AYI NEDJIMI Consultants

Kubernetes pour l'IA : GPU Scheduling, Serving et 2026

Catégorie : Intelligence Artificielle Lecture : 18 min Publié le : 13/02/2026 Auteur : Ayi NEDJIMI

Guide complet Kubernetes pour l'IA : GPU scheduling avancé, MIG, time-slicing, model serving avec vLLM et Triton,. Thèmes : kubernetes IA, kubernetes.

Kubernetes pour l'IA : GPU Scheduling, Serving et 2026 constitue un enjeu majeur pour les professionnels de la sécurité informatique et les équipes techniques. Ce guide détaillé sur ia kubernetes gpu scheduling serving propose une méthodologie structurée, des outils éprouvés et des recommandations opérationnelles directement applicables. L'objectif est de fournir aux praticiens — consultants, ingénieurs sécurité, administrateurs systèmes — les connaissances et les techniques nécessaires pour aborder ce sujet avec rigueur. Chaque section s'appuie sur des retours d'expérience terrain et intègre les évolutions les plus récentes du domaine. Les recommandations présentées sont adaptées aux environnements d'entreprise et tiennent compte des contraintes opérationnelles réelles.

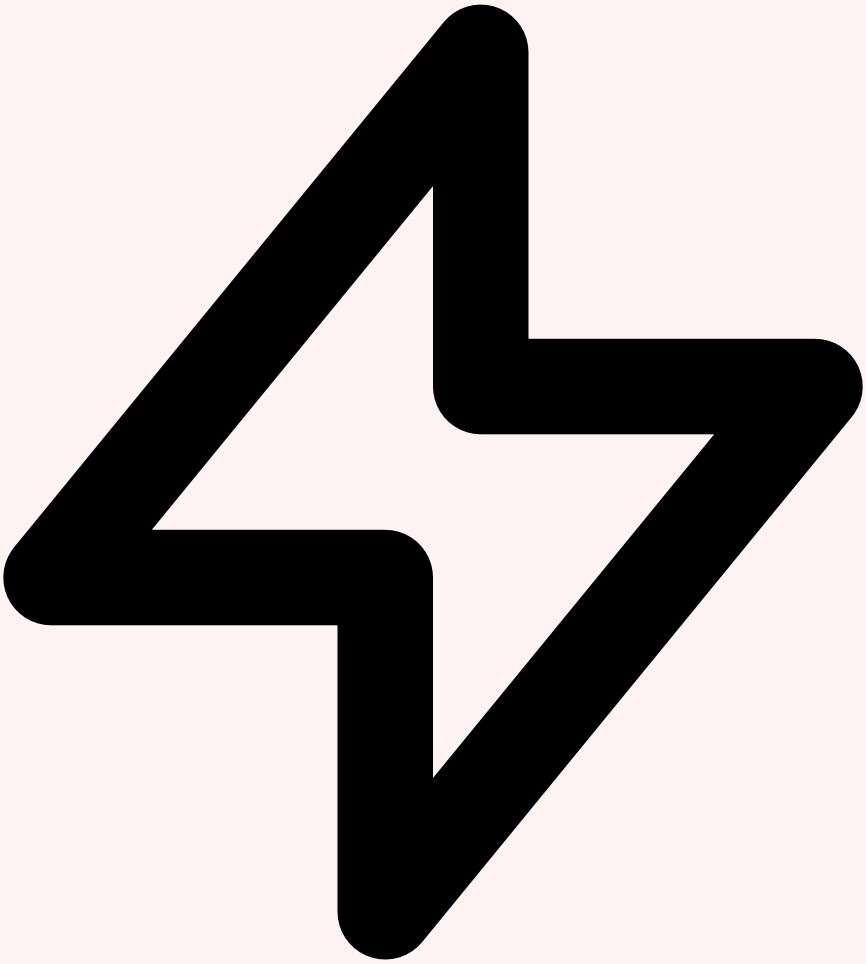
Table des Matières

1 Introduction : Kubernetes et l'ère de l'IA

L'explosion de l'intelligence artificielle générative a fondamentalement transformé les exigences des infrastructures informatiques. Les modèles de langage comme Llama 3, Mistral, ou les architectures de diffusion exigent des ressources GPU considérables, tant pour l'entraînement que pour l'inférence en production. **Kubernetes**, l'orchestrateur de conteneurs devenu standard de l'industrie, s'est naturellement imposé comme la plateforme de choix pour gérer ces workloads d'IA à grande échelle. Guide complet Kubernetes pour l'IA : GPU scheduling avancé, MIG, time-slicing, model serving avec vLLM et Triton,. Thèmes : kubernetes IA, kubernetes. Dans un contexte où l'intelligence artificielle transforme les pratiques de cybersécurité, la maîtrise de ia kubernetes gpu scheduling serving devient un avantage stratégique pour les équipes techniques. Nous abordons notamment : table des matières, 1 introduction : kubernetes et l'ère

de l'ia et 2 architecture gpu cluster kubernetes. Les professionnels y trouveront des recommandations actionnables, des commandes prêtes à l'emploi et des stratégies de mise en œuvre adaptées aux environnements d'entreprise.

Cependant, orchestrer des GPU sur Kubernetes diffère radicalement de la gestion classique de workloads CPU. Un GPU NVIDIA A100 coûte entre 10 000 et 15 000 euros, et un cluster de production peut facilement contenir des dizaines de ces accélérateurs. **L'optimisation de l'utilisation GPU** n'est pas simplement une question technique : c'est un impératif économique. Un GPU inactif pendant une heure représente un coût direct significatif, et les entreprises qui déploient des modèles d'IA en production doivent maîtriser finement l'allocation de ces ressources rares et coûteuses.



Pourquoi Kubernetes pour l'IA ?

Kubernetes apporte plusieurs avantages déterminants pour les workloads d'IA qui justifient sa position dominante dans ce domaine :

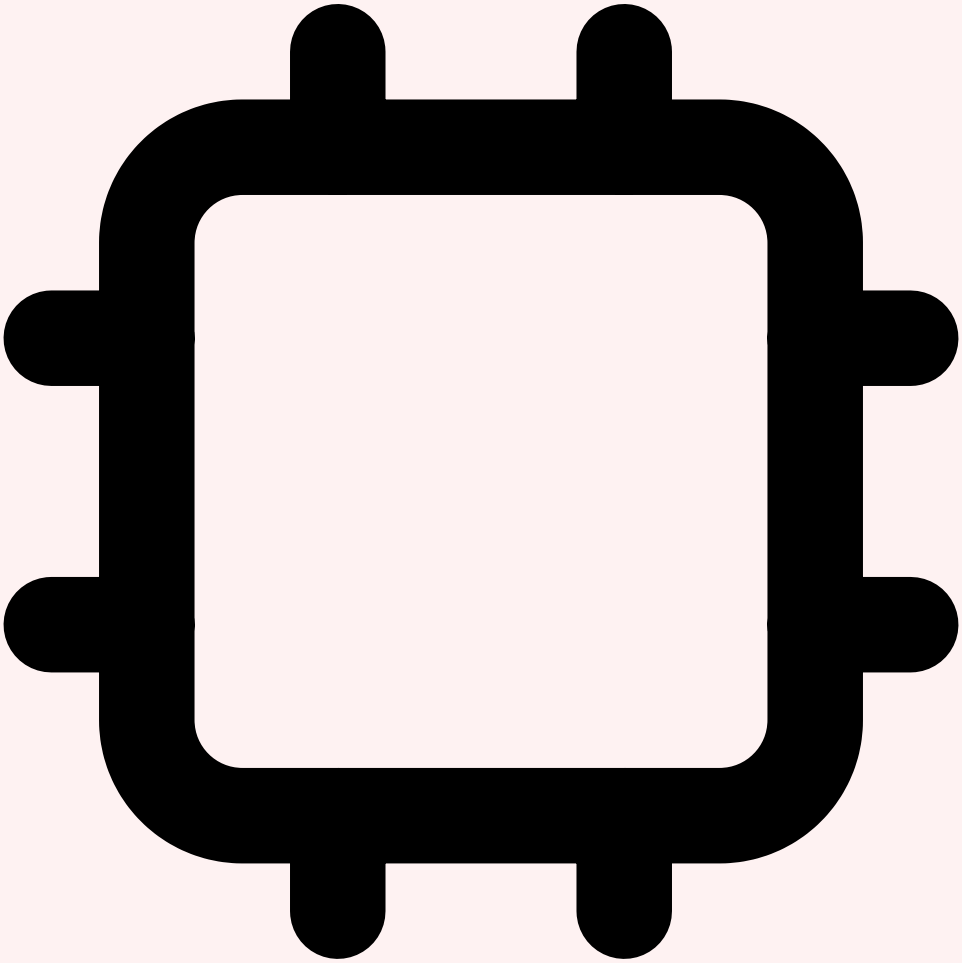
▹**Abstraction de l'infrastructure** : les data scientists décrivent leurs besoins en ressources GPU sans se soucier de l'infrastructure sous-jacente. Kubernetes se charge du placement optimal des pods sur les nœuds disposant des GPU appropriés.

▹**Reproductibilité** : les configurations déclaratives (manifests YAML) garantissent que les environnements d'entraînement et d'inférence sont parfaitement reproductibles d'un déploiement à l'autre, éliminant le classique problème du « ça marchait sur ma machine ».

Vos pipelines de données d'entraînement sont-ils protégés contre l'empoisonnement ?

▹**Scaling dynamique** : Kubernetes permet de scaler horizontalement les serveurs d'inférence en fonction de la demande, et de provisionner automatiquement de nouveaux nœuds GPU via des autoscalers comme Karpenter.

▹**Multi-tenancy** : dans une organisation où plusieurs équipes partagent un cluster GPU, Kubernetes offre des mécanismes de quotas, de namespaces et de priorités pour répartir équitablement les ressources.



Les défis spécifiques du GPU sur Kubernetes

La gestion des GPU sur Kubernetes pose des défis uniques que les administrateurs doivent comprendre avant de déployer leurs premiers workloads d'IA. Contrairement au CPU, un GPU ne se partage pas nativement de manière aussi granulaire. La **topologie NUMA** (Non-

Uniform Memory Access) des serveurs multi-GPU ajoute une couche de complexité : deux GPU connectés via NVLink auront une bande passante d'échange bien supérieure à deux GPU communiquant via le bus PCIe. Le **NVIDIA Device Plugin** pour Kubernetes expose les GPU comme des ressources schedulables, mais la configuration optimale requiert une compréhension fine de l'architecture matérielle sous-jacente.

les workloads d'IA présentent des profils de ressources très variés : un job d'entraînement distribué peut nécessiter 8 GPU H100 pendant plusieurs jours, tandis qu'un serveur d'inférence peut fonctionner avec un seul GPU A10G mais nécessiter un autoscaling rapide pour absorber les pics de trafic. Cette **hétérogénéité des besoins** impose une stratégie de scheduling poussée que nous détaillerons dans cet article.

Périmètre de cet article : Nous couvrirons l'architecture d'un cluster K8s optimisé pour le GPU, le scheduling avancé avec MIG et time-slicing, le model serving avec vLLM, Triton et KServe, les pipelines KubeFlow, l'autoscaling GPU, et les bonnes pratiques de production. Niveau requis : familiarité avec Kubernetes et les concepts GPU de base.

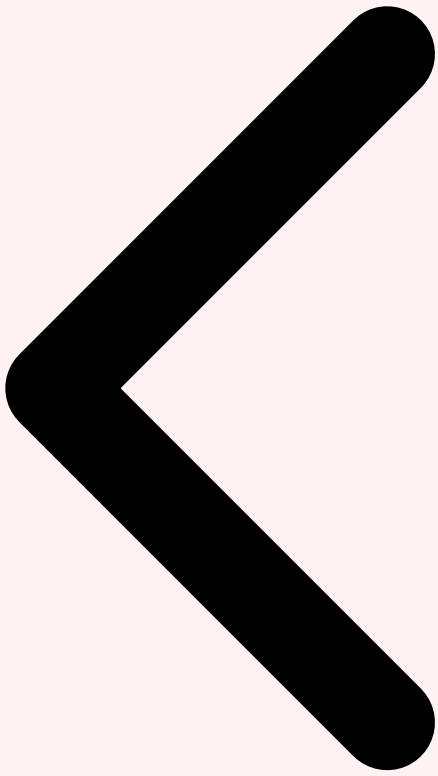
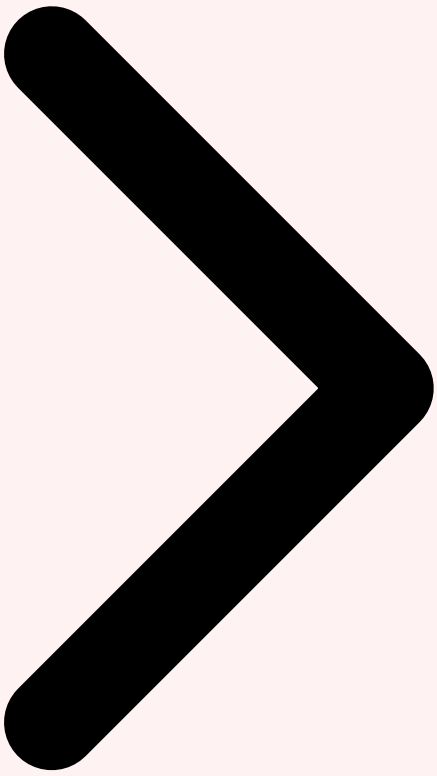


