VMware Cloud Foundation

Deploying NVIDIA Run:ai on VMware Cloud Foundation

White Paper

Table of Contents

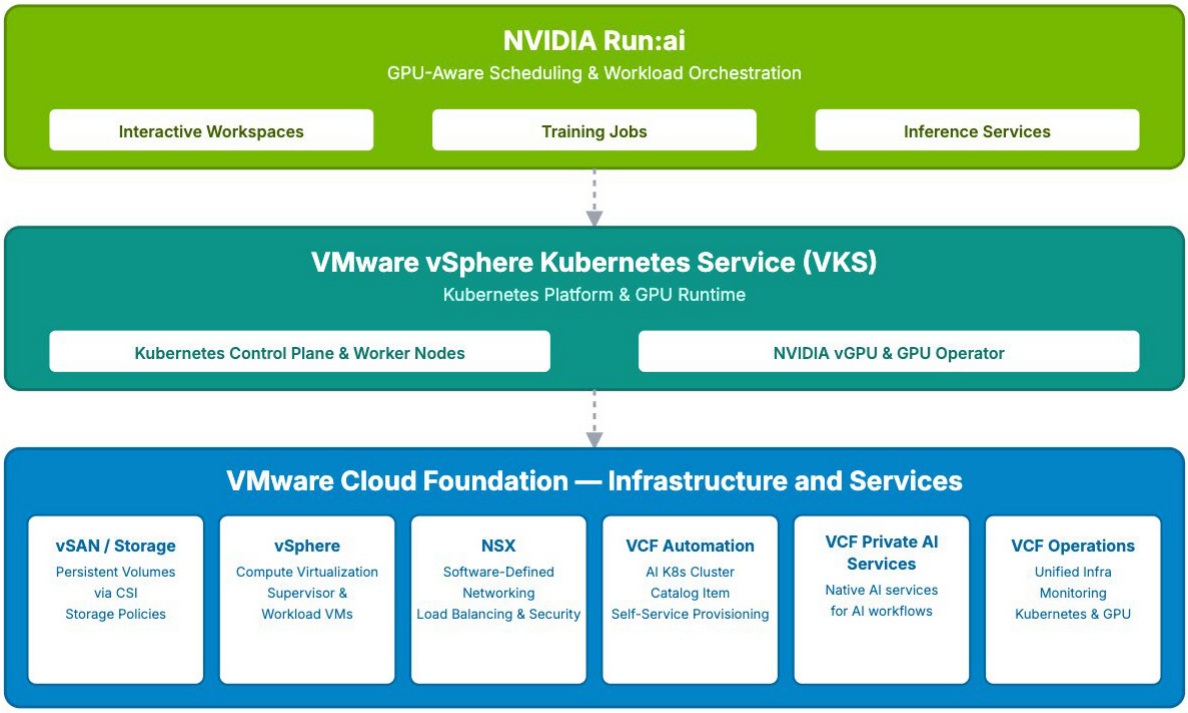
Chapter 1: Introduction 5
1.1 VCF and vSphere VKS 5
1.2 VMware Private AI Foundation with NVIDIA 6
1.3 NVIDIA Run:ai 6
1.4 Why This Matters for NVIDIA AI Enterprise Customers 7
Chapter 2: Reference Architecture 8
2.1 Infrastructure and Services Layer: VCF 8
2.2 Kubernetes and GPU Layer: VKS 9
2.3 Workload Orchestration Layer: NVIDIA Run:ai 9
2.4 Tested Configuration 10
2.5 Network and DNS Topology 11
2.5.1 Internal (In-Cluster) Ingress 11
2.5.2 External Networking (NSX, DNS, and Outbound Access) 12
2.6 Identity and RBAC 12
2.7 Storage 12
2.8 Topology and Flexibility 13
2.9 Multi-tenancy and Tenant Isolation on VCF 14
Chapter 3: Prerequisites 16
Chapter 4: Deployment Steps 17
4.1 Provision the VKS Clusters 17
4.2 Verify the NVIDIA GPU Operator 18
4.3 Install the Ingress Controller (HAProxy) and Create the TLS Secrets 19
4.3.1 Set the Default StorageClass 19
4.3.2 Install the HAProxy Ingress Controller 19
4.3.3 Generate the TLS Material 19
4.3.4 Pod Security Admission Notes 20
4.4 Install Prometheus on the Workload Cluster 20
4.5 Install the NVIDIA Run:ai Self-Hosted Control Plane 21
4.6 Register the Cluster and Install the NVIDIA Run:ai Cluster Component 22
4.7 Install Knative Serving for Inference 25
4.7.1 Create the namespaces 25
4.7.2 Install the Knative Operator 25
4.7.3 Create the Wildcard TLS Secret 25
4.7.4 Apply the KnativeServing Custom Resource 25
4.7.5 Front Kourier with HAProxy 26
4.7.6 Enable Inference in the NVIDIA Run:ai Cluster 27
Chapter 5: Functional Testing 28

5.1 CLI Login and Project Setup 28
5.2 NVIDIA Run:ai GPU Fractions 29
5.3 Over-Quota Borrowing and Fair Share across Departments 30
5.4 Inference Test: Small Language Model with vLLM 32
5.5 Workspace Test: Interactive JupyterLab on GPU 33
5.6 5.6 Training Test: MNIST with PyTorch 34
Chapter 6: Monitoring **37**
6.1 NVIDIA Run:ai UI Dashboards 37
6.1.1 Overview Dashboard 37
6.1.2 Clusters View 38
6.1.3 Nodes View 38
6.1.4 Workloads View 38
6.1.5 Departments and Projects 39
6.1.6 GPU Utilization Metrics 39
6.1.7 Workload-Level Metrics 40
6.1.8 Workload Logs in the UI 41
6.2 NVIDIA Run:ai CLI 41
6.2.1 Cluster and Node Status 41
6.2.2 Department and Project Allocation 42
6.2.3 Workload Status 42
6.2.4 Workload Logs from the CLI 43
6.3 VCF Operations 43
6.3.1 VKS Cluster Dashboard 44
6.3.2 Private AI Dashboards 44
Chapter 7: Conclusion **46**
Chapter 8: Resources **47**

Chapter 1: Introduction

Enterprises running AI on-premises need a private cloud that hosts both traditional applications and GPU-accelerated AI through a single management plane. VMware Cloud Foundation® (VCF) is that private cloud; paired with VMware vSphere® Kubernetes Service (VKS) as its native Kubernetes runtime and VMware® Private AI Foundation with NVIDIA, it becomes a GPU-ready AI platform. NVIDIA Run:ai builds on that foundation, providing AI workload and GPU orchestration that helps platform teams improve utilization of shared GPU resources to support more concurrent users, AI workloads, agentic workflows and inference services.

Figure 1: VCF and VMware Private AI Foundation with NVIDIA



1.1 VCF and vSphere VKS

VCF is the Broadcom® integrated private-cloud platform. A single deployment bundles VMware vSphere® for compute, vSAN (or a supported external array) for storage, and VMware NSX® (NSX) for networking, all managed through VMware Cloud Foundation® Operations (VCF Operations). Compute, storage, networking, and Kubernetes are provisioned, upgraded, and patched as one system rather than four loosely coupled products.

VKS is the Kubernetes runtime that ships natively with VCF. Clusters are provisioned directly from vSphere through the Supervisor, a Kubernetes control plane that runs on the ESXi hypervisor layer and serves as the management plane for all VKS workload clusters, vSphere Namespaces, and Supervisor Services. No separate Kubernetes distribution is needed, no external control plane to operate, and no extra licensing. Worker nodes are vSphere VMs, so GPUs attached with NVIDIA vGPU or with (Dynamic) DirectPath I/O are consumed by Kubernetes through the same vSphere scheduler, with the vMotion and DRS behavior operators already understand. The Kubernetes layer is not a new silo; it is part of the same cluster the platform team already manages.

1.2 VMware Private AI Foundation with NVIDIA

VMware Private AI Foundation with NVIDIA is the joint Broadcom and NVIDIA platform for running AI on VCF.

This platform enables enterprises to fine-tune LLM models, deploy RAG workflows, and run inference workloads in their data centers, addressing privacy, choice, cost, performance, and compliance concerns. This joint platform simplifies AI deployments for enterprises by offering Model Store, Air-Gapped Support, Model runtime, NVIDIA NIMTM, NVIDIA Blueprints, and more.

Built and run on the industry-leading private cloud platform, VCF, Private AI Foundation with NVIDIA includes VCF Private AI Services, NVIDIA AI Enterprise, NVIDIA NIM inference microservices for the latest AI models — including NVIDIA Nemotron models and leading community models — and NVIDIA Blueprints.

It brings together the components a GPU-ready VKS cluster needs, validates them as a whole, and exposes them to tenants as self-service catalog items through VMware Cloud Foundation® Automation (VCF Automation). The AI blueprints are VCF Automation catalog items including the AI Kubernetes Cluster Automation Template, which is used throughout this paper. Requesting an AI Kubernetes Cluster drives the Supervisor to provision a VKS cluster with the right VM class, GPU profile, content library, and add ons (including the NVIDIA GPU Operator and licensing), so the cluster comes up with drivers loaded, the device plugin advertising GPUs, and Data Center GPU Manager (DCGM) metrics flowing.

What Kubernetes does not provide on its own is a workload and GPU scheduler aware of users, projects, quotas, fairness, preemption, and inference autoscaling. That is where NVIDIA Run:ai fits.

1.3 NVIDIA Run:ai

NVIDIA Run:ai is a Kubernetes-native AI workload and GPU orchestration platform for shared AI infrastructure. It gives platform teams a control plane for organizing users, departments, projects, quotas, policies, and GPU capacity, while giving AI practitioners a consistent way to submit, run, and monitor workloads across the AI life cycle. NVIDIA Run:ai adds the following:

- GPU-aware scheduling: Projects, departments, quotas, over-quota and fair-share, priority, preemption, node pools, and fractional-GPU sharing, so multiple teams can share the same accelerators without stepping on each other.
- Workload abstractions: First-class objects for interactive workspaces, training jobs, distributed training (PyTorch, TensorFlow, MPI, XGBoost), and inference services, usable from the UI, the `runai` CLI, the REST API, or raw Kubernetes Custom Resources CRs.
- Composable workloads: Workloads assembled from reusable assets (data sources, credentials, environments, compute templates, policies). The same assets plug into a Workspace today and a Training job or Inference service tomorrow without rewriting the spec.
- Custom workspaces and environments: Admins publish curated environments (Jupyter, VS Code, RStudio, custom CUDA images, vLLM, Triton, NIM) that researchers pick from a drop-down.
- Inference serving: Inference workloads run on Knative Serving and gain request-driven autoscaling, scale-to-zero, revision-based rollout, and a stable OpenAI-style endpoint without hand-written Knative configuration.
- Governance and observability: SSO/OIDC, RBAC at department and project scope, policies, audit logs, and GPU/workload metrics in both the NVIDIA Run:ai UI and Prometheus.

NVIDIA Run:ai is offered in two deployment models: NVIDIA-hosted SaaS and self-hosted. Both models install a lightweight cluster component on each attached Kubernetes cluster to run the scheduler, admission webhooks, and workload controllers next to the workloads. This paper focuses on the self-hosted model, the common choice for VCF and Private AI Foundation with NVIDIA customers who want the control plane inside their own private cloud.

1.4 Why This Matters for NVIDIA AI Enterprise Customers

NVIDIA Run:ai is now part of NVIDIA AI Enterprise. Any customer with an NVIDIA AI Enterprise entitlement can deploy NVIDIA Run:ai without a separate NVIDIA Run:ai license. For a VCF customer who already runs Private AI Foundation with NVIDIA, that simplifies procurement and deployment:

- The GPU platform is already there: VCF, VKS, and Private AI Foundation with NVIDIA's AI Kubernetes Cluster catalog item with the GPU Operator and licensed driver stack out of the box.
- The NVIDIA Run:ai components are already entitled: NVIDIA AI Enterprise + Private AI Foundation with NVIDIA brings the vGPU host driver, Deep Learning VM images, and NVIDIA AI Enterprise-supported runtimes (CUDA, PyTorch, TensorFlow, NIM).
- What remains is a deployment exercise: Install the NVIDIA Run:ai control plane, attach the VKS cluster, wire up Knative Serving for inference, verify the stack.

The bigger story is consolidation. NVIDIA has folded the Kubernetes-native GPU scheduling and workload-orchestration layer into the entitlement that already sits on top of the VCF infrastructure layer. The decision tree collapses: instead of weighing NVIDIA Run:ai against KAI Scheduler (an open-source, Kubernetes-native GPU scheduler and current CNCF Sandbox Project that was initially created for, and still powers, the NVIDIA Run:ai platform), plain kube-scheduler with the GPU Operator, or third-party schedulers, the default becomes NVIDIA Run:ai.

This paper covers the foundational deployment of NVIDIA Run:ai on VCF with Private AI Foundation with NVIDIA:

1. Bring up the NVIDIA Run:ai self-hosted control plane on a VKS cluster.
2. Register one or more GPU-enabled VKS clusters with the control plane.
3. Install Knative Serving for inference endpoints.
4. Validate fractional-GPU sharing.
5. Validate over-quota and fair-share scheduling across departments.
6. Run a single-GPU inference service.
7. Run an interactive workspace.
8. Run a single-GPU training job.
9. Set up monitoring through the NVIDIA Run:ai UI, the runai CLI, and VCF Operations.

