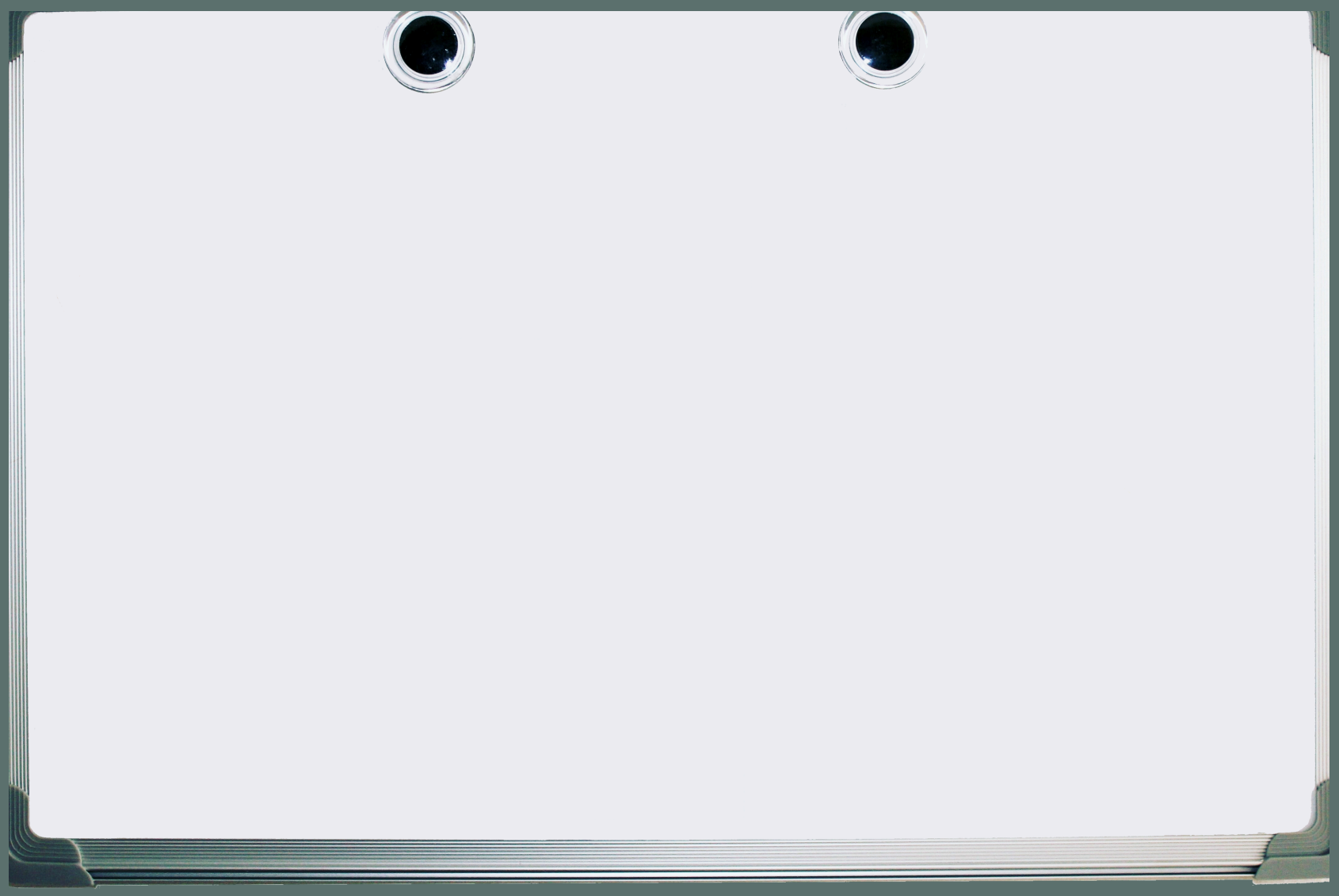
# 🎨 Systems Design: Kubernetes for ML Workloads



"Run distributed ML training on Kubernetes without burning your budget"

**Where do you start?**

Most people deploy to K8s and hope it works.

I start with resource constraints and GPU scheduling.

**Here's the thinking process: 👇**

# 📝 Step 1: Why K8s for ML is DIFFERENT

## ML workloads ≠ Web workloads

**Web services:**
→ Stateless (no data stored locally)
→ CPU-bound (handle requests)
→ Scale horizontally (add more pods)
→ Predictable resource usage

**ML workloads:**
→ Stateful (checkpoints, data, models)
→ GPU-bound (expensive, scarce)
→ Scale vertically (bigger GPUs, not more pods)
→ Unpredictable (training spikes, inference steady)

**Questions that determine everything:**
→ How do you schedule GPUs? (Not all nodes have them)
→ How do you handle multi-node training? (All-to-all communication)
→ How do you prevent OOM? (Models don't fit in memory)
→ How do you manage costs? ($$$)

**Wrong approach = treat ML like web services, waste money.**
**Right approach = design for GPU constraints, distributed training, and state.**

## 🔄 Step 2: Map the ML Workflow on K8s

**Data Prep → Training → Evaluation → Deployment → Inference**

Fill in the [?]:

→ Where does **training data** live? (PVCs, S3, NFS?)