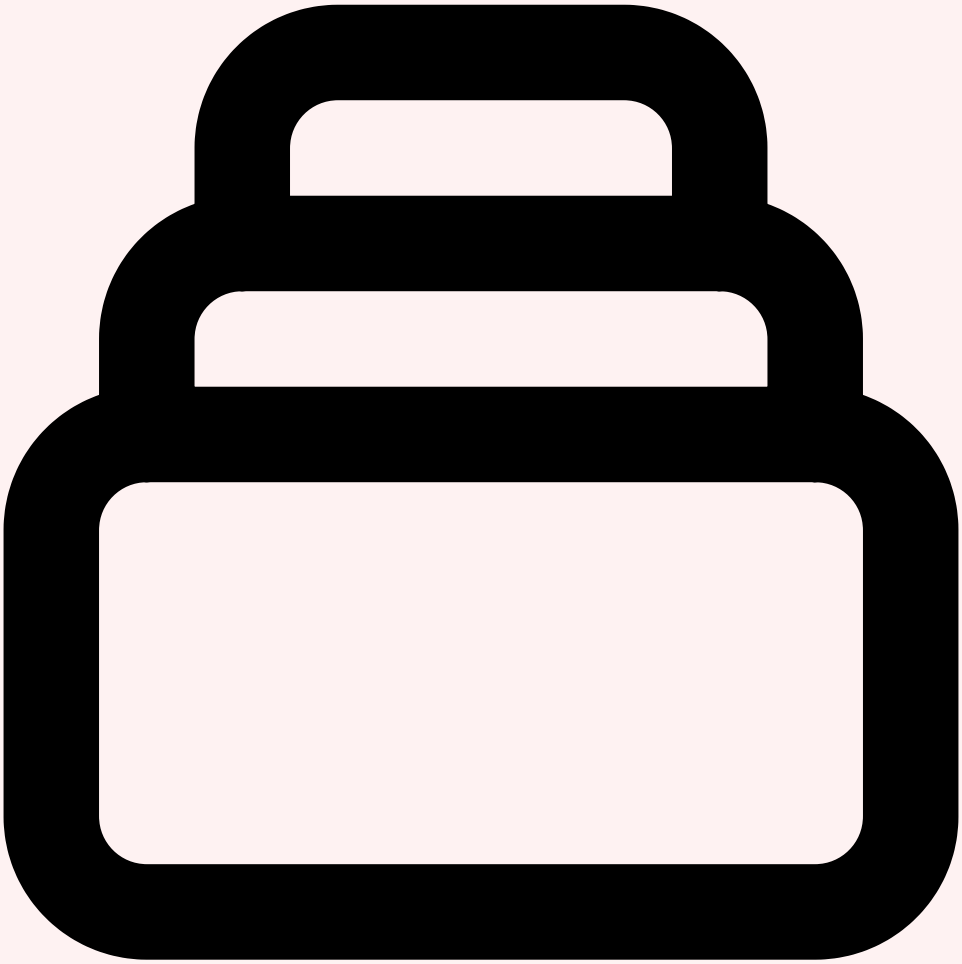
Table des Matières Introduction Architecture GPU Cluster



Critere	Description	Niveau de risque
Confidentialite	Protection des donnees d'entrainement et des prompts	Eleve
Integrite	Fiabilite des sorties et detection des hallucinations	Critique
Disponibilite	Resilience du service et gestion de la charge	Moyen
Conformite	Respect du RGPD, AI Act et politiques internes	Eleve

2 Architecture GPU Cluster Kubernetes

La conception d'un cluster Kubernetes optimisé pour les workloads GPU demande une réflexion architecturale spécifique. Contrairement à un cluster classique où les nœuds sont relativement interchangeables, un cluster GPU présente une **hétérogénéité matérielle** importante : différents types de GPU (A100, H100, L40S, T4), différentes quantités de VRAM, et des topologies d'interconnexion variées. L'architecture doit refléter cette diversité tout en simplifiant la vie des utilisateurs.



Node Pools et séparation des workloads

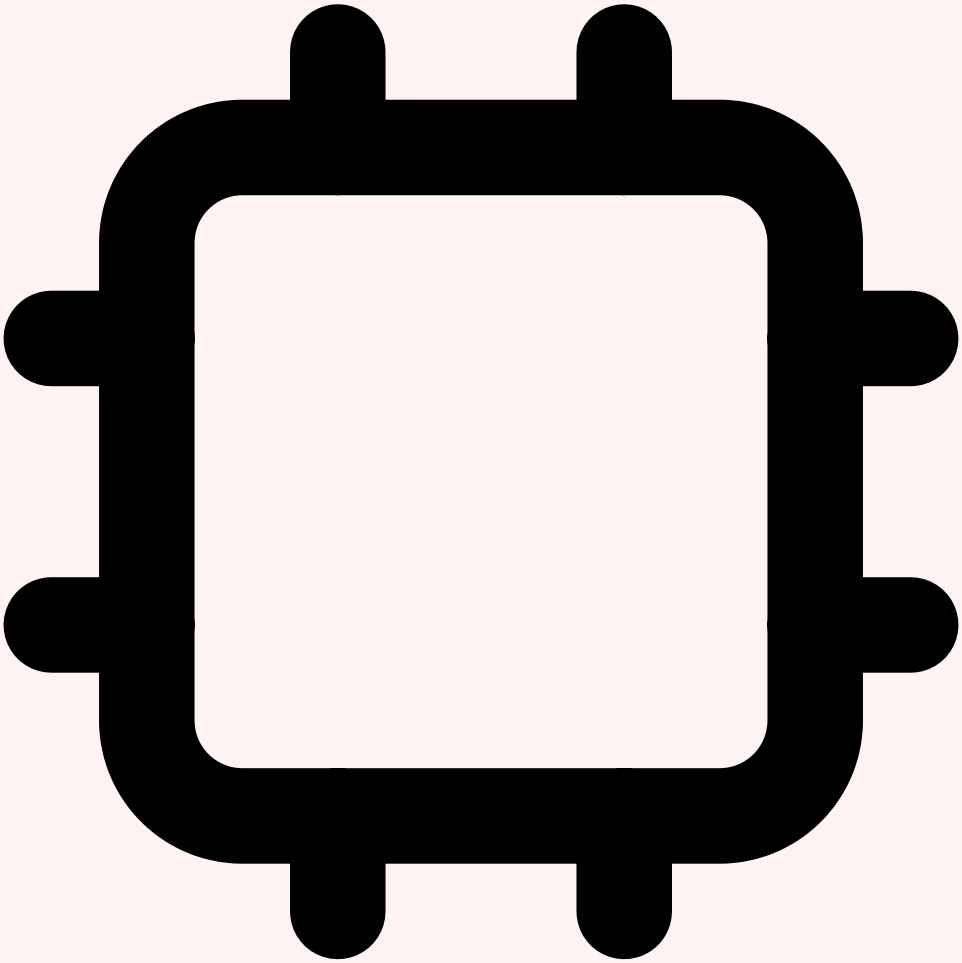
La stratégie fondamentale consiste à organiser le cluster en **node pools** spécialisés. Chaque pool regroupe des nœuds homogènes dédiés à un type de workload spécifique. Cette séparation permet d'optimiser l'utilisation des ressources et de garantir que les workloads critiques ne sont pas perturbés par des tâches de moindre priorité.

▹**Pool système** : nœuds CPU uniquement pour le control plane, les opérateurs, le monitoring (Prometheus, Grafana) et les composants d'infrastructure comme Istio ou Nginx Ingress.

▹**Pool entraînement** : nœuds équipés de GPU haute performance (A100 80GB, H100 80GB) avec interconnexion NVLink/NVSwitch pour l'entraînement distribué multi-GPU. Ces nœuds disposent généralement de 8 GPU par serveur.

▹**Pool inférence** : nœuds avec des GPU optimisés coût/performance (A10G, L4, L40S) pour servir les modèles en production. Le ratio GPU/nœud est souvent plus faible (1-2 GPU) pour permettre un scaling plus granulaire.

▹**Pool spot/préemptible** : nœuds GPU à coût réduit (instances spot AWS, preemptible GCP) pour les jobs d'entraînement tollérants aux interruptions, avec checkpointing automatique.

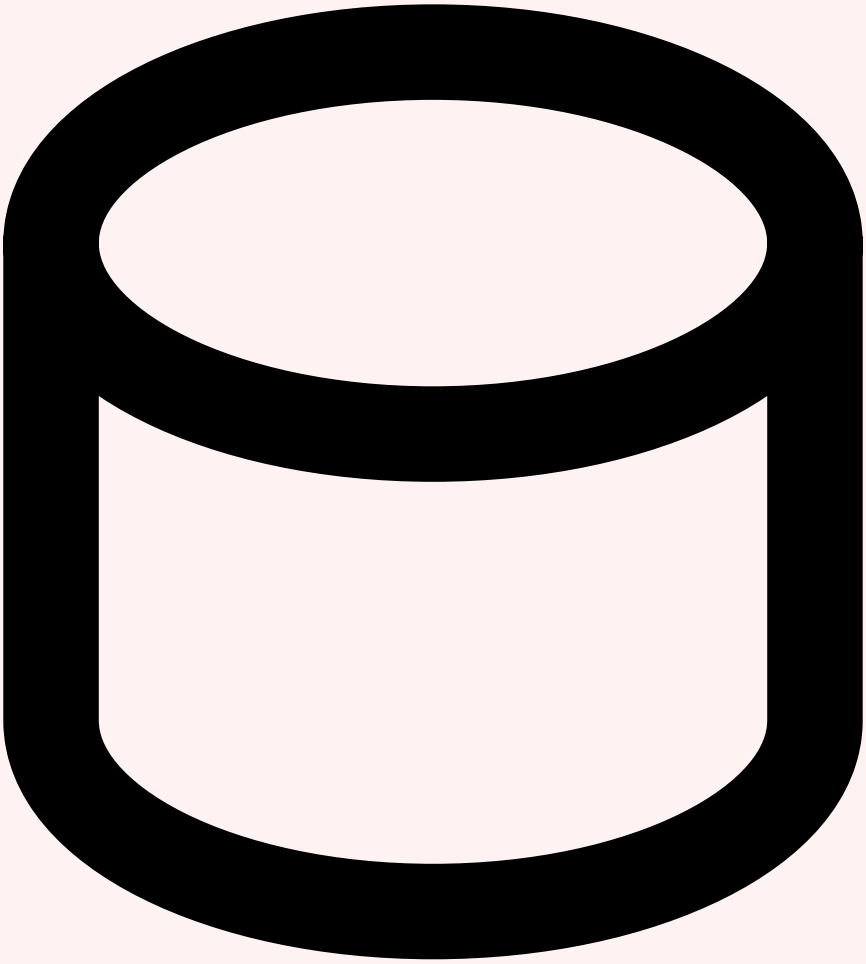


NVIDIA Device Plugin et GPU Feature Discovery

Le **NVIDIA Device Plugin** est le composant central qui permet à Kubernetes de découvrir et d'allouer les GPU. Déployé en tant que DaemonSet, il s'exécute sur chaque nœud GPU et expose la ressource `nvidia.com/gpu` au scheduler Kubernetes. En complément, le **GPU Feature Discovery** (GFD) labellise automatiquement les nœuds avec des informations détaillées sur le matériel : modèle de GPU, quantité de mémoire, capacité de calcul (compute capability), et support MIG. Pour approfondir, consultez Phishing Généré par IA : Nouvelles Menaces.

YAML

```
# Labels automatiques générés par GPU Feature Discovery
nvidia.com/gpu.product: NVIDIA-A100-SXM4-80GB
nvidia.com/gpu.memory: 81920
nvidia.com/gpu.count: 8
nvidia.com/gpu.compute.major: 8
nvidia.com/gpu.compute.minor: 0
nvidia.com/mig.capable: true
nvidia.com/gpu.replicas: 1
```



Topologie NUMA et affinité GPU

Dans les serveurs multi-GPU, la **topologie NUMA** (Non-Uniform Memory Access) impacte significativement les performances. Un serveur DGX A100 possède 8 GPU connectés par NVSwitch avec une bande passante de 600 GB/s par GPU. Cependant, les GPU sont répartis entre deux sockets CPU, et l'accès à la mémoire du socket distant est plus lent. Le **Topology Manager** de Kubernetes, combiné avec le NVIDIA Device Plugin configuré en mode `topology-aware`, garantit que les GPU alloués à un pod sont physiquement proches et connectés via les liens les plus rapides.