This paper does not address the following procedures:

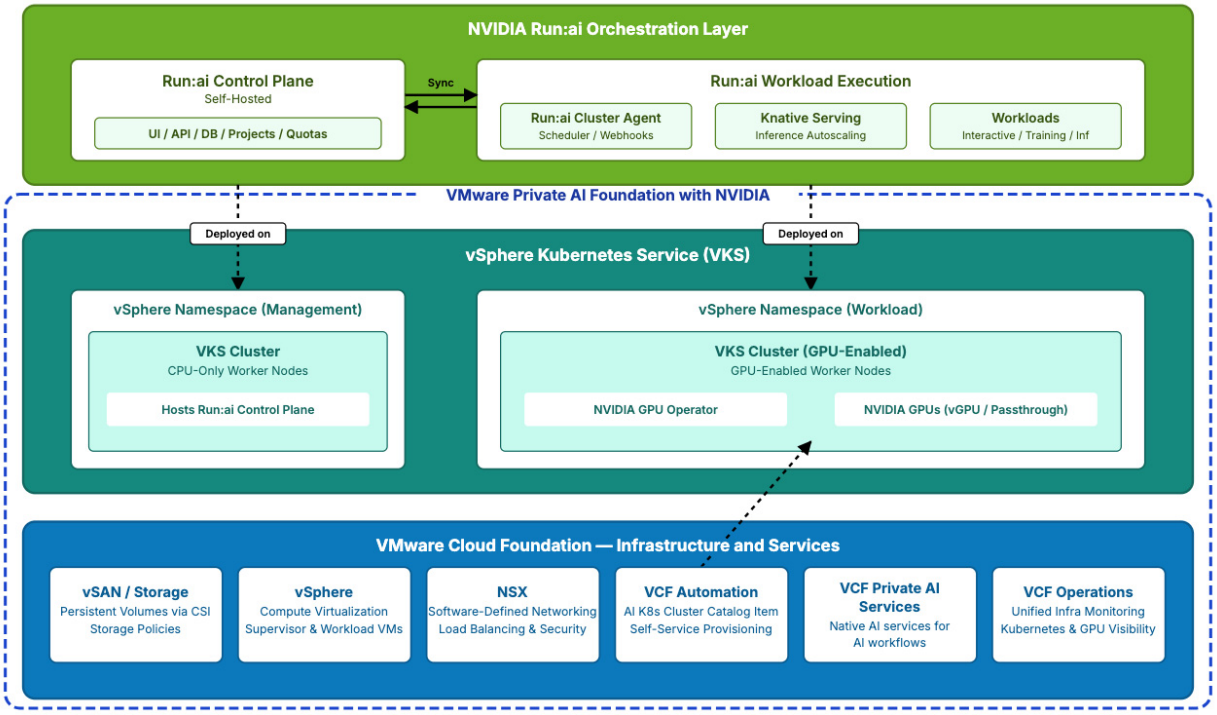
1. Knative request-driven autoscaling and scale-to-zero.
2. Distributed training across multiple nodes (PyTorch DDP, DeepSpeed, MPI).

The remainder of this document assumes the reader has VCF 9.0 with Private AI Foundation with NVIDIA running, an NVIDIA AI Enterprise entitlement, and at least one AI Kubernetes Cluster provisioned from the VCF Automation catalog, and is ready to add NVIDIA Run:ai on top.

Chapter 2: Reference Architecture

The reference architecture is a three-layer stack: VCF with Private AI Foundation with NVIDIA at the bottom (compute, storage, networking, vSphere namespaces, VKS clusters) and NVIDIA Run:ai on top (control plane, cluster components, Knative Serving, workloads). Figure 2 shows the pieces.

Figure 2: NVIDIA Run:ai on VCF with Private AI Foundation with NVIDIA



2.1 Infrastructure and Services Layer: VCF

The bottom layer of the architecture houses a GPU-enabled VCF deployment.

- vSphere virtualizes compute and exposes GPUs to the upper layers as vGPU profiles or as whole devices via Dynamic Direct Path I/O / Direct Path I/O.
- NSX provides software-defined networking, distributed routing, and the load balancers used by VKS clusters and NVIDIA Run:ai endpoints.
- vSAN (or a supported external array surfaced through the vSphere Container Storage Plugin (CSI) provides the persistent volumes consumed by NVIDIA Run:ai's stateful components, the control-plane database, and any persistent workload data.

Nothing here is AI-specific. It is the same VCF that hosts the customer's other workloads, run by the same operations team.

2.2 Kubernetes and GPU Layer: VKS

VKS does not introduce a separate Kubernetes distribution; it consumes the vSphere Supervisor that ships with VCF, and uses VCF Automation to publish self-service catalog items that drive the Supervisor on the tenant's behalf.

This paper assumes two vSphere namespaces, each with its own VKS cluster:

- Management namespace: A CPU-only VKS cluster that contains the NVIDIA Run:ai self-hosted control plane (UI, API, Postgres, Keycloak/OIDC, internal services). Keeping the control plane off the GPU nodes avoids contention with researcher jobs and lets the platform team upgrade it independently.
- Workload namespace: A GPU-enabled VKS cluster for data-science and inference workloads, provisioned from the AI Kubernetes Cluster catalog item. The cluster comes up with the GPU Operator preinstalled, drivers loaded, the device plugin advertising `nvidia.com/gpu`, and NVIDIA Data Center GPU Manager (DCGM) metrics flowing.

2.3 Workload Orchestration Layer: NVIDIA Run:ai

NVIDIA Run:ai is deployed across the two VKS clusters from the previous layer:

- Control plane on the CPU-only cluster, installed via the NVIDIA Run:ai self-hosted Helm chart. System of record for projects, departments, quotas, users, roles, and policies. Reachable through the UI, the `runai` CLI, and the REST API.
- Cluster component on the GPU-enabled cluster, also via Helm. Hosts the NVIDIA NVIDIA Run:ai scheduler, admission webhooks, and workload controllers. Connects to the control plane over HTTPS and registers as a managed cluster.
- Knative Serving on the GPU-enabled cluster, the runtime NVIDIA Run:ai uses for inference workloads. Provides request-driven autoscaling, scale-to-zero, revision-based rollouts, and a stable URL per service. NVIDIA Run:ai generates the Knative `Service` objects from a higher-level `InferenceWorkload` spec.
- Researcher-facing workloads (Workspaces, Training jobs, Distributed Training, Inference services) are submitted against the control plane and scheduled onto the GPU-enabled cluster.

2.4 Tested Configuration

Table 1 details the components used in this exercise. Use the published NVIDIA Run:ai, GPU Operator, and VCF compatibility matrices as the source of truth for production. The components below were cross-checked against the NVIDIA Run:ai 2.24 Cluster System Requirements and Support Matrix; Kubernetes, GPU Operator, Knative, and Prometheus are all in supported ranges.

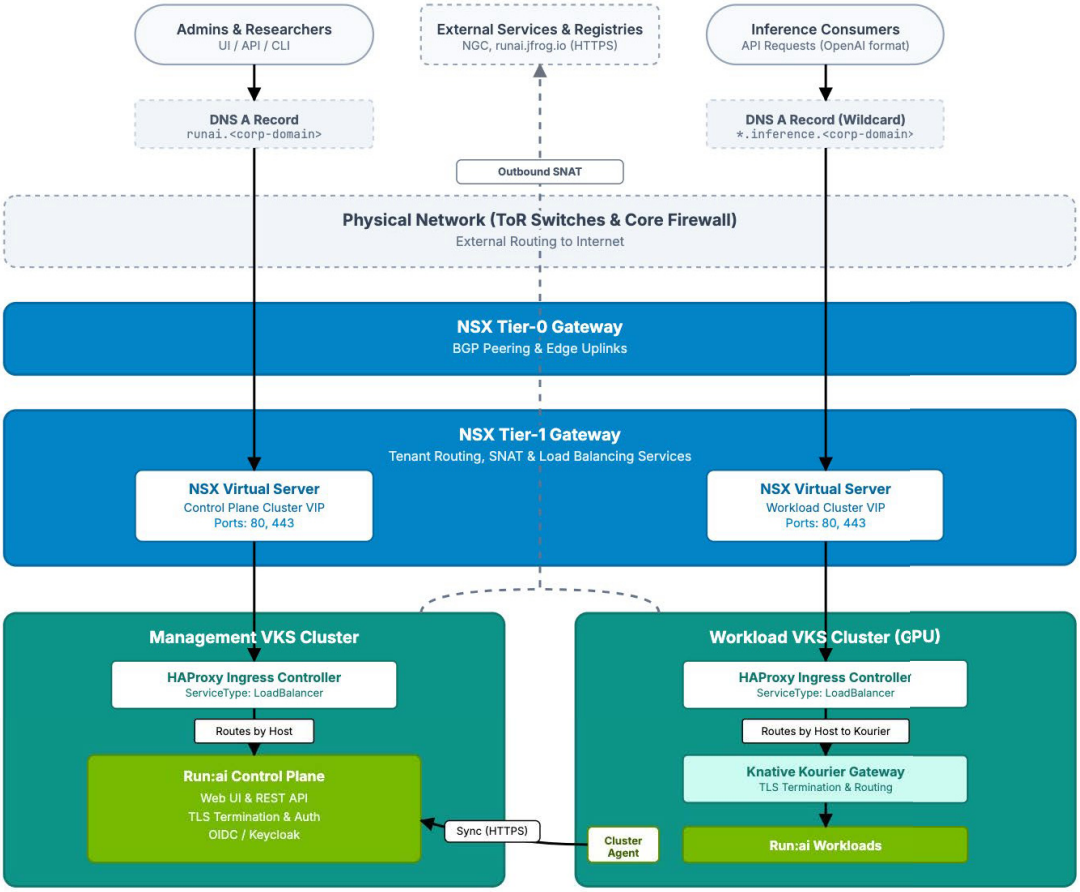
Table 1: Validated Testbed Configuration and Component Versions

Layer	Component	Version
Infrastructure	VCF	9.0
	vSphere / ESXi	9.0
	NSX	9.0
	vSAN	9.0
Kubernetes	vSphere Supervisor	v1.30.5+vmware.4-fips-vsc9.0.0.0-24686447
	VKS (Tanzu Kubernetes release)	v1.34.2+vmware.2
	VKS ClusterClass	builtin-generic-v3.5.0
	Node OS image	Ubuntu 24.04.3 LTS
GPU stack	NVIDIA AI Enterprise entitlement	NVIDIA AI Enterprise 6.x
	NVIDIA vGPU guest driver	580.105.08
	NVIDIA GPU Operator	v25.10.1
	NVIDIA k8s device plugin	v0.18.1
	NVIDIA DCGM Exporter	4.4.2-4.7.0
Orchestration	NVIDIA Run:ai (self-hosted control plane)	2.24.66
	NVIDIA Run:ai (cluster)	2.24.66
	Knative Operator	v1.18.2
	Knative Serving	1.18.1
	Kourier (via net-kourier)	1.18.x
VKS Cluster Add-ons	HAProxy Ingress Controller	chart 1.49.x (haproxytech/kubernetes-ingress)
	Prometheus (kube-prometheus-stack)	83.2.0 (Prometheus v3.11.1)
Storage	vSphere CSI driver	shipped with VKS 1.34

2.5 Network and DNS Topology

Figure 3 shows the end-to-end traffic flow from external clients through NSX and HAProxy into the NVIDIA Run:ai control plane and inference endpoints.

Figure 3: Network and DNS Topology: Inbound and Outbound Routing Paths



2.5.1 Internal (In-Cluster) Ingress

Ingress is a two-part stack:

- HAProxy as the in-cluster controller
- NSX as the external L4 load balancer

The `haproxytech/kubernetes-ingress` controller is installed via Helm on each VKS cluster and exposed through a `Service` of type `LoadBalancer`. NSX provisions a virtual server, assigns an external IP from the cluster's LB pool (referred to as `<haproxy-lb-ip>` throughout this paper), and forwards `:80` and `:443` to the HAProxy pods. HAProxy routes by host and path into the rest of the cluster: the NVIDIA Run:ai control-plane Services on the management cluster, the Knative `kourier` gateway on the workload cluster.

HAProxy is used because NVIDIA Run:ai 2.24 now recommends it. The upstream NGINX Ingress Controller (`kubernetes/ingress-nginx`) has been retired and NVIDIA Run:ai's `global.ingress.ingressClass` defaults to `haproxy` for new installs. The Kubernetes `Ingress` standard is unchanged, so the NVIDIA Run:ai charts are agnostic to which controller is in use as long as the `IngressClass` matches.

2.5.2 External Networking (NSX, DNS, and Outbound Access)

NVIDIA Run:ai docs recommend MetalLB to back `LoadBalancer` Services in self-hosted on-premises deployments. On VCF, MetalLB is not needed: the NSX already implements that contract, so any `LoadBalancer` Service automatically gets an NSX virtual server and an external IP from the Supervisor's pool. NSX is the on-premises equivalent of MetalLB for this use case.

Two externally reachable URLs are required:

- Control-plane URL (`runai.<corp-domain>`): Resolves to the HAProxy LB IP on the control-plane cluster. Used by administrators and researchers for the UI and API, and by the cluster component to register and stream events back.
- Wildcard inference domain (`*.inference.<corp-domain>`): Resolves to the HAProxy LB IP on the GPU-enabled cluster. Knative uses this so each `InferenceWorkload` gets a stable per-revision hostname like `phi3-mini.<project-namespace>.inference.<corp-domain>`; HAProxy hands those requests to Kourier.

Both are DNS Type A records pointing at the relevant NSX virtual-server IP. A wildcard TLS certificate covering the inference domain is required so endpoints are reachable over HTTPS without per-service issuance. Outbound HTTPS (or a configured mirror) is required from both clusters to NVIDIA NGC (GPU Operator, NVIDIA AI Enterprise, NIM images) and to `helm.ngc.nvidia.com` and `nvcr.io` (NVIDIA Run:ai charts and images). The legacy `runai.jfrog.io` endpoint is required only if you still use the deprecated JFrog artifact source.

2.6 Identity and RBAC

This deployment authenticates users against NVIDIA Run:ai's built-in local user database. Administrators are created and managed through the NVIDIA Run:ai UI/API, sign in with username and password, and map onto NVIDIA Run:ai's role model (system admin, department admin, project admin, researcher, viewer) without an external Identity Provider (IdP). Local users keep the lab self contained and let the install be validated end to end before any IdP is wired up.

For production, NVIDIA Run:ai also federates to OpenID Connect (OIDC). The self-hosted control plane bundles its own Keycloak that can act as a relying party against an upstream OIDC or SAML IdP (AD FS, Entra ID, Okta, Ping). Group claims map onto NVIDIA Run:ai departments and projects, and the same OIDC tokens are accepted by the UI, the `runai` CLI, and the REST API. Switching from local users to OIDC only changes control-plane configuration; the cluster component and workloads are untouched.

Inside Kubernetes, NVIDIA Run:ai relies on standard RBAC. The cluster component creates the namespaces, service accounts, roles, and role bindings that workloads run under, and admission webhooks enforce project quota and policy. Researchers do not need direct kubeconfig access to the workload cluster; the NVIDIA Run:ai UI, CLI, and REST API are sufficient for day-to-day work.

2.7 Storage

Both VKS clusters use the CSI driver as their default `StorageClass`, backed by vSAN or whatever array the customer has presented through the Supervisor's storage policies. The driver ships with VKS, so no separate CSI install is required. Three classes of persistent volumes land here:

- Control-plane state on the management cluster: Bundled PostgreSQL, Thanos receive store, NATS streams, Keycloak.
- Cluster-side state on the GPU-enabled cluster: Cluster-local Prometheus that scrapes DCGM and workload metrics, plus any cluster-scoped NVIDIA Run:ai caches.
- Researcher-requested PVCs created indirectly through NVIDIA Run:ai data sources: Attaching a PVC-backed data source to a Workspace or Training job materializes a `PersistentVolumeClaim` against the default `StorageClass`, satisfied by the vSphere CSI driver from vSAN.