→ How many **GPUs** per job? (Single GPU, multi-GPU, multi-node?)

→ How do you **orchestrate?** (Kubeflow, Argo, custom operators?)

→ Where do **models** go? (Registry, S3, PVCs?)

→ How do you **serve** models? (KServe, Seldon, TorchServe?)

**This is the skeleton.**
**Now add the K8s primitives that make ML work.**

# 🎮 Step 3: GPU Scheduling & Resource Management

**The Problem:**

→ Not all nodes have GPUs
→ GPUs are expensive (can't waste them)
→ Training jobs need exclusive GPU access
→ Some jobs need 1 GPU, some need 8, some need 64

**K8s GPU Scheduling:**

Node Labels:
→ Label GPU nodes: `nvidia.com/gpu=true`
→ Node selectors: Schedule GPU jobs ONLY on GPU nodes

**Resource Requests:**

→ Request GPUs: `nvidia.com/gpu: 1`
→ K8s reserves GPU for that pod
→ No oversubscription (1 GPU = 1 job)

**GPU Sharing (Advanced):**

→ Time-slicing (multiple jobs share 1 GPU)
→ MIG (Multi-Instance GPU - partition A100/H100)
→ Fractional GPUs (0.5 GPU per job)

**Taints & Tolerations:**

→ Taint GPU nodes (prevent non-GPU pods)
→ GPU jobs tolerate taint (only they can schedule)

**Attack:** Schedule training job, no GPU available, job pending forever
**Impact:** Wasted developer time, blocked experiments
**Defense:** Node selectors, resource requests, cluster autoscaling

**GPUs are your bottleneck. Schedule them correctly or waste money**

## 💾 Step 4: Storage Strategy for ML

**ML Storage Needs:**

→ Datasets (100GB - 10TB+)
→ Checkpoints (model state during training)
→ Models (weights, configs)
→ Logs (TensorBoard, metrics)

**K8s Storage Options:**

**Persistent Volumes (PVC):**

→ EBS/GCP Persistent Disk/Azure Disk
→ ReadWriteOnce (single pod access)
→ Good for: Checkpoints, model storage
→ Bad for: Multi-pod training (can't share)

**Network File System (NFS/EFS):**

→ ReadWriteMany (multiple pods access)
→ Good for: Shared datasets, distributed training
→ Bad for: Slower than local disk

**S3/GCS/Blob Storage:**

→ Infinite capacity, cheap
→ Good for: Raw data, model registry
→ Bad for: High-latency reads during training

**Strategy:**

1. Store raw data in S3 (cheap, durable)
2. Copy to NFS for training (fast access)
3. Save checkpoints to PVC (persistent)
4. Upload final model to S3 (registry)

**Local NVMe (Fast):**

→ Fastest option (direct attached storage)
→ Good for: Training data (copy from S3 first)
→ Bad for: Ephemeral (lost if pod dies)

**Attack:** Training job crashes, checkpoints on ephemeral storage, lost 3 days of training
**Impact:** Wasted compute ($10K+ GPU hours)
**Defense**: Persistent volumes for checkpoints, S3 for final artifacts

**Storage failures kill ML jobs. Plan for them.**

## Single GPU → Multi-GPU → Multi-Node

## 🌐 Step 5: Distributed Training on K8s

**Single GPU:**
→ Simple: 1 pod, 1 GPU
→ Limit: Model + data must fit in GPU memory

**Multi-GPU (Single Node):**
→ 1 pod, 8 GPUs (e.g., DGX A100)
→ Fast: GPUs communicate via NVLink
→ K8s: Request `nvidia.com/gpu: 8`

**Multi-Node (Distributed):**
→ Multiple pods, each with GPUs
→ Communicate over network (slower)
→ Requires: All-to-all communication

## Disributed Training Frameworks:

**PyTorch DDP (Distributed Data Parallel):**
→ Each pod gets copy of model
→ Sync gradients across pods
→ Requires: All pods can reach each other

**Kubeflow Training Operators:**
→ PyTorchJob, TFJob (K8s CRDs)
→ Handles pod creation, networking
→ Auto-configures MASTER/WORKER roles

**Horovod:**
→ MPI-based (Message Passing Interface)
→ Efficient gradient sync
→ Works with: PyTorch, TensorFlow

**Networking Requirements:**
→ All pods in same namespace (can communicate)
→ Fast network (RDMA, InfiniBand preferred)
→ Network policies (don't block training traffic)

**Attack:** Distributed training slow, network bottleneck, GPU utilization <50%
**Impact**: Wasted GPU time, slow experiments
**Defense**: Fast networking, proper pod placement, network monitorin

**Distributed training on K8s = networking problem. Get networking right or waste GPUs.**

# ⚠️ Step 6: Prevent Out-of-Memory Kills Part 1

## The OOM Problem:

Training job starts → Loads model → Loads batch → Runs forward pass → **OOM KILLED**

**Why?**

→ Model + activations + gradients + optimizer state > GPU memory
→ Or: CPU memory exhausted