YAML

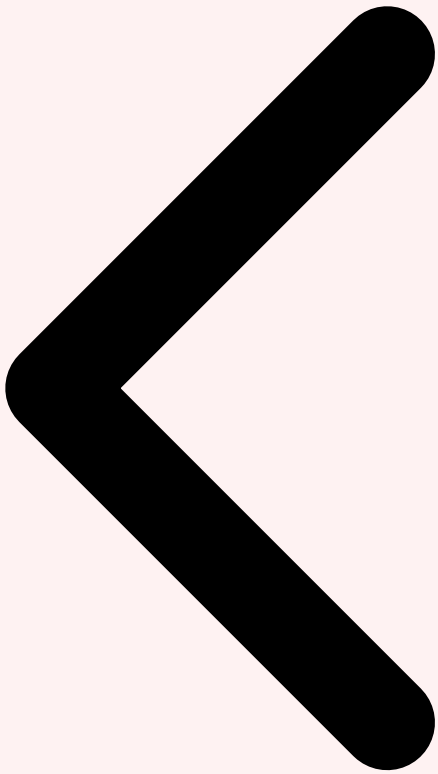
```
# kubelet config pour topology-aware GPU allocation
topologyManagerPolicy: best-effort
topologyManagerScope: pod
# NVIDIA Device Plugin ConfigMap
apiVersion: v1
kind: ConfigMap
metadata:
  name: nvidia-device-plugin
data:
  config.yaml: |
    version: v1
    sharing:
      timeSlicing:
        renameByDefault: false
        resources:
          - name: nvidia.com/gpu
            replicas: 1
    flags:
      migStrategy: none
      gdsEnabled: false
      mofedEnabled: false
```

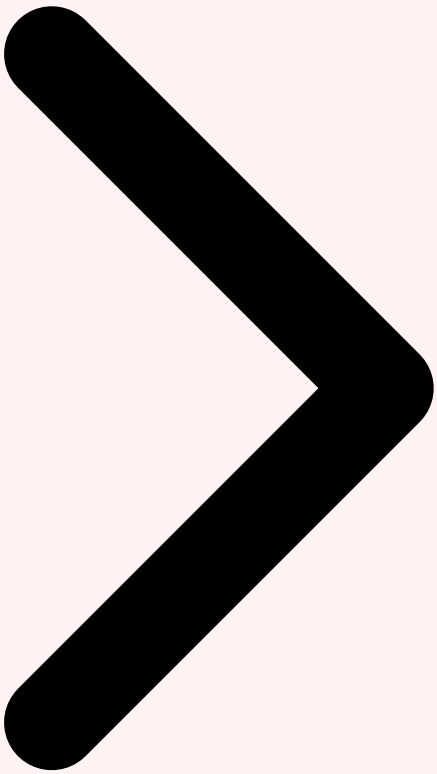
Figure 1 -- Architecture d'un cluster Kubernetes GPU avec 4 node pools specialises, couches stockage, reseau et observabilite

Cas concret
En 2024, des chercheurs de Cornell ont publié une étude démontrant l'empoisonnement de données d'entraînement de modèles de vision par ordinateur avec seulement 0.01% d'images malveillantes, suffisant pour créer des backdoors indétectables par les méthodes de validation standard.

Votre organisation est-elle prête à faire face aux attaques basées sur l'IA ?

Conseil d'architecture : Utilisez des taints et tolerations pour isoler strictement les node pools. Les nœuds GPU d'entraînement doivent avoir un taint `workload=training:NoSchedule` pour empêcher le scheduler d'y placer des pods d'inférence qui gaspilleraient les ressources NVLink.





3 GPU Scheduling avancé

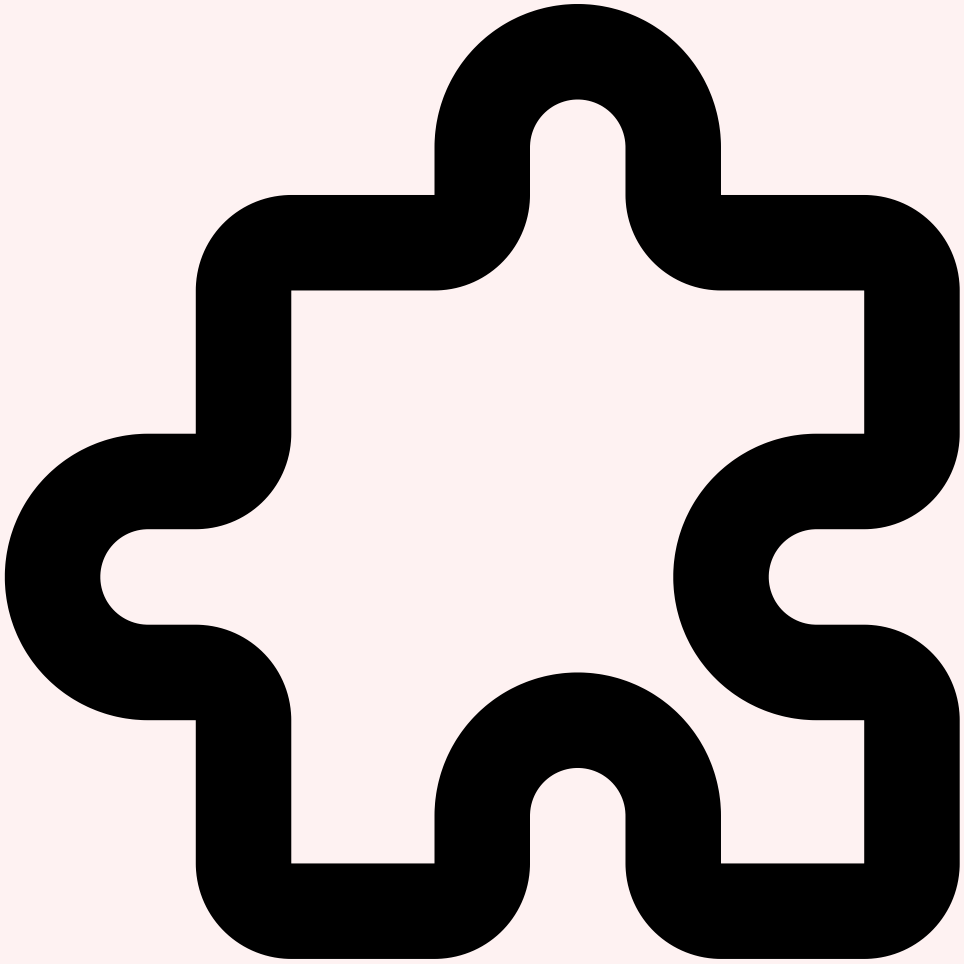
Le scheduling GPU sur Kubernetes va bien au-delà de la simple allocation de GPUs entiers à des pods. Les techniques modernes permettent un **partage fin des ressources GPU**, essentiel pour maximiser l'utilisation et réduire les coûts. Trois approches principales existent : l'allocation de GPU entiers, le partitionnement matériel via **MIG (Multi-Instance GPU)**, et le partage temporel via **time-slicing**.

Allocation de GPU entiers

L'approche la plus simple consiste à allouer un ou plusieurs GPU entiers à un pod. Kubernetes gère cela nativement via les resource requests et limits. Le scheduler garantit que le pod est placé sur un nœud disposant du nombre de GPU demandé, et le NVIDIA Device Plugin monte les devices correspondants dans le conteneur.

YAML

```
# Pod demandant 2 GPU entiers
apiVersion: v1
kind: Pod
metadata:
  name: training-job
spec:
  containers:
  - name: trainer
    image: pytorch/pytorch:2.3.0-cuda12.1-cudnn8-runtime
    resources:
      requests:
        nvidia.com/gpu: 2
        memory: "64Gi"
        cpu: "16"
      limits:
        nvidia.com/gpu: 2
        memory: "64Gi"
    env:
    - name: NVIDIA_VISIBLE_DEVICES
      value: "all"
  nodeSelector:
    nvidia.com/gpu.product: NVIDIA-A100-SXM4-80GB
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```



NVIDIA MIG (Multi-Instance GPU)

Multi-Instance GPU (MIG), disponible sur les A100 et H100, permet de partitionner physiquement un GPU en jusqu'à 7 instances indépendantes. Chaque instance dispose de ses propres **Streaming Multiprocessors (SM)**, de sa propre mémoire et de son propre bus mémoire, offrant une isolation complète au niveau matériel. Contrairement au time-slicing, MIG garantit des performances prédictibles car les partitions ne partagent aucune ressource.

Les profils MIG disponibles sur un A100 80GB sont les suivants :

▹**7x 1g.10gb** : 7 instances de 10 GB chacune avec 1/7 des SM. Idéal pour les petits modèles d'inférence ou les notebooks Jupyter.

▹**3x 2g.20gb** : 3 instances de 20 GB avec 2/7 des SM. Bon compromis pour des modèles de taille moyenne (7B quantizés).

▹**2x 3g.40gb** : 2 instances de 40 GB avec 3/7 des SM. Adapté aux modèles 13B en inférence.

▹**1x 7g.80gb** : GPU entier sans partitionnement. Pour l'entraînement ou les très grands modèles.

YAML

