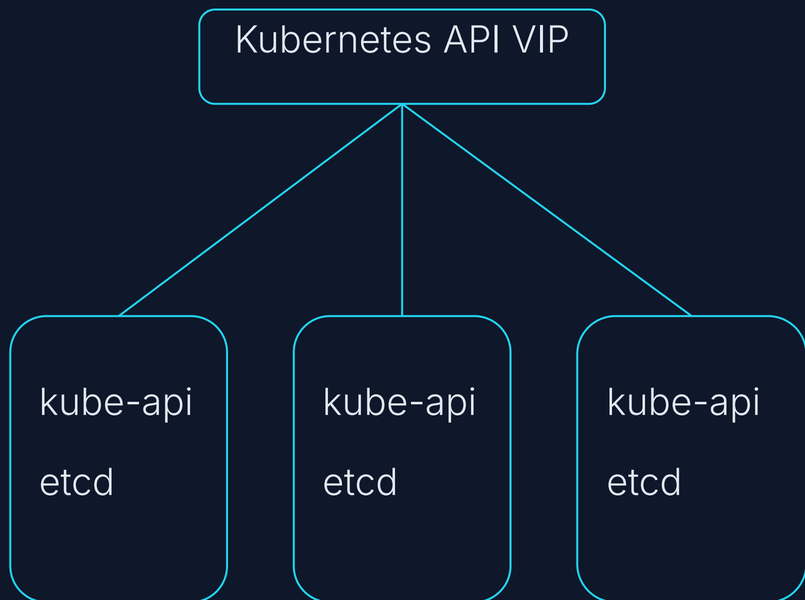
Cluster and failure domains



Design before failure happens:

- Separate control plane from workers
- Spread GPU capacity across nodes and racks
- Avoid one switch, one uplink, one API endpoint
- Keep enough spare capacity for failover

Control plane stability



Control plane stability

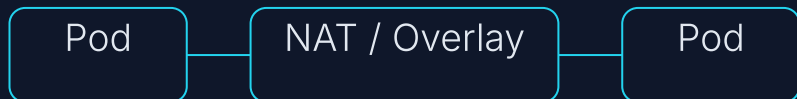
- etcd quorum across failure domains
- Highly available API endpoint
- Health checks before failover
- No dependency loop with networking

Routable networking

Why it matters:

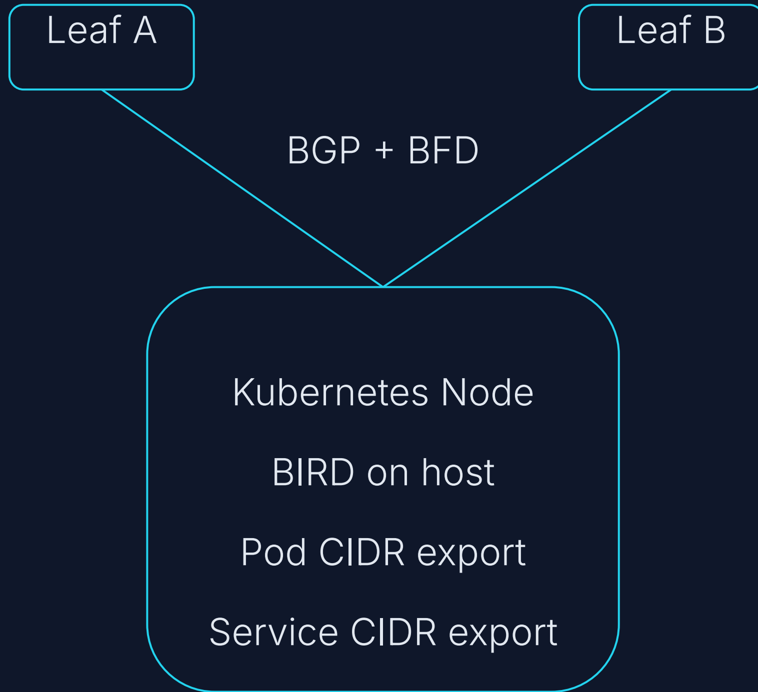
- Real pod IPs are visible end-to-end
- Less NAT and encapsulation
- Easier packet captures and troubleshooting
- Standard routing tools become useful

Bad visibility:



Good visibility:

BGP/ECMP/BFD design



- BIRD runs on the host
- Nodes advertise pod and service CIDRs with BGP
- ECMP uses both uplinks
- BFD removes failed paths quickly

GPU node operational checklist

GPU Node

NVIDIA Driver

Container Toolkit

Device Plugin

DCGM Exporter

Node Labels / Health

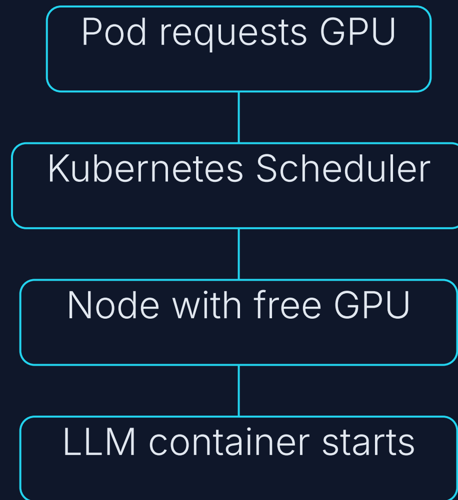
Before scheduling workloads:

- GPU visible on the host
- GPU visible to kubelet
- Pods can request nvidia.com/gpu
- Metrics are collected
- Node upgrades are planned

GPU Pod Scheduling

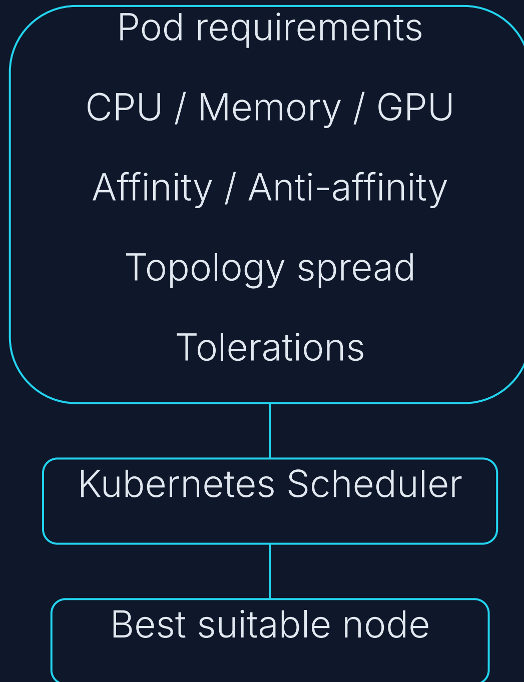
GPU Pod Scheduling

- GPUs are exposed as extended resources
- Pods request GPU with nvidia.com/gpu
- Scheduler places the Pod on a node with available GPU
- For inference: Deployment is usually enough
- For training / batch jobs: Job or queue-based scheduling




```yaml
resources:
  limits:
    nvidia.com/gpu: 1
```

Requests, affinity, topology, taints

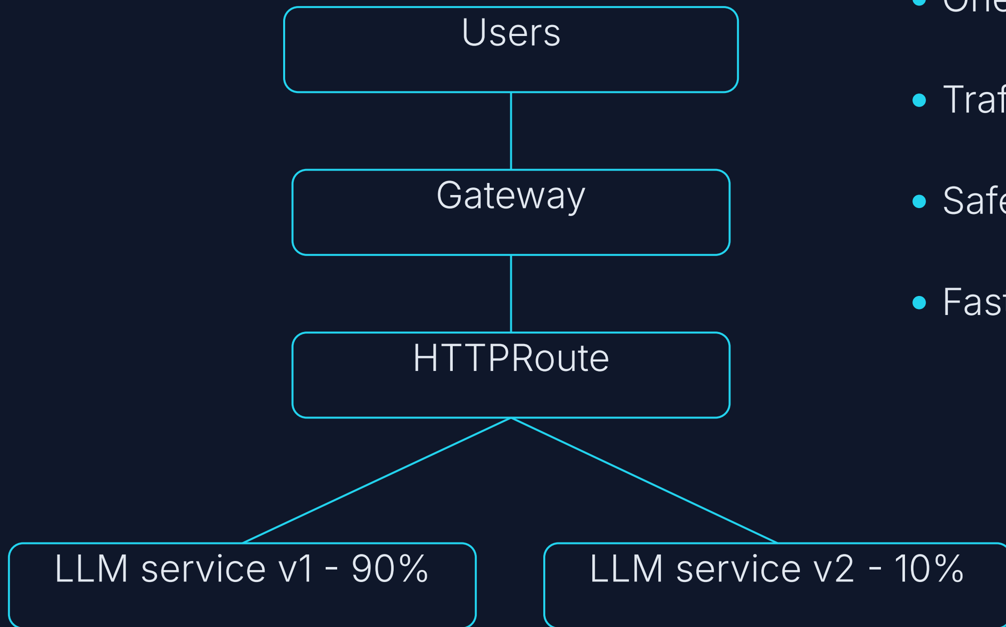


- Requests define required capacity
- Affinity selects the right node pool
- Anti-affinity spreads replicas
- Topology spread avoids single failure domains
- Taints protect GPU nodes from random workloads

```yaml
nodeSelector:
  accelerator: nvidia-gpu

tolerations:
  - key: "gpu"
    operator: "Equal"
    value: "true"
    effect: "NoSchedule"
```

Gateway API and canary



- One stable entry point for users
- Traffic split between model/service versions
- Safer rollout for new models or configs
- Fast rollback by changing routing weights

Canary = expose small traffic first, observe, then increase

Observability and final checklist

Measure:

- Application: latency, errors, queue, tokens/sec
- GPU: utilization, memory, temperature, errors
- Kubernetes: pending pods, restarts, scheduling failures
- Network: drops, retransmits, BGP/BFD state
- Control plane: API latency, etcd health

Final checklist:

- Visible network path
- Stable control plane
- Planned GPU scheduling
- Safe rollout strategy
- Metrics before production

The model is important, but the platform decides how it behaves under failure.