## K8s Resource Management:

### Resource Requests (Guaranteed):

```yaml
resources:
 requests:
 memory: "64Gi"
 cpu: "16"
 nvidia.com/gpu: 1
```

→ K8s guarantees these resources
→ Pod won't schedule if node can't provide

### Resource Limits (Max):

```yaml
resources:
 limits:
 memory: "128Gi" # Can burst to 128GB
 cpu: "32"
```

→ Pod killed if exceeds limits
→ Prevents one job from starving others

## The OOM Problem:

## ⚠ Step 6: Prevent Out-of-Memory Kills Part 2

**Right-size resources:**
→ Profile training job first (how much memory?)
→ Set requests = typical usage
→ Set limits = peak usage + buffer

**Batch size tuning:**
→ Smaller batches = less memory
→ Gradient accumulation (simulate large batch)

**Mixed precision training:**
→ FP16 instead of FP32 (50% memory reduction)
→ Less memory, faster training

**Gradient checkpointing:**
→ Trade compute for memory
→ Recompute activations instead of storing

**Attack:** Training job OOM killed at **90%** completion, no checkpoint, lost 12 hours
**Impact:** Wasted compute, developer frustration
**Defense:** Resource limits, checkpointing, memory profiling

**OOM kills are the #1 ML job failure. Plan memory usage or lose work.**

# 🎛️ Step 7: Orchestrate ML Jobs Part 1

## Options for Running ML on K8s:

### Raw K8s Jobs:
→ `kubectl apply -f training-job.yaml`
→ Manual management
→ Good for: Simple, one-off jobs
→ Bad for: Distributed training, pipelines

### Kubeflow:
→ ML platform on K8s
→ Jupyter notebooks, pipelines, training operators
→ Good for: Full ML lifecycle
→ Bad for: Complex, heavy, overkill for small teams

### Argo Workflows:
→ DAG-based pipeline orchestration
→ Good for: ML pipelines (data → train → deploy)
→ Bad for: Doesn't handle GPU scheduling natively

### Custom Operators:
→ PyTorchJob, TFJob (Kubeflow operators)
→ Handle distributed training setup
→ Auto-configure MASTER/WORKER pods

# 🎛️ Step 7: Orchestrate ML Jobs Part 2

**Job Management:**

**Retry Logic:**
→ `restartPolicy: OnFailure` (retry if job fails)
→ `backoffLimit: 3` (max 3 retries)

**TTL After Finished:**
→ `ttlSecondsAfterFinished: 86400` (delete after 24h)
→ Prevents cluster clutter

**Priority Classes:**
→ High priority: Production inference
→ Low priority: Experimental training (preemptible)

**Preemption:**
→ High priority job can evict low priority
→ Save costs: Run experiments on preemptible nodes

**Attack:** Hundreds of completed jobs clog cluster, can't schedule new work
**Impact:** Cluster instability, quota exhaustion
**Defense:** TTL cleanup, resource quotas, namespaces

**Job lifecycle management = cluster health. Clean up or drown in zombie pods.**

💰 Step 9: Cost Control for K8s ML Part 1

## Where ML costs come from:

**GPUs: 90% of ML infrastructure spend**

→ A100 (80GB): $3-5/hour
→ H100: $8-10/hour
→ Left idle = money burning

## Cost Optimization Strategies:

**Spot/Preemptible Instances:**
→ 70-90% discount
→ Can be interrupted (use for experiments)
→ Not for: Production inference

**Cluster Autoscaling:**
→ Scale up when jobs pending
→ Scale down when idle
→ Don't pay for unused nodes

**Right-size GPU requests:**
→ Don't request 8 GPUs if you need 1
→ Fractional GPUs for small models

**Job prioritization:**
→ Production > experiments
→ Kill low-priority jobs if needed

**Scheduled scaling:**
→ Scale down at night/weekends
→ Scale up during work hours

**Local NVMe caching:**
→ Copy data once, reuse for multiple jobs
→ Don't re-download from S3 every time

**Cost Monitoring:**
→ Kubecost: Track spend per team, per job
→ Alert on runaway jobs
→ Chargeback to teams (incentivize efficiency)

**Attack:** Developer leaves training job running over weekend, $15K wasted
**Impact:** Budget overrun
**Defense:** Auto-shutdown, resource quotas, cost alerts

**K8s for ML without cost controls = budget disaster. Monitor spend or go bankrupt.**

✅ The Complete K8s for ML System

Data Layer (S3/NFS → PVCs)
↓
Training Jobs (PyTorchJob, GPU scheduling)
↓
Distributed Training (Multi-node, RDMA networking)
↓
Resource Management (Requests, limits, OOM prevention)
↓
Monitoring (Prometheus, Grafana, GPU metrics)
↓
Model Registry (S3, MLflow)
↓
Inference (KServe, autoscaling)
↓
Cost Control (Spot instances, autoscaling, quotas)

K8s for ML ≠ K8s for web services.
Treat them the same = waste money.

The difference: Systems thinking.

This is production K8s for ML.  Not "deploy and pray."  9 layers. Each optimized for ML workloads.