```yaml
# Configuration MIG dans le Device Plugin ConfigMap
apiVersion: v1
kind: ConfigMap
metadata:
  name: nvidia-device-plugin
  namespace: kube-system
data:
  config.yaml: |
    version: v1
    flags:
      migStrategy: mixed  # single | mixed | none
    sharing:
      mig:
        resources:
        - name: nvidia.com/mig-1g.10gb
          replicas: 7
        - name: nvidia.com/mig-3g.40gb
          replicas: 2
---
# Pod utilisant une partition MIG
apiVersion: v1
kind: Pod
metadata:
  name: inference-small
spec:
  containers:
  - name: vllm-server
    image: vllm/vllm-openai:latest
    resources:
      requests:
        nvidia.com/mig-1g.10gb: 1
      limits:
        nvidia.com/mig-1g.10gb: 1
```

Time-Slicing GPU

Le **time-slicing** permet de partager un GPU entre plusieurs pods en alternant rapidement les contextes d'exécution. Contrairement à MIG, il n'y a pas d'isolation mémoire : tous les processus partagent la VRAM totale du GPU. L'avantage est sa compatibilité universelle avec tous les GPU NVIDIA (pas seulement A100/H100) et sa flexibilité : on peut configurer n'importe quel nombre de « réplicas » virtuels.

YAML

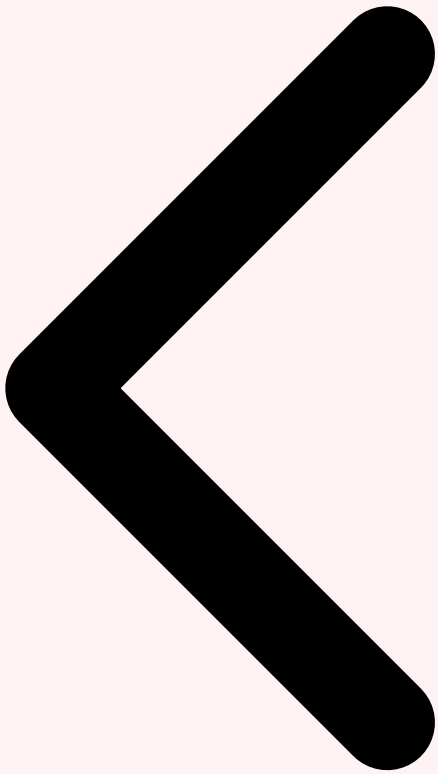
```
# Time-slicing: exposer 4 GPU virtuels par GPU physique
apiVersion: v1
kind: ConfigMap
metadata:
  name: nvidia-device-plugin
  namespace: kube-system
data:
  config.yaml: |
    version: v1
    sharing:
      timeSlicing:
        renameByDefault: true
        failRequestsGreaterThanOne: false
        resources:
        - name: nvidia.com/gpu
          replicas: 4  # 1 GPU physique = 4 GPU virtuels
```

Attention au time-slicing : Le time-slicing ne fournit aucune isolation mémoire. Si un pod consomme toute la VRAM, les autres pods sur le même GPU recevront des erreurs OOM (Out of Memory). En production, combinez le time-slicing avec des limites mémoire applicatives strictes (par exemple `--gpu-memory-utilization 0.25` dans vLLM pour 4 replicas). MIG reste préférable pour les workloads de production critiques.

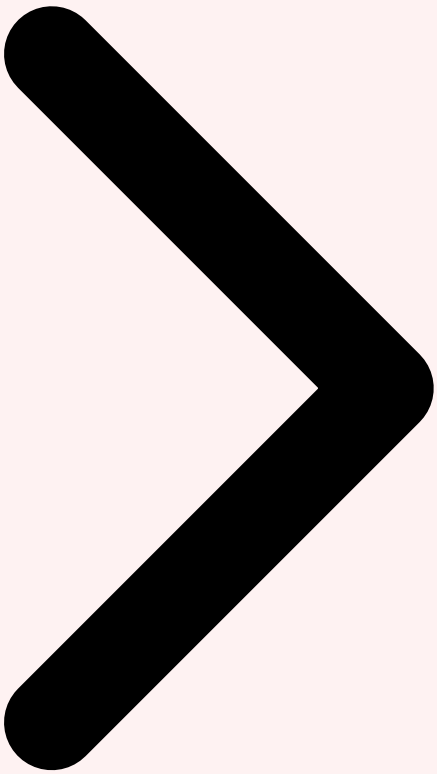
Fractional GPU et GPU Operator

Le **NVIDIA GPU Operator** simplifie considérablement la gestion GPU en automatisant le déploiement de tous les composants nécessaires : drivers NVIDIA, container toolkit, device plugin, DCGM exporter et GFD. Il permet une approche « Day 0 » où un nouveau nœud GPU est automatiquement configuré sans intervention manuelle. En complément, des solutions comme **Run:ai** ou **HAMi (Heterogeneous AI Computing Virtualization)** offrent un fractional GPU encore plus granulaire, permettant d'allouer des fractions arbitraires de VRAM et de compute à chaque pod avec une isolation forte.

Recommandation de scheduling : Pour l'inférence de petits modèles (< 7B), utilisez MIG sur les A100/H100 pour l'isolation garantie, ou le time-slicing sur les GPU grand public (T4, L4). Pour l'entraînement, allouez toujours des GPU entiers. Pour les environnements de développement (notebooks), le time-slicing avec 4-8 replicas est un excellent compromis coût/flexibilité.

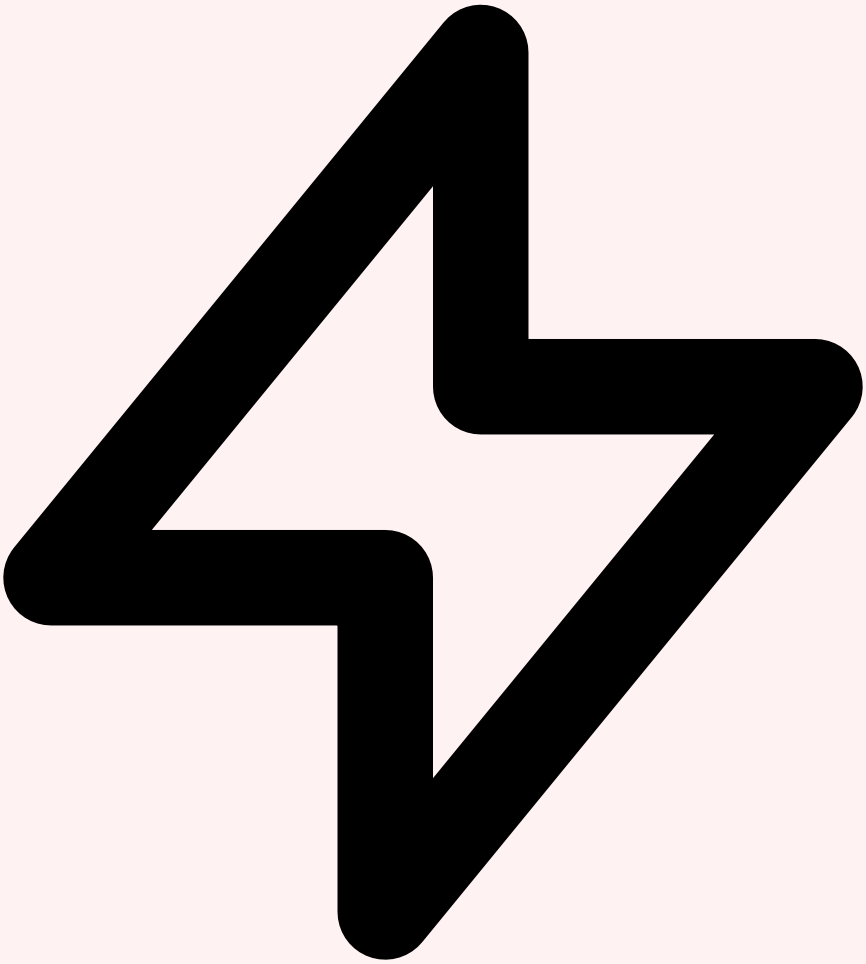


Architecture GPU Cluster GPU Scheduling Model Serving



4 Model Serving sur Kubernetes

Le model serving est l'étape critique qui transforme un modèle entraîné en service accessible par les applications. Sur Kubernetes, plusieurs frameworks de serving se sont imposés, chacun avec ses forces spécifiques. Le choix du framework dépend du type de modèle, des exigences de latence, du throughput requis et de la complexité du pipeline d'inférence.



vLLM : le standard pour les LLM

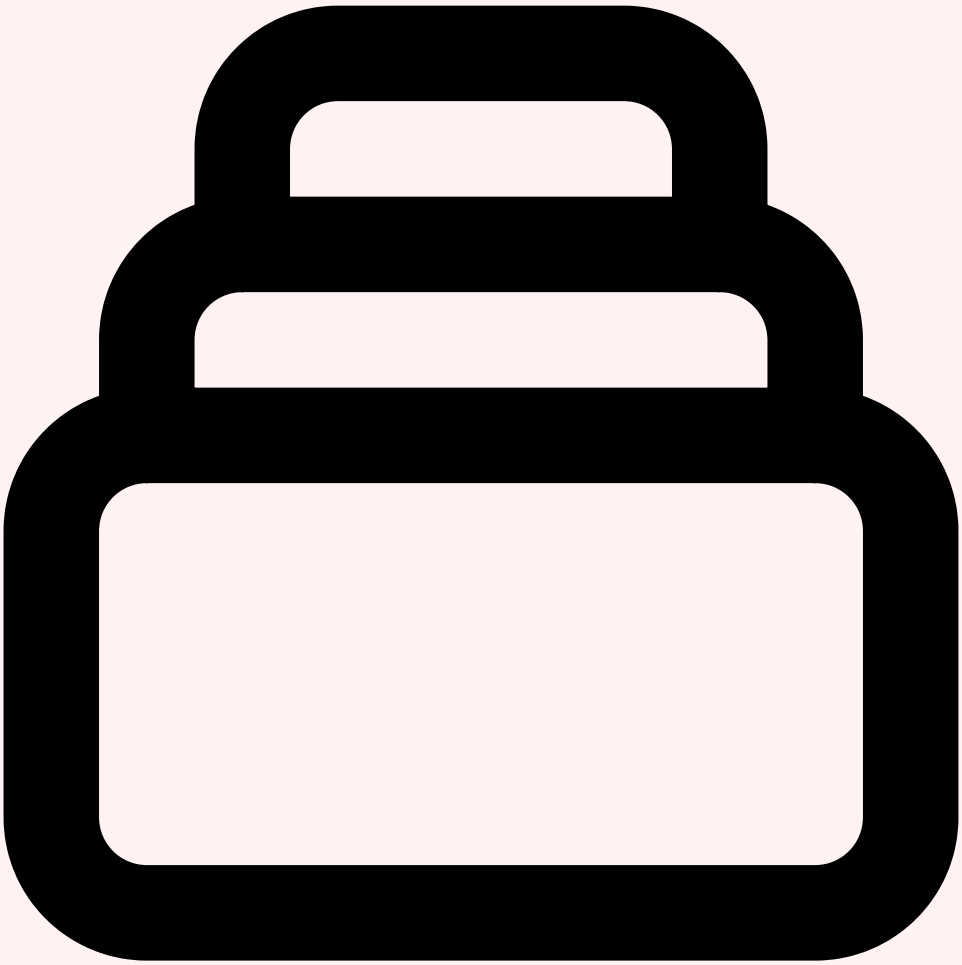
vLLM (Virtual LLM) s'est impose comme le framework de reference pour servir les Large Language Models en production. Son innovation cle est le **PagedAttention**, un algorithme inspire de la gestion de memoire virtuelle des systemes d'exploitation. Au lieu d'allouer un bloc contigu de memoire pour le KV-cache de chaque requete, PagedAttention divise le cache en pages de taille fixe qui peuvent etre allouees dynamiquement, reduisant le gaspillage memoire de 60 a 80% par rapport aux approches traditionnelles. Pour approfondir, consultez Sécurité des Agents IA en Production : Sandboxing et Contrôles.

vLLM supporte le **continuous batching**, qui insere de nouvelles requetes dans un batch des qu'une requete existante termine sa generation, maximisant ainsi l'utilisation GPU. Il offre egalement le **speculative decoding**, le **tensor parallelism** pour distribuer un modele sur plusieurs GPU, et une API compatible OpenAI pour une integration transparente.

YAML

```
# Deployment vLLM sur Kubernetes
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vllm-llama3-70b
  namespace: inference
spec:
  replicas: 2
  selector:
    matchLabels:
      app: vllm-llama3
  template:
    metadata:
      labels:
        app: vllm-llama3
    spec:
      containers:
      - name: vllm
        image: vllm/vllm-openai:v0.6.4
        args:
          - --model=meta-llama/Meta-Llama-3.1-70B-Instruct
          - --tensor-parallel-size=2
          - --gpu-memory-utilization=0.90
          - --max-model-len=8192
          - --enable-chunked-prefill
          - --max-num-batched-tokens=32768
          - --quantization=awq
          - --dtype=half
        ports:
        - containerPort: 8000
          name: http
        resources:
          requests:
            nvidia.com/gpu: 2
            memory: "96Gi"
            cpu: "16"
          limits:
            nvidia.com/gpu: 2
            memory: "96Gi"
        readinessProbe:
          httpGet:
            path: /health
            port: 8000
          initialDelaySeconds: 120
          periodSeconds: 10
        volumeMounts:
        - name: model-cache
          mountPath: /root/.cache/huggingface
        - name: shm
          mountPath: /dev/shm
      volumes:
      - name: model-cache
        persistentVolumeClaim:
          claimName: model-cache-pvc
      - name: shm
        emptyDir:
          medium: Memory
          sizeLimit: 16Gi
      nodeSelector:
        gpu-type: inference
      tolerations:
      - key: nvidia.com/gpu
```

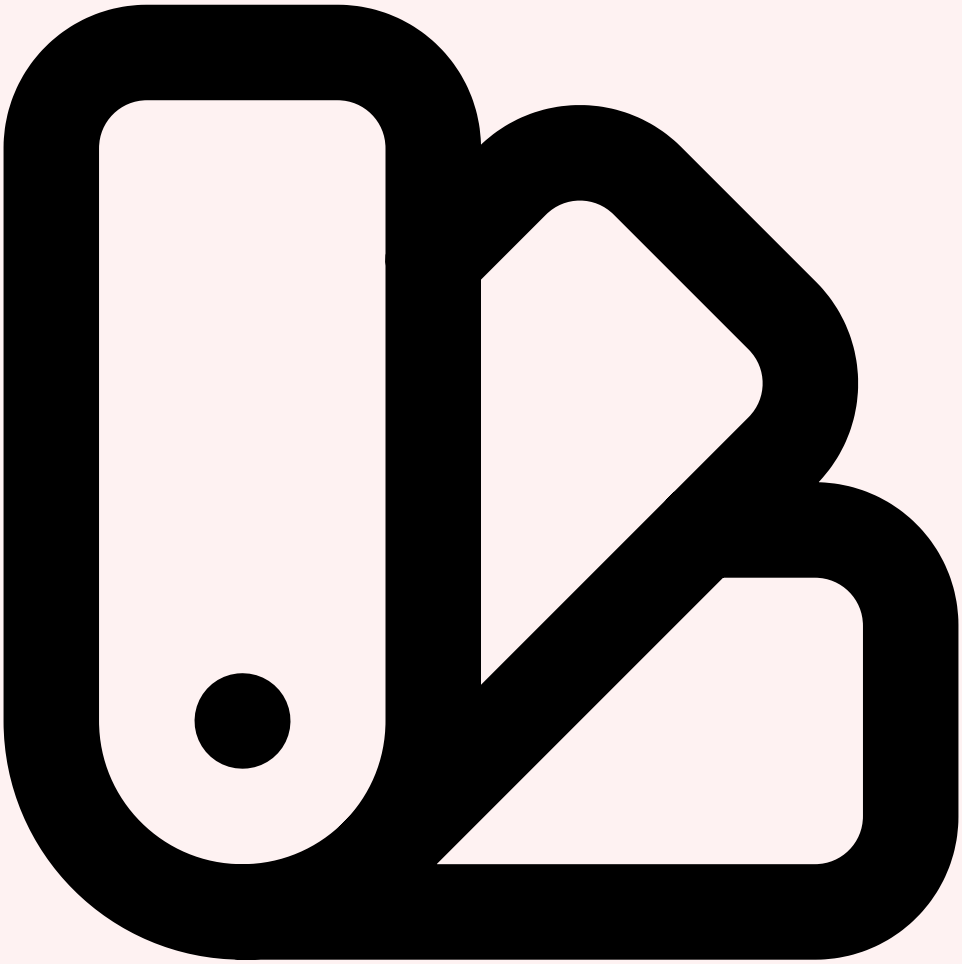
```
      operator: Exists
      effect: NoSchedule
```



NVIDIA Triton Inference Server

Triton Inference Server est la solution la plus polyvalente pour le model serving. Il supporte simultanement des modeles PyTorch, TensorFlow, ONNX, TensorRT, et meme des pipelines Python personnalises via le **Python backend**. Triton excelle dans les scenarios multi-modeles ou un meme serveur doit servir plusieurs modeles differents, en gerant automatiquement le chargement/dechargement en memoire GPU en fonction de la demande.

Le **Model Analyzer** de Triton permet de profiler automatiquement un modele pour trouver la configuration optimale de batch size, concurrence et instances GPU. L'**ensemble scheduling** permet de chainer plusieurs modeles dans un pipeline (par exemple : preprocessing -> modele principal -> postprocessing) en une seule requete.



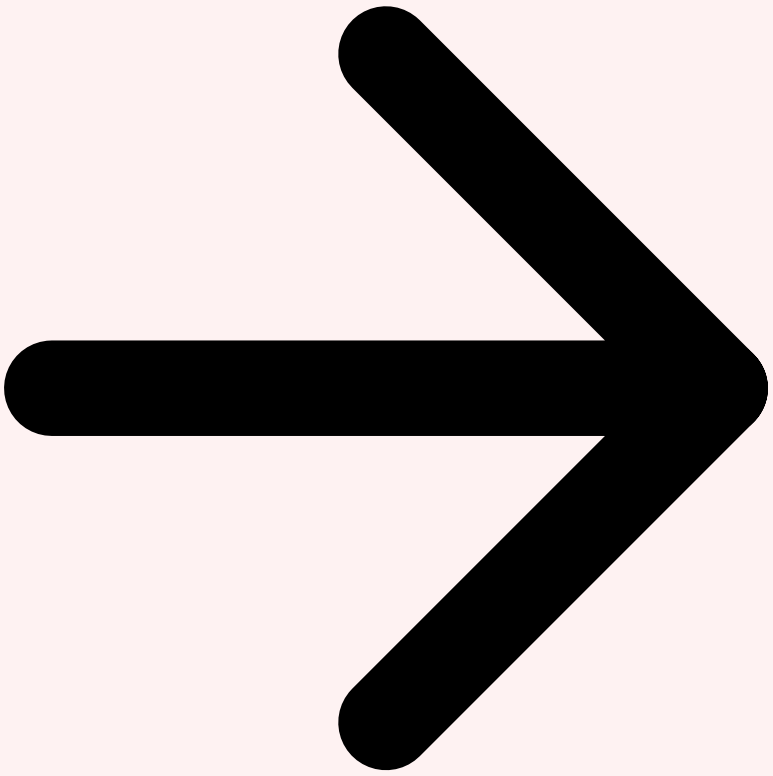
KServe et Seldon Core

KServe (anciennement KFServing) est le composant de serving de reference dans l'ecosysteme KubeFlow. Il ajoute une couche d'abstraction au-dessus de frameworks comme vLLM, Triton ou TGI via le concept d'**InferenceService**. KServe gere automatiquement le canary deployment, le scale-to-zero, le traffic splitting et le model versioning. Son architecture Knative-based permet un autoscaling rapide, y compris jusqu'a zero replicas quand aucun trafic n'est recu, reduisant les couts GPU.

YAML