2.8 Topology and Flexibility

A single self-hosted control plane can manage one or more GPU clusters; the cluster running the control plane can also run GPU workloads or stay CPU-only; GPU clusters can be added or removed over time without rebuilding the control plane.

For the two-cluster reference shown in Figure 2, NVIDIA Run:ai on VCF with Private AI Foundation with NVIDIA (CPU-only management cluster plus GPU-enabled workload cluster) is a clean starting point. It gives platform teams the most operational room: day-2 work on the control plane (upgrades, backups, certificate rotations, OIDC reconfiguration) does not impact the GPU cluster, and the control plane never competes with workloads for GPU capacity. It is the topology recommended for production and the one the deployment steps assume.

A practical note on how those clusters get created on VCF: the AI Kubernetes Cluster catalog item in VCF Automation only produces GPU-enabled VKS clusters. The catalog item drives the GPU Operator install as part of provisioning, and the operator's driver DaemonSet has no GPUs to attach to on a CPU-only worker, so the cluster never reaches a healthy state. The CPU-only management cluster therefore has to come from a different VKS provisioning path: the Supervisor UI, VCF Lifecycle and Configuration workflows, the VCF CLI, or a generic VKS manifest applied to a vSphere Namespace. Only the GPU-enabled workload cluster sits behind the AI Kubernetes Cluster catalog item.

The NVIDIA Run:ai 2.24 documentation explicitly supports both this two-cluster pattern and the simpler single-cluster pattern: the control plane and cluster component are colocated on one Kubernetes cluster. The Cluster System Requirements call out that when the first cluster and the control plane are colocated, a separate ingress controller and FQDN are not required; when they are separate, both are required on each side.

The lab exercises that flexibility on purpose, with two GPU-enabled VKS clusters (both deployed through the AI Kubernetes Cluster catalog item in VCF Automation) registered against a single NVIDIA Run:ai control plane. The cluster names, GPU models, and domain names used throughout this guide reflect the specific lab environment used for validation.

Figure 4: Reference Architecture: Single-Cluster and Multi-Cluster NVIDIA Run:ai Topologies Under One Tenancy

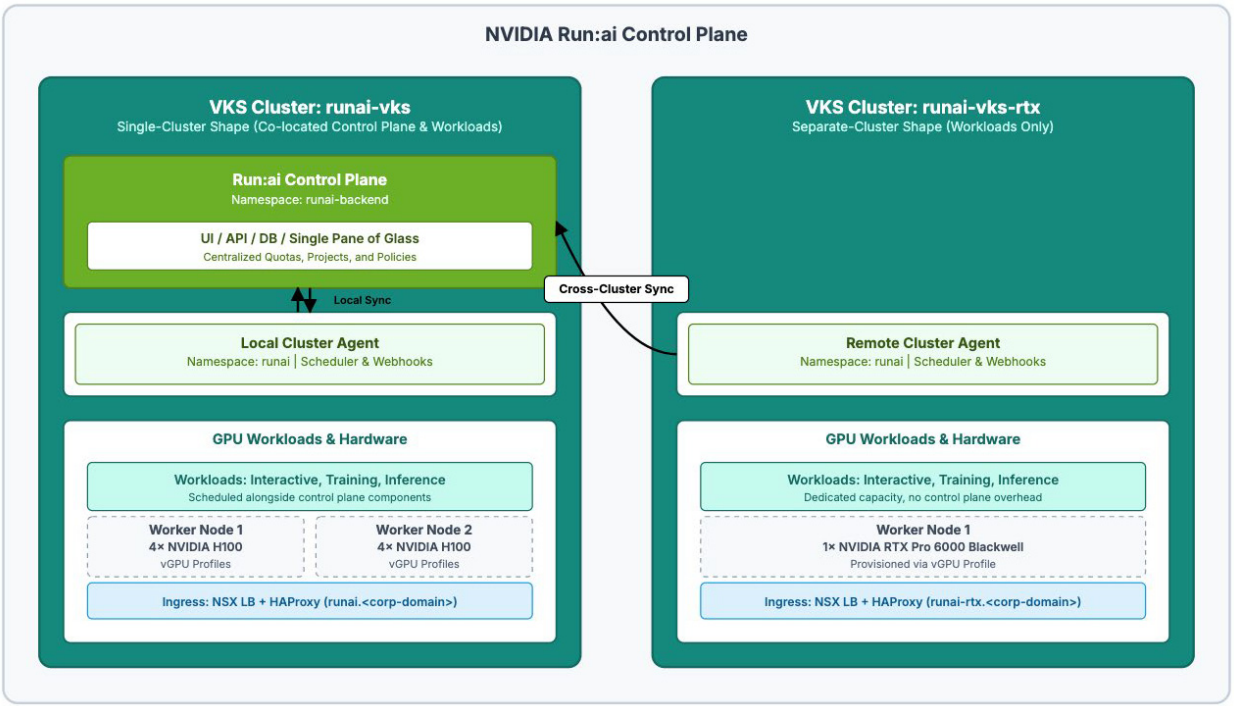


Figure 4 details this reference architecture:

- `runai-vks` hosts the NVIDIA Run:ai control plane (`runai-backend` namespace) and its own cluster component (`runai` namespace) on the same Kubernetes cluster, alongside GPU workloads on NVIDIA H100 in vGPU mode. The same NSX LB endpoint serves both the control-plane URL (`runai.<corp-domain>`) and the cluster's local ingress. This is the single-cluster shape: the control plane sits on a GPU-enabled cluster and shares resources with workloads when that is the desired tradeoff.
- `runai-vks-rtx` is a second AI Kubernetes Cluster, in its own vSphere namespace, with different hardware (an NVIDIA RTX PRO 6000 Blackwell vGPU profile) and its own ingress endpoint (`runai-rtx.<corp-domain>`). It runs only the NVIDIA Run:ai cluster component and registers against the control plane on `runai-vks`. From the UI it appears as a second managed cluster with its own node pools, projects, and quotas.

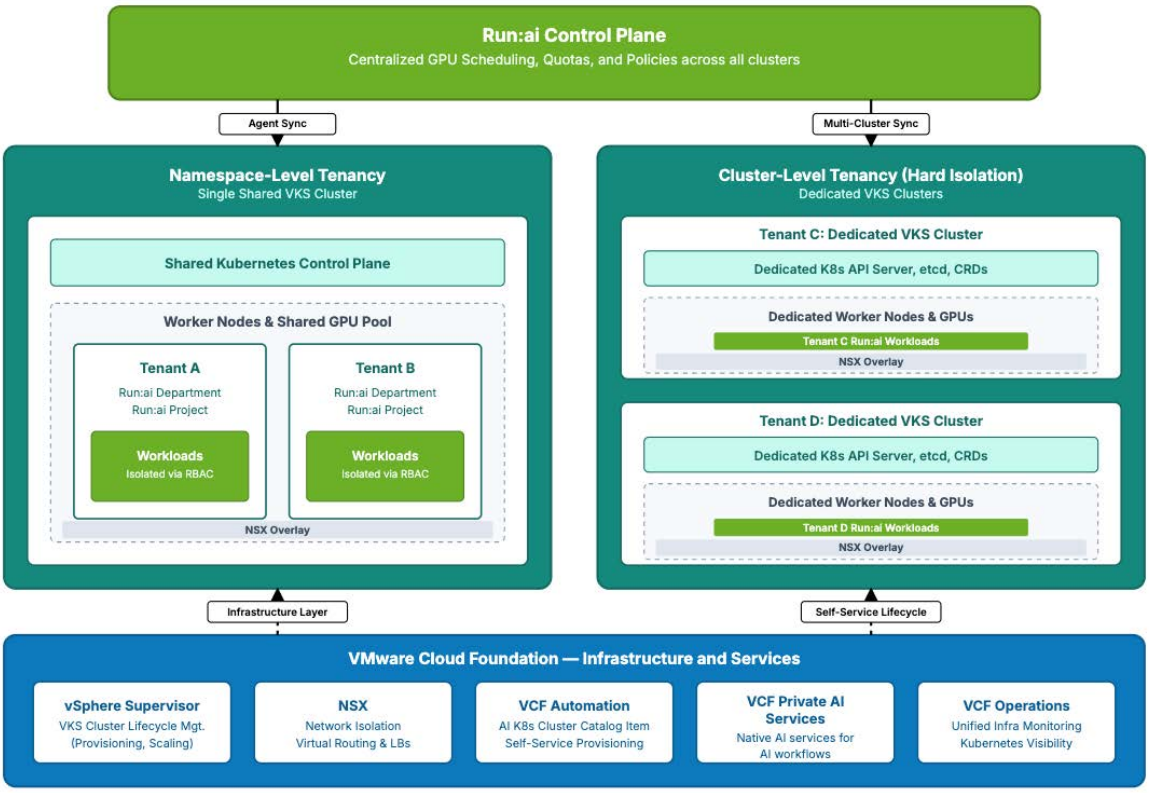
Together, the two clusters cover both supported topologies under one tenancy. The same control plane drives both, so projects, departments, RBAC, policies, integrations, and dashboards are configured once. That is the operational story for VCF and Private AI Foundation with NVIDIA customers: start with whatever footprint fits today, add VKS clusters as new hardware is installed or new tenants come online, mix GPU generations across clusters, manage through a single NVIDIA Run:ai control plane.

2.9 Multi-tenancy and Tenant Isolation on VCF

Within a single VKS cluster, NVIDIA Run:ai segments tenants through departments, projects, and Kubernetes namespaces. This gives each team its own GPU quota, RBAC scope, and workload isolation, but all tenants share the same Kubernetes API server, etcd, and controller manager. For many organizations, this namespace-level segmentation is sufficient and maximizes GPU utilization because the NVIDIA Run:ai scheduler can see every GPU on every node and bin-pack across all of them.

When stronger isolation is required, for example when tenants need to install their own CRDs, run incompatible Kubernetes versions, or demand a hard security boundary at the control-plane level, the answer on VCF is to provision separate VKS clusters. Each VKS cluster is a fully independent Kubernetes control plane (its own API server, etcd, controller manager) running on its own set of worker-node VMs. The Supervisor and VCF Automation make this operationally simple: requesting an additional AI Kubernetes Cluster from the VCF Automation catalog is a self-service action, and lifecycle management (upgrades, patching, scaling) is handled by the Supervisor, not by the platform team manually. Adding the new cluster to NVIDIA Run:ai is a single Helm install of the cluster component followed by registering it in the control-plane UI.

Figure 5: VCF with NVIDIA Run:ai: Achieving Multi-Tenancy and Isolation between Tenants



This is the VCF-native equivalent of the virtual-cluster pattern used on bare-metal Kubernetes, where a lightweight virtual control plane is layered on top of a shared physical cluster to give each tenant its own API surface. On VCF, that abstraction already exists at the infrastructure layer: VKS clusters are the tenant boundary, the Supervisor is the shared infrastructure, and NSX provides the network isolation between them. The result is the same, strong tenant isolation without sacrificing centralized GPU scheduling through NVIDIA Run:ai, but delivered through the platform the customer already operates rather than an additional Kubernetes-in-Kubernetes layer.

Chapter 3: Prerequisites

NVIDIA Run:ai publishes detailed system requirements for both the self-hosted control plane and the cluster component. Treat these as the source of truth and read them end to end before starting:

- Run:ai 2.24 Control Plane System Requirements
- Run:ai 2.24 Cluster System Requirements
- Run:ai 2.24 Support Matrix

On top of those, this paper chooses to use self-hosted control plane and assumes the following pre-requisites are already in place:

- VCF 9.0 with the Supervisor enabled and Private AI Foundation with NVIDIA configured
- Two VKS clusters provisioned through the VCF Automation AI Kubernetes Cluster catalog item, with the GPU Operator deployed and `nvidia-smi` healthy in a test pod
- A `StorageClass` backed by the vSphere CSI driver (default on AI Kubernetes Cluster)
- Two A records, one for the control-plane URL (`runai.<corp-domain>`) and one for the wildcard inference domain (`*.inference.<corp-domain>`), both pointing at NSX-issued LoadBalancer IPs on the relevant clusters
- A wildcard TLS certificate covering the inference domain
- An NGC API key is used to pull the NVIDIA Run:ai Helm charts (`helm.ngc.nvidia.com`) and container images (`nvcr.io`); the same key also covers the NVIDIA AI Enterprise images (GPU Operator, NIM). Starting with v2.24, NGC is the recommended artifact source for all new and existing installations; the legacy JFrog repository (`runai.jfrog.io`) remains supported in 2.24 but will be removed in a future release. For air-gapped clusters, mirror the registries locally.

Chapter 4: Deployment Steps

The following sections walk the install from end to end, with the actual commands and values that produced the lab build. The control-plane FQDN, inference wildcard, and IP addresses are shown as placeholders (`runai.<corp-domain>`, `*.inference.<corp-domain>`, `<haproxy-lb-ip>`); substitute your own. Each step links the relevant NVIDIA Run:ai page for deeper detail.

The lab uses two AI Kubernetes Clusters from VCF Automation: `runai-vks` hosts the NVIDIA Run:ai control plane alongside its own GPU workloads (single-cluster shape), and `runai-vks-rtx` is a second GPU-only cluster registered against that control plane (separate-cluster shape). The order below is written for the two-cluster pattern from Chapter 2 (a control-plane cluster plus one or more GPU workload clusters); for the single-cluster shape used by `runai-vks`, every command runs against the same kubeconfig, the same TLS secret can serve both roles, and Section 4.6, Register the Cluster and Install the NVIDIA Run:ai Cluster Component picks up immediately after Section 4.5, Install the NVIDIA Run:ai Self-Hosted Control Plane on the same cluster.

4.1 Provision the VKS Clusters

Both lab clusters (`runai-vks` and `runai-vks-rtx`) are provisioned as AI Kubernetes Cluster catalog items in VCF Automation, in line with the topology described in Chapter 2. If you are following the recommended two-cluster pattern, also provision the CPU-only management cluster from one of the non-AI VKS paths (the Supervisor UI, VCF Lifecycle and Configuration workflows, the VCF CLI, or a generic VKS manifest); the deployment steps in the rest of this section apply to it unchanged.

When requesting either GPU-enabled cluster from VCF Automation, pick a node-pool VM class whose vGPU profile matches the model and capacity you want to expose to Kubernetes. Once a cluster reports `Ready`, create a kube context with the VCF CLI. The CLI logs you into the Supervisor and scopes the context to the workload VKS cluster in one step:

```
vcf context create <ctx-name> \
  --endpoint https://<supervisor-vip> \
  --username <sso-user> \
  --workload-cluster-name <cluster-name> \
  --workload-cluster-namespace <vsphere-namespace> \
  --type k8s
vcf context use <ctx-name>
```

NOTE: References:

- Deploy a GPU-Accelerated VKS Cluster by Using a Self-Service Catalog Item in VCF Automation (VCF 9.0)
- Workflow for Provisioning VKS Clusters Using VCF CLI

4.2 Verify the NVIDIA GPU Operator

When the cluster is provisioned through the AI Kubernetes Cluster catalog item, the GPU Operator is deployed automatically with the NVIDIA AI Enterprise-licensed vGPU guest driver, the device plugin, and DCGM Exporter. Three checks confirm it is healthy:

1. First, every operator pod should be `Running`:

```
kubectl -n gpu-operator get pods
# gpu-operator-*, nvidia-driver-daemonset-*, nvidia-device-plugin-*,
# nvidia-dcgm-exporter-*, nvidia-container-toolkit-* ... all Running
```

2. Second, exec into the driver DaemonSet pod on a GPU worker and run `nvidia-smi -q` to confirm the driver is loaded, the right vGPU profile is bound, and the license is `Licensed` against your NVIDIA License System (NLS) server:

```
DRIVER_POD=$(kubectl -n gpu-operator get pod \
  -l app.kubernetes.io/component=nvidia-driver \
  -o jsonpath='{.items[0].metadata.name}')

kubectl -n gpu-operator exec "$DRIVER_POD" -c nvidia-driver-ctr -- nvidia-smi -q | \
  grep -E "Driver Version|vGPU Software Licensed Product|License Status"
# Driver Version                              : 580.105.08
# vGPU Software Licensed Product
#     Product Name                            : NVIDIA AI Enterprise
#     License Status                          : Licensed (Expiry: ...)
```

`License Status`: `Licensed` against the expected NVIDIA AI Enterprise product is the signal that the worker VM can reach NLS and that the catalog item handed it the right token. If it shows `Unlicensed`, the most common cause on Private AI Foundation with NVIDIA is NLS reachability from the worker VM, not Kubernetes.

3. Third, run a throwaway CUDA pod to confirm the device plugin is advertising GPUs and the container toolkit is wiring them through. VKS clusters enforce Pod Security Admission (PSA) `restricted` on the `default` namespace, so the pod spec must include a compliant `securityContext`:

```
kubectl run cuda-smoke --image=nvcr.io/nvidia/cuda:12.4.1-base-ubuntu22.04 \
  --rm -it --restart=Never \
  --overrides='{
    "spec":{
      "containers":[{
        "name":"cuda-smoke",
        "image":"nvcr.io/nvidia/cuda:12.4.1-base-ubuntu22.04",
        "command":["nvidia-smi"],
        "resources":{"limits":{"nvidia.com/gpu":"1"}},
        "securityContext":{
          "runAsNonRoot":true,
          "runAsUser":1000,
          "allowPrivilegeEscalation":false,
          "capabilities":{"drop":["ALL"]},
          "seccompProfile":{"type":"RuntimeDefault"}
        }
      }]
    }
  }'
```

NOTE: Reference: NVIDIA GPU Operator with NVIDIA AI Enterprise

4.3 Install the Ingress Controller (HAProxy) and Create the TLS Secrets

Run this on the control-plane cluster and on each workload cluster. The `IngressClass` (`haproxy`) is the same everywhere.

4.3.1 Set the Default StorageClass

NVIDIA Run:ai requires a default `StorageClass`. VKS clusters provisioned through Private AI Foundation with NVIDIA already have the vSphere CSI driver and a `vsan-default-storage-policy` class:

```
kubectl get storageclass
kubectl patch storageclass vsan-default-storage-policy \
  -p '{"metadata":{"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
```

4.3.2 Install the HAProxy Ingress Controller

The HAProxy chart's images live on Docker Hub, which now rate-limits anonymous pulls. Override the image repo to Google's pull-through cache (`mirror.gcr.io`):

```
helm repo add haproxytech https://haproxytech.github.io/helm-charts
helm repo update

helm install haproxy-kubernetes-ingress haproxytech/kubernetes-ingress \
  --create-namespace --namespace haproxy-controller \
  --set controller.service.type=LoadBalancer \
  --set controller.image.repository=mirror.gcr.io/haproxytech/kubernetes-ingress \
  --wait --timeout 5m
```

NSX assigns the external IP automatically:

```
kubectl -n haproxy-controller get svc haproxy-kubernetes-ingress \
  -o jsonpath='{.status.loadBalancer.ingress[0].ip}'
```

That returns the cluster's HAProxy LB IP, referred to as `<haproxy-lb-ip>` from here on. Update the public A record `runai.<corp-domain>` to point at the control-plane cluster's `<haproxy-lb-ip>`. On the GPU workload cluster, point the wildcard inference record `*.inference.<corp-domain>` at that cluster's `<haproxy-lb-ip>`.

```
kubectl get ingressclass haproxy
# NAME      CONTROLLER                             PARAMETERS   AGE
# haproxy   haproxy.org/ingress-controller/haproxy   <none>       30s
```

4.3.3 Generate the TLS Material

Two TLS bundles are needed before the NVIDIA Run:ai installs in the next sections can run:

- Control-plane bundle (consumed on the control-plane cluster): a server certificate covering the control-plane FQDN (`runai.<corp-domain>`) and its private key. If the certificate is signed by a private CA, the CA bundle is also required so the cluster component on the workload cluster can validate it.
- Inference bundle (consumed on the GPU workload cluster by Knative): a wildcard server certificate covering `*.inference.<corp-domain>` and its private key.

Each bundle should end up as three PEM files in whatever working directory you keep the install assets:

1. `fullchain.pem` (server cert plus any intermediate CAs)
2. `private.pem` (matching key)
3. `ca-bundle.pem` (the issuing CA chain, only required for private CAs)

How you obtain them is up to you: an enterprise PKI such as Active Directory Certificate Services, a public ACME provider with DNS-01 (Let's Encrypt, ZeroSSL), or a self-signed Local CA generated with `cfssl` or `openssl` for non-production builds. The rest of the procedure is identical regardless of issuer; only the paths you pass to `kubectl create secret tls` and `kubectl create secret generic` change.

4.3.4 Pod Security Admission Notes

VKS clusters in this lab enforce PSA `restricted` by default. Component installs in this section (NVIDIA Run:ai control plane and cluster, Knative, GPU Operator) require `privileged` on their namespaces, and each subsection labels its namespace explicitly; do not skip those `kubectl label namespace ... pod-security.kubernetes.io/enforce=privileged --overwrite` lines. The HAProxy chart already complies with `restricted`.

Test workloads created outside those privileged namespaces stay under `restricted` and need an explicit `securityContext`:

```
securityContext:
  runAsNonRoot: true
  allowPrivilegeEscalation: false
  capabilities:
    drop: [ALL]
  seccompProfile:
    type: RuntimeDefault
```

NOTE: References:

- Domain Name
- Install Ingress

4.4 Install Prometheus on the Workload Cluster

NVIDIA Run:ai's cluster-side telemetry feeds off `kube-prometheus-stack`. NVIDIA Run:ai has its own UI, so disable Grafana:

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

helm install prometheus prometheus-community/kube-prometheus-stack \
  -n monitoring --create-namespace \
  --set grafana.enabled=false

kubectl -n monitoring get pods
# kube-prometheus-stack components Running, prometheus-prometheus-kube-prometheus-prometheus-0 Ready 2/2
```

The DCGM Exporter scrape config from the GPU Operator is picked up automatically through the `nvidia-dcgm-exporter ServiceMonitor`.

NOTE: Reference: Install Prometheus

4.5 Install the NVIDIA Run:ai Self-Hosted Control Plane

Switch to the control-plane cluster and create the namespace:

```
kubectl create namespace runai-backend
kubectl label namespace runai-backend \
  pod-security.kubernetes.io/enforce=privileged --overwrite
```

Create the TLS secret for the control-plane FQDN, the CA bundle secret (required when using a private CA), and the NGC image-pull secret:

```
kubectl -n runai-backend create secret tls runai-backend-tls \
  --cert /path/to/fullchain.pem --key /path/to/private.pem

kubectl -n runai-backend create secret generic runai-ca-cert \
  --from-file=runai-ca.pem=/path/to/ca-bundle.pem

kubectl -n runai-backend create secret docker-registry runai-reg-creds \
  --docker-server=https://nvcr.io \
  --docker-username='$oauthtoken' \
  --docker-password='<ngc-api-key>' \
  --docker-email=<your-email>
```

Add the NGC Helm repo for the control-plane chart, authenticating with the NGC API key:

```
helm repo add runai-backend https://helm.ngc.nvidia.com/nvidia/runai --force-update--
username='$oauthtoken' --password='<ngc-api-key>'
helm repo update
```

Install the control plane. For a private CA add `--set global.customCA.enabled=true`; for any ingress controller other than HAProxy override `--set global.ingress.ingressClass=`:

```
helm upgrade -i runai-backend -n runai-backend \
  runai-backend/control-plane \
  --version <2.24.x> \
  --set global.domain=runai.<corp-domain> \
  --set global.customCA.enabled=true \
  --set global.ingress.ingressClass=haproxy \
  --set tenantsManager.config.adminUsername=admin@<corp-domain> \
  --set tenantsManager.config.adminPassword='<initial-admin-password>' \
  --wait --timeout 15m
```

The initial administrator password must be at least 8 characters and include an uppercase letter, a lowercase letter, a digit, and a special character.

Confirm all pods are `Running`:

```
kubectl -n runai-backend get pods
# keycloak-0, runai-backend-{frontend,backend,bff-service,authorization,tenants-manager,
# postgresql-0,nats-0..2,traefik,thanos-receive-0,...} all Running
```

Browse to `https://runai.<corp-domain>/` (allow up to 15 minutes after the Helm install completes for the UI to become reachable) and sign in with the administrator credentials. The cluster should be listed with a Connected status on the UI.

Figure 6: The Cluster Displays as Connected

On a fresh install the first signin launches a four-step onboarding wizard:

1. Install your cluster
2. Set up SSO
3. Set up an email server
4. Create your first research team

Step 2 through Step 4 can be skipped and configured later from the NVIDIA Run:ai UI. Step 1 is covered in Section 4.6, Register the Cluster and Install the NVIDIA Run:ai Cluster Component.

NOTE: Reference: Install the Control Plane

4.6 Register the Cluster and Install the NVIDIA Run:ai Cluster Component

The onboarding wizard (Step 1) walks you through registering the first cluster: enter a name, such as `runai-vks`, choose an option: *Same as the control plane* or *Remote control plane*, enter the cluster URL (`runai.<corp-domain>` for the single-cluster shape, the cluster's own FQDN for the separate-cluster shape), and click through. The UI prints a Helm install command prefilled with `controlPlane.url`, a fresh `controlPlane.clientSecret`, `cluster.uid`, and `cluster.url`; copy it.

To add subsequent clusters after the wizard completes, navigate to *Resources* > *Clusters* > *New Cluster* and walk through the same flow.

Switch to the workload cluster and prepare the namespace and TLS material:

```
kubectl create namespace runai
kubectl label namespace runai pod-security.kubernetes.io/enforce=privileged --overwrite
kubectl label namespace runai pod-security.kubernetes.io/warn=privileged    --overwrite
kubectl label namespace runai pod-security.kubernetes.io/audit=privileged   --overwrite
```

The NVIDIA Run:ai cluster installer additionally creates two namespaces of its own on the workload cluster: `runai-reservation` (GPU reservation pods used by fractional-GPU workloads) and `runai-scale-adjust` (node-scale-adjuster scaling pods). Because VKS enforces the restricted Pod Security Standard on any namespace without explicit PSA labels, pre-create both namespaces with the privileged labels, including the Helm ownership metadata so the runai-cluster chart adopts them during installation:

```
for ns in runai-reservation runai-scale-adjust; do
  kubectl create namespace $ns --dry-run=client -o yaml | kubectl apply -f -
  kubectl label namespace $ns \
    pod-security.kubernetes.io/enforce=privileged \
    pod-security.kubernetes.io/warn=privileged \
    pod-security.kubernetes.io/audit=privileged \
```

```
    app.kubernetes.io/managed-by=Helm --overwrite
  kubectl annotate namespace $ns \
    meta.helm.sh/release-name=runai-cluster \
    meta.helm.sh/release-namespace=runai --overwrite
done

kubectl -n runai create secret tls runai-cluster-domain-tls-secret \
  --cert /path/to/fullchain.pem --key /path/to/private.pem

kubectl -n runai create secret generic runai-ca-cert \
  --from-file=runai-ca.pem=/path/to/ca-bundle.pem
kubectl -n runai label secret runai-ca-cert \
  run.ai/cluster-wide=true run.ai/name=runai-ca-cert --overwrite

kubectl -n runai create secret docker-registry runai-reg-creds \
  --docker-server=https://nvcr.io \
  --docker-username='$oauthtoken' \
  --docker-password='<ngc-api-key>' \
  --docker-email=<your-email>
```

Add the NGC Helm repo for the cluster chart:

NOTE: This is the same NGC repository as the control plane, added under a different alias.

```
helm repo add runai https://helm.ngc.nvidia.com/nvidia/runai --force-update --username='$oauthtoken'
--password='<ngc-api-key>'
helm repo update
```

Run the command the UI gave you, plus two extra flags: `global.customCA.enabled=true` (private CA) and `clusterConfig.global.ingress.ingressClass=haproxy` (the IngressClass the cluster operator stamps on any `Ingress` it generates). On Helm 4, also append `--server-side=false` to avoid server-side apply conflicts with the NVIDIA Run:ai operator:

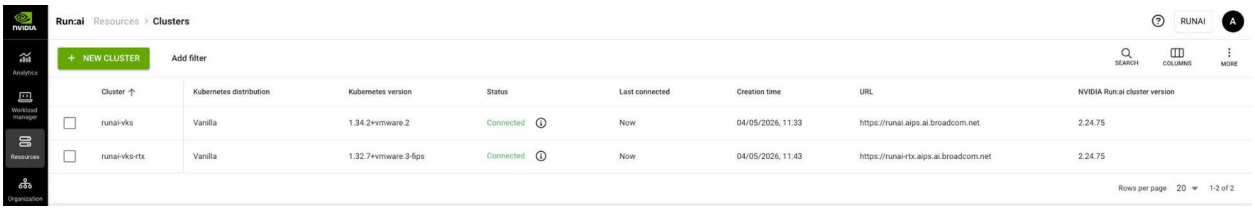
```
helm upgrade -i runai-cluster runai/runai-cluster -n runai \
  --version <2.24.x> \
  --set controlPlane.url=runai.<corp-domain> \
  --set controlPlane.clientSecret='<from-UI>' \
  --set cluster.uid='<from-UI>' \
  --set cluster.url=runai.<corp-domain> \
  --set global.customCA.enabled=true \
  --set-string clusterConfig.global.ingress.ingressClass=haproxy \
  --server-side=false \
  --wait --timeout 15m
```

Confirm the cluster-side pods are `Running`:

```
kubectl -n runai get pods
# runai-agent, cluster-sync, cluster-api, runai-operator, engine-operator,
# admission-controller, *-controllers, runai-node-exporter (DaemonSet),
# init-ca, prometheus-runai-0, status-updater, ... all Running
```

The wizard in the UI changes from *Waiting for cluster to connect* to *Cluster connected* once the agent reaches the control plane. After that the cluster is listed in *Resources* > *Clusters* as `Connected`.

Figure 7: The Cluster Displays as Connected



NOTE: Reference: Install the Cluster

4.7 Install Knative Serving for Inference

Knative Serving on the GPU-enabled cluster materializes NVIDIA Run:ai inference workloads. NVIDIA Run:ai 2.24 supports Knative 1.11 to 1.18; the lab uses 1.18.x with Kourier as the network layer and HAProxy in front.

4.7.1 Create the namespaces

```
kubectl create ns knative-operator
kubectl label ns knative-operator pod-security.kubernetes.io/enforce=privileged --overwrite

kubectl create ns knative-serving
kubectl label ns knative-serving pod-security.kubernetes.io/enforce=privileged --overwrite
```

4.7.2 Install the Knative Operator

```
helm repo add knative-operator https://knative.github.io/operator
helm repo update

helm install knative-operator \
  --create-namespace --namespace knative-operator \
  --version 1.18.2 \
  knative-operator/knative-operator

kubectl -n knative-operator get deployment knative-operator
```

4.7.3 Create the Wildcard TLS Secret

NVIDIA Run:ai expects the wildcard cert at a specific name (`runai-cluster-inference-tls-secret`) in the `knative-serving` namespace:

```
kubectl -n knative-serving create secret tls runai-cluster-inference-tls-secret \
  --cert /path/to/wildcard-fullchain.pem --key /path/to/wildcard-private.pem
```

4.7.4 Apply the KnativeServing Custom Resource

Replace the `domain` value with your real inference base domain (no leading `*.`):

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  config:
    config-autoscaler:
      enable-scale-to-zero: "true"
    config-features:
      kubernetes.podspec-affinity: enabled
      kubernetes.podspec-init-containers: enabled
      kubernetes.podspec-persistent-volume-claim: enabled
      kubernetes.podspec-persistent-volume-write: enabled
      kubernetes.podspec-schedulername: enabled
      kubernetes.podspec-securitycontext: enabled
      kubernetes.podspec-tolerations: enabled
      kubernetes.podspec-volumes-emptydir: enabled
```

```
      kubernetes.podspec-fieldref: enabled
      kubernetes.containerspec-addcapabilities: enabled
      kubernetes.podspec-nodeselector: enabled
      multi-container: enabled
    domain:
      inference.<corp-domain>: ""
    network:
      domainTemplate: '{{.Name}}-{{.Namespace}}.{{.Domain}}'
      ingress-class: kourier.ingress.networking.knative.dev
      default-external-scheme: https
  high-availability:
    replicas: 2
  ingress:
    kourier:
      enabled: true

kubectl apply -f knative-serving.yaml
kubectl get knativeserving -n knative-serving
kubectl get svc kourier -n knative-serving
```

Wait for `Ready: True`.

4.7.5 Front Kourier with HAProxy

Kourier's `Service` is `ClusterIP` here, so HAProxy is the public entry point. Apply an `Ingress` that routes the wildcard host into Kourier with the inference TLS secret:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  ingressClassName: haproxy
  rules:
    - host: '*.inference.<corp-domain>'
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: kourier
                port:
                  number: 80
  tls:
    - hosts:
        - '*.inference.<corp-domain>'
      secretName: runai-cluster-inference-tls-secret

kubectl apply -f knative-ingress.yaml
kubectl -n knative-serving get ingress knative-serving
# CLASS    HOSTS                       ADDRESS            PORTS
# haproxy  *.inference.<corp-domain>   <haproxy-lb-ip>    80, 443
```

4.7.6 Enable Inference in the NVIDIA Run:ai Cluster

```
kubectl -n runai patch runaiconfig runai --type merge \
  -p '{"spec":{"global":{"inference":{"enabled":true},"subdomainSupport":true}}}'
```

Confirm the cluster picks it up:

```
kubectl -n runai logs deploy/inference-workload-controller | grep -i "Knative available"
kubectl -n runai logs deploy/cluster-sync               | grep -i "serving-knative"
```

You should see `Knative available: true` and successful syncs to the control plane.

NOTE: References:

- Install Optional NVIDIA Run:ai Components: Inference
- External access to containers

Chapter 5: Functional Testing

With the control plane, cluster component, and Knative Serving installed, the stack is ready for workloads. The tests below exercise the three workload types (inference, workspace, training) end to end using the `runai` CLI v2 and confirm that GPU scheduling, Knative autoscaling, and external connectivity all work. Each test includes explicit verification steps so the operator knows the test passed. All three tests are self-contained and can be torn down after verification.

5.1 CLI Login and Project Setup

Log in to the control plane and set the target cluster and project. The CLI accepts local credentials directly or can open a browser for OIDC:

```
runai login user -u admin@<corp-domain> -p '<admin-password>'
runai cluster set <cluster-name>
runai project set <project-name>
```

If the project does not exist yet, create one with a GPU quota:

```
runai project create <project-name> --resources gpu-deserved=2,gpu-limit=4
```

NVIDIA Run:ai creates a `runai-<project-name>` namespace on the cluster. On VKS clusters with PSA `restricted`, label it `privileged` so workload pods are not rejected:

```
kubectl label namespace runai-<project-name> \
  pod-security.kubernetes.io/enforce=privileged --overwrite
```

NVIDIA Run:ai's fractional-GPU scheduler creates short-lived reservation pods in the `runai-reservation` namespace. On VKS clusters this namespace also needs the `privileged` label, or the reservation pods will be rejected by PSA and fractional workloads will fail to bind:

```
kubectl label namespace runai-reservation \
  pod-security.kubernetes.io/enforce=privileged --overwrite
```

Confirm the cluster's GPU capacity is visible:

```
runai node list
runai nodepool list
# The node pool should show the expected GPU count under Allocated/Total GPUs
```

5.2 NVIDIA Run:ai GPU Fractions

NVIDIA Run:ai lets multiple workloads share a single physical GPU by requesting a fraction instead of a whole device. Submit two workloads that each request 50% of a GPU:

```
runai training standard submit frac-half-1 \
  -p <project-name> \
  -i nvcr.io/nvidia/cuda:12.4.1-base-ubuntu22.04 \
  --gpu-portion-request 0.5 \
  --command -- bash -c "nvidia-smi -L; sleep 300"

runai training standard submit frac-half-2 \
  -p <project-name> \
  -i nvcr.io/nvidia/cuda:12.4.1-base-ubuntu22.04 \
  --gpu-portion-request 0.5 \
  --command -- bash -c "nvidia-smi -L; sleep 300"
```

Verify both workloads are `Running`, each allocated 0.50 GPU:

```
runai training standard list -p <project-name>
# Workload     Type      Status   Running/Req. Pods   GPU Alloc.
# frac-half-1  Training  Running  1/1                 0.50
# frac-half-2  Training  Running  1/1                 0.50
```

Verify both pods landed on the same physical GPU by comparing the GPU UUID:

```
kubectl -n runai-<project-name> exec frac-half-1-0-0 -- \
  nvidia-smi --query-gpu=gpu_uuid,name --format=csv,noheader
kubectl -n runai-<project-name> exec frac-half-2-0-0 -- \
  nvidia-smi --query-gpu=gpu_uuid,name --format=csv,noheader
# GPU-1ea7dcd9-..., NVIDIA H100XM-80C
# GPU-1ea7dcd9-..., NVIDIA H100XM-80C
```

Identical GPU UUIDs confirm that both workloads share the same physical device. The scheduler placed them together because each requested only half a GPU, leaving the remaining GPUs free for other workloads.

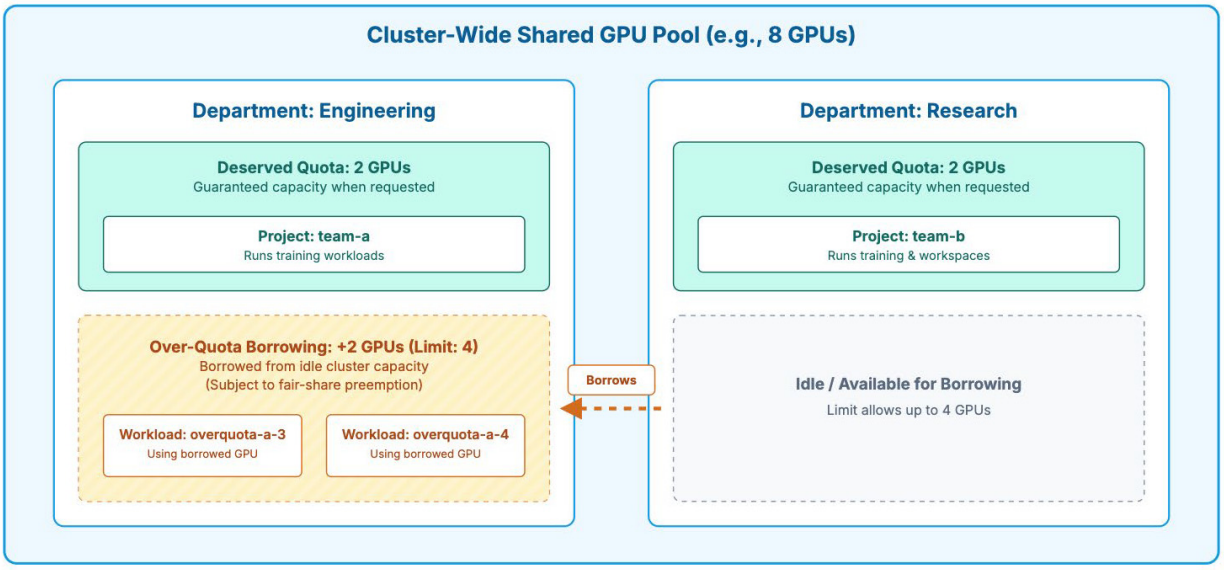
Clean up:

```
runai training standard delete frac-half-1 -p <project-name>
runai training standard delete frac-half-2 -p <project-name>
```

5.3 Over-Quota Borrowing and Fair Share across Departments

NVIDIA Run:ai organizes GPU capacity into departments and projects. Each department has a deserved GPU quota and a limit. Deserved represents a guaranteed, minimum allocation of resources, that a department or project is entitled to use within a node pool. Limit places an upper bound on the combined deserved quota + over-quota resources. When a department's GPUs sit idle, other departments can borrow them up to their limit (over-quota). When the owning department submits work, the scheduler reclaims capacity through fair-share arbitration. This test creates two departments on the same cluster and demonstrates over-quota borrowing.

Figure 8: Over-Quota Borrowing and Fair Share across Departments



Create two departments via the NVIDIA Run:ai UI (Settings > Departments > New Department) or the REST API. Each department gets a deserved quota of 2 GPUs and a limit of 4:

Department	Deserved GPUs	Limit GPUs
Engineering	2	4
Research	2	4

Create a project under each department:

```
runai project create team-a --department engineering --resources gpu-deserved=2,gpu-limit=4
runai project create team-b --department research    --resources gpu-deserved=2,gpu-limit=4
```

Label the project namespaces as `privileged`:

```
kubectl label namespace runai-team-a pod-security.kubernetes.io/enforce=privileged --overwrite
kubectl label namespace runai-team-b pod-security.kubernetes.io/enforce=privileged --overwrite
```

Step 1: Team-a submits four jobs, borrowing beyond its quota.

With team-b idle, the cluster has spare GPUs. Team-a's limit of 4 allows it to borrow up to twice its deserved quota.

NOTE: In this example we use a simple `sleep 600` command to simulate team-a's workloads occupying the GPUs for 10 minutes, but in reality, these would be the team's actual training scripts or interactive sessions.

```
for i in 1 2 3 4; do
  runai training standard submit overquota-a-$i \
    -p team-a \
    -i nvcr.io/nvidia/cuda:12.4.1-base-ubuntu22.04 \
    -g 1 \
    --command -- bash -c "nvidia-smi -L; sleep 600"
done
```

Step 2: Team-b submits two jobs, claiming its fair share.

```
for i in 1 2; do
  runai training standard submit fairshare-b-$i \
    -p team-b \
    -i nvcr.io/nvidia/cuda:12.4.1-base-ubuntu22.04 \
    -g 1 \
    --command -- bash -c "nvidia-smi -L; sleep 600"
done
```

Verify the project and department allocation:

```
runai project list
# Name    Status  Department    GPU Quota  Allocated GPUs  GPU Allocation Ratio
# team-a  Ready   engineering   2.00       4.00            200.00%
# team-b  Ready   research      2.00       2.00            100.00%

runai department list
# Name         GPU Quota  Allocated GPUs  GPU Allocation Ratio
# engineering  2.00       4.00            200.00%
# research     2.00       2.00            100.00%
```

Team-a is at 200% of its deserved quota: it borrowed two idle GPUs from the cluster-wide pool. Team-b is at 100%, running exactly at its entitlement. Both departments' jobs are running concurrently because the cluster has enough total capacity. If team-b were to submit additional jobs that push the cluster past capacity, the scheduler would preempt team-a's over-quota (preemptible) workloads to reclaim GPUs for team-b's deserved allocation.

Clean up:

```
for i in 1 2 3 4; do runai training standard delete overquota-a-$i -p team-a; done
for i in 1 2; do runai training standard delete fairshare-b-$i -p team-b; done
```

5.4 Inference Test: Small Language Model with vLLM

Deploy a small HuggingFace model through `vllm/vllm-openai` as a NVIDIA Run:ai inference workload. The workload lands on Knative Serving, gets a stable HTTPS endpoint, and supports scale-to-zero. This test uses `Qwen/Qwen3-0.6B` (ungated, ~1.5 GB); substitute any model that fits in your GPU memory.

If the cluster nodes cannot reach `huggingface.co` directly (common in firewalled environments), pre-download the model to a PVC and point vLLM at the local path with `--existing-pvc` and `--model /models/<model-dir>`. If the nodes can reach HuggingFace, the `--model` argument accepts the HuggingFace model ID directly and vLLM downloads at startup.

```
runai inference submit qwen3-06b \
  -p <project-name> \
  -i vllm/vllm-openai:latest \
  -g 1 \
  --serving-port 8000 \
  --min-replicas 1 --max-replicas 1 \
  --initialization-timeout-seconds 600 \
  -e HF_TOKEN='<huggingface-token>' \
  -- --model Qwen/Qwen3-0.6B --max-model-len 2048
```

`--initialization-timeout-seconds 600` gives vLLM time to download the model weights and compile CUDA graphs on first start. `--min-replicas 1` keeps the pod warm; set to `0` for scale-to-zero after validation.

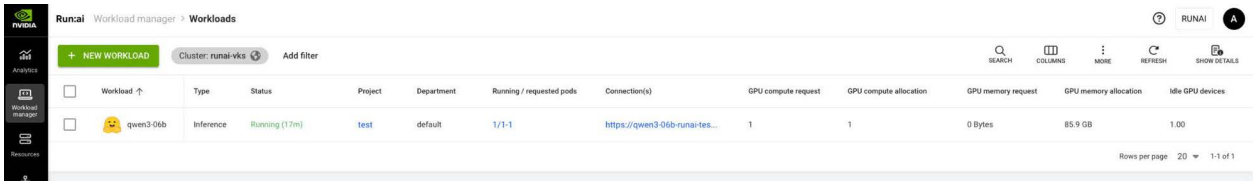
Verify the workload reaches `Running` with 1/1 pods:

```
runai inference list -p <project-name>
# Workload    Type       Status    Running/Req. Pods   GPU Alloc.
# qwen3-06b   Inference  Running   1/1-1               1.00
```

Verify the model is loaded and serving:

```
runai inference describe qwen3-06b -p <project-name>
# Phase:  Running
# URL:    https://qwen3-06b-runai-<project>.inference.<corp-domain>
```

Figure 9: Verify the Model is Loaded and Serving



Send a chat completion request to the endpoint:

```
curl -sk https://qwen3-06b-runai-<project>.inference.<corp-domain>/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "Qwen/Qwen3-0.6B",
    "messages": [{"role":"user","content":"What is VMware Cloud Foundation?"}],
    "max_tokens": 64
  }'
# A JSON response with "choices":[{"message":{"content":"..."}}] confirms the full
# inference path: Run:ai scheduling, GPU allocation, Knative Serving, Kourier
```

```
# routing through HAProxy, and TLS termination on the wildcard certificate.
```

You can also verify the model list endpoint:

```
curl -sk https://qwen3-06b-runai-<project>.inference.<corp-domain>/v1/models
# Returns {"data":[{"id":"Qwen/Qwen3-0.6B", ...}]}
```

Clean up:

```
runai inference delete qwen3-06b -p <project-name>
```

5.5 Workspace Test: Interactive JupyterLab on GPU

Submit a JupyterLab workspace with one GPU. The `--external-url` flag exposes port 8888 through the NVIDIA Run:ai ingress so the notebook is reachable from a browser. NVIDIA Run:ai fronts the workspace with its oauth2-proxy, so access requires authentication through the NVIDIA Run:ai UI.

```
runai workspace submit jupyter-test \
  -p <project-name> \
  -i jupyter/scipy-notebook:latest \
  -g 1 \
  --external-url container=8888 \
  --command -- start-notebook.sh --NotebookApp.token=''
```

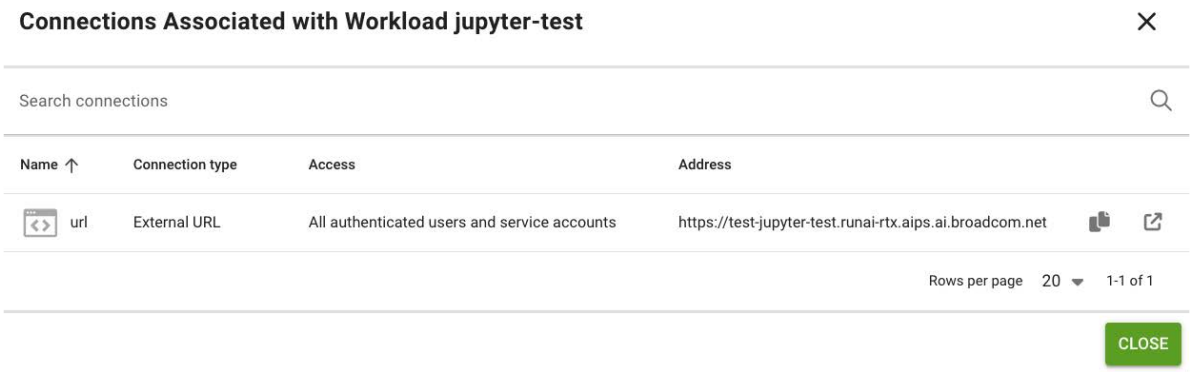
Verify the workspace reaches `Running` with 1/1 pods:

```
runai workspace list -p <project-name>
# Workload      Type       Status    Running/Req. Pods   GPU Alloc.
# jupyter-test  Workspace  Running   1/1                 1.00
```

Verify the URL is assigned:

```
runai workspace describe jupyter-test -p <project-name>
# Phase:  Running
# URL:    https://<project>-jupyter-test.<cluster-fqdn>
```

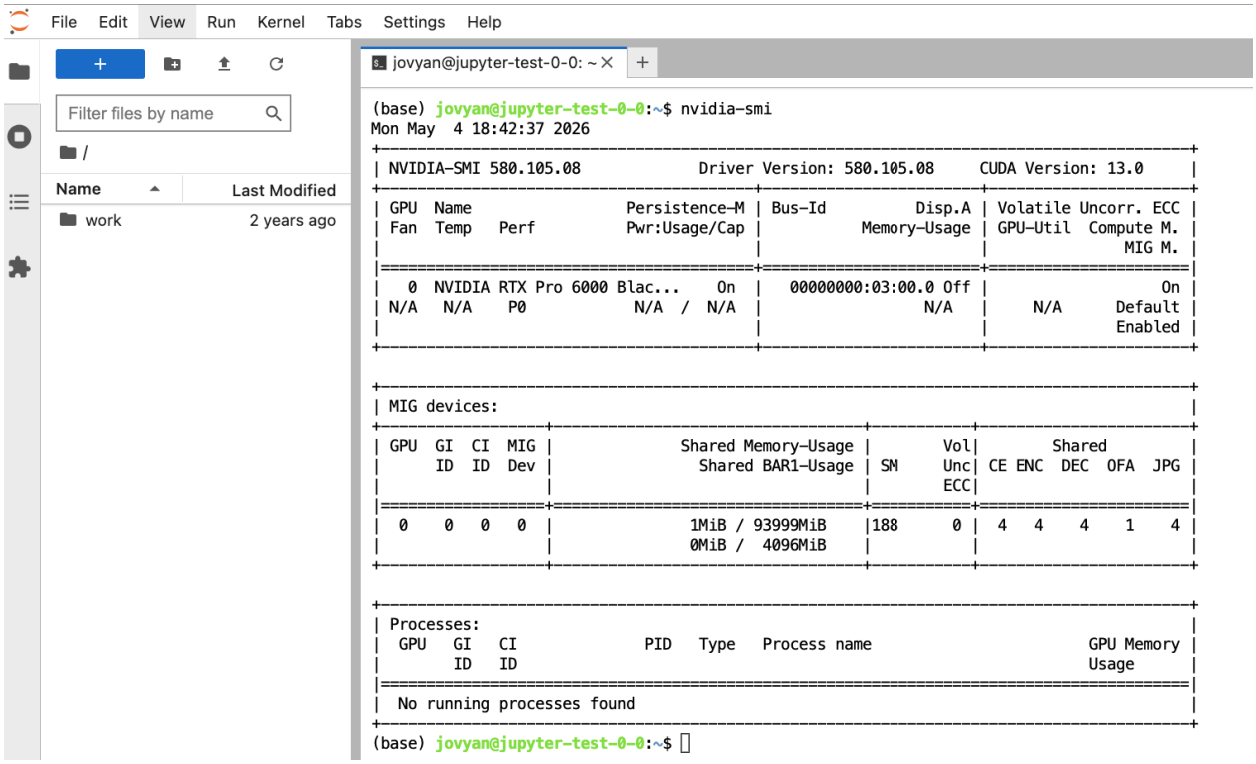
Figure 10: Verify the URL is Assigned



Verify GPU visibility inside the workspace:

```
kubectl -n runai-<project-name> exec jupyter-test-0-0 -c jupyter-test -- \
  nvidia-smi --query-gpu=name,driver_version --format=csv,noheader
# NVIDIA H100 80GB HBM3, 580.105.08    (or whatever GPU the cluster has)
```

Figure 11: Verify GPU Visibility inside the Workspace



Clean up:

```
runai workspace delete jupyter-test -p <project-name>
```

5.6 5.6 Training Test: MNIST with PyTorch

Submit a real training job that trains a small CNN on the MNIST handwritten-digit dataset for three epochs and reports test accuracy. The NVIDIA PyTorch NGC container is used as the base image. `--large-shm` mounts a large `/dev/shm`, required by PyTorch `DataLoader` workers for shared-memory IPC.

```
runai training standard submit mnist-train \
  -p <project-name> \
  -i nvcr.io/nvidia/pytorch:24.05-py3 \
  -g 1 \
  --large-shm \
  --command -- python -c "
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import time

device = torch.device('cuda')
print(f'GPU: {torch.cuda.get_device_name(0)}')
print(f'CUDA version: {torch.version.cuda}')
print(f'PyTorch version: {torch.__version__}')
```

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
train_ds = datasets.MNIST('/tmp/data', train=True, download=True, transform=transform)
test_ds  = datasets.MNIST('/tmp/data', train=False, transform=transform)
train_loader = DataLoader(train_ds, batch_size=256, shuffle=True, num_workers=2)
test_loader  = DataLoader(test_ds,  batch_size=1000, num_workers=2)

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.fc1   = nn.Linear(9216, 128)
        self.fc2   = nn.Linear(128, 10)
    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        return self.fc2(x)

model = Net().to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()

epochs = 3
for epoch in range(1, epochs + 1):
    model.train()
    t0 = time.time()
    running_loss = 0.0
    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        loss = criterion(model(data), target)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    elapsed = time.time() - t0
    avg_loss = running_loss / len(train_loader)
    print(f'Epoch {epoch}/{epochs}  loss={avg_loss:.4f}  time={elapsed:.1f}s')

model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        pred = model(data).argmax(dim=1)
        correct += pred.eq(target).sum().item()
        total += target.size(0)

accuracy = 100.0 * correct / total
print(f'Test accuracy: {correct}/{total} ({accuracy:.1f}%)')
print('MNIST training completed successfully')
"
```

The job downloads the MNIST dataset (~11 MB, falls back to S3 mirrors if `yann.lecun.com` is unreachable), trains a two-layer CNN with Adam for three epochs, then evaluates on the 10k test set.

Verify the job transitions to `Completed`:

```
runai training standard list -p <project-name>
# Workload     Type      Status     Running/Req. Pods   GPU Alloc.
# mnist-train  Training  Completed  0/1                 0.00
```

Verify the logs show per-epoch loss, test accuracy, and a successful exit:

```
kubectl -n runai-<project-name> logs mnist-train-0-0 --tail=10
# GPU: NVIDIA H100 80GB HBM3
# CUDA version: 12.4
# PyTorch version: 2.4.0a0+07cecf4168.nv24.05
# Epoch 1/3  loss=0.1782  time=91.9s
# Epoch 2/3  loss=0.0456  time=1.3s
# Epoch 3/3  loss=0.0285  time=1.3s
# Test accuracy: 9852/10000 (98.5%)
# MNIST training completed successfully
```

Test accuracy above 98% after three epochs confirms that the GPU, the CUDA runtime, the PyTorch framework, and the NVIDIA Run:ai scheduler are all working correctly. If the job stays in `Pending`, check that the project has available GPU quota (`runai project list`) and that no other workload is consuming the cluster's GPUs.

Clean up:

```
runai training standard delete mnist-train -p <project-name>
```

Chapter 6: Monitoring

Once the platform is running, three monitoring surfaces cover different audiences and use cases: the Run:ai UI for platform teams and researchers, the `runai` CLI for quick terminal checks, and VCF Operations for infrastructure teams who manage GPU capacity alongside the rest of the VCF estate.

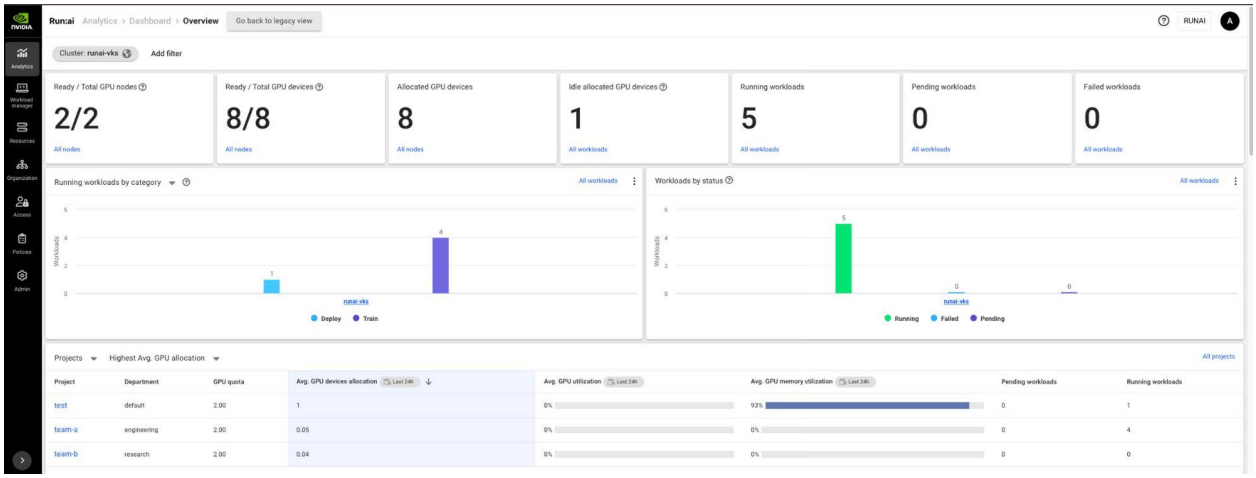
6.1 NVIDIA Run:ai UI Dashboards

The NVIDIA Run:ai control-plane UI is the primary monitoring interface for GPU workloads. It is reachable at `https://runai.<corp-domain>/` and requires an NVIDIA Run:ai login. The dashboards update in near-real time from DCGM and scheduler telemetry collected by the cluster component.

6.1.1 Overview Dashboard

The overview dashboard is the landing page after login. It shows cluster-wide GPU allocation, utilization, and workload counts across all registered clusters, departments, and projects in a single view. Use this to confirm at a glance that GPU capacity is being consumed and that no cluster is disconnected.

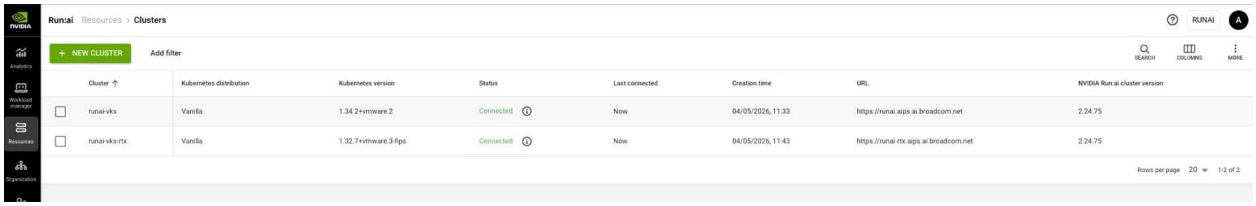
Figure 12: Overview Dashboard



6.1.2 Clusters View

The clusters view lists every registered cluster with its connection status, GPU count, and version. Drill into a cluster to see per-node GPU allocation and health.

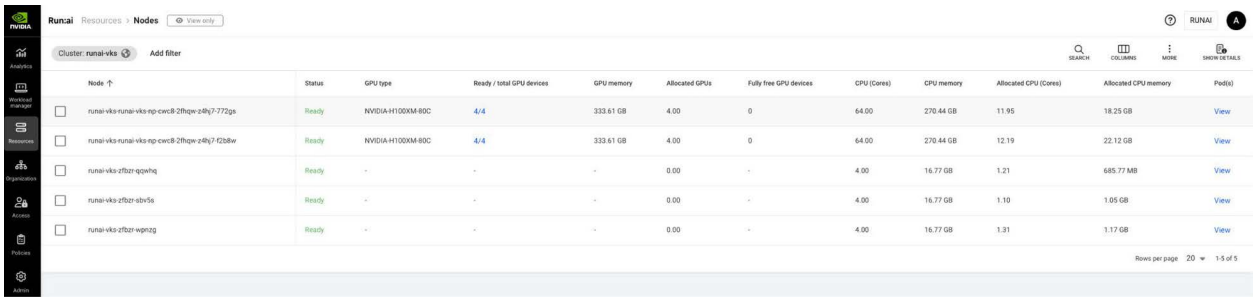
Figure 13: Clusters View



6.1.3 Nodes View

The nodes view is a per-node breakdown of GPU type, allocation, and utilization. It shows which GPUs are idle, partially allocated (fractional), or fully consumed.

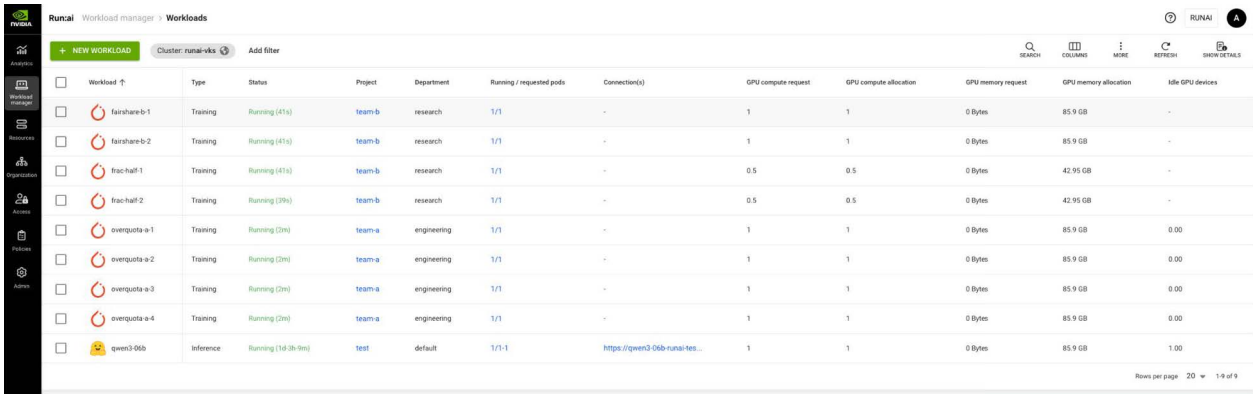
Figure 14: Nodes View



6.1.4 Workloads View

The workloads view is a central table of all active, pending, and completed workloads across projects. Each row shows the workload type (Training, Inference, Workspace), status, GPU allocation, project, and submission time. Click a workload to see its pods, events, resource usage, and logs.

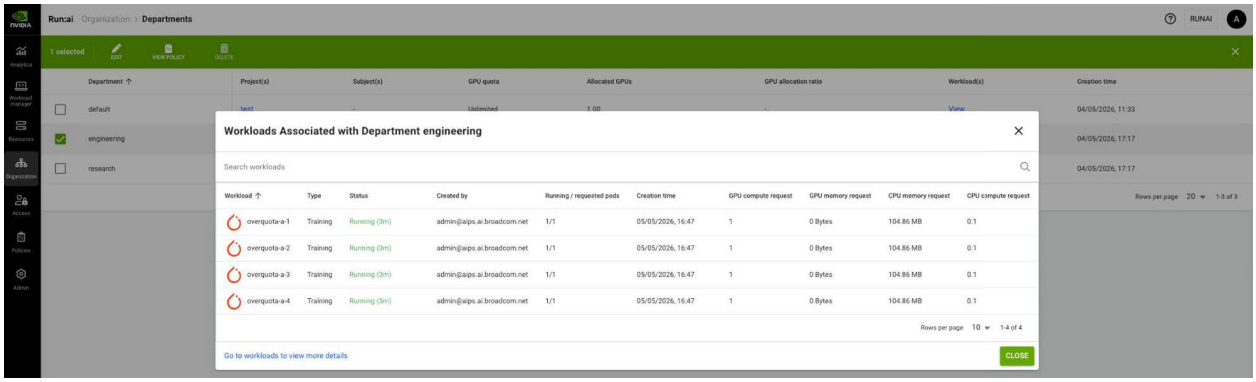
Figure 15: Workloads View



6.1.5 Departments and Projects

The Departments view shows per-department GPU quota, allocation, and utilization ratio. The Projects view drills into individual projects within a department. These views are the operational proof of the over-quota and fair-share behavior tested in Section 5.3, Over-Quota Borrowing and Fair Share across Departments.

Figure 16: Departments and Projects View



6.1.6 GPU Utilization Metrics

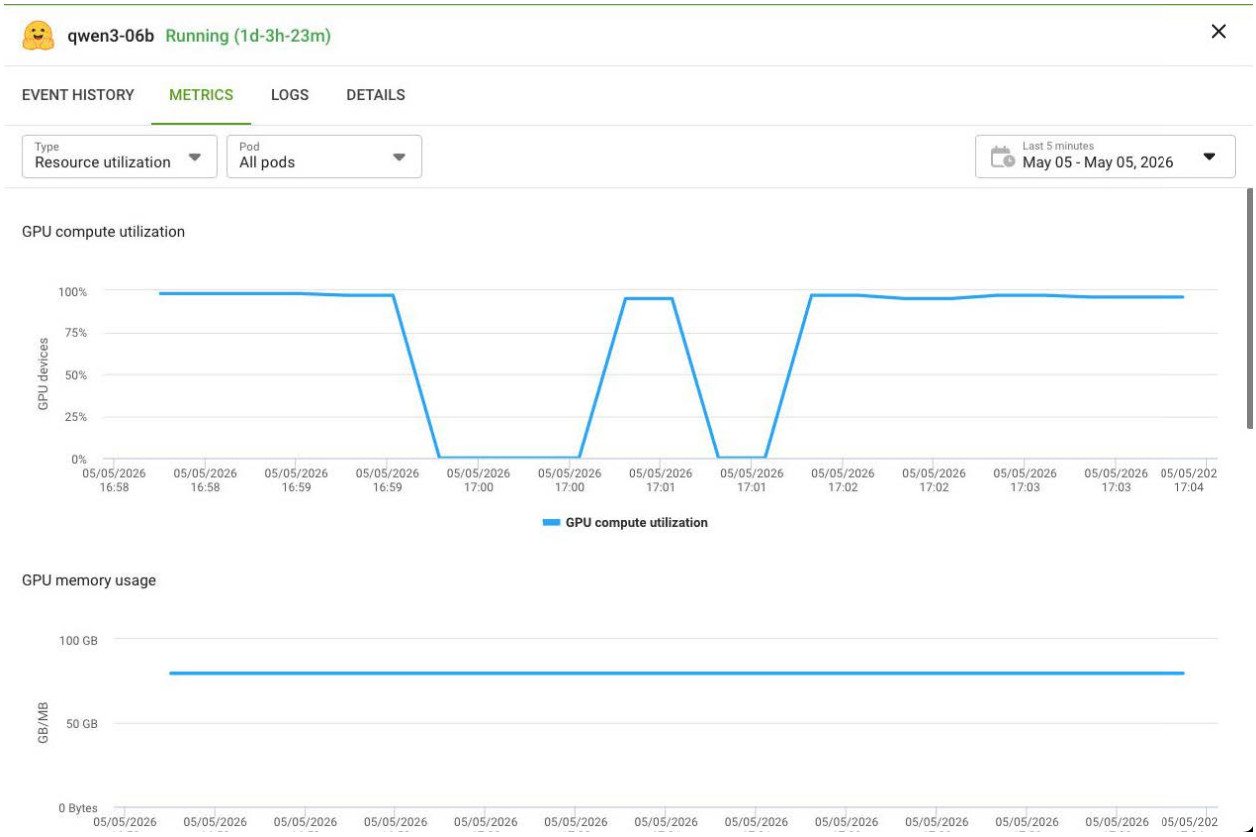
Use the GPU Utilization Metrics view to drill into any node or workload to see per-GPU DCGM metrics: compute utilization, memory utilization, memory used/free, temperature, SM clock, and encoder/decoder utilization. These are the same DCGM Exporter metrics scraped by the NVIDIA Run:ai cluster Prometheus, rendered in the UI without additional configuration.

Figure 17: GPU Utilization Metrics View

6.1.7 Workload-Level Metrics

Select any workload from the Workloads view and click the Metrics tab in the side panel. View training and workspace workloads, the UI shows GPU utilization, GPU memory usage, CPU usage, and memory usage over time for each pod. For inference workloads, the Metrics tab additionally shows throughput (requests/sec) and latency percentiles, drawn from Knative and vLLM telemetry.

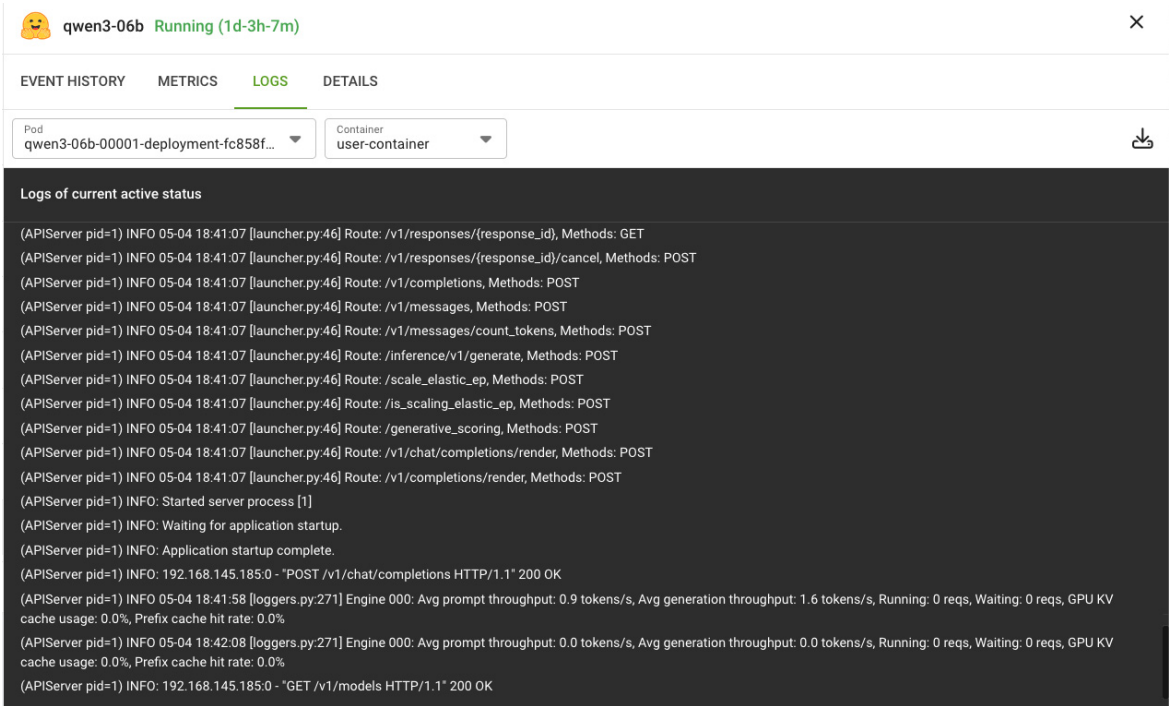
Figure 18: Workload-Level Metrics View



6.1.8 Workload Logs in the UI

Select any workload and click the Logs tab in the side panel. The UI streams container logs in real time from the workload's pods. Use the container dropdown to switch between containers (`user-container` for the main workload, sidecar containers for NVIDIA Run:ai agents). Logs are available for running, completed, and failed workloads as long as the pod has not been garbage-collected.

Figure 19: Workload Logs in the UI View



6.2 NVIDIA Run:ai CLI

The `runai` CLI provides the same information as the UI in a terminal-friendly format. These commands are useful for scripting, quick checks, and environments where browser access is not available.

6.2.1 Cluster and Node Status

```
runai node list
# Name                                        Status  GPU Type           Allocated/Total GPUs  Free
GPUs  Nodepool
# runai-vks-runai-vks-np-cwc8-2fhqw-z4hj7-772gs  Ready   NVIDIA-H100XM-80C  4/4                   0
default
# runai-vks-runai-vks-np-cwc8-2fhqw-z4hj7-f2b8w  Ready   NVIDIA-H100XM-80C  4/4                   0
default
# runai-vks-zfbzr-qqwhq                      Ready   N/A             N/A                N/A       default
# runai-vks-zfbzr-sbv5s                      Ready   N/A             N/A                N/A       default
# runai-vks-zfbzr-wpnzg                      Ready   N/A             N/A                N/A       default

runai nodepool list
# Name     Status  Allocated/Total GPUs  Nodes  MNNVL detection  Network Topology
# default  Ready   8/8                   5      Not Detected
```

6.2.2 Department and Project Allocation

```
runai department list
# Name         GPU Quota  Allocated GPUs  GPU Allocation Ratio
# research     2.00       3.00            150.00%
# engineering  2.00       4.00            200.00%
# default      -1.00      1.00            0.00%

runai project list
# Name    Status  Department    GPU Quota  Allocated GPUs  GPU Allocation Ratio
# team-a  Ready   engineering   2.00       4.00            200.00%
# team-b  Ready   research      2.00       3.00            150.00%
# test    Ready   default       2.00       1.00            50.00%
```

Both `engineering` and `research` are over their deserved quota (200% and 150% respectively), borrowing idle GPUs from the cluster-wide pool. The `default` department shows unlimited quota (`-1.00`) with the `test` project running the inference workload.

6.2.3 Workload Status

```
runai training standard list -p team-a
# Workload       Type      Status   Running/Req. Pods  GPU Alloc.
# overquota-a-1  Training  Running  1/1                1.00
# overquota-a-2  Training  Running  1/1                1.00
# overquota-a-3  Training  Running  1/1                1.00
# overquota-a-4  Training  Running  1/1                1.00

runai training standard list -p team-b
# Workload       Type      Status   Running/Req. Pods  GPU Alloc.
# fairshare-b-1  Training  Running  1/1                1.00
# fairshare-b-2  Training  Running  1/1                1.00
# frac-half-1    Training  Running  1/1                0.50
# frac-half-2    Training  Running  1/1                0.50

runai inference list -p test
# Workload   Type       Status   Running/Req. Pods  GPU Alloc.
# qwen3-06b  Inference  Running  1/1-1              1.00

runai inference describe qwen3-06b -p test
# Name:       qwen3-06b
# Type:       Inference
# Phase:      Running
# Project:    test
# Department: default
# Image:      vllm/vllm-openai:latest
# ...
# Serving
#  Minimum Replicas:  1
#  Maximum Replicas:  1
# ...
# Network
#  URL:  https://qwen3-06b-runai-test.runai-inference.<corp-domain>
# ...
# Pods
#  Running Pods: 1/1
```

6.2.4 Workload Logs from the CLI

The `runai` CLI supports a `logs` subcommand for each workload type, but in environments where the cluster-api TLS certificate is signed by a private CA the CLI may reject the connection. In that case, fall back to `kubectl logs` against the workload pod directly:

```
kubectl -n runai-team-a logs overquota-a-1-0-0 --tail=5
# Iteration 365900, GPU memory: 0.8GB
# Iteration 366000, GPU memory: 0.8GB
# Iteration 366100, GPU memory: 0.8GB
# Iteration 366200, GPU memory: 0.8GB
# Iteration 366300, GPU memory: 0.8GB

kubectl -n runai-test logs qwen3-06b-00001-deployment-<pod-hash> -c user-container --tail=5
# INFO: 192.168.145.185:0 - "POST /v1/chat/completions HTTP/1.1" 200 OK
# INFO: 192.168.145.185:0 - "POST /v1/chat/completions HTTP/1.1" 200 OK
# INFO: 192.168.145.242:0 - "POST /v1/chat/completions HTTP/1.1" 200 OK
# INFO: 192.168.145.242:0 - "POST /v1/chat/completions HTTP/1.1" 200 OK
# INFO: 192.168.145.185:0 - "POST /v1/chat/completions HTTP/1.1" 200 OK
```

Training workload logs show the GPU stress loop output with iteration counts and memory usage. Inference workload logs show vLLM handling incoming requests with HTTP 200 responses. These are the same logs visible in the UI's Logs tab.

6.3 VCF Operations

VCF Operations can monitor the VKS clusters that NVIDIA Run:ai workloads run on, surfacing Kubernetes-level metrics (node health, pod counts, container resource usage) alongside the existing vSphere, NSX, and vSAN telemetry. This gives infrastructure teams a unified view of GPU cluster health without logging into the NVIDIA Run:ai UI.

The setup requires three components:

1. Supervisor Management Proxy service on the Supervisor
2. Telegraf and Prometheus VKS standard packages on each VKS cluster
3. Pod And Container Monitoring toggle enabled in VCF Operations

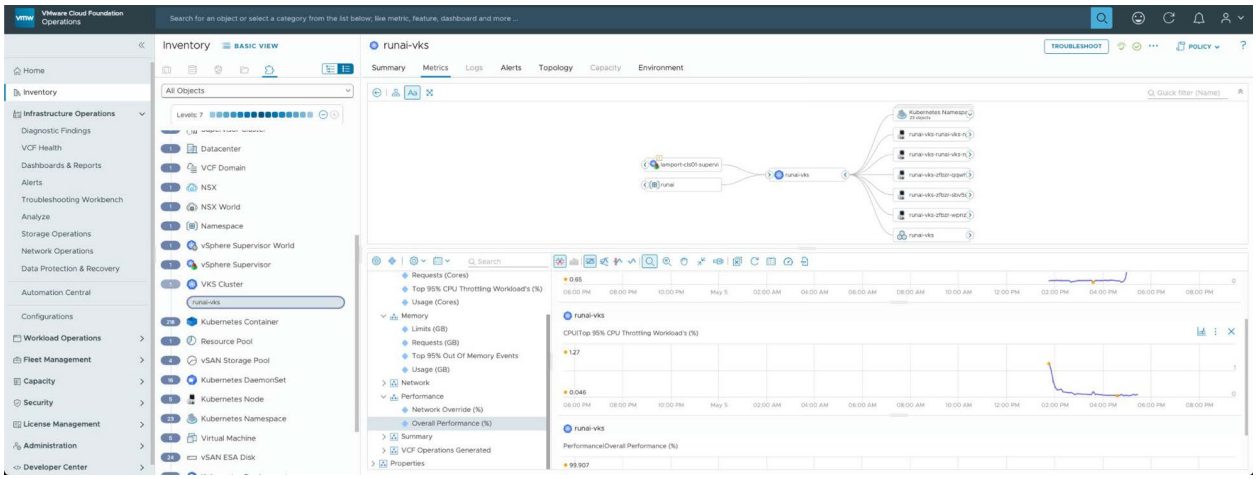
The Telegraf package includes a built-in DCGM Exporter input plugin that can be enabled in its data values to forward GPU metrics through the Supervisor proxy pipeline. For the full installation procedure, refer to the following resources:

- Enabling Monitoring for VKS Clusters (VCF 9.0 documentation)
- Install the Supervisor Management Proxy Service
- Install Telegraf and Telegraf Package Reference (includes DCGM input plugin configuration)
- Install Prometheus

6.3.1 VKS Cluster Dashboard

Once monitoring is enabled and metrics start flowing (allow 10 to 15 minutes), the built-in Kubernetes VKS Cluster dashboard in VCF Operations shows node health, pod counts, container resource utilization, and workload status for each monitored VKS cluster.

Figure 20: VKS Cluster Dashboard



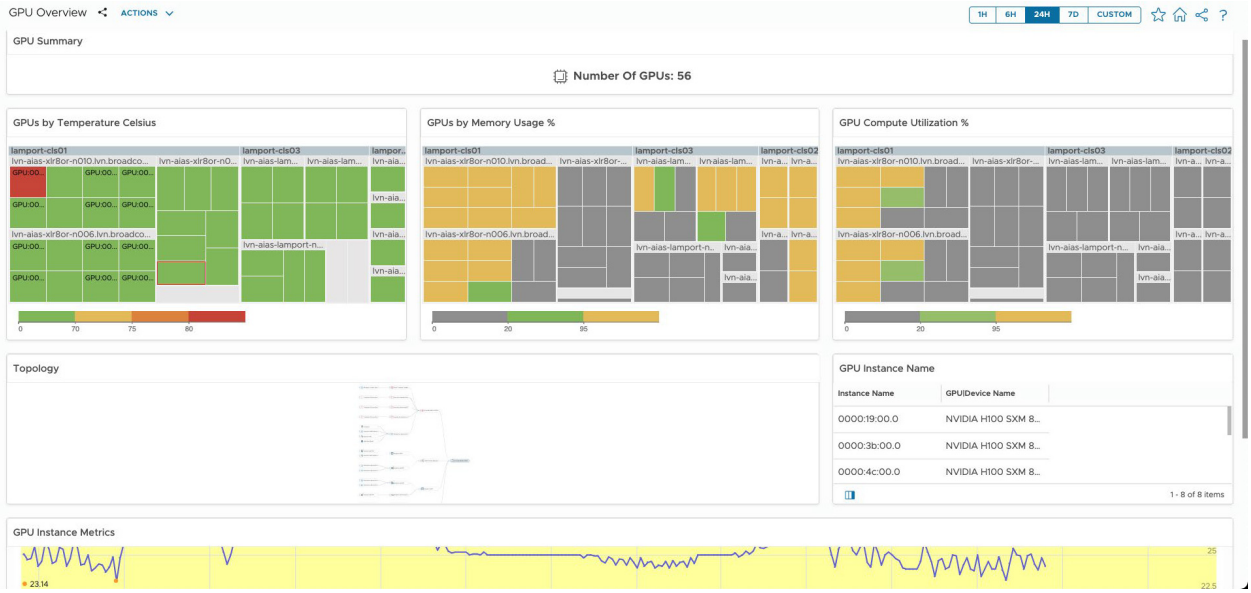
6.3.2 Private AI Dashboards

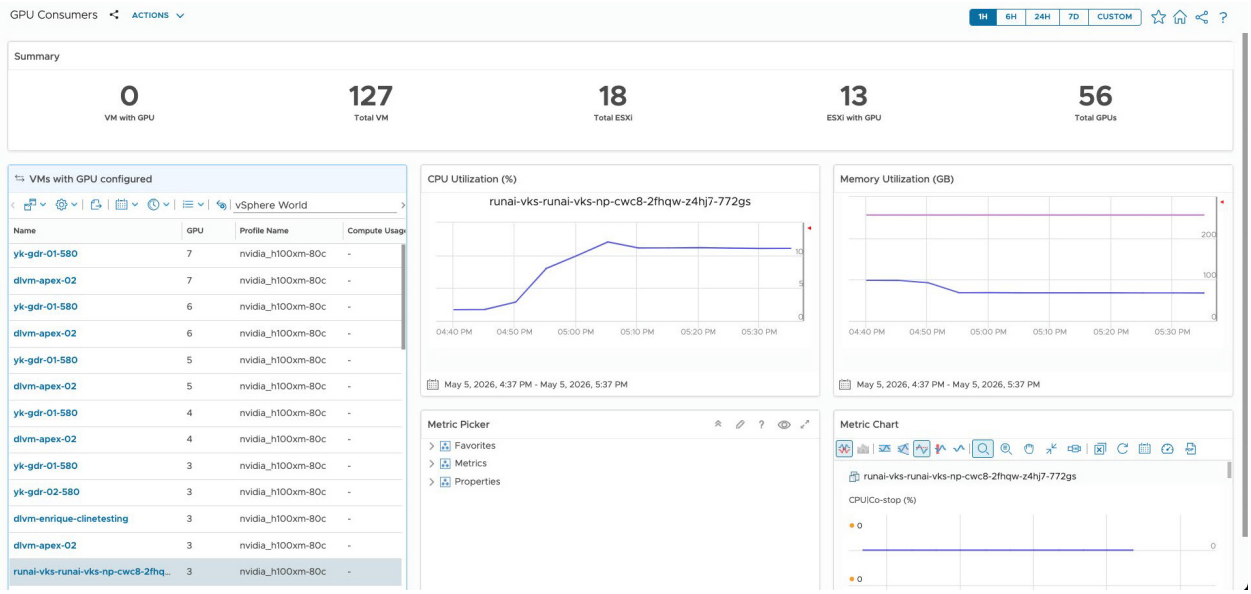
VCF Operations includes a set of built-in Private AI dashboards under Dashboards > Private AI that provide GPU-centric views of the infrastructure without any additional configuration beyond the vSphere Supervisor adapter:

- GPU Overview: Aggregated view of all GPU resources across the VCF estate, including total GPU count, allocation status, and utilization trends.
- GPU Providers: Per-host breakdown of GPU hardware, showing which ESXi hosts have GPUs, the GPU model, driver version, and current assignment (passthrough, vGPU profile, or unassigned).
- GPU Consumers: Maps GPU assignments to the VMs and VKS worker nodes consuming them, linking vGPU profiles or passthrough devices to specific workloads.
- GPU Equipped Clusters: Shows which vSphere clusters have GPU-equipped hosts and their aggregate GPU capacity.

These dashboards operate at the vSphere layer and reflect GPU hardware inventory, vGPU profile assignment, and host-level utilization. They complement NVIDIA Run:ai's workload-level GPU metrics (Section 6.1, NVIDIA Run:ai UI Dashboards) by giving infrastructure teams visibility into the physical GPU topology, driver health, and capacity planning data that sits below the Kubernetes and NVIDIA Run:ai layers.

Figure 21: Private AI Dashboards





Chapter 7: Conclusion

This paper walked through a full end-to-end deployment of NVIDIA Run:ai on VCF with Private AI Foundation with NVIDIA.

The following workflow was used to get started with NVIDIA Run:ai on VCF:

1. Install the self-hosted control plane on a VKS cluster
2. Attach one or more GPU-enabled AI Kubernetes Clusters provisioned from the VCF Automation catalog
3. Set up Knative Serving for inference
4. Validate with real workloads
5. Integrate monitoring across the NVIDIA Run:ai and VCF Operations surfaces

NVIDIA Run:ai is included in every NVIDIA AI Enterprise entitlement. For VCF customers who already run Private AI Foundation with NVIDIA, that means the GPU scheduling and workload-orchestration layer is not an additional procurement decision; it ships with the stack they already own.

Chapter 8: Resources

VMware Private AI Foundation with NVIDIA Solution Brief:
https://www.vmware.com/docs/vmware-privateai-foundation-with-nvidia-solutions-brief

NVIDIA Run:ai Documentation:
https://run-ai-docs.nvidia.com/

VCF Private AI Services:
https://www.vmware.com/products/cloud-infrastructure/vcf-private-ai-services

AI workloads with VCF 9.1 Release Blog:
https://blogs.vmware.com/cloud-foundation/2026/05/05/streamline-simplify-and-protect-all-your-ai-workloads-with-vcf-9-1/

BROADCOM®