```
# InferenceService KServe avec vLLM backend
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: llama3-70b-service
  namespace: inference
  annotations:
    serving.kserve.io/autoscalerClass: hpa
    serving.kserve.io/targetUtilizationPercentage: "70"
spec:
  predictor:
    minReplicas: 1
    maxReplicas: 8
    scaleTarget: 5         # requests per second
    scaleMetric: rps
    containers:
    - name: kserve-container
      image: vllm/vllm-openai:v0.6.4
      args:
        - --model=meta-llama/Meta-Llama-3.1-70B-Instruct
        - --tensor-parallel-size=2
      resources:
        requests:
          nvidia.com/gpu: 2
          memory: "96Gi"
        limits:
          nvidia.com/gpu: 2
    canaryTrafficPercent: 10  # 10% du trafic vers la nouvelle version
```

Seldon Core offre une approche similaire avec une emphase particuliere sur les **inference graphs**, permettant de definir des pipelines complexes avec des routeurs, des combiners et des transformateurs. Seldon est particulierement adapte aux cas d'usage ou l'inference implique plusieurs modeles orchestres (ensemble de modeles, A/B testing, multi-armed bandits).

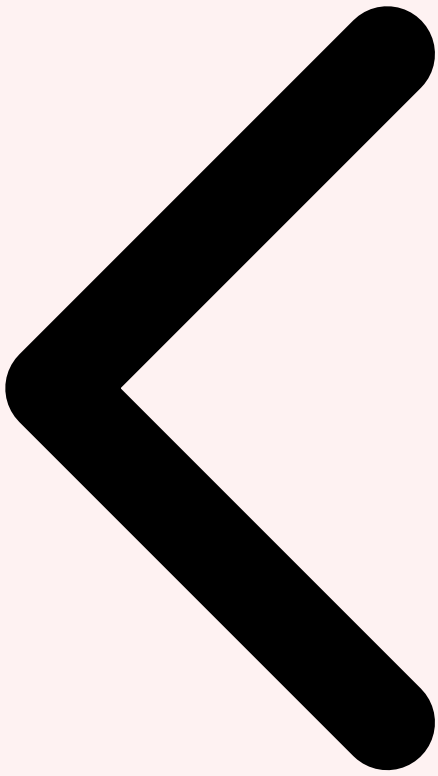


Text Generation Inference (TGI)

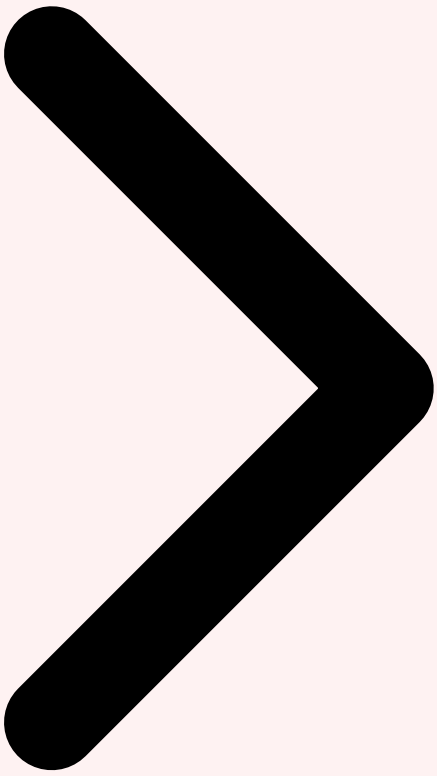
TGI de Hugging Face est un framework de serving optimise specifiquement pour la generation de texte. Il integre nativement le **Flash Attention 2**, le continuous batching et le tensor parallelism. TGI est plus simple a deployer que vLLM pour les modeles Hugging Face standard, avec une API gRPC performante et un support natif du streaming token par token.

Figure 2 -- Pipeline complet KubeFlow : de l'ingestion des donnees au model serving, avec comparatif des frameworks de serving GPU

Choix du framework : Pour les LLM en production, vLLM est le choix par defaut grace a son PagedAttention et son throughput superieur. Triton est prefere pour les pipelines multi-modeles ou quand TensorRT est necessaire. KServe ajoute une couche MLOps indispensable (canary, scale-to-zero, versioning) et peut utiliser vLLM ou Triton comme backend.

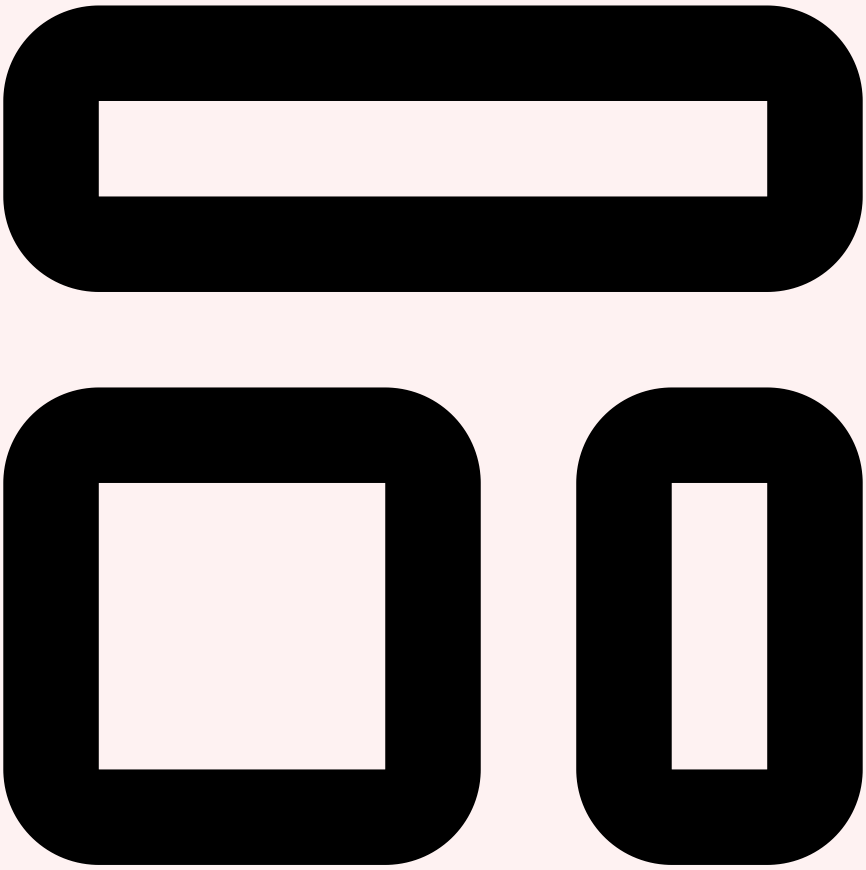


GPU Scheduling Model Serving KubeFlow & MLOps



5 KubeFlow et MLOps sur Kubernetes

KubeFlow est la plateforme MLOps native Kubernetes qui orchestre l'ensemble du cycle de vie des modeles d'IA, depuis l'experimentation jusqu'au deploiement en production. Ne chez Google en 2017 comme un outil interne pour simplifier le deploiement de TensorFlow sur Kubernetes, KubeFlow a evolue en un ecosysteme complet couvrant chaque etape du machine learning industrialise.

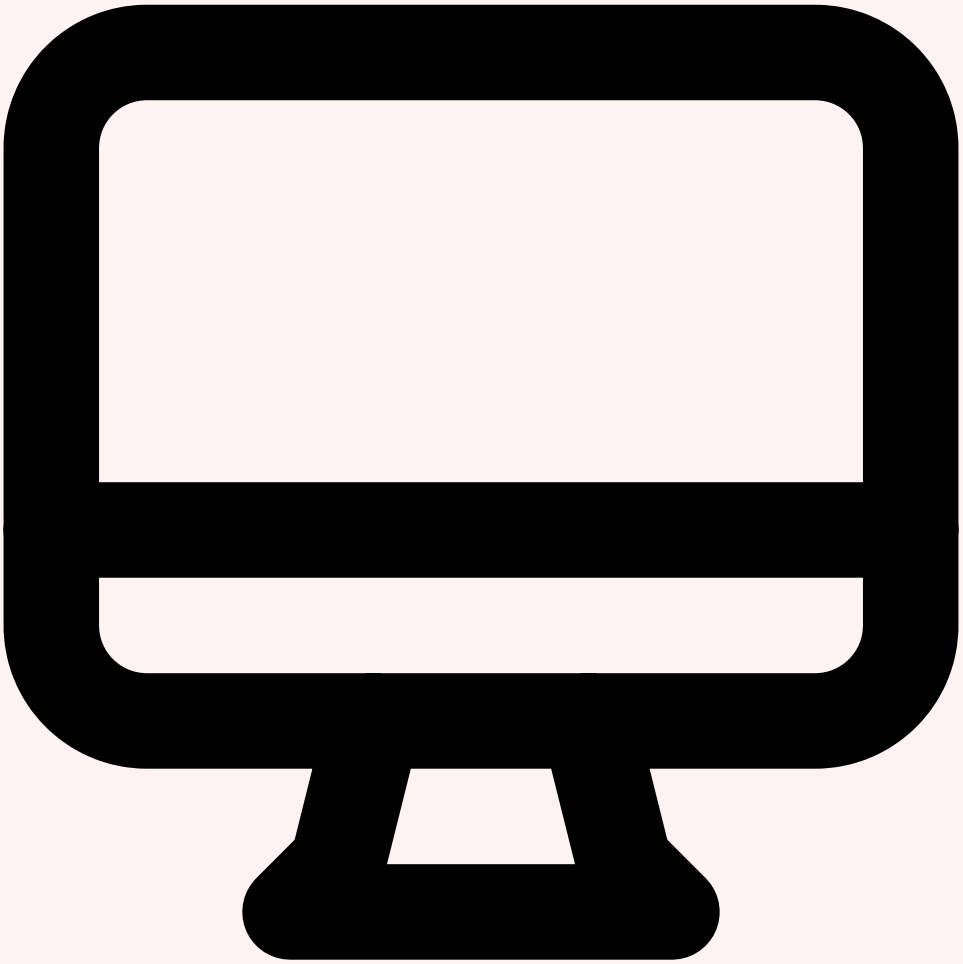


KubeFlow Pipelines

KubeFlow Pipelines (KFP) est le composant central qui permet de definir, deployer et monitorer des workflows ML sous forme de DAG (Directed Acyclic Graphs). Chaque etape du pipeline s'execute dans un conteneur isole avec ses propres ressources GPU, et les artefacts (datasets, modeles, metriques) sont automatiquement versiones et trackes. La version 2 de KFP utilise le SDK Python pour definir les pipelines de maniere programmatique.

Python

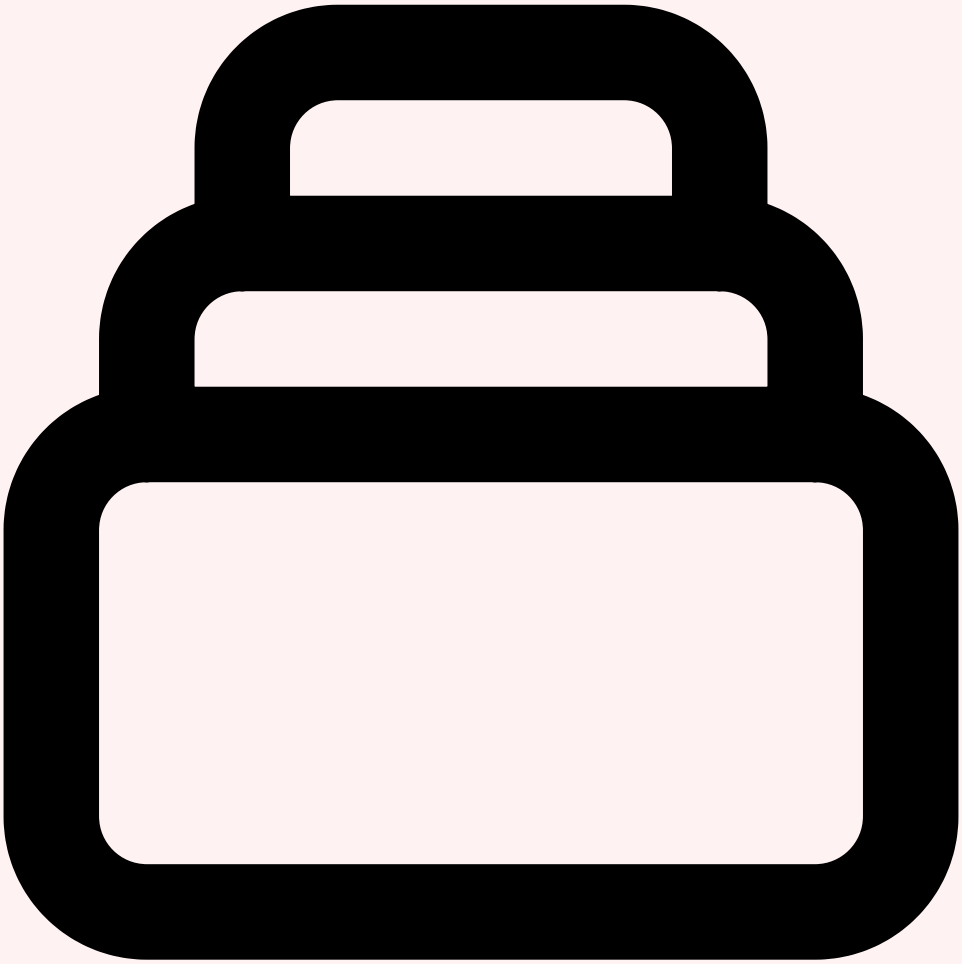
```
# Pipeline KubeFlow : fine-tuning + evaluation + deploiement
from kfp import dsl
from kfp.dsl import Input, Output, Artifact, Model, Metrics

@dsl.component(
    base_image="pytorch/pytorch:2.3.0-cuda12.1-cudnn8-runtime",
    packages_to_install=["transformers", "peft", "bitsandbytes"]
)
def train_model(
    base_model: str,
    dataset_path: Input[Artifact],
    trained_model: Output[Model],
    metrics: Output[Metrics],
    num_epochs: int = 3,
    learning_rate: float = 2e-4,
):
    # Fine-tune un LLM avec QLoRA sur GPU
    from transformers import AutoModelForCausalLM, TrainingArguments
    from peft import LoraConfig, get_peft_model
    # ... training logic ...
    metrics.log_metric("eval_loss", eval_loss)
    metrics.log_metric("eval_perplexity", perplexity)

@dsl.component(base_image="python:3.11")
def evaluate_model(
    model_path: Input[Model],
    metrics: Output[Metrics],
    threshold: float = 0.85,
) -> bool:
    # Evalue le modele et decide du deploiement
    # ... evaluation logic ...
    return accuracy > threshold

@dsl.component(base_image="bitnami/kubectl:latest")
def deploy_model(
    model_path: Input[Model],
    serving_namespace: str = "inference",
):
    # Deploie le modele via KServe InferenceService
    # ... kubectl apply InferenceService ...

@dsl.pipeline(name="llm-fine-tuning-pipeline")
def llm_pipeline(base_model: str = "meta-llama/Llama-3.1-8B"):
    train_task = train_model(base_model=base_model)
    train_task.set_gpu_limit(2)
    train_task.set_memory_limit("64Gi")
    train_task.set_cpu_limit("16")

    eval_task = evaluate_model(
        model_path=train_task.outputs["trained_model"]
    )

    with dsl.Condition(eval_task.output == True):
        deploy_model(
            model_path=train_task.outputs["trained_model"]
        )
```



KubeFlow Notebooks et environnements GPU

KubeFlow Notebooks fournit des environnements Jupyter directement dans le cluster Kubernetes avec un acces GPU transparent. Chaque data scientist peut lancer un notebook avec les ressources GPU souhaitees, sans avoir besoin de configurer des drivers ou des environnements CUDA. Les notebooks sont persistes via des PVCs (Persistent Volume Claims) et survivent aux redemarrages de pods.

La configuration des notebooks GPU se fait via le **Notebook Controller** qui gere les CRDs (Custom Resource Definitions) et integre automatiquement les tolerations GPU, les limites de ressources et les volumes de cache des modeles.



Training Operator pour l'entrainement distribue

Le **Training Operator** de KubeFlow est un controleur Kubernetes qui gere les jobs d'entrainement distribue via des CRDs specifiques a chaque framework. Le `PyTorchJob` orchestre l'entrainement distribue PyTorch avec DDP (Distributed Data Parallel) ou FSDP (Fully Sharded Data Parallel), en creant automatiquement les pods master et worker avec la configuration NCCL appropriee.

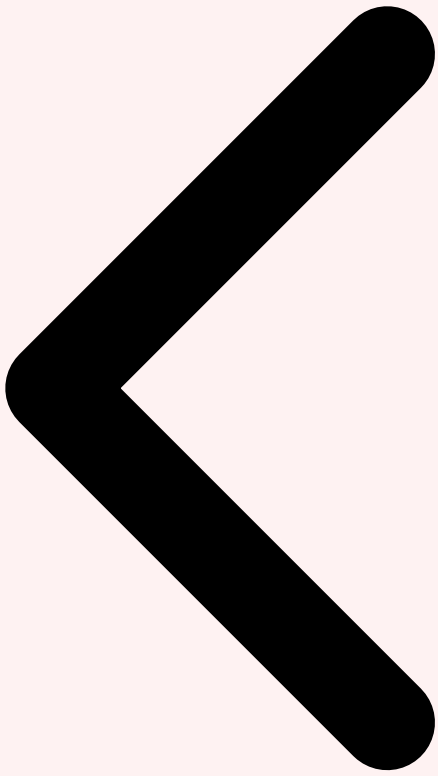
YAML

```
# PyTorchJob pour entrainement distribue multi-noeud
apiVersion: kubeflow.org/v1
kind: PyTorchJob
metadata:
  name: llama3-finetune-distributed
  namespace: training
spec:
  elasticPolicy:
    rdzvBackend: etcd
    minReplicas: 2
    maxReplicas: 4
  pytorchReplicaSpecs:
    Master:
      replicas: 1
      template:
        spec:
          containers:
          - name: pytorch
            image: my-registry/llm-trainer:v2.3
            command:
              - torchrun
              - --nproc_per_node=8
              - --nnodes=$(WORLD_SIZE)
              - --node_rank=$(RANK)
              - train.py
              - --deepspeed=ds_config_zero3.json
              - --model=meta-llama/Llama-3.1-70B
            resources:
              requests:
                nvidia.com/gpu: 8
                memory: "512Gi"
                rdma/rdma_shared_device_a: 1
              limits:
                nvidia.com/gpu: 8
            env:
            - name: NCCL_IB_DISABLE
              value: "0"
            - name: NCCL_NET_GDR_LEVEL
              value: "5"
    Worker:
      replicas: 3
      template:
        spec:
          containers:
          - name: pytorch
            image: my-registry/llm-trainer:v2.3
            resources:
              requests:
                nvidia.com/gpu: 8
                memory: "512Gi"
              limits:
                nvidia.com/gpu: 8
```

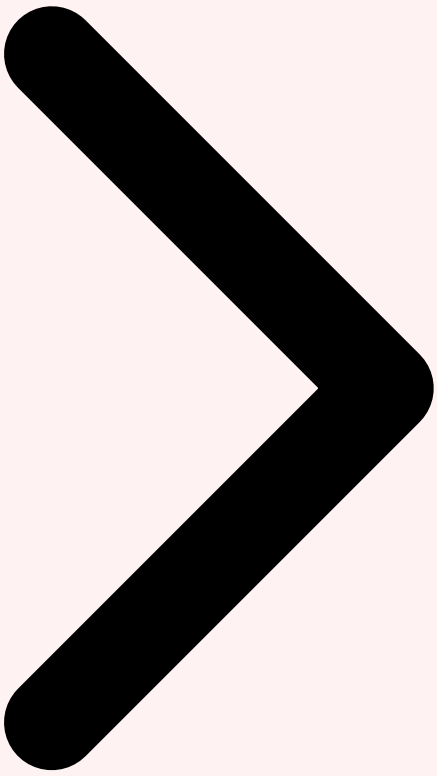
Le Training Operator supporte egalement les `TFJob` pour TensorFlow, `MXJob` pour MXNet, et `XGBoostJob` pour XGBoost. Chaque type de job gere automatiquement la decouverte des pairs, la configuration du backend de communication (NCCL pour GPU, Gloo pour CPU), et

le redemarrage en cas de defaillance de noeud. L'**elastic training** permet meme de redimensionner dynamiquement le nombre de workers pendant l'entrainement, ce qui est particulierement utile avec les instances spot.

Integration MLflow : KubeFlow s'integre naturellement avec MLflow pour le tracking d'experiences et le model registry. Chaque run KubeFlow Pipeline peut logger automatiquement ses metriques, parametres et artefacts dans MLflow, creant une tracabilite complete du pipeline. Utilisez l'annotation `mlflow.log_param()` dans vos composants KFP pour un tracking sans effort.

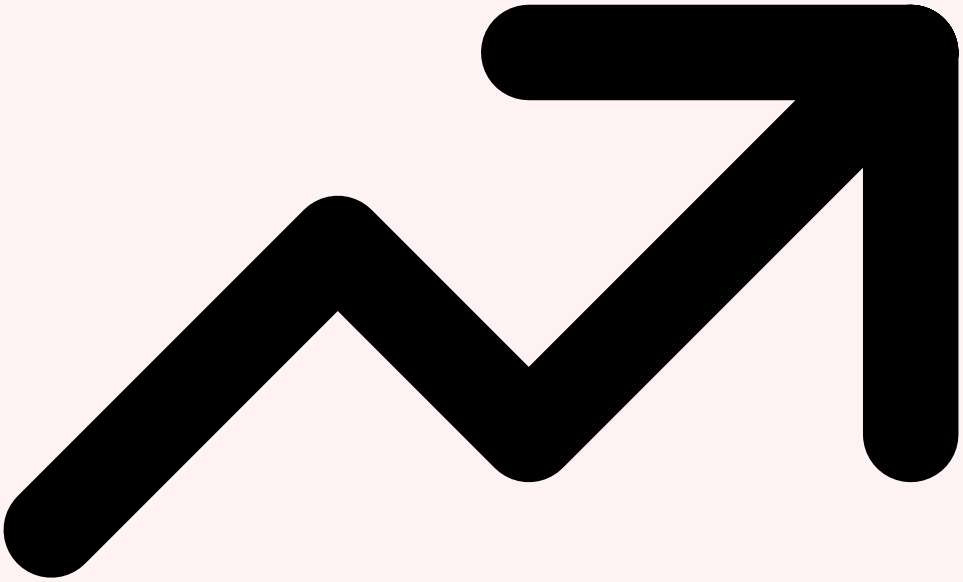


Model Serving KubeFlow & MLOps Autoscaling & Optimisation



6 Autoscaling et optimisation GPU

L'autoscaling des workloads GPU represente un defi unique par rapport aux workloads CPU classiques. Le provisionnement d'un nouveau noeud GPU peut prendre plusieurs minutes (telechargement d'images, initialisation des drivers NVIDIA, warm-up du modele), et chaque noeud GPU represente un cout significatif. Une strategie d'autoscaling efficace doit **anticiper la demande** plutot que simplement reagir, tout en minimisant le gaspillage de ressources couteuses.

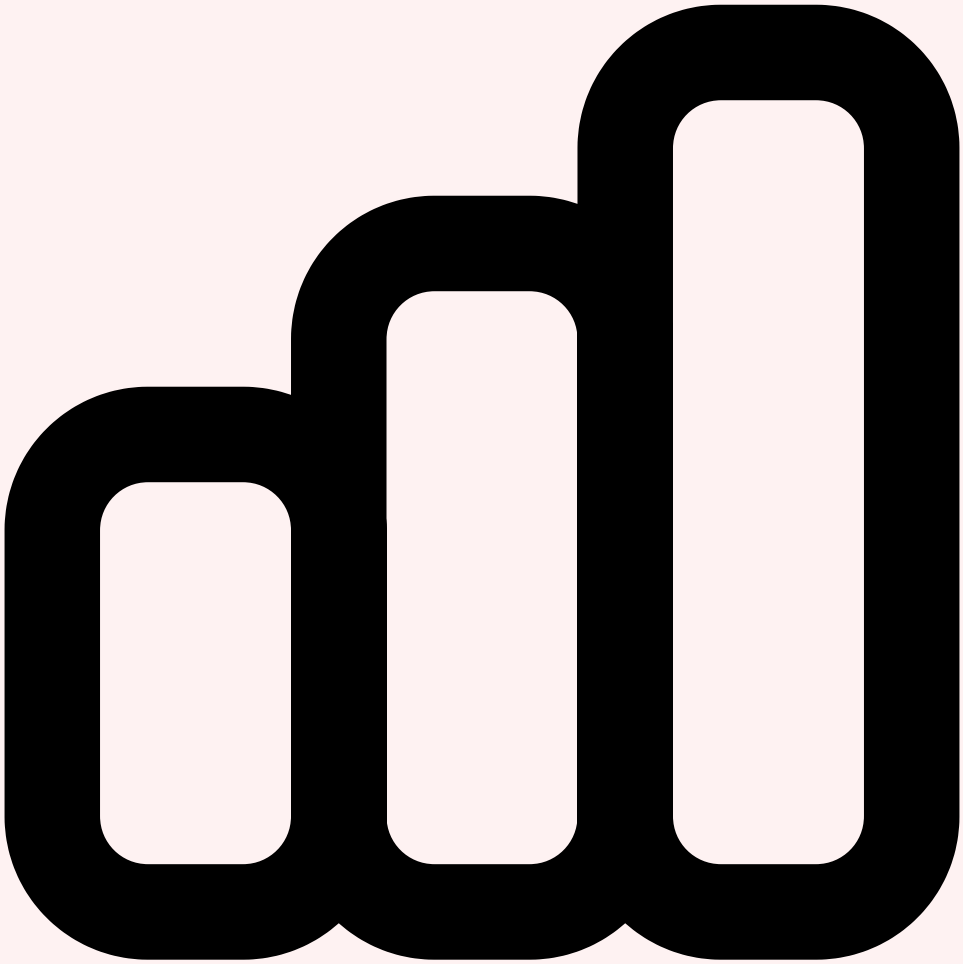


Horizontal Pod Autoscaler (HPA) pour le GPU

Le **HPA** natif de Kubernetes peut scaler les deploiements d'inference en se basant sur des metriques custom exposees par le DCGM Exporter ou les metriques applicatives. Cependant, les metriques CPU/memoire par defaut sont insuffisantes pour les workloads GPU. Il faut configurer des **metriques custom** via Prometheus Adapter pour exposer des metriques GPU pertinentes au HPA : utilisation GPU, profondeur de la file d'attente des requetes, latence P99, et throughput en tokens par seconde. Pour approfondir, consultez Agents IA Edge 2026 : Privacy, Latence et Architecture PLAM.

YAML

```
# HPA base sur les metriques GPU custom
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: vllm-hpa
  namespace: inference
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: vllm-llama3-70b
  minReplicas: 2
  maxReplicas: 12
  behavior:
    scaleUp:
      stabilizationWindowSeconds: 60
      policies:
      - type: Pods
        value: 2
        periodSeconds: 120  # max +2 pods toutes les 2 min
    scaleDown:
      stabilizationWindowSeconds: 300  # 5 min avant scale-down
      policies:
      - type: Pods
        value: 1
        periodSeconds: 180
  metrics:
  - type: Pods
    pods:
      metric:
        name: vllm_requests_running  # file d'attente vLLM
      target:
        type: AverageValue
        averageValue: "10"           # scale si > 10 req en cours
  - type: Pods
    pods:
      metric:
        name: dcgm_gpu_utilization   # utilisation GPU via DCGM
      target:
        type: AverageValue
        averageValue: "80"           # scale si GPU > 80%
```

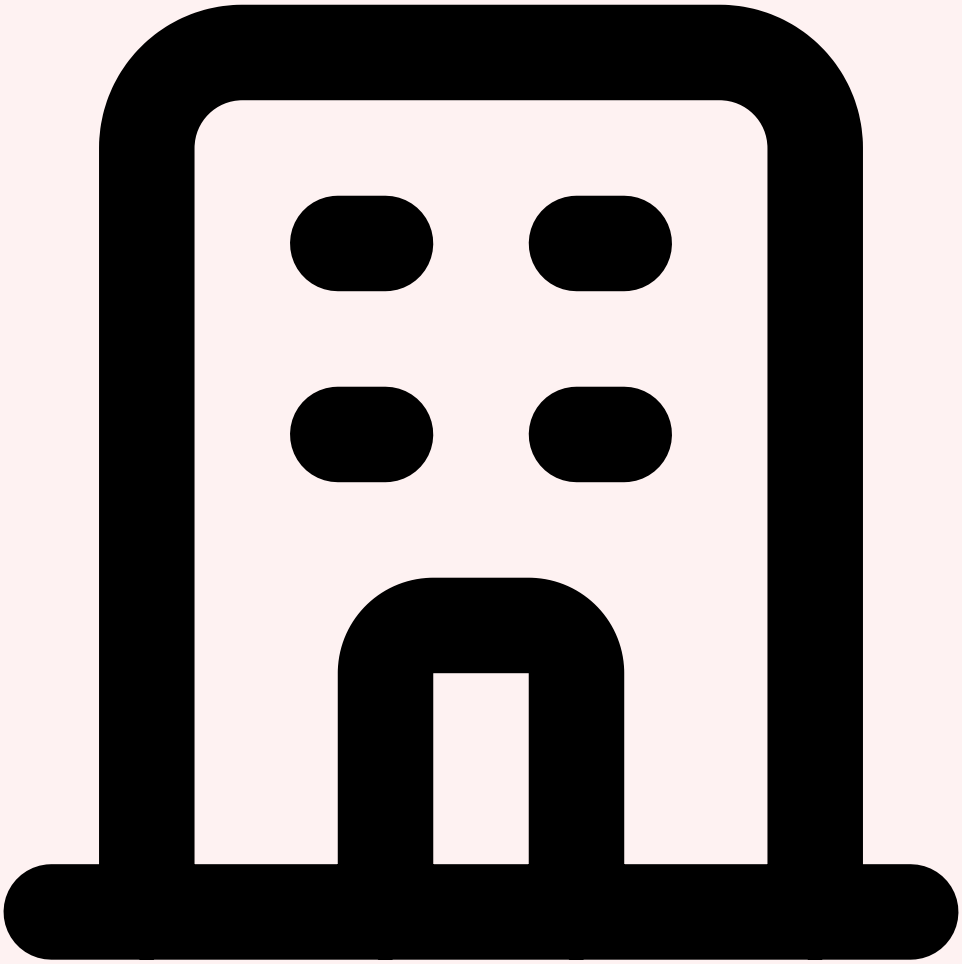


KEDA pour l'autoscaling evenementiel

KEDA (Kubernetes Event-Driven Autoscaling) va au-dela du HPA en offrant un scaling base sur des sources d'evenements externes. Pour les workloads d'inference IA, KEDA est particulierement precieux car il peut scaler en se basant sur la taille d'une file d'attente (RabbitMQ, Kafka, SQS), le nombre de requetes HTTP en attente, ou des metriques Prometheus custom. L'avantage majeur de KEDA est sa capacite de **scale-to-zero** : quand aucune requete n'arrive, KEDA peut reduire les replicas a zero, eliminant completement le cout GPU des modeles rarement utilises.

YAML

```
# KEDA ScaledObject pour inference evenementielle
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: vllm-scaler
  namespace: inference
spec:
  scaleTargetRef:
    name: vllm-llama3-70b
  minReplicaCount: 0     # Scale to zero !
  maxReplicaCount: 10
  cooldownPeriod: 300     # 5 min avant scale-down
  triggers:
  - type: prometheus
    metadata:
      serverAddress: http://prometheus:9090
      metricName: http_requests_pending
      query: |
        sum(rate(vllm_request_success_total{namespace="inference"}[2m]))
      threshold: "50"     # scale si > 50 req/min
  - type: rabbitmq
    metadata:
      queueName: inference-queue
      host: amqp://rabbitmq.default:5672
      queueLength: "5"    # scale si > 5 messages en attente
```



Karpenter : provisionnement GPU intelligent

Karpenter est l'autoscaler de noeuds de nouvelle generation qui remplace avantageusement le Cluster Autoscaler pour les workloads GPU. Contrairement au Cluster Autoscaler qui travaille avec des node groups predefinies et des tailles fixes, Karpenter selectionne **dynamiquement le type d'instance optimal** pour chaque pod en attente. Par exemple, si un pod necessite 2 GPU avec 48 GB de VRAM minimum, Karpenter peut choisir entre une instance p4d.24xlarge (8x A100), une g5.12xlarge (4x A10G), ou une g6.xlarge (1x L40S) en fonction du cout et de la disponibilite.

YAML

```
# Karpenter NodePool pour GPU inference
apiVersion: karpenter.sh/v1beta1
kind: NodePool
metadata:
  name: gpu-inference
spec:
  template:
    spec:
      requirements:
      - key: karpenter.k8s.aws/instance-family
        operator: In
        values: ["g5", "g6", "p4d"]
      - key: karpenter.k8s.aws/instance-gpu-count
        operator: Gt
        values: ["0"]
      - key: karpenter.sh/capacity-type
        operator: In
        values: ["on-demand", "spot"]
      - key: kubernetes.io/arch
        operator: In
        values: ["amd64"]
      nodeClassRef:
        name: gpu-node-class
    metadata:
      labels:
        gpu-pool: inference
  limits:
    nvidia.com/gpu: 32    # max 32 GPU dans ce pool
    cpu: "512"
  disruption:
    consolidationPolicy: WhenUnderutilized
    consolidateAfter: 10m  # consolider apres 10 min
```

Optimisation des couts GPU avec les instances Spot

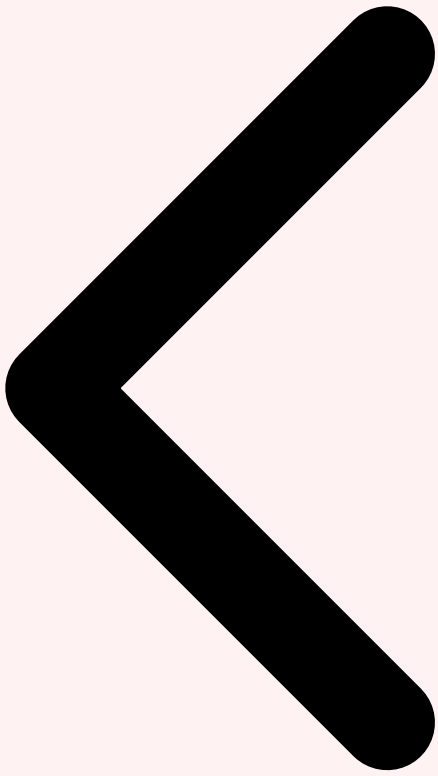
Les **instances GPU spot** (AWS) ou **preemptible** (GCP) offrent des reductions de 60 a 90% par rapport aux instances on-demand. Pour les jobs d'entrainement tolerants aux interruptions, c'est une strategie d'economie majeure. La cle est d'implementer un **checkpointing robuste** : sauvegarder l'etat de l'entrainement toutes les N etapes dans un stockage persistant (S3, GCS) et reprendre automatiquement depuis le dernier checkpoint en cas d'interruption.

▹**Entrainement** : utilisez les instances spot avec checkpointing toutes les 15-30 minutes. Le surcoute de checkpointing est negligeable compare a l'economie realisee.

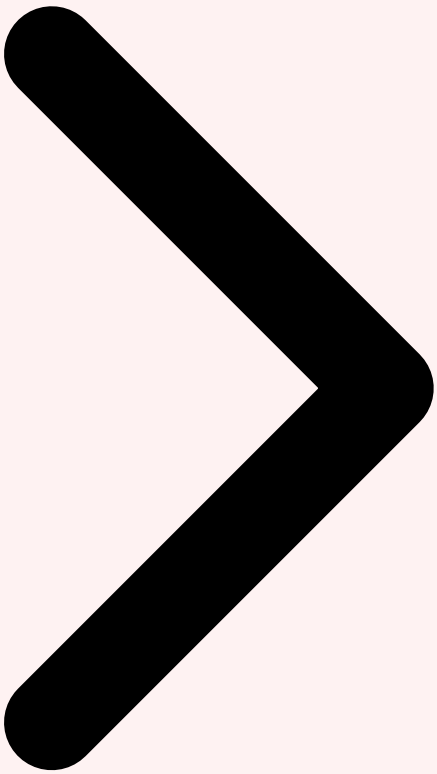
▹**Inference non-critique** : les services internes ou les environnements de staging peuvent tourner sur spot. Utilisez les PDB (Pod Disruption Budgets) pour garantir un minimum de replicas disponibles.

▹**Inference critique** : restez sur on-demand pour les services de production avec SLA. Utilisez Karpenter en mode mixte (on-demand base + spot burst) pour absorber les pics.

Piege du scale-to-zero GPU : Le cold start d'un serveur d'inference LLM peut prendre 2 a 5 minutes (telechargement du modele + chargement en VRAM). Le scale-to-zero n'est recommande que pour les modeles rarement utilises. Pour les services avec un trafic regulier, maintenez au minimum 1 replica on-demand pour eviter les timeouts client.

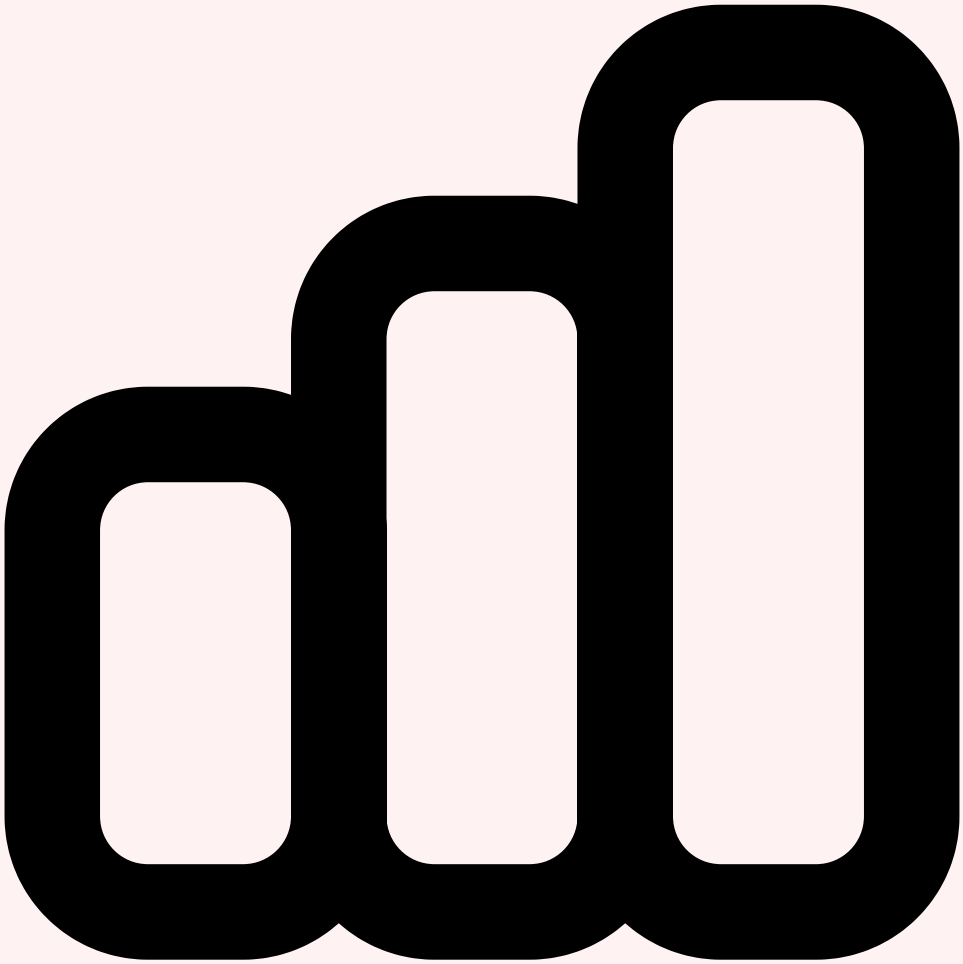


KubeFlow & MLOps Autoscaling & Optimisation Bonnes Pratiques



7 Bonnes pratiques et production

Deployer des workloads GPU sur Kubernetes en production va bien au-dela de la simple configuration technique. Il faut considerer le **monitoring GPU granulaire**, la **gestion des couts**, la **securite des modeles** et la **multi-tenancy** entre equipes. Cette section consolide les lecons apprises sur des clusters GPU de production a grande echelle.



Monitoring GPU avec DCGM et Prometheus

Le **DCGM Exporter** (Data Center GPU Manager) est l'outil officiel NVIDIA pour exposer les metriques GPU vers Prometheus. Il collecte plus de 50 metriques par GPU, incluant l'utilisation des SM, la temperature, la consommation electrique, les erreurs ECC, et l'utilisation memoire. Les metriques critiques a monitorer en production sont les suivantes.

▹**DCGM_FI_DEV_GPU_UTIL** : pourcentage d'utilisation des SM. En inference, visez 60-80%. En entrainement, visez > 90%.

▹**DCGM_FI_DEV_FB_USED / DCGM_FI_DEV_FB_FREE** : memoire GPU utilisee/libre. Un manque de memoire libre indique un risque d'OOM.

▹**DCGM_FI_DEV_GPU_TEMP** : temperature GPU. Alertez au-dessus de 85 degres Celsius, car le thermal throttling reduit les performances.

▹**DCGM_FI_DEV_POWER_USAGE** : consommation electrique. Un A100 consomme jusqu'a 400W TDP, un H100 jusqu'a 700W. Essentiel pour le capacity planning electrique du datacenter.

▹**DCGM_FI_DEV_XID_ERRORS** : erreurs GPU (XID errors). Les erreurs XID 48 et 63 indiquent des problemes memoire necessitant un remplacement du GPU.

YAML

```
# Alertes Prometheus pour GPU en production
groups:
- name: gpu-alerts
  rules:
  - alert: GPUTemperatureCritical
    expr: DCGM_FI_DEV_GPU_TEMP > 85
    for: 5m
    labels:
      severity: critical
    annotations:
      summary: "GPU temperature critical sur {{ $labels.gpu }}"
  - alert: GPUMemoryAlmostFull
    expr: (DCGM_FI_DEV_FB_USED / (DCGM_FI_DEV_FB_USED + DCGM_FI_DEV_FB_FREE)) > 0.95
    for: 2m
    labels:
      severity: warning
    annotations:
      summary: "GPU memory > 95% sur {{ $labels.gpu }}"
  - alert: GPUUtilizationLow
    expr: DCGM_FI_DEV_GPU_UTIL < 20
    for: 30m
    labels:
      severity: info
    annotations:
      summary: "GPU sous-utilise (< 20%) - verifier le workload"
  - alert: GPUXidError
    expr: DCGM_FI_DEV_XID_ERRORS > 0
    for: 1m
    labels:
      severity: critical
    annotations:
      summary: "Erreur XID detectee sur GPU {{ $labels.gpu }}"
```

Gestion des couts et FinOps GPU

La **gestion des couts GPU** est un enjeu critique pour les organisations deployant de l'IA a grande echelle. Un cluster de 8 nœuds DGX H100 represente un investissement de plus de 2 millions d'euros, et le cout operationnel (electricite, refroidissement, maintenance) ajoute 20-30% par an. Les outils comme **Kubecost** ou **OpenCost** permettent d'attribuer les couts GPU a chaque equipe, projet ou namespace, creant une visibilite indispensable pour la gouvernance financiere.

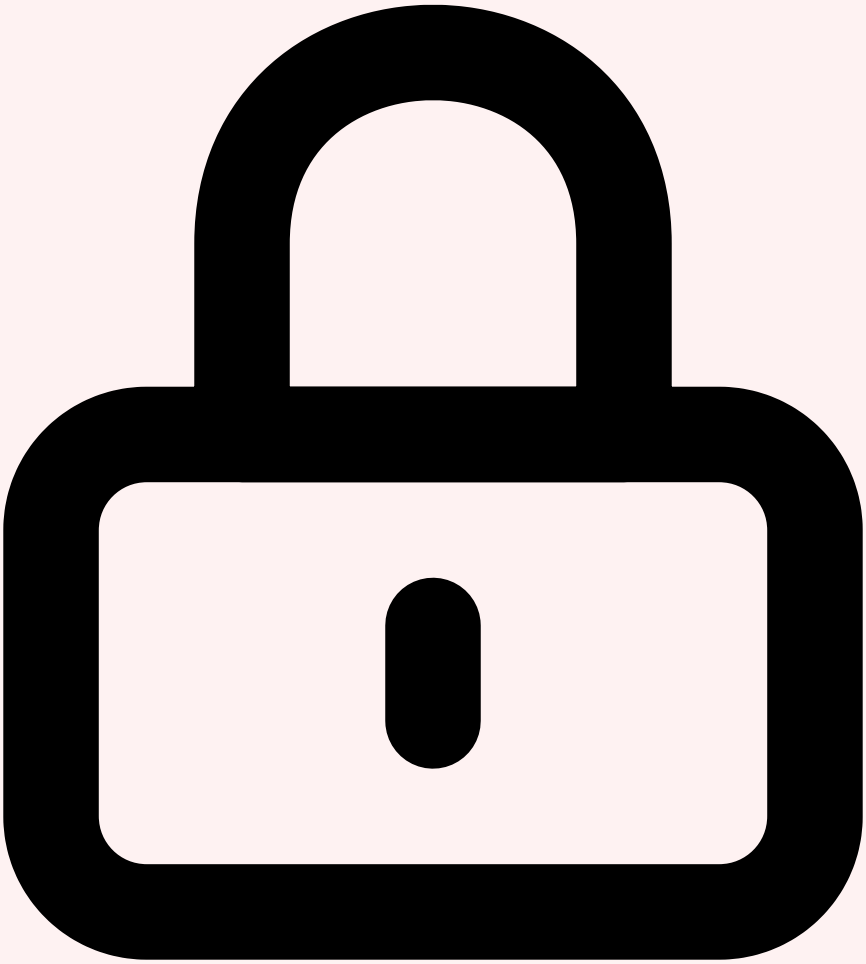
▹**Quotas par namespace** : definissez des ResourceQuotas limitant le nombre de GPU par equipe pour eviter la monopolisation.

▹**Priority Classes** : configurez des priorites pour que les workloads de production puissent premter les jobs de developpement en cas de contention. Pour approfondir, consultez La Fin des Moteurs.

▹**Idle detection** : alertez sur les GPU inactifs depuis plus de 30 minutes. Un GPU A100 inactif coute environ 3 euros par heure sur AWS.

YAML

```
# ResourceQuota GPU par namespace
apiVersion: v1
kind: ResourceQuota
metadata:
  name: gpu-quota
  namespace: team-nlp
spec:
  hard:
    requests.nvidia.com/gpu: "8"     # max 8 GPU pour cette equipe
    limits.nvidia.com/gpu: "8"
    persistentvolumeclaims: "20"
---
# PriorityClass pour workloads de production
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: production-inference
value: 1000000
globalDefault: false
preemptionPolicy: PreemptLowerPriority
description: "Priorite maximale pour l'inference en production"
```



Securite des modeles et des donnees

La securite des workloads GPU sur Kubernetes necessite une attention particuliere. Les modeles d'IA representent une propriete intellectuelle considerable, et les donnees d'entrainement peuvent contenir des informations sensibles. Les bonnes pratiques incluent l'application de **Network Policies** strictes pour isoler les namespaces d'entrainement et d'inference, l'utilisation de **Pod Security Standards** restrictives (pas de conteneurs privilegies sauf le device plugin), le chiffrement des modeles au repos dans le stockage, et l'authentification mTLS via un service mesh pour les communications entre le serveur d'inference et les clients.

▹**RBAC granulaire** : limitez l'acces aux CRDs de training et d'inference par role. Un data scientist ne devrait pas pouvoir modifier les InferenceServices de production.

▹**Image scanning** : les images de ML contiennent souvent des dependances vulnerables (anciennes versions de CUDA, bibliothèques Python). Integrez Trivy ou Snyk dans votre pipeline CI/CD.

▹**Secrets management** : les tokens d'acces HuggingFace, les credentials cloud et les clés API doivent etre geres via des solutions comme HashiCorp Vault ou AWS Secrets Manager, jamais en dur dans les manifests.

Checklist de production

Avant de mettre un cluster GPU Kubernetes en production, validez les points suivants :

▹**GPU Health Checks** : implementez des liveness et readiness probes specifiques GPU qui verifient que le device est fonctionnel et que le modele repond correctement.

▹**Pod Disruption Budgets** : definissez des PDB pour garantir un minimum de replicas pendant les mises a jour ou les drains de noeuds.

▹**Preloading des modeles** : utilisez des init containers ou des DaemonSets pour precacher les modeles sur les noeuds, reduisant le cold start de minutes a secondes.

▹**Graceful shutdown** : configurez un terminationGracePeriodSeconds suffisant (120-300s) pour permettre aux requetes en cours de se terminer avant l'arret du pod.

▹**Disaster recovery** : sauvegardez regulierement les configurations KubeFlow, les pipelines et les modeles dans un stockage externe. Testez la restauration complete du cluster GPU trimestriellement.

Conclusion : Kubernetes est devenu la plateforme incontournable pour orchestrer les workloads d'IA en production. La combinaison du NVIDIA Device Plugin, de MIG/time-slicing, des frameworks de serving comme vLLM et Triton, et des outils MLOps comme KubeFlow permet de construire des pipelines IA robustes, scalables et economiquement viables. La cle du succes reside dans une approche progressive : commencez par un pool d'inference simple, puis ajoutez l'entrainement distribue, le monitoring GPU et l'autoscaling au fur et a mesure que vos besoins evoluent.

Besoin d'un accompagnement expert ?

Nos consultants en cybersécurité et IA vous accompagnent dans vos projets. Devis personnalisé sous 24h.

Références et ressources externes

- MITRE ATT&CK T1610 — Deploy Container
- vLLM — Moteur d'inférence LLM haute performance
- llama.cpp — Inférence LLM optimisée en C/C++
- MLflow — Plateforme open source de gestion du cycle de vie ML
- Kubernetes Docs — Documentation officielle Kubernetes

Pour approfondir ce sujet, consultez notre outil open-source ai-threat-detection qui facilite la détection de menaces basée sur l'IA.

Sources et références : ArXiv IA · Hugging Face Papers

FAQ

Qu'est-ce que Kubernetes pour l'IA ?

Le concept de Kubernetes pour l'IA est détaillé dans les premières sections de cet article, qui couvrent les fondamentaux, les enjeux et le contexte opérationnel. Pour un accompagnement sur ce sujet, contactez nos experts.

Pourquoi Kubernetes pour l'IA est-il important en cybersécurité ?

La compréhension de Kubernetes pour l'IA permet aux équipes de sécurité d'améliorer leur posture défensive. Les sections « Table des Matières » et « 1 Introduction : Kubernetes et l'ère de l'IA » détaillent les raisons de cette importance. Pour un accompagnement sur ce sujet, contactez nos experts.

Comment mettre en œuvre les recommandations de cet article ?

Les recommandations pratiques sont détaillées tout au long de l'article, avec des commandes, des outils et des méthodologies éprouvées. La section « Conclusion » fournit une synthèse actionnable. Pour un accompagnement sur ce sujet, contactez nos experts.

Conclusion

Cet article a couvert les aspects essentiels de Table des Matières, 1 Introduction : Kubernetes et l'ère de l'IA, 2 Architecture GPU Cluster Kubernetes. La mise en pratique de ces recommandations permet de renforcer significativement la posture de securite de votre organisation.

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr