Offline export. Prefer the **PDF** for airgapped handoff (section headings preserve navigation). This single HTML file is secondary: open it directly in a browser (`file://`). Interactive features (schema validator, in-site search) are not included.

NKP Catalog documentation

Offline export of the NKP Catalog docs. Prefer the **PDF** for airgapped use; the companion **HTML** file is a secondary option.

Generated: 2026-11-28T07:12:24.053Z

NKP Catalog

The **NKP Catalog** helps you manage application catalogs and releases on the Nutanix Kubernetes Platform (NKP). Use it to discover and deploy apps from Nutanix, partner, and custom catalogs — from the command line or in automation.

Applications Browse applications from the Nutanix, partner, and AI catalogs. Filter by NKP version, category, and certification.

Getting started Create and license NKP clusters, browse the Nutanix product, partner, and AI catalogs.

Concepts OCI artifacts, Flux, HelmRelease, Kustomization, and configuration overrides.

Guides Initialize catalogs and applications, build and push OCI artifacts, set up releases.

Examples Simple, complex, and declarative input patterns for catalog applications.

API Application metadata, user inputs, and release specification schemas.

CLI Versioned nkp command reference generated from released CLIs.

Get help

- **GitHub:** nutanix-cloud-native/nkp-partner-catalog
- Open an issue or discussion for support.

Getting started

End-to-end workflow

This diagram shows the user-facing journey from discovery to deployment.

```
sequenceDiagram
    actor Alice as Alice (Partner Developer)
    actor John as John (Repo Admin)
    participant GitHub as GitHub Repository
    participant CI as CI Pipeline
    participant CD as CD Pipeline
    participant Catalog as Catalog Portal (gh-pages)
    participant NKP as Nutanix Kubernetes Platform
    actor Bob as Bob (End User)

    Note over GitHub: Drafting an App
    Alice->>GitHub: Submit application (PR)
    activate GitHub
    GitHub->>CI: Trigger validation workflow
    deactivate GitHub
    activate CI

    CI->>CI: Run security scans
    CI->>CI: Validate manifest format
    CI->>CI: Test deployment

    alt Validation Failed
        CI-->>GitHub: Post validation results
        activate GitHub
        GitHub-->>Alice: Request changes
        deactivate GitHub
    else Validation Passed
        CI->>GitHub: Approve submission
        deactivate CI
        activate GitHub

        GitHub->>John: Notification of pending PR
        activate John
        John->>GitHub: Review and merge PR
        deactivate John

        GitHub->>CD: Trigger deployment workflow
        deactivate GitHub
        activate CD
```

```
        Note over CD: Artifact Processing
        CD->>CD: Perform Validation
        CD->>CD: Index new application
        CD->>CD: Generate metadata
        CD->>CD: Publish non airgapped artifacts
        CD->>CD: Publish airgapped artifacts (if applicable)

        CD->>Catalog: Deploy to catalog portal
        deactivate CD
        activate Catalog
    end

    Note over NKP: User Discovers an App
    Bob->>NKP: Browse catalog
    activate NKP
    NKP->>Catalog: Request available applications
    Catalog-->>NKP: Return application listing
    deactivate Catalog

    Bob->>NKP: Select application to install

    alt Airgapped environments
      activate Bob
      Bob->>Bob: Download airgapped bundle
      Bob->>NKP: Push images to private registry
      deactivate Bob
    end

    Bob->>NKP: Deploy application
    NKP->>NKP: Validate AppDeployment
    NKP->>NKP: Check upgradesFrom (for upgrades)

    Note over Catalog: Telemetry
    activate Catalog
    NKP-->>Bob: Display deployment status
    NKP-->>Catalog: Send usage statistics (Pulse ?)
    Catalog-->>NKP: Send CVE information and upgrade prompts
    Catalog-->>Catalog: Analyze usage metrics (Pulse ?) for public visiblity
    deactivate Catalog
    deactivate NKP
```

Create an NKP cluster

To use the NKP Catalog, you need a running NKP management cluster. Refer to the official Nutanix documentation for complete instructions on creating and configuring NKP clusters.

<a href={ `https://portal.nutanix.com/page/documents/details?targetId=Nutanix-Kubernetes-Platform-v2_19:top-get-started-nkp-t.html` }> Getting Started with NKP v2.19

Quick overview

NKP uses Cluster API (CAPI) to provision and manage Kubernetes clusters. The typical workflow is:

1. **Create a management cluster** — A self-managed cluster that hosts NKP components (Kommander, Flux, platform applications). For Nutanix infrastructure, use:

   ```
   nkp create cluster nutanix --self-managed
   ```

2. **Create or attach workload clusters** — From the management cluster, create managed clusters (Nutanix, AWS, vSphere, Azure, GCP) or attach existing Kubernetes clusters using the NKP UI or CLI.

3. **Install Kommander** — The management cluster runs Kommander which provides the NKP UI, catalog, workspace/project management, and GitOps (Flux) capabilities.

Supported infrastructure providers

Provider	CLI flag
Nutanix (AHV)	`nkp create cluster nutanix`
AWS	`nkp create cluster aws`
vSphere	`nkp create cluster vsphere`
Azure	`nkp create cluster azure`
GCP	`nkp create cluster gcp`
Pre-provisioned	`nkp create cluster preprovisioned`
EKS (attach)	`nkp create cluster eks`
AKS (attach)	`nkp create cluster aks`

Next steps

- <a href={`https://portal.nutanix.com/page/documents/details?targetId=Nutanix-Kubernetes-Platform-v2_19:top-nkp-licenses-t.html`}>Apply an NKP license
- Browse catalogs after your cluster is ready

Apply an NKP license

Overview

The license type you use determines which NKP features are available. You must replace the default license key with one obtained from the Nutanix Support Portal (or from your fulfillment source). Only administrators can add or remove licenses.

Add a license through the UI

1. Log into the NKP UI dashboard.
2. In the workspace header drop-down list, select **Global**.
3. In the navigation menu, select **Settings** > **Licensing**.
4. Choose based on your current and procured license:
   - **Default license:** Select **Activate License**, enter a valid license key, then **Activate**.
   - **Non-default Nutanix license:** Select **Update License**, enter the key from the Nutanix Support Portal, then **Update**.
   - **Non-default D2IQ license:** Select **Update License**, enter a valid D2IQ or Nutanix key, then **Update**.

The license card will show **Valid** when activation succeeds.

Add a license through kubectl

1. Create a secret with your Nutanix-provided token:

   ```
   kubectl create secret generic my-license-secret --from-literal=jwt=MY_LICENSE -n kommander
   kubectl label secret my-license-secret kommanderType=license -n kommander
   ```

2. Create the License object pointing at that secret (apply the YAML for `kind: License`, `spec.licenseRef.name: my-license-secret` in namespace `kommander`).
3. Check the license page in the NKP UI to confirm it shows as valid.

License types (from the guide)

- **NKP Starter** — Basic cluster and NKP UI; Nutanix infrastructure only.
- **NKP Pro** — Single-cluster or single Management cluster; platform applications, GitOps (FluxCD), NKP UI.
- **NKP Ultimate** — Multi-cluster (Management plus Attached or Managed clusters); catalog applications (e.g. Kafka, Zookeeper), custom applications, project and workspace management.
- **NKP Full Stack** — For Prism Central clusters (e.g. CAPX, CAPV); separate enablement steps in the guide.

You cannot downgrade after registering a higher-tier license.

Browse the Nutanix product catalog

Overview

The **Nutanix Product Catalog** contains Nutanix-supported applications that extend NKP with capabilities such as AI inference, networking, observability, and data services. These are first-party applications validated and certified by Nutanix.

Repository: nutanix-cloud-native/nkp-nutanix-product-catalog

Browse the repository's `applications/` directory for the current list of available applications and their versions.

Characteristics

All applications in this catalog:

- Require **NKP Pro** or **NKP Ultimate** license
- Are **Nutanix-supported** with official certification
- Support **airgapped** deployments
- Deploy at **workspace** scope

Deploying from the catalog

1. Log into the NKP UI and navigate to your workspace.
2. Go to **Applications** in the sidebar.
3. Find the application under the Nutanix catalog section.
4. Click **Enable** and configure any required values.

Resources

- <a href={`https://portal.nutanix.com/page/documents/details?targetId=Nutanix-Kubernetes-Platform-v2_19:top-nkp-applications-c.html`}>NKP documentation

Browse the partner catalog

Overview

The **NKP Partner Catalog** contains validated third-party applications from Nutanix technology partners. These applications have been integrated and tested for deployment on NKP through the partner ecosystem.

Repository: nutanix-cloud-native/nkp-partner-catalog

Browse the repository's `applications/` directory for the current list of available applications and their versions.

Characteristics

- Require **NKP Pro** or **NKP Ultimate** license

- Some applications may require a separate partner license in addition to the NKP license
- Deploy at **workspace** scope
- Applications span categories such as API gateway, backup and DR, virtual desktops, and more

Deploying from the catalog

1. Log into the NKP UI and navigate to your workspace.
2. Go to **Applications** in the sidebar.
3. Browse applications under the Partner catalog section.
4. Click **Enable** and configure required values (e.g. license keys, domain names).

Contributing a partner application

Partner applications follow the same catalog structure as all NKP catalog applications. See the Workflows section for adding applications to a catalog repository.

Resources

- NKP Partner program
- Catalog CLI documentation

Browse the AI catalog

Overview

The **NKP AI Applications Catalog** provides a curated set of open-source and community AI/ML applications packaged for one-click deployment on NKP. Applications span inference engines, model registries, ML pipelines, vector databases, notebooks, agent frameworks, and more.

Repository: nutanix-cloud-native/nkp-ai-applications-catalog

Browse the repository's `applications/` directory for the current list of available applications and their versions.

Characteristics

- Require **NKP Pro** or **NKP Ultimate** license
- Deploy at **workspace** or **project** scope depending on the application
- Applications are packaged as Flux HelmReleases with OCI-based chart sources
- Each application deploys to its own dedicated namespace
- GPU support is available for inference and training workloads (configure via Helm values)

Categories

Applications in this catalog cover areas such as:

- **Inference** — LLM serving engines with OpenAI-compatible APIs
- **Agent frameworks** — Kubernetes-native AI agent orchestration
- **ML pipelines** — End-to-end machine learning workflow management
- **Notebooks** — Interactive development environments for data science
- **Vector databases** — Similarity search for RAG and embedding workloads
- **Model registries** — Versioned model artifact management
- **GPU management** — Device plugins, metrics exporters, and operator support

Deploying from the catalog

1. Log into the NKP UI and navigate to your workspace.
2. Go to **Applications** in the sidebar.
3. Browse applications under the AI catalog section.
4. Click **Enable** and configure required values (e.g. model names, GPU settings, storage).

Dependencies

Some AI applications declare soft or hard dependencies on other catalog applications (e.g. an agent framework depending on an inference engine). Check each application's `metadata.yaml` for the `dependencies` and `requiredDependencies` fields.

Resources

- NKP AI Applications Catalog repository
- Catalog CLI documentation

Concepts

OCI artifacts

Overview

OCI (Open Container Initiative) artifacts are the packaging and distribution format used by NKP Catalog for delivering application manifests, Helm charts, and catalog bundles. By leveraging OCI registries, catalogs can be pushed, pulled, and versioned using the same infrastructure used for container images.

How NKP Catalog uses OCI artifacts

NKP Catalog uses OCI artifacts at two levels:

1. **Helm charts stored in OCI registries** — Each application's Helm chart is pushed to an OCI registry (e.g. `ghcr.io`). The `OCIRepository` resource in each application's `helmrelease.yaml` references the chart by OCI URL and tag.
2. **Catalog bundles** — The `nkp create catalog-bundle` command packages one or more applications (manifests, metadata, and optionally container images) into a tar archive. This bundle can then be pushed to an OCI registry as a collection artifact using `nkp push bundle`.

OCI registry URLs

A typical OCI URL for a catalog collection follows this pattern:

```
oci://<registry>/<org>/<catalog-name>/collection
```

For Applications within a Catalog:

```
oci://<registry>/<org>/<catalog-name>/<app-name>
```

Airgapped deployments

For airgapped (disconnected) environments, catalog bundles can include container images alongside manifests. Build an airgapped bundle with:

```
nkp create catalog-bundle --airgapped --collection-tag <tag>
```

The resulting tar file can be transferred to the airgapped environment and pushed to a local registry.

Relationship to Flux

Flux's `OCIRepository` source controller pulls OCI artifacts from the registry at the interval specified in the application's manifests. When Flux detects a new artifact revision, it reconciles the application's `HelmRelease` or `Kustomization` to the updated state.

Flux

Overview

NKP uses Flux (FluxCD) for GitOps: continuous deployment and reconciliation of configurations from Git. Flux is available with NKP Pro and NKP Ultimate. It lets you manage applications and platform configurations declaratively; changes in the Git repository are reconciled to the clusters. Project deployments in the platform guide use GitOps to deploy resources across clusters in a project.

HelmRelease

A HelmRelease is a Flux custom resource that installs or upgrades a Helm chart. NKP uses HelmReleases when deploying applications (e.g. platform or catalog apps) to workspaces or projects. You can verify deployments by checking HelmReleases in the target namespace (e.g. `kubectl get helmreleases -n <namespace>`). Configuration overrides can be applied via a ConfigMap referenced by the HelmRelease or AppDeployment.

Kustomization

A Kustomization is a Flux resource that applies Kubernetes manifests from a Git repository or other source, optionally using Kustomize. In NKP, Kustomizations support GitOps workflows for deploying and managing resources across clusters in a project or workspace. NKP fills in per-deployment values on a Kustomization's `spec.postBuild.substitute`; see Substitution variables.

Resources

- Flux documentation
- Flux concepts
- HelmRelease API reference
- Kustomization API reference
- <a href={`https://github.com/mesosphere/kommander-applications/tree/release-$2.19/applications/kommander-flux/`}>Flux configuration shipped with NKP 2.19

Linter config

This page documents the Bloodhound config structure used for catalog application validation. Note that this file is purely used for CI/CD tooling and is not used for application deployments.

File location

Use either `.bloodhound.yaml` or `.bloodhound.yml` (both are supported). You can place the file in one of these locations:

- **Repository-level config** (repository root): defines defaults shared by all applications and versions.
- **Application-level config** (`applications/<app>/<version>/`): overrides repository defaults for a specific application version when validation needs differ.

If both files are present, the application-level file takes precedence.

Required field

- `k8s_versions` (array of strings, required)
  - Each entry must be a valid semver-style Kubernetes version.
  - Bloodhound validates each version against the upstream Kubernetes JSON schema index.

Example:

```
k8s_versions:
  - 1.33.0
```

Optional fields

- `paths` (array of strings)
  - Paths to load/parse, in order.
- `strict` (boolean)
  - Enable strict schema validation.
  - Default in generated config is `true`.
- `skip_types` (array of strings)
  - API groups/types to skip (for example `group/version` or `group/version/Kind`).
- `additional_crds` (array of strings)
  - Additional CRD YAML URLs to load.
- `substitution_vars` (map of string to string)
  - Variable values used for substitutions such as `${releaseNamespace}`.

Default config structure

The default config emitted by Bloodhound contains:

```
k8s_versions:
  - 1.33.0

strict: true
skip_types: []
additional_crds: []
substitution_vars:
  releaseNamespace: kommander
  workspaceNamespace: workspace
```

Validation behavior

- Unknown fields are rejected (`KnownFields(true)` in YAML decoder).
- `k8s_versions` must be present.
- Invalid or unknown Kubernetes versions fail validation.

Substitution variables

Overview

When NKP deploys a catalog application, it fills in a set of variables on the Flux Kustomization's `spec.postBuild.substitute`. Reference them in your manifests as `${variable}` instead of hardcoding values, and Flux replaces them at deploy time.

Available variables

Each variable becomes available in the NKP version noted below and remains available in all later versions.

Variable	Introduced in NKP	Availability	Description
`releaseName`	2.16	Always	The deployment's release name: the AppDeployment's release-name annotation if set, otherwise its `metadata.name`.
`releaseNamespace`	2.16	Always	The namespace the application is deployed into: the workspace namespace for workspace deployments, or the project namespace for project deployments.

Variable	Introduced in NKP	Availability	Description
`workspaceNamespace`	2.16	Always	The workspace's namespace.
`appName`	2.17	Always	The catalog application's ID, derived from its `applications/<app-name>` directory (e.g. `jupyterhub`).
`appVersion`	2.17	Always	The catalog application's version, derived from its `applications/<app-name>/<version>` directory (e.g. `4.3.2`).
`projectNamespace`	2.17	Project deployments only	The project's namespace.
`clusterName`	2.19	Always	The name of the target cluster (the KommanderCluster's name).
`workspaceName`	2.19	When the target namespace resolves to a workspace or project	The name of the Workspace the deployment belongs to (for a project deployment, its parent workspace).
`projectName`	2.19	Project deployments only	The name of the project the deployment belongs to.
`clusterUUID`	2.19	When the `kommander.mesosphere.io/cluster-id` label is set	The target cluster's ID, taken from that label on the KommanderCluster.
`clusterType`	2.19	When a value is available	The target cluster's type, resolved in precedence order: (1) for CAPI-managed clusters, the referenced CAPI Cluster's `spec.infrastructureRef.kind` (for example `NutanixCluster`, `AWSCluster`, `AzureCluster`, `DockerCluster`); (2) for non-CAPI clusters, the `kommander.mesosphere.io/cluster-type` label, one of `EKS`, `GKE`, `AKS`, `Unknown`.

Configuration overrides in Flux HelmRelease

Overview

You can override default Helm chart values for applications by providing a ConfigMap. When creating an AppDeployment (e.g. via the NKP CLI), use a `--config-overrides` flag with the name of a ConfigMap that contains your custom values (e.g. in a `values.yaml` key). Kommander waits for the ConfigMap to exist before deploying the application. In the UI, use **Configure Application** or upload a YAML file with overrides when enabling an application.

Guides

Initialize a catalog repository

Overview

A catalog repository holds the definitions and metadata for applications that appear in the NKP catalog (workspace or project). Setting it up is the first step before adding applications and building artifacts.

Repository layout

A catalog repository follows this structure:

```
<repo-root>/
  .bloodhound.yml          # Global validation config
  .release.yaml            # Release specification (maps NKP versions to app sets)
  applications/
    <app-name>/
      <version>/
        metadata.yaml      # Application metadata (display name, description, etc.)
        kustomization.yaml # Top-level Kustomize config
        helmrelease.yaml   # Flux Kustomization resource
        helmrelease/
          kustomization.yaml
          helmrelease.yaml  # OCIRepository + HelmRelease
          cm.yaml           # Default Helm values ConfigMap
```

Steps

1. **Create the repository** — Initialize a new Git repository or use an existing one.
2. **Generate the scaffold** — Use the CLI to generate the directory structure:

   ```
   nkp generate catalog-repository --apps=<app-name>=<version>
   ```

   This creates the `applications/<app-name>/<version>/` directory with template files.
3. **Configure validation** — Add a `.bloodhound.yml` at the repository root to configure schema validation settings (e.g. Kubernetes version, strict mode).
4. **Add the release spec** — Create `.release.yaml` at the repository root to define how releases are tagged and which applications are included. See Continuous Delivery for details.
5. **Validate** — Run validation to ensure the repository is well-formed:

```
nkp validate catalog-repository --repo-dir=.
```

Initialize a catalog application

Overview

Adding a new application to a catalog repository involves generating the required files (metadata, Kustomize config, HelmRelease manifests) and customizing them for the application.

Steps

1. **Generate the application scaffold:**

   ```
   nkp generate catalog-repository --apps=<app-name>=<version>
   ```

   This creates `applications/<app-name>/<version>/` with template files.

2. **Edit** `metadata.yaml` — Fill in the required fields:

   - `displayName`, `description`, `category`, `licensing`, `scope`
   - `overview` (Markdown), `supportLink`, `icon`
   - `dependencies` and `requiredDependencies`
   - `allowMultipleInstances`, `type`
   - Set `scope` to `workspace` unless a project-only requirement is explicitly defined.
   - Provide an application `icon` (required for submission review).

3. **Configure the HelmRelease** — In `helmrelease/helmrelease.yaml`:

   - Set the `OCIRepository` URL to point to your chart's OCI registry location
   - Set `targetNamespace` to the application's dedicated namespace
   - Configure `install.createNamespace: true`

4. **Set default values** — Edit `helmrelease/cm.yaml` to provide default Helm values for the application.

5. **Add optional files** as needed:

   - Per-app `.bloodhound.yaml` for validation overrides
   - Additional ConfigMaps for UI dashboard integration

6. **Validate the repository:**

   ```
   nkp validate catalog-repository --repo-dir=.
   ```

7. **Build:**

```
nkp create catalog-bundle --collection-tag <tag>
```

Key conventions

- Use Flux substitution variables (`${releaseName}` , `${releaseNamespace}`) instead of hardcoded values.
- Every YAML document must end with `---` on its own line.

Convert Helm charts to OCI

Overview

When an application's Helm chart is hosted in a traditional Helm repository (not OCI-native), you need to pull it locally and push it to an OCI registry so NKP and Flux can consume it.

Steps

1. **Add the Helm repository:**

   ```
   helm repo add <repo-name> <repo-url>
   helm repo update
   ```

2. **Search for available versions:**

   ```
   helm search repo <repo-name>/<chart> --versions
   ```

3. **Pull the chart locally:**

   ```
   helm pull <repo-name>/<chart> --version <version>
   ```

   This downloads a `.tgz` file to the current directory.

4. **Push to your OCI registry:**

   ```
   helm push <chart>-<version>.tgz oci://<your-oci-registry>/<chart>
   ```

5. **Reference the OCI URL** in your application's `helmrelease/helmrelease.yaml` :

   ```
   apiVersion: source.toolkit.fluxcd.io/v1beta2
   kind: OCIRepository
   metadata:
     name: ${releaseName}-source
   ```

```
  namespace: ${releaseNamespace}
spec:
  interval: 6h0m0s
  url: oci://<your-oci-registry>/<chart>
  ref:
    tag: <version>
```

Authentication

If your OCI registry requires authentication, configure credentials before pushing:

```
helm registry login <registry> --username <user> --password <token>
```

For GHCR (GitHub Container Registry):

```
echo $GITHUB_TOKEN | helm registry login ghcr.io --username <github-user> --password-stdin
```

Build catalog application

Overview

The build step packages one or more applications from a catalog repository into a bundle (tar archive) that can be pushed to an OCI registry for deployment on NKP clusters.

Build command

```
nkp create catalog-bundle [flags]
```

Common usage

Build a bundle for all applications matching an NKP version constraint:

```
nkp create catalog-bundle --constraint nkpVersion=<nkp-version> --collection-tag v<nkp-version>.0
```

Build a bundle for specific applications:

```
nkp create catalog-bundle --apps=<app1>=<version1> --apps=<app2>=<version2> --collection-tag v1.0.0
```

Airgapped bundles

Include container images for disconnected environments:

```
nkp create catalog-bundle --airgapped --collection-tag v1.0.0
```

Output

By default the bundle is written to the current directory. Use `--output-file` to specify a custom path:

```
nkp create catalog-bundle --collection-tag v1.0.0 -o ./dist/my-catalog.tar
```

Prerequisites

- The catalog repository must pass validation (`nkp validate catalog-repository --repo-dir=.`)
- Helm charts referenced by applications must be accessible (either in OCI registries or local)

Push artifacts

Overview

After building a catalog bundle, push it to an OCI registry so NKP clusters can pull and deploy the catalog collection.

Push command

```
nkp push bundle --bundle
 --to-registry <registry-url>
```

Example

```
nkp push bundle \
  --bundle ./nkp-ai-app-catalog.tar \
  --to-registry ghcr.io/nutanix-cloud-native/nkp-ai-applications-catalog
```

Authentication

Ensure you are authenticated to the target OCI registry before pushing:

```
echo $GITHUB_TOKEN | docker login ghcr.io --username <user> --password-stdin
```

Or for other registries:

```
docker login <registry> --username <user> --password <token>
```

Deploying the pushed catalog on a cluster

Once the bundle is pushed, deploy it on an NKP cluster:

```
nkp create catalog-collection \
  --url oci://<registry>/<catalog-name>/<catalog-name>/collection \
  --tag <version> \
  --workspace <workspace-name>
```

Run `nkp push bundle --help` for additional configuration flags.

Partner catalog publishing flow

For partner repository (`nkp-partner-catalog`), contributor flow is:

1. Add/update app definitions and metadata.
2. Open a PR to `main` .
3. After merge, the repository release pipeline publishes catalog artifacts using the repo release spec files (for example, `stable.yaml`).

In other words, for partner-catalog submissions you typically do **not** run manual `nkp push bundle` to a personal target registry as the final publication step. You can however publish to your own registry for testing before submission.

AppDeployment for Catalog App

Overview

Once a catalog collection is available on an NKP cluster, applications are deployed by creating an AppDeployment resource in a workspace or project. Kommander reconciles the AppDeployment and deploys the application (via Flux HelmRelease) to the selected clusters.

How it works

```
flowchart LR
    A[OCI Registry] -->|Flux pulls| B[OCIRepository]
    B --> C[HelmRelease]
    C --> D[Application Pods]
    E[AppDeployment] -->|Manages| C
```

1. **Catalog collection** is registered on the management cluster via `nkp create catalog-collection` .
2. Flux's `OCIRepository` controller pulls manifests from the OCI registry at the configured interval.
3. The **AppDeployment** resource (created via UI or CLI) triggers Kommander to deploy the application's `HelmRelease` to target clusters.

4. Flux reconciles the `HelmRelease`, installing or upgrading the Helm chart in the application's dedicated namespace.

Deploying an application

Using the NKP UI

1. Navigate to your workspace or project → **Applications**.
2. Find the application in the catalog listing.
3. Click **Enable** (or **Deploy**).
4. Optionally configure values overrides.

Using the NKP CLI

```
nkp create appdeployment <app-name> \
  --workspace <workspace-name> \
  --config-overrides <configmap-name>
```

Configuration overrides

Override default Helm values by providing a ConfigMap:

```
kubectl create configmap my-app-overrides \
  --from-file=values.yaml=./my-values.yaml \
  -n <workspace-namespace>
```

Then reference it when creating the AppDeployment. Kommander waits for the ConfigMap to exist before deploying.

Upgrading applications

Catalog applications can be upgraded individually (unlike platform applications which are upgraded as a set per workspace). When a new version is available in the catalog collection, use the UI or CLI to trigger the upgrade.

Continuous Delivery

This page describes how to set up a release pipeline for a given catalog repository.

Overview

The **catalog release specification** maps NKP (Nutanix Kubernetes Platform) versions to the set of applications supported in each release. You define this in a release spec file that drives how catalog artifacts are built and tagged. See the Specification for the JSON schema and validator.

Release spec file: By default, the release spec file is named `.release.yaml` and lives at the root of your catalog repository. Use this file (or the path your pipeline expects) when running the release command.

Catalog release spec

The top-level structure is CatalogReleaseSpec. It contains:

- **Schema** — Identifies the schema version (e.g. `catalog.nkp.nutanix.com/v1/catalog-release`).
- **Releases** — A list of **Release** entries, one per catalog version (typically aligned to an NKP version).

Release

Each **Release** describes a single catalog collection release:

Field	Description
collectionName	Name of the catalog collection artifact (e.g. `nkp-partner-catalog`). Defaults to the repository name.
tagName	OCI tag for this release (e.g. `2.19`). Must be a valid OCI tag (alphanumeric, `_`, `.`, `-`, up to 128 chars).
fileName	Name of the built artifact (e.g. `nkp-partner-catalog-2.19.tar`).
airgapped	If `true`, artifacts are built for airgapped use. Defaults to `false`.
constraints	Optional. Filters applications by NKP version (see below).
applications	Optional. Explicit list of applications (name + version) included in this release.

You must define **either** `constraints` **or** `applications` (or both); at least one is required. Validation fails if both are empty.

Application

Each application in a release is defined by:

- **name** — Application name.
- **version** — Application version.

Use **applications** when you want to pin exactly which apps (and versions) are in the release:

- Each entry has **name** (application name) and **version** (application version).

This gives full control over the set of applications and their versions for that release.

Constraints (filter by NKP version)

Use **constraints** when you want the release to include all applications that match an NKP version:

- **nkpVersion** — NKP version in `MAJOR.MINOR` or `MAJOR.MINOR.PATCH` format (e.g. `2.19` or `2.19.0`). Optional leading `v` is allowed.

The build process uses this to filter which applications are included in the catalog for that release.

Validation

- Every **Release** must have at least one of: **constraints** or **applications** (non-empty).
- **tagName** must match the standard OCI tag pattern.
- **nkpVersion** (when used) must match the semver-style pattern. E.g.: `2.19`.

These rules are enforced when the release spec is processed by the CLI.

Creating release artifacts

Once `.release.yaml` (or your release spec file) is in place, generate the release artifacts with:

```
nkp experimental catalog release
```

This command produces the catalog tar files from the spec. It does **not** push them to a registry. After the build, push the artifacts to your OCI registry using your usual workflow (for example, the steps in Push Artifacts).

Partner catalog note

For `nkp-partner-catalog`, publication is handled by repository release automation after changes are merged. The release mapping (NKP versions and included apps/versions) is defined by the release spec files, including `stable.yaml`.

Custom OCI Catalog Update and Verification

This guide outlines the end-to-end workflow for updating a custom application (like the Slurm Operator) in a private OCI registry and verifying its successful deployment across NKP Management and Workload clusters.

1. GitOps resource hierarchy and order of operations

NKP uses a declarative GitOps pipeline powered by FluxCD. Each step in the "get" sequence represents a specific state of the application lifecycle.

Layer 1: Source (OCIRepository)

What it is: The link to your external registry (GHCR). **What is inside:** The URL to your OCI artifact and the specific tag/digest. It validates that NKP can successfully "reach" and "download" the package.

Sample output:

```
NAME                    URL                                             READY   STATUS
slurm-operator-1.1.1    oci://ghcr.io/navidpadid/slurm-operator         True    stored artifact for digest 'v0.1
```

Troubleshooting: If `READY` is `False`, check your `secretRef` (credentials).

Layer 2: Bridge (Kustomization)

What it is: The logic that tells Kubernetes how to apply the manifests. **What is inside:** It points to the `OCIRepository` as its source and defines which namespace to target and which post-build variables (like cluster-specific labels) to inject.

Sample output:

```
NAME              READY   STATUS                                  AGE
slurm-operator    True    Applied revision: v0.1.0@sha256:4b41...    5m
```

Troubleshooting: If it hangs, the YAML manifests inside your OCI bundle may have syntax errors.

Layer 3: Deployment (HelmRelease)

What it is: The installer that manages the Helm chart lifecycle. **What is inside:** It contains the `values.yaml` overrides and the specific version of the chart to be installed. It tracks whether the `helm install` or `helm upgrade` command succeeded.

Sample output:

```
NAME              READY   STATUS                              AGE
slurm-operator    True    Release reconciliation succeeded    3m
```

Troubleshooting: If `READY` is `False`, check the status message for Helm hooks failing or OOM issues during install.

Layer 4: Result (Pods)

What it is: The actual running code. **What is inside:** The Linux containers executing your application logic.

Sample output:

```
NAME                              READY   STATUS    RESTARTS   AGE
slurm-operator-8b4cff689-tqj6v    1/1     Running   0          2m
```

Order of refresh: When you repush, you must see the digest change in Layer 1, which then triggers Layer 2 to re-apply, Layer 3 to upgrade, and finally Layer 4 to restart the pods.

2. Workflow: Updating a custom app (same tag)

When you repush an image to the same tag (for example, `v0.1.0`), NKP may not automatically detect the change due to caching. Follow these steps to force a refresh.

Step 1: Purge stale metadata (management cluster)

Remove the old registry pointers to ensure a clean re-scan.

```
### Delete the collection source
kubectl delete ocirepository nkp-ai-applications-catalog-collection \
  -n workload-cluster-01-jtnmw \
  --kubeconfig="mgmt.conf"

### Clear cached app list metadata
kubectl delete apps -n workload-cluster-01-jtnmw --all --kubeconfig="mgmt.conf"
```

Step 2: Re-register the collection (management cluster)

```
nkp create catalog-collection \
  --url oci://ghcr.io/navidpadid/nkp-ai-applications-catalog/collection \
  --tag v0.1.0 \
  --workspace "workload-cluster-01-ghfvk" \
  --kubeconfig="mgmt.conf"
```

Step 3: Enable and patch (workload cluster)

1. Enable in UI: refresh the Kommander UI and click **Enable** for your app (for example, Slurm) in the App Catalog.

   If you repush to the same tag (for example, `v0.1.0`), NKP might use a cached version and the app may not be visible in some cases. Use the `reconcileAt` annotation to bypass the cache and force an immediate pull from GitHub.

```
kubectl annotate ocirepository nkp-ai-applications-catalog-collection \
  fluxcd.io/reconcileAt=$(date +%s) \
  -n workload-cluster-01-jtnmw \
  --kubeconfig="mgmt.conf" --overwrite
```

Now enable the app using the UI. Confirm app list by using this command:

```
kubectl get apps -n workload-cluster-01-jtnmw --kubeconfig="nkp.cluster.navid.vlan173.final.conf"

NAME                                APP ID                          APP VERSION   SOURCE
agentgateway-2.2.0                  agentgateway                    2.2.0         agentgateway-2.2.0
amd-device-metrics-exporter-1.4.2   amd-device-metrics-exporter     1.4.2         amd-device-metrics-exporter-1
coder-2.30.2                        coder                           2.30.2        coder-2.30.2
```

2. Patch for private registry: tell the workload cluster to use your credentials.

Problem:

```
kubectl get ocirepository -n workload-cluster-01-jtnmw --kubeconfig=workload-cluster-01.conf

slurm-operator-source  oci://ghcr.io/navidpadid/nkp-ai-applications-catalog/slurm-operator   False   failed t
```

Solution:

```
kubectl patch ocirepository slurm-operator-source \
  -n workload-cluster-01-jtnmw \
  --kubeconfig=workload-cluster-01.conf \
  --type=merge -p '{"spec":{"secretRef":{"name":"host-cluster-oci-creds"}}}'

ocirepository.source.toolkit.fluxcd.io/slurm-operator-source patched
```

3. Step-by-step verification commands

Use these commands to confirm the update has propagated through all layers.

Phase A: Source verification

Verify the SHA256 digest has updated and the source is ready.

```
kubectl get ocirepository -n workload-cluster-01-jtnmw --kubeconfig=workload-cluster-01.conf
```

Sample output:

```
NAME                URL                                READY   STATUS
slurm-operator-1.1.1 oci://ghcr.io/navidpadid/slurm   True    stored artifact for digest 'v0.1.0@sha256:4b417
```

Phase B: Config and deployment verification

Check that the Kustomization and HelmRelease have reconciled.

```
### Check Kustomization
kubectl get kustomization -n workload-cluster-01-jtnmw --kubeconfig=workload-cluster-01.conf

### Check HelmRelease
kubectl get helmrelease -n workload-cluster-01-jtnmw --kubeconfig=workload-cluster-01.conf
```

Sample output:

```
NAME            READY   STATUS                             AGE
slurm-operator  True    Release reconciliation succeeded   2m
```

Phase C: Workload verification

The final confirmation that the containers are healthy.

```
kubectl get pods -n slurm-operator --kubeconfig=workload-cluster-01.conf
```

Sample output:

```
NAME                                  READY   STATUS    RESTARTS   AGE
slurm-operator-8b4cff689-tqj6v        1/1     Running   0          90s
slurm-operator-webhook-5658954b8d-drfnl   1/1     Running   0          90s
```

Tip: If `AGE` is very low, it confirms the update just rolled out.

Examples

Simple applications

Overview

A simple catalog application is a single Helm chart with no mandatory dependencies. In the NKP UI you enable it from the Applications list for a workspace or project; the platform deploys it via HelmRelease.

Use this pattern when the app does not require other platform or catalog apps to be installed first.

When to use

- The application is self-contained (one chart, one namespace)
- No CRDs or cluster-scoped resources that would conflict with other instances
- No required dependencies on Istio, GPU operators, or other catalog apps

File structure

```
applications/<app-name>/<version>/
  metadata.yaml
  kustomization.yaml
  helmrelease.yaml
  helmrelease/
    kustomization.yaml
    helmrelease.yaml      # OCIRepository + HelmRelease
    cm.yaml               # Default Helm values ConfigMap
```

Key points

- Set `allowMultipleInstances: true` in `metadata.yaml` if the app does not install CRDs or cluster-scoped resources.
- Use `targetNamespace: <app-namespace>` and `install.createNamespace: true` so the app deploys into its own namespace.
- Keep the default values ConfigMap (`cm.yaml`) minimal — users can override via AppDeployment config overrides.
- The `dependencies` and `requiredDependencies` lists in `metadata.yaml` should both be empty (`[]`).

Example

Refer to the AI Applications Catalog repository for examples of simple applications such as inference engines and vector databases.

Complex applications

Overview

Complex catalog applications have dependencies on other platform or catalog applications. For example, an application may require Istio for service mesh, Gateway API CRDs for ingress, or a specific operator to be installed first.

When to use

- The application depends on CRDs provided by another catalog or platform app
- The application requires a service mesh, ingress controller, or GPU operator
- The application installs cluster-scoped resources (set `allowMultipleInstances: false`)

Declaring dependencies

In `metadata.yaml`, use:

- **`requiredDependencies`** — Hard dependencies. NKP ensures these are installed first. Use platform app names from kommander-applications (e.g. `istio`, `gateway-api-crds`).
- **`dependencies`** — Soft dependencies. Recommended but not enforced.

```
requiredDependencies:
  - istio
  - gateway-api-crds
dependencies:
  - prometheus
```

Key points

- Never disable application features (e.g. `USE_ISTIO=false`) to avoid dependency crashes — declare the dependency instead.
- Deploy dependencies first; if a dependency fails, the dependent application will not deploy successfully.
- For applications that install CRDs, set `allowMultipleInstances: false` to prevent conflicts.
- Some complex applications use a Job-based installer (`<app>-install.yaml`) instead of a plain HelmRelease for multi-step setup.

Example

Applications that follow this pattern include ML pipelines (which depend on training operators) and central dashboards (which depend on authentication and Istio). Browse the AI Applications Catalog for examples.

Applications with declarative input

Overview

Catalog applications can accept user-provided configuration at deploy time through declarative inputs. This lets users customize values (e.g. model names, replica counts, storage settings) without modifying the default Helm chart.

How it works

1. The application defines an **application user inputs** schema (see Application User Inputs) that describes the configurable fields, their types, defaults, and validation rules.
2. At deploy time, the NKP UI renders a form from this schema, or the user provides a ConfigMap via the CLI.
3. Kommander merges the user-provided values with the chart defaults and deploys the application.

Providing overrides via the UI

1. Navigate to **Applications** in your workspace.
2. Click **Enable** on the application.
3. Fill in the configuration form (generated from the user inputs schema).
4. Click **Deploy**.

Providing overrides via the CLI

Create a ConfigMap with your custom values:

```
kubectl create configmap my-app-config \
  --from-file=values.yaml=./my-values.yaml \
  -n <workspace-namespace>
```

Then reference it when deploying:

```
nkp create appdeployment <app-name> \
  --workspace <workspace-name> \
  --config-overrides my-app-config
```

Kommander waits for the ConfigMap to exist before deploying the application.

Key points

- The user inputs schema is optional — applications work without it, but it provides a better UI experience.
- Overrides are merged with the default values ConfigMap (`cm.yaml`); user values take precedence.

- See Configuration overrides for how overrides interact with Flux HelmReleases.

API

Application Metadata

JSON Schema for **application metadata**: catalog and application manifest structure and fields.

Schema file: applicationmetadata.schema.json

Use the reference and validator below to explore the schema and validate your YAML or JSON in the browser.

> **Note:** The interactive schema validator is only on the live docs site. Use the schema JSON link above when working offline.

Application User Inputs

JSON Schema for **application user inputs**: declarative user input definitions for catalog applications.

Schema file: applicationuserinputs.schema.json

Use the reference and validator below to explore the schema and validate your YAML or JSON in the browser.

> **Note:** The interactive schema validator is only on the live docs site. Use the schema JSON link above when working offline.

Catalog Release Specification

JSON Schema for **CatalogReleaseSpec**: the catalog release specification that maps NKP versions to supported applications. Each release defines a catalog collection (e.g. tag, file name, airgapped flag) and either **constraints** (filter by NKP version) or an explicit **applications** list.

Schema file: catalogreleasespec.schema.json

Use the reference and validator below to explore the schema and validate your YAML or JSON in the browser. A standalone page is also available at /schema-validator.

> **Note:** The interactive schema validator is only on the live docs site. Use the schema JSON link

> above when working offline.

CLI

CLI reference

CNCF AI Conformance

CNCF AI Conformance



The CNCF Kubernetes AI Conformance program defines extra capabilities, APIs, and configurations a Kubernetes distribution must provide — on top of standard CNCF Kubernetes Conformance — to run AI/ML workloads reliably.

NKP participates in this program. Use the table below to open the conformance evidence for each NKP release.

NKP versions

NKP version	Kubernetes	Status	Details
2.19	v1.36	Certified	NKP 2.19 AI Conformance
2.18	v1.35	Certified	NKP 2.18 AI Conformance

Upstream submissions

- v1.36 NKP · v1.35 NKP (cncf/k8s-ai-conformance)
- CNCF Certified Kubernetes (AI Conformance builds on this)

CNCF AI Conformance



The CNCF Kubernetes AI Conformance defines a set of additional capabilities, APIs, and configurations that a Kubernetes cluster MUST offer, on top of standard CNCF Kubernetes Conformance, to reliably and efficiently run AI/ML workloads.

This page shows how NKP 2.18 meets these requirements using Kubernetes v1.35.2 and the NVIDIA GPU Operator v26.3.0, deployed via Flux CD as platform HelmReleases.

> **NOTE:** NKP is **CNCF Kubernetes AI Conformant** for v1.35; see the v1.35 NKP AI conformance submission. NKP also holds **CNCF Certified Kubernetes** conformance, which AI Conformance builds on. That certification was submitted under NKP's former product name, **Konvoy**; see the v1.35 Konvoy conformance submission. For installation, configuration, and operations guidance, see the official NKP 2.18 documentation.

Requirements Overview

NKP 2.18 satisfies all 12 CNCF AI Conformance requirements — 8 MUST and 4 SHOULD. Each row below maps a requirement — identified by its upstream ID and grouped by upstream category — to its level and the section of this page that shows how NKP meets it.

Requirement	Level	Supported	Section
`accelerators`			
`dra_support`	MUST	✅	Support Dynamic Resource Allocation (DRA)
`driver_runtime_management`	SHOULD	✅	Driver Runtime Management
`gpu_sharing`	SHOULD	✅	GPU Sharing
`virtualized_accelerator`	SHOULD	✅	Virtualized Accelerator (vGPU)
`networking`			
`ai_inference`	MUST	✅	Support the Gateway API
`schedulingOrchestration`			
`gang_scheduling`	MUST	✅	Gang Scheduling
`cluster_autoscaling`	MUST	✅	Cluster Autoscaling
`pod_autoscaling`	MUST	✅	Horizontal Pod Autoscaler
`observability`			
`accelerator_metrics`	MUST	✅	Accelerator Performance Metrics
`ai_service_metrics`	MUST	✅	AI Job and Inference Service Metrics
`security`			
`secure_accelerator_access`	MUST	✅	Secure Accelerator Access
`operator`			
`robust_controller`	MUST	✅	Robust CRD and Controller Operation

Verification Setup

Environment

The steps below were verified on the following NKP 2.18 cluster and workspace. Commands throughout this page target the `kommander-workspace` (namespace `kommander`).

```
kubectl get clusters -A
```

```
NAMESPACE   NAME          CLUSTERCLASS          AVAILABLE   CP DESIRED   CP AVAILABLE   CP UP-TO-DATE   W DES
kommander   dlipovetsky   nkp-nutanix-v2.18.0   True        3            3              3               5
```

```
kubectl get workspaces -A
```

```
NAME                 DISPLAY NAME                  WORKSPACE NAMESPACE           AGE
default-workspace    Default Workspace             kommander-default-workspace   3d3h
kommander-workspace  Management Cluster Workspace  kommander                     3d3h
```

Prerequisites

Several checks use operators from the NKP AI Applications catalog. Add the catalog collection to your workspace before running them:

```
nkp create catalog-collection \
  --url oci://ghcr.io/nutanix-cloud-native/nkp-ai-applications-catalog/collection \
  --tag 2.18 \
  --workspace kommander-workspace
```

Once the collection is available, its applications are listed as `App` resources in the workspace namespace and can be deployed from the NKP catalog:

```
kubectl get apps -n kommander
```

▶ Expected output — catalog applications (`App` resources) in the workspace

Support Dynamic Resource Allocation (DRA)

This requirement is a **MUST**: *the platform must support the Dynamic Resource Allocation (DRA) APIs to enable more flexible, fine-grained resource requests beyond simple counts.*

Dynamic Resource Allocation (DRA) is a Kubernetes API for flexible, fine-grained resource requests beyond simple counts. The DRA v1 API group (`resource.k8s.io/v1`) has been **generally available**

since Kubernetes v1.34 and is enabled by default on NKP 2.18 (Kubernetes v1.35.2) — no additional configuration or driver is required for the APIs to be served.

Verify that all `resource.k8s.io/v1` DRA API resources are served:

```
kubectl api-resources --api-group=resource.k8s.io
```

Expected output:

```
NAME                     SHORTNAMES   APIVERSION           NAMESPACED   KIND
deviceclasses                         resource.k8s.io/v1   false        DeviceClass
resourceclaims                        resource.k8s.io/v1   true         ResourceClaim
resourceclaimtemplates                resource.k8s.io/v1   true         ResourceClaimTemplate
resourceslices                        resource.k8s.io/v1   false        ResourceSlice
```

> **TIP:** NKP serves the DRA APIs (`resource.k8s.io/v1`) out of the box, enabling structured, fine-grained resource requests beyond simple counts. On the shipped NKP configuration, GPUs are allocated through the NVIDIA device plugin, and the DRA APIs are available for DRA-aware drivers that consume them.

Driver Runtime Management

This requirement is a **SHOULD**: *the platform should provide a verifiable mechanism ensuring that compatible accelerator drivers and the corresponding container-runtime configuration are correctly installed and maintained on accelerator nodes.*

NKP ships the NVIDIA GPU Operator as a platform component to manage accelerator drivers, container toolkit, device plugin, and runtime configuration on GPU nodes. Node Feature Discovery (NFD) and GPU Feature Discovery (GFD) provide node metadata attestation.

Verify Container Runtime Version

```
kubectl get nodes -l nvidia.com/gpu.present=true \
  -o jsonpath='{.items[*].status.nodeInfo.containerRuntimeVersion}'
```

Expected output:

```
containerd://2.1.6-d2iq.1
```

Verify Accelerator Driver and Runtime Configuration

Query node labels to verify driver installation and container toolkit injection:

```
kubectl get nodes -l nvidia.com/gpu.present=true \
  -o jsonpath='{range .items[*].metadata.labels}{range $k,$v := @}{$k}={$v}{"\n"}{end}{end}' \
  | grep nvidia.com
```

Key labels:

```
nvidia.com/cuda.driver-version.full=580.126.18
nvidia.com/gpu.deploy.container-toolkit=true
nvidia.com/gpu.deploy.device-plugin=true
nvidia.com/gpu.deploy.driver=true
nvidia.com/gpu.deploy.gpu-feature-discovery=true
nvidia.com/gpu.deploy.dcgm-exporter=true
nvidia.com/gpu.family=ampere
nvidia.com/gpu.memory=46068
nvidia.com/gpu.product=NVIDIA-A40
```

- `gpu.deploy.container-toolkit=true` confirms the container runtime configuration (the toolkit patch to `containerd`) is injected and active.
- `gpu.deploy.driver=true` confirms the NVIDIA driver is managed by the operator.

Verify GPU Operator Pods

```
kubectl get pods -n kommander | grep nvidia
```

Expected output:

```
nvidia-container-toolkit-daemonset-7m6g7          1/1     Running     0   2d
nvidia-cuda-validator-6jdzd                       0/1     Completed   0   2d
nvidia-dcgm-77msj                                 1/1     Running     0   2d
nvidia-dcgm-exporter-9nnws                        1/1     Running     2   2d
nvidia-device-plugin-daemonset-x8k4p              1/1     Running     0   2d
nvidia-operator-validator-dqvks                   1/1     Running     0   2d
```

Verify HelmReleases

```
kubectl get helmreleases -n kommander | grep nvidia
```

Expected output:

```
nvidia-gpu-operator   2d   True   Helm install succeeded for release kommander/nvidia-gpu-operator.v1 with ch
```

> **TIP:** The NVIDIA GPU Operator automates the installation, configuration, and lifecycle of accelerator drivers, container toolkit, device plugin, and runtime on GPU nodes, with Node Feature Discovery and GPU Feature Discovery publishing node metadata for attestation.

GPU Sharing

This requirement is a **SHOULD**: *for accelerators that support GPU sharing, the platform must provide a well-defined mechanism for at least one sharing strategy so workloads that do not need a full dedicated GPU can improve utilization. If software-based sharing (e.g. time-slicing) is supported, oversubscription of GPUs should be allowed.*

NKP satisfies this through the **NVIDIA GPU Operator** — the same platform component that manages the driver, toolkit, and device plugin — using **time-slicing**, a software-based sharing strategy in which one physical GPU is advertised as *N* schedulable `nvidia.com/gpu` replicas, allowing *N* pods to oversubscribe a single GPU. This is exactly the software-based, oversubscription path the requirement calls out, and it works on every NVIDIA GPU (including the **NVIDIA A40** on the verification cluster).

The evidence below proves the end-to-end path: enable time-slicing → the node advertises a single physical GPU as multiple schedulable slices → multiple independent pods each request a whole `nvidia.com/gpu` and are all placed on that one physical GPU.

Enable Time-Slicing on the GPU Operator

Create a device-plugin sharing config that advertises each physical GPU as 4 replicas, and point the `ClusterPolicy` at it. The GPU Operator rolls out the change automatically.

```
kubectl apply -f - <<'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: time-slicing-config
  namespace: kommander
data:
  any: |-
    version: v1
    sharing:
      timeSlicing:
        resources:
        - name: nvidia.com/gpu
          replicas: 4
EOF

kubectl patch clusterpolicy cluster-policy --type merge \
  -p '{"spec":{"devicePlugin":{"config":{"name":"time-slicing-config","default":"any"}}}}'
```

Verify the Node Advertises Oversubscribed GPU Slices

Once the device plugin and GPU Feature Discovery pods have restarted, the node reports the sharing strategy and the oversubscription factor:

```
kubectl get node <gpu-node> -o json \
  | jq -r '.metadata.labels | to_entries[] | select(.key|test("nvidia.com/gpu.(count|product|replicas|sharing
```

```
nvidia.com/gpu.count=1
nvidia.com/gpu.product=NVIDIA-A40-SHARED
nvidia.com/gpu.replicas=4
nvidia.com/gpu.sharing-strategy=time-slicing
```

The node has **one physical GPU** (`gpu.count=1`) but now advertises it as **4 schedulable slices**. This is reflected directly in the node's schedulable capacity:

```
kubectl get node <gpu-node> \
  -o jsonpath='capacity={.status.capacity.nvidia\.com/gpu} allocatable={.status.allocatable.nvidia\.com/gpu}{
```

```
capacity=4 allocatable=4
```

The `-SHARED` product suffix and `sharing-strategy=time-slicing` label make the sharing mode explicit and machine-discoverable for schedulers and operators.

Deploy Multiple Pods That Each Request a Whole GPU

Deploy 4 replicas, each requesting `nvidia.com/gpu: 1`. Without sharing, only one pod could ever run on a single-GPU node; with time-slicing, all four schedule onto the same physical GPU.

```
kubectl create namespace gpu-sharing-demo
kubectl apply -f - <<'EOF'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: time-slicing-verify
  namespace: gpu-sharing-demo
spec:
  replicas: 4
  selector:
    matchLabels:
      app: time-slicing-verify
  template:
    metadata:
```

```
      labels:
        app: time-slicing-verify
    spec:
      tolerations:
      - key: nvidia.com/gpu
        operator: Exists
        effect: NoSchedule
      containers:
      - name: cuda
        image: nvcr.io/nvidia/cuda:12.5.0-base-ubuntu22.04
        command: ["/bin/sh","-c","nvidia-smi -L; sleep 3600"]
        resources:
          limits:
            nvidia.com/gpu: 1
EOF
```

All 4 pods reach `Running` on the single GPU node:

```
kubectl -n gpu-sharing-demo get pods -o wide
```

```
NAME                                  READY   STATUS    NODE
time-slicing-verify-6f76775cb9-8pvgb   1/1     Running   dlipovetsky-gpu-jz4tk-t2dqf-f4mp2
time-slicing-verify-6f76775cb9-gfg8h   1/1     Running   dlipovetsky-gpu-jz4tk-t2dqf-f4mp2
time-slicing-verify-6f76775cb9-mzbqv   1/1     Running   dlipovetsky-gpu-jz4tk-t2dqf-f4mp2
time-slicing-verify-6f76775cb9-sxhcd   1/1     Running   dlipovetsky-gpu-jz4tk-t2dqf-f4mp2
```

The node confirms all four slices are consumed (`4` requested against `4` allocatable):

```
kubectl describe node <gpu-node> | grep -A2 'Allocated resources' | grep nvidia.com/gpu
```

```
  nvidia.com/gpu     4                4
```

Prove the Pods Share One Physical GPU

Each pod runs `nvidia-smi -L`. All four report the **identical physical GPU UUID** — the single A40 on the host — proving true oversubscription rather than four separate devices:

```
for p in $(kubectl -n gpu-sharing-demo get pods -o jsonpath='{.items[*].metadata.name}'); do
  echo "--- $p ---"; kubectl -n gpu-sharing-demo logs $p | grep -i UUID
done
```

```
--- time-slicing-verify-6f76775cb9-8pvgb ---
GPU 0: NVIDIA A40 (UUID: GPU-ff400afb-03b6-3cd9-2fa9-5b8d002ad839)
```

```
--- time-slicing-verify-6f76775cb9-gfg8h ---
GPU 0: NVIDIA A40 (UUID: GPU-ff400afb-03b6-3cd9-2fa9-5b8d002ad839)
--- time-slicing-verify-6f76775cb9-mzbqv ---
GPU 0: NVIDIA A40 (UUID: GPU-ff400afb-03b6-3cd9-2fa9-5b8d002ad839)
--- time-slicing-verify-6f76775cb9-sxhcd ---
GPU 0: NVIDIA A40 (UUID: GPU-ff400afb-03b6-3cd9-2fa9-5b8d002ad839)
```

The host-side view confirms there is exactly one physical GPU behind those four slices:

```
kubectl exec -n kommander <nvidia-dcgm-pod> -- nvidia-smi -L
```

```
GPU 0: NVIDIA A40 (UUID: GPU-ff400afb-03b6-3cd9-2fa9-5b8d002ad839)
```

> **TIP:** Four independent pods, each granted a full `nvidia.com/gpu`, are all scheduled onto — and share — the same physical A40, the software-based sharing / oversubscription mechanism the requirement calls for.

Virtualized Accelerator (vGPU)

NVIDIA vGPU lets a hypervisor partition a physical GPU into multiple virtual GPUs that are assigned to separate VMs. It requires vGPU-capable hardware, the NVIDIA vGPU host driver on the hypervisor, and the NVIDIA License System. This requirement is a **SHOULD**: *for accelerators that support vGPU, the platform must provide well-defined mechanisms to expose and manage them consistently with physical fractional GPUs.*

NKP addresses this requirement as a first-class platform capability. Beginning with NKP 2.16, worker node pools may be provisioned as **vGPU-enabled node pools** — in addition to full GPU passthrough — on both **Nutanix AHV** and **VMware vSphere**. Virtual GPUs are managed through the same control path NKP uses for physical GPUs:

- Each virtual machine in a vGPU node pool is allocated an NVIDIA vGPU profile by the hypervisor.
- The **NVIDIA GPU Operator**, configured for vGPU with the licensed vGPU guest driver and an NVIDIA License System token (supplied via a `licensing-config` ConfigMap), installs the driver and advertises each virtual GPU as a standard **`nvidia.com/gpu`** schedulable resource.
- Because virtual GPUs are surfaced through the **same device plugin and resource name** as physical GPUs, they are scheduled, monitored via DCGM (see Accelerator Performance Metrics), autoscaled as ordinary Cluster API `MachineDeployment` node groups (see Cluster Autoscaling), and consumed by workloads **identically** to physical GPUs. This delivers the API and lifecycle consistency the requirement specifies.

Consistent Management Across Physical and Virtual GPUs

The verification cluster runs the NVIDIA GPU Operator on a Nutanix **AHV** worker node. GPU Feature Discovery advertises the accelerator's state — including the `nvidia.com/vgpu.present` label, which reports `true` on vGPU node pools — and the device plugin exposes the accelerator as `nvidia.com/gpu`:

```
kubectl get nodes -l nvidia.com/gpu.present=true \
  -o jsonpath='{range .items[*].metadata.labels}{@}{"\n"}{end}' \
  | tr ',' '\n' | grep -E 'nvidia.com/(gpu.machine|gpu.product|gpu.mode|gpu.count|vgpu.present|gpu.sharing-st
```

```
nvidia.com/gpu.machine=AHV
nvidia.com/gpu.product=NVIDIA-A40
nvidia.com/gpu.mode=compute
nvidia.com/gpu.count=1
nvidia.com/gpu.sharing-strategy=none
nvidia.com/vgpu.present=false
```

```
kubectl get node dlipovetsky-gpu-jz4tk-t2dqf-f4mp2 \
  -o jsonpath='{.status.allocatable.nvidia\.com/gpu}{"\n"}'
### 1
```

On this node the A40 is attached in **passthrough mode** (`gpu.mode=compute`, `vgpu.present=false`) and therefore advertises one full `nvidia.com/gpu`. On a vGPU node pool, the same operator stack reports `nvidia.com/vgpu.present=true` and advertises the assigned vGPU profile as `nvidia.com/gpu` in exactly the same manner. A workload's specification (`resources.limits: nvidia.com/gpu: 1`) remains unchanged whether the backing device is a physical GPU or a virtual GPU, preserving full API consistency across both.

> **NOTE:** Dynamic Resource Allocation (DRA) is the emerging Kubernetes mechanism for structured accelerator allocation. The optional **NVIDIA DRA driver** can be deployed on NKP to allocate GPUs via DRA `DeviceClass` / `ResourceClaim` objects; as the NVIDIA DRA driver adds vGPU support, NKP will expose vGPU through the same DRA path, per the requirement's forward-looking guidance.

Resources

- NKP 2.18 — Configure GPU node pools for Nutanix clusters
- vGPU on Nutanix (TN-2046)
- NVIDIA GPU Operator — Using NVIDIA vGPU

> **TIP:** NKP provides a well-defined, first-class mechanism (vGPU-enabled node pools on AHV/vSphere with the NVIDIA GPU Operator) that exposes and manages virtualized accelerators

> consistently with physical GPUs — the same `nvidia.com/gpu` device-plugin resource, scheduling, metrics, and node-pool autoscaling — with a clear forward path to DRA-based vGPU allocation.

Support the Gateway API

This requirement is a **MUST**: *the platform must support the Kubernetes Gateway API with an implementation that provides advanced traffic management for inference services — weighted traffic splitting, header-based routing (e.g. OpenAI protocol headers), and optional service-mesh integration.*

NKP ships the Kubernetes **Gateway API** CRDs (`gateway.networking.k8s.io/v1`) out of the box, and provides Envoy Gateway — a CNCF Gateway API implementation — in the Nutanix Products catalog for advanced traffic management of inference services. Envoy Gateway delivers exactly the capabilities the requirement calls for: **weighted traffic splitting** (canary/blue-green model rollouts), **header-based routing** (e.g. routing on OpenAI-style protocol headers), and optional **service-mesh integration**.

The following was verified on the NKP 2.18 cluster.

Verify Gateway API Resources

```
kubectl api-resources --api-group=gateway.networking.k8s.io
```

The standard Gateway API kinds are served at `v1`:

```
NAME              SHORTNAMES   APIVERSION                          NAMESPACED   KIND
gatewayclasses    gc           gateway.networking.k8s.io/v1        false        GatewayClass
gateways          gtw          gateway.networking.k8s.io/v1        true         Gateway
grpcroutes                     gateway.networking.k8s.io/v1        true         GRPCRoute
httproutes                     gateway.networking.k8s.io/v1        true         HTTPRoute
referencegrants   refgrant     gateway.networking.k8s.io/v1beta1   true         ReferenceGrant
```

Install the Gateway Implementation (Envoy Gateway)

Envoy Gateway is delivered from the Nutanix Products catalog (it depends on `cert-manager`, which ships with NKP). Install it by creating an `AppDeployment` — no additional configuration is required:

```
nkp create appdeployment envoy-gateway \
  --app envoy-gateway-1.6.3 \
  --workspace kommander-workspace
```

Confirm the control plane is running:

```
kubectl get pods -n envoy-gateway-system -l control-plane=envoy-gateway
```

```
NAME                            READY   STATUS    RESTARTS   AGE
envoy-gateway-6c4d474877-d75gw   1/1     Running   0          8m
```

Register a `GatewayClass` for the Envoy Gateway controller and confirm it is accepted:

```
kubectl apply -f - <<'EOF'
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: eg
spec:
  controllerName: gateway.envoyproxy.io/gatewayclass-controller
EOF

kubectl get gatewayclass eg
```

```
NAME   CONTROLLER                                      ACCEPTED   AGE
eg     gateway.envoyproxy.io/gatewayclass-controller   True       7m
```

Deploy Two Model Backends and a Gateway

Deploy two mock inference services (`model-v1`, `model-v2`) that identify themselves in their responses, and a `Gateway` that listens for HTTP traffic:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: model-v1
  namespace: gw-inference
spec:
  replicas: 1
  selector: { matchLabels: { app: model-v1 } }
  template:
    metadata: { labels: { app: model-v1 } }
    spec:
      containers:
      - name: echo
        image: hashicorp/http-echo:1.0
        args: ["-text=model-v1", "-listen=:5678"]
---
apiVersion: v1
kind: Service
metadata:
  name: model-v1
  namespace: gw-inference
spec:
```

```
  selector: { app: model-v1 }
  ports: [{ port: 80, targetPort: 5678 }]
### ... model-v2 Deployment + Service are identical, with "-text=model-v2"
---
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: inference-gw
  namespace: gw-inference
spec:
  gatewayClassName: eg
  listeners:
  - name: http
    protocol: HTTP
    port: 80
    allowedRoutes:
      namespaces:
        from: Same
```

The platform load balancer assigns the Gateway an address and it reports `PROGRAMMED=True`:

```
kubectl get gateway inference-gw -n gw-inference
```

```
NAME           CLASS   ADDRESS        PROGRAMMED   AGE
inference-gw   eg      10.22.203.92   True         7m
```

Weighted Traffic Splitting and Header-Based Routing

A single `HTTPRoute` demonstrates both required capabilities — an **80/20 weighted split** across the two model versions for normal traffic, and a **header-based rule** that pins requests carrying `x-model-version: canary` to `model-v2`:

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: inference-route
  namespace: gw-inference
spec:
  parentRefs:
  - name: inference-gw
  hostnames:
  - "models.example.com"
  rules:
  # Header-based routing (OpenAI-style header) -> always model-v2 (canary)
  - matches:
    - path: { type: PathPrefix, value: / }
```

```
      headers:
      - name: x-model-version
        value: canary
    backendRefs:
    - name: model-v2
      port: 80
  # Default -> 80/20 weighted traffic split across v1/v2
  - matches:
    - path: { type: PathPrefix, value: / }
    backendRefs:
    - name: model-v1
      port: 80
      weight: 80
    - name: model-v2
      port: 80
      weight: 20
```

```
kubectl get httproute inference-route -n gw-inference \
  -o jsonpath='Accepted={.status.parents[0].conditions[?(@.type=="Accepted")].status} ResolvedRefs={.status.p
```

```
Accepted=True ResolvedRefs=True
```

Weighted split — 50 requests through the gateway with no special header land ~80% on `model-v1` and ~20% on `model-v2`, matching the configured weights:

```
for i in $(seq 1 50); do
  curl -s -H 'Host: models.example.com' http://<gateway-address>/ ;
done | sort | uniq -c
```

```
  40 model-v1
  10 model-v2
```

Header-based routing — every request carrying the `x-model-version: canary` header is routed to `model-v2`, regardless of the split:

```
for i in $(seq 1 5); do
  curl -s -H 'Host: models.example.com' -H 'x-model-version: canary' http://<gateway-address>/ ;
done | sort | uniq -c
```

```
   5 model-v2
```

> **TIP:** NKP supports the Kubernetes Gateway API and ships a conformant implementation (Envoy Gateway) that provides advanced traffic management for inference services — demonstrated with **weighted traffic splitting** (80/20 across model versions) and **header-based routing** (OpenAI-style protocol header). Envoy Gateway also supports gRPC routing (`GRPCRoute`), TLS, and integration with service meshes such as Istio for the optional mesh-integration capability.

Gang Scheduling

This requirement is a **MUST**: *the platform must allow the installation and successful operation of at least one gang-scheduling solution that provides all-or-nothing scheduling for distributed AI workloads (e.g. Kueue, Volcano).*

NKP ships Kueue in the AI Applications catalog. Kueue is a CNCF gang scheduler that admits a distributed workload **atomically**: it reserves quota for the *entire* job (all of its pods) at once and unsuspends the job only when the whole gang fits. If the gang does not fit, **nothing** is scheduled — no pod of that job is placed, avoiding the partial-placement deadlock where some workers of a distributed job occupy accelerators while waiting forever for the rest.

The following was verified on the NKP 2.18 cluster. Kueue is installed exactly as shown in AI Job and Inference Service Metrics (`nkp create appdeployment kueue --app kueue-0.18.0 --workspace kommander-workspace`).

Verify Kueue Is Running and Its APIs Are Available

```
kubectl get pods -n kueue-system
kubectl api-resources --api-group=kueue.x-k8s.io
```

```
NAME                                                READY   STATUS    RESTARTS   AGE
kueue-system-kueue-controller-manager-786b5b8678-42cn2   1/1     Running   0          43m

NAME             SHORTNAMES        APIVERSION               NAMESPACED   KIND
clusterqueues    cq                kueue.x-k8s.io/v1beta2   false        ClusterQueue
localqueues      queue,queues,lq   kueue.x-k8s.io/v1beta2   true         LocalQueue
resourceflavors  flavor,flavors    kueue.x-k8s.io/v1beta2   false        ResourceFlavor
workloads        kwl               kueue.x-k8s.io/v1beta2   true         Workload
...
```

Define a Quota Pool and Queue

Create a `ClusterQueue` with a bounded quota (4 CPU) and a namespaced `LocalQueue` that feeds it. Jobs opt in by carrying the `kueue.x-k8s.io/queue-name` label.

```
apiVersion: kueue.x-k8s.io/v1beta2
kind: ResourceFlavor
metadata:
  name: default-flavor
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata:
  name: gang-cq
spec:
  namespaceSelector: {}
  resourceGroups:
  - coveredResources: ["cpu", "memory"]
    flavors:
    - name: default-flavor
      resources:
      - name: cpu
        nominalQuota: "4"
      - name: memory
        nominalQuota: 4Gi
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: LocalQueue
metadata:
  name: gang-lq
  namespace: gang-demo
spec:
  clusterQueue: gang-cq
```

Gang that fits → admitted atomically

Submit a 3-pod job (each pod requests 1 CPU, total 3 ≤ 4 quota). Kueue reserves quota for the whole job and unsuspends it, so **all three pods start together**:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: gang-ok
  namespace: gang-demo
  labels:
    kueue.x-k8s.io/queue-name: gang-lq
spec:
  parallelism: 3
  completions: 3
  suspend: true          # Kueue manages suspension; it unsuspends only when the gang is admitted
  template:
    spec:
      restartPolicy: Never
```

```
      containers:
      - name: worker
        image: registry.k8s.io/e2e-test-images/agnhost:2.53
        args: ["pause"]
        resources:
          requests:
            cpu: "1"
            memory: 256Mi
```

```
kubectl get workloads -n gang-demo
kubectl get pods -n gang-demo
kubectl get job gang-ok -n gang-demo -o jsonpath='suspend={.spec.suspend} active={.status.active}{"\n"}'
```

The workload is admitted and all three pods are `Running`:

```
NAME                QUEUE     RESERVED IN   ADMITTED   FINISHED   AGE
job-gang-ok-47da4   gang-lq   gang-cq       True                  13s

NAME            READY   STATUS    RESTARTS   AGE
gang-ok-586dl   1/1     Running   0          12s
gang-ok-8bxmj   1/1     Running   0          12s
gang-ok-kdr4v   1/1     Running   0          12s

suspend=false active=3
```

Oversized gang → refused in full

Submit a 4-pod job where each pod requests 2 CPU (total 8 > 4 quota). Even though two of these pods (2 × 2 = 4 CPU) would individually fit in the remaining quota, Kueue refuses the **entire gang** rather than placing part of it:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: gang-toobig
  namespace: gang-demo
  labels:
    kueue.x-k8s.io/queue-name: gang-lq
spec:
  parallelism: 4
  completions: 4
  suspend: true
  template:
    spec:
      restartPolicy: Never
```

```
      containers:
      - name: worker
        image: registry.k8s.io/e2e-test-images/agnhost:2.53
        args: ["pause"]
        resources:
          requests:
            cpu: "2"
            memory: 256Mi
```

```
kubectl get workloads -n gang-demo
kubectl get pods -n gang-demo
kubectl get job gang-toobig -n gang-demo -o jsonpath='suspend={.spec.suspend} active={.status.active}{"\n"}'
```

The oversized workload is **not** admitted, the job stays suspended, and it contributes **zero** pods — only the earlier `gang-ok` pods exist. There is no partial placement:

```
NAME                    QUEUE     RESERVED IN   ADMITTED   FINISHED   AGE
job-gang-ok-47da4       gang-lq   gang-cq       True                  36s
job-gang-toobig-f7b00   gang-lq                                       13s

NAME            READY   STATUS    RESTARTS   AGE
gang-ok-586dl   1/1     Running   0          35s
gang-ok-8bxmj   1/1     Running   0          35s
gang-ok-kdr4v   1/1     Running   0          35s

suspend=true active=
```

Kueue records exactly why the gang was held back — the whole pod set (8 CPU) exceeds capacity (4 CPU):

```
kubectl describe workload job-gang-toobig-f7b00 -n gang-demo
```

```
Status:
  Conditions:
    Type:     QuotaReserved
    Status:   False
    Reason:   Pending
    Message:  couldn't assign flavors to pod set main: insufficient quota for cpu
              in flavor default-flavor, previously considered podsets requests (0)
              + current podset request (8) > maximum capacity (4)
Events:
  Type     Reason   Age   From             Message
```

```
----     ------   ----  ----              -------
Warning  Pending  24s   kueue-admission   couldn't assign flavors to pod set main: insufficient quota for cp
```

The `ClusterQueue` accounting confirms the state — one workload admitted (using 3 of 4 CPU), one pending, none partially placed:

```
kubectl get clusterqueue gang-cq \
  -o jsonpath='admitted={.status.admittedWorkloads} pending={.status.pendingWorkloads} cpuUsed={.status.flavo
```

```
admitted=1 pending=1 cpuUsed=3
```

The pending gang is admitted automatically — as a whole — as soon as enough quota is freed (for example, when `gang-ok` finishes or the quota is raised).

> **TIP:** NKP installs and successfully runs a gang scheduler (Kueue) that provides **all-or-nothing** scheduling for distributed AI workloads — a fitting gang is admitted atomically, and an oversized gang is refused in full with no partial placement. The same quota-based admission governs GPU (`nvidia.com/gpu`) resources for multi-GPU distributed training and inference jobs.

Cluster Autoscaling

This requirement is a **MUST**: *if the platform provides a cluster autoscaler (or equivalent), it must scale node groups containing specific accelerator types up and down based on pending pods requesting those accelerators.*

NKP is built on **Cluster API (CAPI)** and deploys the upstream Kubernetes **Cluster Autoscaler** (with the `clusterapi` cloud provider) for its managed clusters. Because it scales through Cluster API rather than one cloud's proprietary API, the same mechanism works across every environment NKP targets — public clouds (AWS, Azure, GCP), Nutanix (AHV/Prism Central), vSphere, and pre-provisioned/bare-metal — including hybrid and multi-cloud deployments.

Each NKP node pool is a CAPI `MachineDeployment`, and GPU node pools are separate pools. The Cluster Autoscaler treats each pool as an independent **node group** with configurable min/max bounds, so it can grow and shrink an **accelerator** node pool on its own — adding GPU nodes when pods requesting `nvidia.com/gpu` are `Pending`, and removing GPU nodes when they are idle.

The following was verified on the NKP 2.18 cluster (running on Nutanix infrastructure).

The Cluster Autoscaler Is Running

```
kubectl get deploy -n kommander | grep cluster-autoscaler
```

```
cluster-autoscaler-019f70dd-75e6-76bc-b192-45cb4a040a9d   1/1   1   1   3d
```

It runs the upstream image with the Cluster API provider and node-group auto-discovery for the managed cluster:

```
kubectl get deploy -n kommander -l app.kubernetes.io/name=clusterapi-cluster-autoscaler \
  -o jsonpath='{.items[0].spec.template.spec.containers[0].image}{"\n"}'
### registry.k8s.io/autoscaling/cluster-autoscaler:v1.34.3

### key args
--cloud-provider=clusterapi
--node-group-auto-discovery=clusterapi:clusterName=dlipovetsky,namespace=kommander
--enforce-node-group-min-size=true
```

Node Pools Are Autoscaling Node Groups

Each node pool (`MachineDeployment`) carries the standard Cluster API autoscaler min/max annotations. GPU and CPU pools are distinct groups:

```
kubectl get machinedeployments -n kommander \
  -o custom-columns='NAME:.metadata.name,REPLICAS:.spec.replicas,MIN:.metadata.annotations.cluster\.x-k8s\.io
```

```
NAME                    REPLICAS   MIN   MAX
dlipovetsky-gpu-jz4tk   1          1     1
dlipovetsky-md-0-95rrn  4          4     4
```

Setting a GPU pool's `max` higher than its `min` enables the autoscaler to add GPU nodes on demand.

> **NOTE:** Where you set the autoscaler `min` / `max` bounds depends on the infrastructure. On **Nutanix (AHV/Prism Central)** — the infrastructure used for this verification — clusters are managed by a Cluster API **ClusterClass**, so the topology controller owns the generated `MachineDeployment` and its annotations. Set the bounds on the **`Cluster` topology**; editing the annotations directly on the `MachineDeployment` is reverted on the next reconcile. Update the matching entry under `spec.topology.workers.machineDeployments[...]` and the bounds propagate to the pool:
>
> ```
> kubectl patch cluster dlipovetsky -n kommander --type merge -p '{
>   "spec": {"topology": {"workers": {"machineDeployments": [
>     {"name": "dlipovetsky-gpu", "metadata": {"annotations": {
>       "cluster.x-k8s.io/cluster-api-autoscaler-node-group-min-size": "1",
>       "cluster.x-k8s.io/cluster-api-autoscaler-node-group-max-size": "3"
>     }}}
> ```

```
  ]}}}
}'
```

> On non-ClusterClass infrastructure (for example **AWS**), the `MachineDeployment` is not owned by a topology controller, so set the same `cluster.x-k8s.io/cluster-api-autoscaler-node-group-min-size` / `-max-size` annotations directly on the `MachineDeployment`.

The GPU Node Pool Is a Managed, Accelerator-Specific Node Group

The Cluster Autoscaler discovers the GPU pool as its own node group and knows which node belongs to it — this is what lets it scale accelerator capacity based on pending GPU pods:

```
kubectl logs -n kommander deploy/cluster-autoscaler-019f70dd-75e6-76bc-b192-45cb4a040a9d | grep -iE 'node gro
```

```
discovered node group: MachineDeployment/kommander/dlipovetsky-gpu-jz4tk (min: 1, max: 1, replicas: 1)
discovered node group: MachineDeployment/kommander/dlipovetsky-md-0-95rrn (min: 4, max: 4, replicas: 4)
node "dlipovetsky-gpu-jz4tk-t2dqf-f4mp2" is in nodegroup "MachineDeployment/kommander/dlipovetsky-gpu-jz4tk"
```

That node advertises the accelerator resource the autoscaler keys on:

```
kubectl get node dlipovetsky-gpu-jz4tk-t2dqf-f4mp2 -o jsonpath='{.status.allocatable.nvidia\.com/gpu}{"\n"}'
### 1
```

On this particular cluster the GPU pool is pinned to `min = max = 1`, so it holds steady at one GPU node; raising `max` on the `Cluster` topology (as shown above) is all that is needed for the autoscaler to provision additional GPU nodes (through Cluster API → the infrastructure provider) whenever GPU-requesting pods are `Pending`, and to scale the pool back down to `min` when the GPUs fall idle.

> **TIP:** NKP provides a cluster autoscaler (the upstream Cluster Autoscaler via Cluster API) that scales accelerator-specific node groups up and down based on pending pods requesting those accelerators — uniformly across cloud, hybrid, Nutanix, vSphere, and pre-provisioned infrastructure.

Horizontal Pod Autoscaler

This requirement is a **MUST**: *if the platform supports the HorizontalPodAutoscaler, it must function correctly for pods that use accelerators, including the ability to scale them on custom AI/ML metrics.*

The HorizontalPodAutoscaler (HPA, `autoscaling/v2`) is a core Kubernetes API. NKP ships the **Prometheus Adapter**, which serves **both** the resource metrics API (`metrics.k8s.io`, for CPU/memory) **and** the custom metrics API (`custom.metrics.k8s.io`) — bridging Prometheus series (including the DCGM GPU metrics) to the HPA. This lets HPA scale pods that use accelerators on standard resource metrics *and* on custom AI/ML metrics such as GPU utilization.

The following was verified on the NKP 2.18 cluster, which has a single NVIDIA A40 GPU node.

Verify Both Metrics APIs Are Served

```
kubectl get apiservices | grep -E 'metrics.k8s.io'
```

Both APIs are backed by the Prometheus Adapter and available:

```
v1beta1.custom.metrics.k8s.io   kommander/prometheus-adapter   True   3h
v1beta1.metrics.k8s.io          kommander/prometheus-adapter   True   3h
```

Resource metrics work (so CPU/memory-based HPA works for accelerator pods too):

```
kubectl top pod -n kommander -l app=gpu-inference
```

```
NAME                             CPU(cores)   MEMORY(bytes)
gpu-inference-bfdff5c85-c5gzz   999m         901Mi
```

GPU Utilization Is Available as a Custom Metric

The GPU Operator's DCGM exporter metrics are surfaced through the custom metrics API automatically:

```
kubectl get --raw '/apis/custom.metrics.k8s.io/v1beta1' | jq -r '.resources[].name' | grep DCGM_FI_DEV_GPU_UT
```

```
namespaces/DCGM_FI_DEV_GPU_UTIL
pods/DCGM_FI_DEV_GPU_UTIL
services/DCGM_FI_DEV_GPU_UTIL
```

The DCGM exporter runs in the GPU Operator's namespace (`kommander`) and labels each series with the consuming workload under `exported_pod` / `exported_namespace`, so out of the box the metric is associated with the exporter's `Service` / `Namespace`. Query its current value while a GPU workload is running:

```
kubectl get --raw '/apis/custom.metrics.k8s.io/v1beta1/namespaces/kommander/services/nvidia-dcgm-exporter/DCG
```

```
{"kind":"MetricValueList","apiVersion":"custom.metrics.k8s.io/v1beta1",
 "items":[{"describedObject":{"kind":"Service","namespace":"kommander","name":"nvidia-dcgm-exporter"},
 "metricName":"DCGM_FI_DEV_GPU_UTIL","value":"100"}]}
```

HPA Scales a GPU Workload on the Custom GPU Metric

Deploy a GPU-consuming workload and an HPA that scales it on `DCGM_FI_DEV_GPU_UTIL` (referenced as an `Object` metric on the DCGM exporter Service):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gpu-inference
  namespace: kommander
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gpu-inference
  template:
    metadata:
      labels:
        app: gpu-inference
    spec:
      runtimeClassName: nvidia
      containers:
      - name: nbody
        image: nvcr.io/nvidia/k8s/cuda-sample:nbody-cuda11.7.1
        args: ["nbody", "-benchmark", "-numbodies=4096000", "-numdevices=1"]
        resources:
          limits:
            nvidia.com/gpu: 1
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: gpu-inference-hpa
  namespace: kommander
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: gpu-inference
  minReplicas: 1
```

```
  maxReplicas: 3
  metrics:
  - type: Object
    object:
      describedObject:
        apiVersion: v1
        kind: Service
        name: nvidia-dcgm-exporter
      metric:
        name: DCGM_FI_DEV_GPU_UTIL
      target:
        type: Value
        value: "70"
```

The HPA reads the GPU utilization (`100`) against its target (`70`) and scales the deployment up:

```
kubectl get hpa gpu-inference-hpa -n kommander
```

```
NAME               REFERENCE                   TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
gpu-inference-hpa   Deployment/gpu-inference   100/70    1         3         2          83s
```

`kubectl describe hpa` confirms the scaling decision was driven by the GPU custom metric:

```
Metrics:                                                                    ( current / target )
  "DCGM_FI_DEV_GPU_UTIL" on Service/nvidia-dcgm-exporter (target value):  100 / 70
Conditions:
  Type            Status  Reason              Message
  ----            ------  ------              -------
  AbleToScale     True    ReadyForNewScale    recommended size matches current size
  ScalingActive   True    ValidMetricFound    the HPA was able to successfully calculate a replica count from
  ScalingLimited  False   DesiredWithinRange  the desired count is within the acceptable range
Events:
  Type    Reason             Age   From                       Message
  ----    ------             ----  ----                       -------
  Normal  SuccessfulRescale  47s   horizontal-pod-autoscaler  New size: 2; reason: Service metric DCGM_FI_DEV
```

Because this node has a single physical GPU, the additional replica remains `Pending` until GPU capacity is available (in production, additional GPU nodes — or cluster autoscaling of GPU node pools — provide that capacity).

> **TIP:** HPA **functions correctly for accelerator pods** and **scales them based on a custom AI/ML metric** (GPU utilization), served through NKP's Prometheus Adapter.

Accelerator Performance Metrics

This requirement is a **MUST**: *the platform must allow the installation and operation of at least one accelerator-metrics solution that exposes fine-grained per-accelerator utilization and memory usage — and, where the hardware makes them available, temperature, power draw, and interconnect bandwidth — on a standardized, machine-readable endpoint.*

NKP ships the NVIDIA GPU Operator, which includes the **DCGM Exporter** — a DaemonSet that exposes fine-grained, per-GPU performance metrics in Prometheus exposition format on a standardized `/metrics` endpoint (port `9400`). This covers the required core metrics (per-accelerator **utilization** and **memory usage**) and additionally exposes temperature, power draw, clocks, and PCIe/interconnect counters when the hardware reports them. The platform Prometheus discovers and scrapes this endpoint automatically via a `ServiceMonitor` (verified under Automatic Collection by the Platform Prometheus below).

The following was verified on the NKP 2.18 cluster, which has a single NVIDIA A40 GPU node.

Verify the DCGM Exporter Is Running

```
kubectl get pods -n kommander | grep dcgm
```

```
nvidia-dcgm-exporter-dhbrk   1/1     Running   2   3h57m
nvidia-dcgm-v86n6            1/1     Running   0   3h57m
```

Verify the Metrics Service

```
kubectl get svc -n kommander | grep dcgm
```

```
nvidia-dcgm            ClusterIP   10.96.142.33     <none>   5555/TCP   3h57m
nvidia-dcgm-exporter   ClusterIP   10.104.172.240   <none>   9400/TCP   3h57m
```

Confirm a Standardized, Machine-Readable Endpoint

The exporter serves metrics in Prometheus exposition format, with typed and self-describing metric families:

```
curl -sL http://nvidia-dcgm-exporter.kommander.svc.cluster.local:9400/metrics \
  | grep -E '^# (HELP|TYPE) DCGM_FI_DEV_(GPU_UTIL|FB_USED|GPU_TEMP|POWER_USAGE)'
```

```
### HELP DCGM_FI_DEV_GPU_UTIL GPU utilization (in %).
### TYPE DCGM_FI_DEV_GPU_UTIL gauge
### HELP DCGM_FI_DEV_FB_USED Framebuffer memory used (in MiB).
```

```
### TYPE DCGM_FI_DEV_FB_USED gauge
### HELP DCGM_FI_DEV_GPU_TEMP GPU temperature (in C).
### TYPE DCGM_FI_DEV_GPU_TEMP gauge
### HELP DCGM_FI_DEV_POWER_USAGE Power draw (in W).
### TYPE DCGM_FI_DEV_POWER_USAGE gauge
```

Metrics Reflect Actual GPU Activity

Idle GPU — with no workload scheduled, utilization and framebuffer usage read zero, while the exporter still reports temperature and baseline power:

```
DCGM_FI_DEV_GPU_UTIL{gpu="0",modelName="NVIDIA A40",Hostname="dlipovetsky-gpu-jz4tk-t2dqf-f4mp2",...} 0
DCGM_FI_DEV_FB_USED{gpu="0",modelName="NVIDIA A40",...} 0
DCGM_FI_DEV_GPU_TEMP{gpu="0",modelName="NVIDIA A40",...} 52
DCGM_FI_DEV_POWER_USAGE{gpu="0",modelName="NVIDIA A40",...} 37.629000
```

Deploy a GPU workload to drive the accelerator:

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-load
  namespace: kommander
spec:
  restartPolicy: Never
  runtimeClassName: nvidia
  containers:
  - name: nbody
    image: nvcr.io/nvidia/k8s/cuda-sample:nbody-cuda11.7.1
    args: ["nbody", "-benchmark", "-numbodies=2560000", "-numdevices=1"]
    resources:
      limits:
        nvidia.com/gpu: 1
```

Under load — re-scraping while the workload runs shows utilization, framebuffer usage, temperature, power, and clocks all rise, and each sample is now attributed to the consuming pod (`pod` , `namespace` , `container` labels):

```
DCGM_FI_DEV_GPU_UTIL{gpu="0",UUID="GPU-ff400afb-03b6-3cd9-2fa9-5b8d002ad839",modelName="NVIDIA A40",Hostname=
DCGM_FI_DEV_FB_USED{gpu="0",modelName="NVIDIA A40",container="nbody",namespace="kommander",pod="gpu-load"} 63
DCGM_FI_DEV_GPU_TEMP{gpu="0",modelName="NVIDIA A40",container="nbody",namespace="kommander",pod="gpu-load"} 8
DCGM_FI_DEV_POWER_USAGE{gpu="0",modelName="NVIDIA A40",container="nbody",namespace="kommander",pod="gpu-load"
DCGM_FI_DEV_SM_CLOCK{gpu="0",modelName="NVIDIA A40",container="nbody",namespace="kommander",pod="gpu-load"} 1
DCGM_FI_DEV_MEM_CLOCK{gpu="0",modelName="NVIDIA A40",container="nbody",namespace="kommander",pod="gpu-load"}
```

Beyond the core utilization and memory metrics, the exporter publishes memory-copy utilization (`DCGM_FI_DEV_MEM_COPY_UTIL`), free framebuffer (`DCGM_FI_DEV_FB_FREE`), memory temperature (`DCGM_FI_DEV_MEMORY_TEMP`), and PCIe replay counters (`DCGM_FI_DEV_PCIE_REPLAY_COUNTER`) — 23 metric families in total on the A40.

Automatic Collection by the Platform Prometheus

The GPU Operator ships a `ServiceMonitor` for the DCGM exporter, so the platform Prometheus discovers and scrapes it without any manual configuration:

```
kubectl get servicemonitors -n kommander | grep nvidia-dcgm-exporter
```

```
nvidia-dcgm-exporter   3h
```

The in-cluster Prometheus reports the target as up and has the accelerator metrics stored and queryable:

```
curl -s 'http://kube-prometheus-stack-prometheus.kommander.svc:9090/api/v1/query?query=up{job="nvidia-dcgm-ex
curl -s 'http://kube-prometheus-stack-prometheus.kommander.svc:9090/api/v1/query?query=DCGM_FI_DEV_GPU_UTIL'
```

```
// up{job="nvidia-dcgm-exporter"}
{"metric":{"__name__":"up","job":"nvidia-dcgm-exporter","namespace":"kommander","pod":"nvidia-dcgm-exporter-d

// DCGM_FI_DEV_GPU_UTIL
{"metric":{"__name__":"DCGM_FI_DEV_GPU_UTIL","modelName":"NVIDIA A40","job":"nvidia-dcgm-exporter","namespace
```

> **TIP:** A per-accelerator metrics solution exposing utilization and memory usage — plus temperature, power, clocks, and interconnect counters — on a standardized, machine-readable Prometheus endpoint that the platform monitoring stack discovers and collects automatically.

AI Job and Inference Service Metrics

This requirement is a **MUST**: *the platform must provide a monitoring system capable of discovering and collecting metrics from workloads that expose them in a standard format (e.g. the Prometheus exposition format).*

NKP ships a fully integrated monitoring system based on **kube-prometheus-stack** (chart v82.13.6, Grafana + Prometheus + Alertmanager), **Loki** (v6.55.0) for logs, **Thanos** for long-term storage, and **Prometheus Adapter** (v5.3.0). Prometheus automatically discovers and scrapes any workload that

exposes metrics in the standard Prometheus exposition format via `ServiceMonitor` / `PodMonitor` resources — no platform-specific glue is required.

This is demonstrated end-to-end below with two AI workloads from the AI Applications catalog: **Kueue** (AI/ML job queueing → *job* metrics) and **KServe** (model inference serving → *inference service* metrics). Both were installed and exercised on the NKP 2.18 cluster.

Verify the Monitoring Stack

```
kubectl get helmreleases -n kommander | grep -E 'kube-prometheus-stack|centralized-grafana|prometheus-adapter
```

```
centralized-grafana     True   Helm install succeeded for release kommander/centralized-grafana.v1 with chart
grafana-loki-v3         True   Helm install succeeded for release kommander/grafana-loki-v3.v1 with chart lok
kube-prometheus-stack   True   Helm install succeeded for release kommander/kube-prometheus-stack.v1 with cha
prometheus-adapter      True   Helm install succeeded for release kommander/prometheus-adapter.v1 with chart
thanos                  True   Helm install succeeded for release kommander/thanos.v1 with chart thanos@17.3.
```

AI Job Metrics — Kueue

Kueue is a Kubernetes-native job queueing and quota system for batch and AI/ML workloads. Its controller is natively instrumented and serves Prometheus metrics on a secured `/metrics` endpoint (port `8443`).

Install Kueue from the catalog by creating an `AppDeployment` for the `kueue-0.18.0` app (the app is listed by `kubectl get apps -n kommander` once the AI Applications catalog is added — see the prerequisite note above):

```
nkp create appdeployment kueue \
  --app kueue-0.18.0 \
  --workspace kommander-workspace
```

Expected output:

```
AppDeployment "kommander/kueue" created in namespace "kommander".
```

Confirm the controller is running:

```
kubectl get pods -n kueue-system
```

```
NAME                                                READY   STATUS    RESTARTS   AGE
kueue-system-kueue-controller-manager-786b5b8678-42cn2   1/1     Running   0          78s
```

Next, create a quota pool and a queue, then submit jobs (full manifests in the Kueue catalog README). Submitting three 1-CPU jobs against a 2-CPU `ClusterQueue` quota admits two and queues the third:

```
kubectl get workloads -n default
```

```
NAME                    QUEUE      RESERVED IN   ADMITTED   FINISHED   AGE
job-smoke-job-1-d008b   smoke-lq   smoke-cq      True                  9s
job-smoke-job-2-0a2d0   smoke-lq   smoke-cq      True                  9s
job-smoke-job-3-8ffd4   smoke-lq                                       9s
```

Scrape the Kueue metrics endpoint (the secure endpoint requires a bearer token bound to the `kueue-metrics-reader` ClusterRole):

```
TOKEN=$(kubectl create token kueue-metrics-scraper -n kueue-system)
curl -sk -H "Authorization: Bearer $TOKEN" \
  https://kueue-system-kueue-controller-manager-metrics-service.kueue-system.svc:8443/metrics \
  | grep -E '^kueue_(admitted_workloads_total|pending_workloads|reserving_active_workloads|cluster_queue_stat
```

The metrics reflect the current queue state — two workloads admitted, one pending (inadmissible until quota frees up):

```
kueue_admitted_workloads_total{cluster_queue="smoke-cq",priority_class="",replica_role="leader"} 2
kueue_admitted_active_workloads{cluster_queue="smoke-cq",replica_role="leader"} 2
kueue_reserving_active_workloads{cluster_queue="smoke-cq",replica_role="leader"} 2
kueue_pending_workloads{cluster_queue="smoke-cq",replica_role="leader",status="inadmissible"} 1
kueue_cluster_queue_status{cluster_queue="smoke-cq",replica_role="leader",status="active"} 1
```

The endpoint is standard, typed Prometheus exposition format (24 `kueue_*` metric families in total):

```
### HELP kueue_admitted_workloads_total The total number of admitted workloads per 'cluster_queue'
### TYPE kueue_admitted_workloads_total counter
### HELP kueue_pending_workloads The number of pending workloads, per 'cluster_queue' and 'status'.
### TYPE kueue_pending_workloads gauge
```

Inference Service Metrics — KServe

KServe is a standard model inference platform. Deploying an `InferenceService` (here a scikit-learn iris model in KServe's `RawDeployment` mode) brings up a model server that both serves predictions and exposes per-model Prometheus metrics.

Install KServe from the catalog by creating an `AppDeployment` for the `kserve-0.15.0` app:

```
nkp create appdeployment kserve \
  --app kserve-0.15.0 \
  --workspace kommander-workspace
```

Expected output:

```
AppDeployment "kommander/kserve" created in namespace "kommander".
```

Confirm the controller is running:

```
kubectl get pods -n kserve
```

```
NAME                                        READY   STATUS    RESTARTS   AGE
kserve-controller-manager-7dc64f5f88-jv5sb   2/2     Running   0          3m51s
```

Deploy a sample `InferenceService` (in a dedicated `kserve-test` namespace, `kubectl create namespace kserve-test`):

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: sklearn-iris
  namespace: kserve-test
spec:
  predictor:
    model:
      modelFormat:
        name: sklearn
      storageUri: gs://kfserving-examples/models/sklearn/1.0/model
```

Confirm it becomes ready:

```
kubectl get isvc -n kserve-test
```

```
NAME          URL                                          READY
sklearn-iris   http://sklearn-iris-kserve-test.example.com   True
```

Send an inference request:

```
curl -s http://sklearn-iris-predictor.kserve-test.svc.cluster.local/v1/models/sklearn-iris:predict \
  -H "Content-Type: application/json" \
```

```
  -d '{"instances": [[6.8,2.8,4.8,1.4],[6.0,3.4,4.5,1.6]]}'
```

```
{"predictions":[1,1]}
```

The model server exposes per-model request metrics — latency histograms for the pre-process, predict, and post-process phases — in Prometheus format:

```
curl -s http://sklearn-iris-predictor.kserve-test.svc.cluster.local/metrics \
  | grep -E '^request_(preprocess|predict|postprocess)_seconds_(count|sum)'
```

```
request_preprocess_seconds_count{model_name="sklearn-iris"} 15.0
request_preprocess_seconds_sum{model_name="sklearn-iris"} 0.000102310033980757
request_predict_seconds_count{model_name="sklearn-iris"} 15.0
request_predict_seconds_sum{model_name="sklearn-iris"} 0.003556511946953833
request_postprocess_seconds_count{model_name="sklearn-iris"} 15.0
request_postprocess_seconds_sum{model_name="sklearn-iris"} 7.373595144599676e-05
```

`request_predict_seconds` is a histogram, so latency percentiles are directly derivable in Prometheus/Grafana:

```
### HELP request_predict_seconds predict request latency
### TYPE request_predict_seconds histogram
request_predict_seconds_bucket{le="0.005",model_name="sklearn-iris"} 15.0
request_predict_seconds_bucket{le="0.01",model_name="sklearn-iris"} 15.0
```

KServe annotates each predictor pod with the scrape discovery contract, so the platform Prometheus collects these metrics automatically:

```
kubectl get pod -n kserve-test -l serving.kserve.io/inferenceservice=sklearn-iris \
  -o jsonpath='{.items[0].metadata.annotations}' | tr ',' '\n' | grep prometheus
```

```
prometheus.kserve.io/path: /metrics
prometheus.kserve.io/port: 8080
```

The Platform Prometheus Discovers and Collects the Metrics

Exposing `/metrics` is only half of the requirement; the platform monitoring system must actually **discover and collect** it. That is done through the `ServiceMonitor` / `PodMonitor` contract that Prometheus watches. Attaching a `ServiceMonitor` to the inference service (labeled for the Kommander Prometheus to select) is all that is needed — no other configuration:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: sklearn-iris-predictor
  namespace: kserve-test
  labels:
    prometheus.kommander.d2iq.io/select: "true"
spec:
  selector:
    matchLabels:
      serving.kserve.io/inferenceservice: sklearn-iris
  endpoints:
  - port: sklearn-iris-predictor
    path: /metrics
    interval: 15s
```

The `ServiceMonitor` appears alongside the workload:

```
kubectl get servicemonitors -n kserve-test
```

```
NAME                    AGE
sklearn-iris-predictor   2m
```

Verify the platform Prometheus is actively scraping the inference service — the target reports `up = 1` (querying the in-cluster Prometheus API):

```
curl -s 'http://kube-prometheus-stack-prometheus.kommander.svc:9090/api/v1/query?query=up{job="sklearn-iris-p
```


```
{"status":"success","data":{"resultType":"vector","result":[
  {"metric":{"__name__":"up","job":"sklearn-iris-predictor","namespace":"kserve-test","pod":"sklearn-iris-pre
]}}
```


Confirm the inference-service metric is stored and queryable in Prometheus, so the collection loop is closed end-to-end — the request count observed at the workload's `/metrics` endpoint above is now a first-class series in the platform Prometheus:

```
curl -s 'http://kube-prometheus-stack-prometheus.kommander.svc:9090/api/v1/query?query=request_predict_second
```


```
{"metric":{"__name__":"request_predict_seconds_count","model_name":"sklearn-iris","job":"sklearn-iris-predict
```

The same `ServiceMonitor` / `PodMonitor` mechanism collects any workload exposing Prometheus-format metrics — including the Kueue endpoint above, and the DCGM exporter's out-of-the-box `ServiceMonitor` (see Accelerator Performance Metrics).

> **TIP:** NKP's monitoring system **discovers** workload endpoints via the standard `ServiceMonitor` / `PodMonitor` contract and **collects and stores** their metrics end-to-end — proven here with AI **job** metrics (Kueue) and **inference service** metrics (KServe, `request_predict_seconds_count = 35`) in the standard Prometheus exposition format.

Secure Accelerator Access

This requirement is a **MUST**: *access to accelerators from within containers must be isolated and mediated by the Kubernetes resource-management framework (device plugin or DRA) and the container runtime, preventing unauthorized access or interference between workloads.*

Access to accelerators from within containers is isolated and mediated by the Kubernetes resource management framework (the NVIDIA device plugin, installed by the GPU Operator) and the container runtime. Only pods that explicitly request `nvidia.com/gpu` receive device access, GPU capacity is enforced per device, and pods without a request cannot reach the accelerator or its device files.

The following was verified on the NKP 2.18 cluster, which has a single NVIDIA A40 GPU node.

Authorized → GPU access

Deploy a pod that explicitly requests a GPU under the `nvidia` RuntimeClass:

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-authorized
  namespace: saa-test
spec:
  restartPolicy: Never
  runtimeClassName: nvidia
  containers:
  - name: cuda
    image: nvcr.io/nvidia/k8s/cuda-sample:vectoradd-cuda12.5.0-ubuntu22.04
    command: ["sleep", "3600"]
    resources:
      limits:
        nvidia.com/gpu: 1
```

The container sees exactly its allocated GPU:

```
kubectl exec gpu-authorized -n saa-test -- nvidia-smi -L
```

```
GPU 0: NVIDIA A40 (UUID: GPU-ff400afb-03b6-3cd9-2fa9-5b8d002ad839)
```

Capacity enforced → second pod Pending

The node has a single GPU. A second pod that also requests `nvidia.com/gpu: 1` remains `Pending` — the scheduler does not oversubscribe the device:

```
kubectl get pod gpu-authorized-2 -n saa-test
```

```
NAME               READY   STATUS    RESTARTS   AGE
gpu-authorized-2   0/1     Pending   0          25s
```

```
kubectl describe pod gpu-authorized-2 -n saa-test
```

```
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  25s   default-scheduler  0/8 nodes are available: 3 node(s) had untolerated
           taint(s), 5 Insufficient nvidia.com/gpu. ... preemption: 0/8 nodes are available: ...
```

Unauthorized → no GPU access

Deploy a pod that does not request a GPU:

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-unauthorized
  namespace: saa-test
spec:
  restartPolicy: Never
  containers:
  - name: plain
    image: nvcr.io/nvidia/k8s/cuda-sample:vectoradd-cuda12.5.0-ubuntu22.04
    command: ["sleep", "3600"]
```

The pod has no access to the GPU — `nvidia-smi` is not present in the container, and no NVIDIA device files are mounted:

```
kubectl exec gpu-unauthorized -n saa-test -- nvidia-smi
```

```
OCI runtime exec failed: exec failed: unable to start container process:
exec: "nvidia-smi": executable file not found in $PATH
```

```
kubectl exec gpu-unauthorized -n saa-test -- sh -c 'ls -l /dev/ | grep -i nvidia'
```

```
(no NVIDIA device files present)
```

> **TIP:** GPU access is mediated exclusively by the Kubernetes device plugin and container runtime — only pods that explicitly request `nvidia.com/gpu` receive device access, capacity is enforced per GPU, and pods without a request can neither run `nvidia-smi` nor reach the underlying device files.

Robust CRD and Controller Operation

This requirement is a **MUST**: *the platform must prove that at least one complex AI operator with a CRD (e.g. Ray, Kubeflow) installs and functions reliably — its pods run correctly, its webhooks are operational, and its custom resources reconcile.*

NKP demonstrates robust CRD and controller operation with the **Slurm Operator** (the Slinky project) from the AI Applications catalog — a complex operator that installs its own CRDs, runs mutating and validating admission webhooks, and reconciles custom resources into a fully running Slurm cluster.

Install the Slurm Operator

Once the AI Applications catalog is added (see Prerequisites), the Slurm Operator is available as the `slurm-operator-1.1.1` app. Deploy it by creating an `AppDeployment`:

```
nkp create appdeployment slurm-operator \
  --app slurm-operator-1.1.1 \
  --workspace kommander-workspace
```

Expected output:

```
AppDeployment "kommander/slurm-operator" created in namespace "kommander".
```

Verify the Operator Pods Are Running

The Slurm Operator installs a controller and a dedicated admission-webhook deployment into the `slurm-operator` namespace:

```
kubectl get pods -n slurm-operator
```

Expected output:

```
NAME                                      READY   STATUS    RESTARTS   AGE
slurm-operator-6b8f9c7d4b-2xk9p           1/1     Running   0          2d
slurm-operator-webhook-7c9d5f8b6c-h4m2t   1/1     Running   0          2d
```

Verify the CRDs Are Registered

The operator registers six custom resource definitions in the `slinky.slurm.net` API group:

```
kubectl api-resources --api-group=slinky.slurm.net
```

Expected output:

```
NAME          SHORTNAMES    APIVERSION                 NAMESPACED   KIND
accountings   slurmdbd      slinky.slurm.net/v1beta1   true         Accounting
controllers   slurmctld     slinky.slurm.net/v1beta1   true         Controller
loginsets     lss,sackd     slinky.slurm.net/v1beta1   true         LoginSet
nodesets      nss,slurmd    slinky.slurm.net/v1beta1   true         NodeSet
restapis      slurmrestd    slinky.slurm.net/v1beta1   true         RestApi
tokens        jwt           slinky.slurm.net/v1beta1   true         Token
```

Verify the Admission Webhooks Are Operational

The operator's webhooks are served by the `slurm-operator-webhook` deployment using cert-manager-issued certificates. A validating webhook guards every Slurm CRD, and a mutating webhook handles pod binding for Slurm-scheduled workloads:

```
kubectl get validatingwebhookconfigurations,mutatingwebhookconfigurations | grep slurm-operator
```

Expected output:

```
validatingwebhookconfiguration.admissionregistration.k8s.io/slurm-operator-webhook   6         2d
mutatingwebhookconfiguration.admissionregistration.k8s.io/slurm-operator-webhook     1         2d
```

The webhook serving certificates are provisioned by cert-manager (a hard dependency of the operator):

```
kubectl get certificate,issuer -n slurm-operator
```

Expected output:

```
NAME                                                          READY   SECRET                           AGE
certificate.cert-manager.io/slurm-operator-webhook-root-ca    True    slurm-operator-webhook-root-ca   2d
certificate.cert-manager.io/slurm-operator-webhook-ca         True    slurm-operator-webhook-ca        2d

NAME                                                          READY   AGE
issuer.cert-manager.io/slurm-operator-webhook-self-issuer     True    2d
issuer.cert-manager.io/slurm-operator-webhook-root-issuer     True    2d
```

The validating webhook (`ValidatingWebhookConfiguration/slurm-operator-webhook`) registers one rule per CRD — `accounting-v1beta1.kb.io` , `controller-v1beta1.kb.io` , `loginset-v1beta1.kb.io` , `nodeset-v1beta1.kb.io` , `restapi-v1beta1.kb.io` , and `token-v1beta1.kb.io` — while the mutating webhook (`podsbinding-v1.kb.io`) intercepts `pods/binding` . Submitting an invalid custom resource is intercepted and rejected by the webhook:

```
kubectl apply -f - <<'EOF'
apiVersion: slinky.slurm.net/v1beta1
kind: NodeSet
metadata:
  name: invalid-nodeset
  namespace: default
spec: {}
EOF
```

Expected output (request denied by the `nodeset-v1beta1.kb.io` webhook):

```
Error from server (Forbidden): error when creating "STDIN": admission webhook
"nodeset-v1beta1.kb.io" denied the request: <validation error>
```

The exact message reflects the specific validation rule violated; the key point is that the request is intercepted and rejected by the webhook rather than being persisted.

Verify Custom Resource Reconciliation

This end-to-end smoke test mirrors the "E2E Smoke Test Validation" section of the catalog README, which is the source of truth for the procedure. It creates a temporary Slurm workload (reconciled by the already-deployed operator), verifies scheduling works, then cleans up.

Create a temporary smoke workload; the operator reconciles the custom resources into running workloads:

```
kubectl create namespace slurm-smoke
helm upgrade --install slurm-smoke oci://ghcr.io/slinkyproject/charts/slurm \
```

```
  --version 1.1.1 \
  --namespace slurm-smoke \
  --set controller.persistence.enabled=false \
  --set nodesets.slinky.replicas=1 \
  --wait --timeout 12m
kubectl wait --for=condition=Ready pod -n slurm-smoke --all --timeout=8m
```

Confirm the operator reconciled the custom resources into workloads:

```
kubectl get controllers,nodesets,loginsets -n slurm-smoke
kubectl get pods -n slurm-smoke
```

Expected output:

```
NAME                                     AGE
controller.slinky.slurm.net/slurm-smoke    3m

NAME                                     AGE
nodeset.slinky.slurm.net/slurm-smoke       3m

NAME                              READY   STATUS    RESTARTS   AGE
slurm-smoke-controller-0          1/1     Running   0          3m
slurm-smoke-slinky-0              1/1     Running   0          3m
```

Validate the Slurm scheduling path from the controller pod:

- `sinfo` verifies Slurm sees the partition and worker node.
- `srun` verifies an interactive job can be placed and executed immediately.
- `sbatch` verifies batch submission/queueing/completion metadata.

```
### 1) Cluster view from Slurm (partitions + node state)
kubectl exec -n slurm-smoke slurm-smoke-controller-0 -- sinfo

### 2) Immediate scheduling check (should print worker hostname)
kubectl exec -n slurm-smoke slurm-smoke-controller-0 -- srun -N1 -n1 hostname

### 3) Batch job check with detailed status
kubectl exec -n slurm-smoke slurm-smoke-controller-0 -- sh -lc \
  'JOBID=$(sbatch --parsable --wrap="hostname"); echo JOBID=$JOBID; \
   squeue -j "$JOBID" || true; \
   scontrol show job "$JOBID"; \
   sacct -j "$JOBID" --format=JobID,State,ExitCode -n || true'
```

Expected result:

- `sinfo` shows a schedulable node (for example `slinky-0`).

- `srun` returns a worker hostname.
- `scontrol show job` reports `JobState=COMPLETED` and `ExitCode=0:0`.
- `squeue` may show the job briefly in queue/running states before completion.
- `sacct` may print `Slurm accounting storage is disabled` in minimal smoke setups; this is expected when accounting is not configured.

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
all*         up   infinite      1   idle slinky-0

slinky-0

JobState=COMPLETED Reason=None
ExitCode=0:0
```

Clean up:

```
helm uninstall slurm-smoke -n slurm-smoke
kubectl delete namespace slurm-smoke
kubectl get controllers,nodesets,loginsets,restapis -A
kubectl get pods -n slurm-operator
```

> **TIP:** The platform installs a complex AI operator, its CRDs and admission webhooks are operational, and its custom resources are reconciled into a working cluster.

Additional AI operators available in the NKP catalog include:

- Slurm Operator — Kubernetes operator (Slinky project) for running and managing Slurm clusters, with CRDs (Controller, NodeSet, LoginSet, RestApi, Accounting, Token) and mutating/validating admission webhooks.
- kagent — Declarative AI agent lifecycle management with CRDs (Agent, Tool, ModelConfig, Team) and admission webhooks.
- Milvus Operator — Manages vector database instances.

CNCF AI Conformance



The CNCF Kubernetes AI Conformance defines a set of additional capabilities, APIs, and configurations that a Kubernetes cluster MUST offer, on top of standard CNCF Kubernetes Conformance, to reliably and efficiently run AI/ML workloads.

This page shows how NKP 2.19 meets these requirements using Kubernetes v1.36.2 and the NVIDIA GPU Operator v26.3.3, deployed via Flux CD as platform HelmReleases.

> **NOTE:** NKP is **CNCF Kubernetes AI Conformant** for v1.36; see the v1.36 NKP AI conformance submission. NKP also holds **CNCF Certified Kubernetes** conformance, which AI Conformance builds on. That certification was submitted under NKP's former product name, **Konvoy**; see the v1.36 Konvoy conformance submission.

Requirements Overview

On this cluster, NKP 2.19 satisfies all 8 MUST requirements and 5 of 6 SHOULD requirements (`high_performance_networking` is Partial). Each row below maps a requirement — identified by its upstream ID and grouped by upstream category — to its level and the section of this page that shows the evidence.

This cluster serves DRA v1 and runs the NVIDIA DRA driver; that evidence is in Support Dynamic Resource Allocation (DRA). Driver Runtime Management, GPU Sharing, Virtualized Accelerator (vGPU), High-Performance Networking, and Secure Accelerator Access use DRA where the checklist requires it.

Requirement	Level	Supported	Section
`accelerators`			
`driver_runtime_management`	SHOULD	✅	Driver Runtime Management
`gpu_sharing`	SHOULD	✅	GPU Sharing
`virtualized_accelerator`	SHOULD	✅	Virtualized Accelerator (vGPU)
`networking`			
`ai_inference`	MUST	✅	Support the Gateway API
`advanced_inference_ingress`	SHOULD	✅	Gateway API Inference Extension
`high_performance_networking`	SHOULD	Partial	High-Performance Networking
`disaggregated_inference`	SHOULD	✅	Disaggregated Inference
`schedulingOrchestration`			
`gang_scheduling`	MUST	✅	Gang Scheduling
`cluster_autoscaling`	MUST	✅	Cluster Autoscaling
`pod_autoscaling`	MUST	✅	Horizontal Pod Autoscaler
`observability`			
`accelerator_metrics`	MUST	✅	Accelerator Performance Metrics
`ai_service_metrics`	MUST	✅	AI Job and Inference Service Metrics

Requirement	Level	Supported	Section
`security`			
`secure_accelerator_access`	MUST	☐	Secure Accelerator Access
`operator`			
`robust_controller`	MUST	☐	Robust CRD and Controller Operation

Verification Setup

Environment

The steps below were verified on NKP **2.19.0-dev.34** (Kubernetes v1.36.2) on cluster `rocky-98-ootb-rocky-starter-fad3`. Commands throughout this page target the `kommander-workspace` (namespace `kommander`).

```
kubectl get clusters -A
```

```
NAMESPACE   NAME                              CLUSTERCLASS                 AVAILABLE   CP DESIRED   CP AVAIL
kommander   rocky-98-ootb-rocky-starter-fad3   nkp-nutanix-v2.19.0-dev.34   True        3            3
```

```
kubectl get workspaces -A
```

```
NAME                 DISPLAY NAME                 WORKSPACE NAMESPACE           AGE
default-workspace    Default Workspace            kommander-default-workspace   3h24m
kommander-workspace  Management Cluster Workspace  kommander                     3h24m
```

Prerequisites

Several checks use operators from the NKP AI Applications catalog. On NKP 2.19 that collection is available by default in the workspace; its applications are listed as `App` resources and can be deployed from the NKP catalog:

```
kubectl get apps -n kommander
```

▶ Expected output — catalog applications (`App` resources) in the workspace

SGLang is not part of the 2.19 catalogs used for this page. For Disaggregated Inference it was installed manually (`sglang-0.5.15` / `lmsysorg/sglang:v0.5.15`).

Deploy an application by creating an `AppDeployment` :

```
nkp create appdeployment <name> \
  --app <app-id>-<version> \
  --workspace kommander-workspace
```

The checks on this page deployed `nvidia-gpu-operator`, `nvidia-network-operator`, `kube-prometheus-stack`, `prometheus-adapter`, `centralized-grafana`, `thanos`, `kueue`, `kserve`, `envoy-gateway-nai`, `slurm-operator`, `opentelemetry-operator`, `leaderworkerset`, and `nutanix-ai` (NAI 2.8.0).

Support Dynamic Resource Allocation (DRA)

v1.36 uses Dynamic Resource Allocation as the mechanism for driver verification, GPU sharing, vGPU, and specialized networks once the accelerator supports it. DRA v1 (`resource.k8s.io/v1`) is served on this Kubernetes v1.36.2 cluster, and the NVIDIA DRA driver is installed and publishing the A40.

```
kubectl api-resources --api-group=resource.k8s.io
```

```
NAME                     SHORTNAMES   APIVERSION           NAMESPACED   KIND
deviceclasses                         resource.k8s.io/v1   false        DeviceClass
resourceclaims                        resource.k8s.io/v1   true         ResourceClaim
resourceclaimtemplates                resource.k8s.io/v1   true         ResourceClaimTemplate
resourceslices                        resource.k8s.io/v1   false        ResourceSlice
```

The driver chart is `dra-driver-nvidia-gpu@0.4.1` (HelmRelease `nvidia-dra-driver`, Ready). It creates these DeviceClasses:

```
kubectl get deviceclasses
```

```
NAME
compute-domain-daemon.nvidia.com
compute-domain-default-channel.nvidia.com
gpu.nvidia.com
mig.nvidia.com
vfio.gpu.nvidia.com
```

The `gpu.nvidia.com` ResourceSlice on the GPU node publishes device `gpu-0`:

```
driver: gpu.nvidia.com
node: rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n-bgqqg-25m89
device: gpu-0
uuid: GPU-0384e187-141b-b2ef-e56e-c92a783f6fd8
productName: NVIDIA A40
```

```
architecture: Ampere
driverVersion: 580.126.18
```

Driver pods in `kommander`:

```
kubectl get pods -n kommander | grep dra-driver
```

```
dra-driver-nvidia-gpu-controller-5766489b58-llkm2     1/1   Running
dra-driver-nvidia-gpu-kubelet-plugin-skjzq            2/2   Running
dra-driver-nvidia-gpu-webhook-6bfb76d88d-2sgwt        1/1   Running
```

The device plugin is also enabled, so the same GPU is schedulable as `nvidia.com/gpu`. See Driver Runtime Management and Secure Accelerator Access.

> **TIP:** NKP 2.19 on this cluster serves `resource.k8s.io/v1` and runs the NVIDIA DRA driver. The A40 is published as device `gpu-0` on DeviceClass `gpu.nvidia.com`, with the host driver version on the ResourceSlice.

Driver Runtime Management

This requirement is a **SHOULD**: *provide a verifiable mechanism for ensuring that compatible accelerator drivers and corresponding container runtime configurations are correctly installed and maintained on nodes with accelerators. Once the accelerator supports exposing driver and runtime version information as part of DRA, the platform should use the DRA mechanism for verification.*

NKP ships the NVIDIA GPU Operator to manage the container toolkit, device plugin, DCGM exporter, and runtime configuration on GPU nodes. On this cluster the GPU image already contains the host driver, so the chart default leaves `driver.enabled: false`. The NVIDIA DRA driver publishes the same driver version on the ResourceSlice, which is the DRA verification path the requirement describes.

Verify Container Runtime Version

```
kubectl get nodes -l nvidia.com/gpu.present=true \
  -o jsonpath='{.items[*].status.nodeInfo.containerRuntimeVersion}'
```

```
containerd://2.3.1-d2iq.1
```

Verify Accelerator Driver and Runtime Configuration

```
kubectl get nodes -l nvidia.com/gpu.present=true -o json
```

Key labels on `rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n-bgqqg-25m89`:

```
nvidia.com/cuda.driver-version.full=580.126.18
nvidia.com/cuda.runtime-version.full=13.0
nvidia.com/gpu.count=1
nvidia.com/gpu.deploy.container-toolkit=true
nvidia.com/gpu.deploy.dcgm=true
nvidia.com/gpu.deploy.dcgm-exporter=true
nvidia.com/gpu.deploy.device-plugin=true
nvidia.com/gpu.deploy.driver=true
nvidia.com/gpu.deploy.gpu-feature-discovery=true
nvidia.com/gpu.family=ampere
nvidia.com/gpu.machine=AHV
nvidia.com/gpu.memory=46068
nvidia.com/gpu.mode=compute
nvidia.com/gpu.present=true
nvidia.com/gpu.product=NVIDIA-A40
nvidia.com/gpu.replicas=1
nvidia.com/gpu.sharing-strategy=none
nvidia.com/mig.capable=false
nvidia.com/vgpu.present=false
```

`gpu.deploy.container-toolkit=true` confirms the container runtime configuration is injected. `cuda.driver-version.full=580.126.18` is the host driver. The same version is on the DRA device attribute `driverVersion` in Support Dynamic Resource Allocation (DRA).

The chart default also sets `devicePlugin.enabled: false`, so `nvidia.com/gpu` is not advertised until that value is overridden. An optional values ConfigMap was applied and the HelmRelease was reconciled so the device-plugin checks on this page could run next to DRA:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nvidia-gpu-operator-config-deviceplugin-defaults
  namespace: kommander
data:
  operator-deviceplugin-enabled-values.yaml: |
    devicePlugin:
      enabled: true
```

After that reconcile the GPU node advertised `nvidia.com/gpu` capacity `1` and allocatable `1`.

Verify GPU Operator Pods

```
kubectl get pods -n kommander | grep nvidia
```

```
nvidia-container-toolkit-daemonset-kx44n     1/1   Running
nvidia-cuda-validator-cq66s                  0/1   Completed
nvidia-dcgm-tgdrx                            1/1   Running
nvidia-dcgm-exporter-5999b                   1/1   Running
nvidia-device-plugin-daemonset-pv6wn         2/2   Running
nvidia-network-operator-7c777996d-wnwpl      1/1   Running
nvidia-operator-validator-ffjlx              1/1   Running
gpu-operator-69b86d44d8-4kfs6                1/1   Running
```

Verify HelmReleases

```
kubectl get helmreleases -n kommander | grep nvidia
```

```
nvidia-dra-driver         True   Helm install succeeded for release kommander/nvidia-dra-driver.v1 with chart
nvidia-gpu-operator       True   Helm install succeeded for release kommander/nvidia-gpu-operator.v1 with cha
nvidia-network-operator   True   Helm install succeeded for release kommander/nvidia-network-operator.v1 with
```

> **TIP:** The GPU node reports driver 580.126.18, CUDA runtime 13.0, and containerd 2.3.1-d2iq.1. The NVIDIA DRA driver publishes that same driver version on `gpu.nvidia.com`.

GPU Sharing

This requirement is a **SHOULD**: *for accelerators that support static GPU sharing, provide a well-defined mechanism for at least one sharing strategy so workloads that do not need a full dedicated GPU can improve utilization. If software-based sharing (for example time-slicing) is supported, oversubscription of GPUs should be allowed.*

NKP satisfies this through the NVIDIA GPU Operator using time-slicing. One physical GPU is advertised as *N* schedulable `nvidia.com/gpu` replicas. The forward-looking DRA clause in v1.36 is covered by the installed NVIDIA DRA driver; this demonstration uses the device-plugin time-slicing path, which is what schedules `nvidia.com/gpu` today.

Enable Time-Slicing on the GPU Operator

```
kubectl apply -f - <<'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: time-slicing-config
  namespace: kommander
data:
```

```
  any: |-
    version: v1
    sharing:
      timeSlicing:
        resources:
        - name: nvidia.com/gpu
          replicas: 4
EOF

kubectl patch clusterpolicy cluster-policy --type merge \
  -p '{"spec":{"devicePlugin":{"config":{"name":"time-slicing-config","default":"any"}}}}'
```

Verify the Node Advertises Oversubscribed GPU Slices

```
kubectl get node rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n-bgqqg-25m89 -o json
```

```
nvidia.com/gpu.count=1
nvidia.com/gpu.product=NVIDIA-A40-SHARED
nvidia.com/gpu.replicas=4
nvidia.com/gpu.sharing-strategy=time-slicing
capacity=4 allocatable=4
```

The node has one physical GPU (`gpu.count=1`) and advertises it as four schedulable slices.

Deploy Multiple Pods That Each Request a Whole GPU

```
kubectl create namespace gpu-sharing-demo
kubectl apply -f - <<'EOF'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: time-slicing-verify
  namespace: gpu-sharing-demo
spec:
  replicas: 4
  selector:
    matchLabels:
      app: time-slicing-verify
  template:
    metadata:
      labels:
        app: time-slicing-verify
    spec:
      runtimeClassName: nvidia
      tolerations:
      - key: nvidia.com/gpu
```

```
        operator: Exists
        effect: NoSchedule
      containers:
      - name: cuda
        image: nvcr.io/nvidia/cuda:12.5.0-base-ubuntu22.04
        command: ["/bin/sh","-c","nvidia-smi -L; sleep 3600"]
        resources:
          limits:
            nvidia.com/gpu: 1
EOF
```

All four pods reached `Running` on the single GPU node:

```
kubectl -n gpu-sharing-demo get pods -o wide
```

```
NAME                                   READY   STATUS    NODE
time-slicing-verify-745956486-l5mh4   1/1     Running   rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n-bgqqg-25m8
time-slicing-verify-745956486-ppxwz   1/1     Running   rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n-bgqqg-25m8
time-slicing-verify-745956486-v9hd8   1/1     Running   rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n-bgqqg-25m8
time-slicing-verify-745956486-x6p9t   1/1     Running   rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n-bgqqg-25m8
```

Allocated resources on the node showed `nvidia.com/gpu 4` requested against `4` allocatable.

Prove the Pods Share One Physical GPU

```
for p in $(kubectl -n gpu-sharing-demo get pods -o jsonpath='{.items[*].metadata.name}'); do
  echo "--- $p ---"; kubectl -n gpu-sharing-demo logs $p | grep -i UUID
done
```

```
--- time-slicing-verify-745956486-l5mh4 ---
GPU 0: NVIDIA A40 (UUID: GPU-0384e187-141b-b2ef-e56e-c92a783f6fd8)
--- time-slicing-verify-745956486-ppxwz ---
GPU 0: NVIDIA A40 (UUID: GPU-0384e187-141b-b2ef-e56e-c92a783f6fd8)
--- time-slicing-verify-745956486-v9hd8 ---
GPU 0: NVIDIA A40 (UUID: GPU-0384e187-141b-b2ef-e56e-c92a783f6fd8)
--- time-slicing-verify-745956486-x6p9t ---
GPU 0: NVIDIA A40 (UUID: GPU-0384e187-141b-b2ef-e56e-c92a783f6fd8)
```

That UUID is the same device the DRA ResourceSlice publishes as `gpu-0`.

Secure access and the HPA check ran while this node still advertised one `nvidia.com/gpu`. Time-slicing was enabled after those checks and left in place. The node then reported `nvidia.com/gpu.product=NVIDIA-A40-SHARED`, `sharing-strategy=time-slicing`, capacity `4`, allocatable `4`.

> **TIP:** Four pods, each granted a full `nvidia.com/gpu`, ran on one physical A40 and reported one UUID.

Virtualized Accelerator (vGPU)

NVIDIA vGPU lets a hypervisor partition a physical GPU into multiple virtual GPUs that are assigned to separate VMs. It requires vGPU-capable hardware, the NVIDIA vGPU host driver on the hypervisor, and the NVIDIA License System. This requirement is a **SHOULD**: *for accelerators that support virtualized accelerator technologies (for example vGPU), provide well-defined mechanisms for these to be exposed and managed, maintaining consistency with physical fractional GPUs.*

NKP can provision vGPU-enabled node pools, in addition to full GPU passthrough, on Nutanix AHV and VMware vSphere. Virtual GPUs use the same GPU Operator path and the same `nvidia.com/gpu` resource name as physical GPUs. v1.36 says that once the accelerator supports vGPU through DRA, the platform should use that mechanism. The NVIDIA DRA driver is installed on this cluster; the published device is the passthrough A40, not a vGPU profile.

Consistent Management Across Physical and Virtual GPUs

```
kubectl get nodes -l nvidia.com/gpu.present=true -o json
```

```
nvidia.com/gpu.machine=AHV
nvidia.com/gpu.product=NVIDIA-A40
nvidia.com/gpu.mode=compute
nvidia.com/gpu.count=1
nvidia.com/gpu.sharing-strategy=none
nvidia.com/vgpu.present=false
```

```
kubectl get node rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n-bgqqg-25m89 \
  -o jsonpath='{.status.allocatable.nvidia\.com/gpu}{"\n"}'
```

```
1
```

On this node the A40 is attached in passthrough mode (`gpu.mode=compute`, `vgpu.present=false`) and advertises one full `nvidia.com/gpu`. On a vGPU node pool, the same operator stack reports `nvidia.com/vgpu.present=true` and advertises the assigned vGPU profile as `nvidia.com/gpu`. A workload's `resources.limits: nvidia.com/gpu: 1` stays the same either way.

> **NOTE:** The NVIDIA DRA driver is installed. DeviceClasses include `gpu.nvidia.com`, `mig.nvidia.com`, and `vfio.gpu.nvidia.com`. The ResourceSlice for this node describes the

> passthrough A40 (`productName: NVIDIA A40` , `architecture: Ampere`). A vGPU profile was not attached, so `nvidia.com/vgpu.present` stays `false` . As the NVIDIA DRA driver adds vGPU support, NKP can expose vGPU through the same DRA path, per the requirement's forward-looking guidance.

Resources

- vGPU on Nutanix (TN-2046)
- NVIDIA GPU Operator — Using NVIDIA vGPU

> **TIP:** NKP provides a well-defined mechanism (vGPU-enabled node pools on AHV/vSphere with the NVIDIA GPU Operator) that exposes and manages virtualized accelerators consistently with physical GPUs — the same `nvidia.com/gpu` device-plugin resource, scheduling, metrics, and node-pool autoscaling. This verification node is passthrough; the DRA driver is present for the forward-looking vGPU path.

Support the Gateway API

This requirement is a **MUST**: *support the Kubernetes Gateway API with an implementation for advanced traffic management for inference services, which enables weighted traffic splitting, header-based routing (for OpenAI protocol headers), and optional integration with service meshes.*

NKP ships the Kubernetes Gateway API CRDs. Envoy Gateway for NKP 2.19 is catalog app `envoy-gateway-nai` 1.8.1 (chart `gateway-helm@v1.8.1`). Traefik remains the in-box GatewayClass.

The following was verified on the NKP 2.19 cluster. The weighted and header request counts are in Weighted Traffic Splitting and Header-Based Routing.

Verify Gateway API Resources

```
kubectl api-resources --api-group=gateway.networking.k8s.io
```

```
NAME                 SHORTNAMES   APIVERSION                          NAMESPACED   KIND
backendtlspolicies   btlspolicy   gateway.networking.k8s.io/v1        true         BackendTLSPolicy
gatewayclasses       gc           gateway.networking.k8s.io/v1        false        GatewayClass
gateways             gtw          gateway.networking.k8s.io/v1        true         Gateway
grpcroutes                        gateway.networking.k8s.io/v1        true         GRPCRoute
httproutes                        gateway.networking.k8s.io/v1        true         HTTPRoute
listenersets         lset         gateway.networking.k8s.io/v1        true         ListenerSet
referencegrants      refgrant     gateway.networking.k8s.io/v1        true         ReferenceGrant
tcproutes                         gateway.networking.k8s.io/v1alpha2  true         TCPRoute
```

```
tlsroutes                       gateway.networking.k8s.io/v1         true         TLSRoute
udproutes                       gateway.networking.k8s.io/v1alpha2   true         UDPRoute
```

The in-box GatewayClass is accepted:

```
NAME       CONTROLLER                     ACCEPTED
traefik    traefik.io/gateway-controller  True
```

Install the Gateway Implementation (Envoy Gateway)

Envoy Gateway NAI depends on `cert-manager`, which ships with NKP. Install it from the product catalog:

```
nkp create appdeployment envoy-gateway-nai \
  --app envoy-gateway-nai-1.8.1 \
  --workspace kommander-workspace
```

```
envoy-gateway-nai   True   Helm install succeeded for release envoy-gateway-system/eg.v1 with chart gateway-h
```

```
kubectl get pods -n envoy-gateway-system
```

The control-plane pod reached `1/1` after the InferencePool CRD and RBAC steps below. Nutanix AI 2.8.0 was installed later; with Valkey available, `envoy-ratelimit` can reach `nai-valkey-sentinel.nai-system.svc.cluster.local`. See Gateway API Inference Extension.

This Envoy Gateway build watches `inference.networking.k8s.io/v1` InferencePool. That CRD is not created by the `envoy-gateway-nai` AppDeployment alone, and the `envoy-gateway` ServiceAccount cannot list it until granted. Without both, the controller exits while its informers sync (`Could not wait for Cache to sync`). For this check, the upstream InferencePool CRD from Gateway API Inference Extension v1.5.0 was applied, and the ServiceAccount was granted get/list/watch on `inferencepools` . NAI operators also ship InferencePool CRDs; the live object used for GAIE evidence is described in the GAIE section.

Register a GatewayClass and confirm it is accepted:

```
kubectl apply -f - <<'EOF'
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: eg
spec:
  controllerName: gateway.envoyproxy.io/gatewayclass-controller
```

```
EOF

kubectl get gatewayclass eg
```

After the CRD and RBAC change:

```
NAME   CONTROLLER                                      ACCEPTED
eg     gateway.envoyproxy.io/gatewayclass-controller   True
```

Deploy Two Model Backends and a Gateway

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: model-v1
  namespace: gw-inference
spec:
  replicas: 1
  selector: { matchLabels: { app: model-v1 } }
  template:
    metadata: { labels: { app: model-v1 } }
    spec:
      containers:
      - name: echo
        image: hashicorp/http-echo:1.0
        args: ["-text=model-v1", "-listen=:8080"]
---
apiVersion: v1
kind: Service
metadata:
  name: model-v1
  namespace: gw-inference
spec:
  selector: { app: model-v1 }
  ports: [{ port: 80, targetPort: 8080 }]
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: model-v2
  namespace: gw-inference
spec:
  replicas: 1
  selector: { matchLabels: { app: model-v2 } }
  template:
    metadata: { labels: { app: model-v2 } }
    spec:
```

```
      containers:
      - name: echo
        image: hashicorp/http-echo:1.0
        args: ["-text=model-v2", "-listen=:8080"]
---
apiVersion: v1
kind: Service
metadata:
  name: model-v2
  namespace: gw-inference
spec:
  selector: { app: model-v2 }
  ports: [{ port: 80, targetPort: 8080 }]
---
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: inference-gw
  namespace: gw-inference
spec:
  gatewayClassName: eg
  listeners:
  - name: http
    protocol: HTTP
    port: 80
    allowedRoutes:
      namespaces:
        from: Same
```

`model-v1` and `model-v2` rolled out. The `envoy-gateway-nai` chart points `extensionManager` at `ai-gateway-controller.nai-system.svc.cluster.local:1063`. Nutanix AI is not installed here, so that name does not resolve. With the hook in place, Envoy opened an xDS connection but never received a listener, port 80 stayed closed, and `Programmed` was `False` with reason `NoResources`. The data-plane startup probe on port `19003` was refused.

For this Gateway API (weighted / header) check, `extensionManager` was removed from the EnvoyGateway config and HelmRelease `envoy-gateway-nai` was suspended so Flux did not restore the hook. That let the controller publish listeners without Nutanix AI. The Gateway then became ready:

```
NAME           CLASS   ADDRESS         PROGRAMMED
inference-gw   eg      10.23.134.247   True
```

```
Accepted      True    Accepted       The Gateway has been scheduled by Envoy Gateway
Programmed    True    Programmed     Address assigned to the Gateway, 1/1 envoy replicas available
listener http
  Programmed    True    Programmed     Sending translated listener configuration to the data plane
  Accepted      True    Accepted       Listener has been successfully translated
```

```
  ResolvedRefs  True    ResolvedRefs   Listener references have been resolved
  Attached Routes: 1
```

Weighted Traffic Splitting and Header-Based Routing

The HTTPRoute used for this check requests both capabilities. An 80/20 weighted split is the default rule. A header rule pins `x-model-version: canary` to `model-v2`:

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: inference-route
  namespace: gw-inference
spec:
  parentRefs:
  - name: inference-gw
  hostnames:
  - "models.example.com"
  rules:
  - matches:
    - path: { type: PathPrefix, value: / }
      headers:
      - name: x-model-version
        value: canary
    backendRefs:
    - name: model-v2
      port: 80
  - matches:
    - path: { type: PathPrefix, value: / }
    backendRefs:
    - name: model-v1
      port: 80
      weight: 80
    - name: model-v2
      port: 80
      weight: 20
```

From a pod in `gw-inference`, 50 requests to `http://10.23.134.247/` with `Host: models.example.com` returned `model-v1` 40 times and `model-v2` 10 times. Ten requests with header `x-model-version: canary` all returned `model-v2`.

> **TIP:** GatewayClass `eg` is Accepted. Gateway `inference-gw` was Programmed at `10.23.134.247` for this check. The HTTPRoute split 50 requests 40/10, and the canary header sent all 10 requests

> to `model-v2`. After Nutanix AI was installed, MetalLB assigned `10.23.134.247` to `nai-ingress-gateway` instead.

Gateway API Inference Extension

This requirement is a **SHOULD**: *support an implementation of the Gateway API Inference Extension (GAIE), which can route requests to models hosted on Kubernetes. The implementation supports serving LLMs and making advanced routing decisions (for example KV-cache-aware routing) based on metrics and capabilities advertised by the model serving platform.*

Nutanix AI 2.8.0 (`nutanix-ai` Helm release revision 2 in `nai-system`) installs the AI Gateway controller and ships `docker.io/nutanix/nai-epp-inference-scheduler` as the endpoint picker. Gateway `nai-ingress-gateway` (`nai-gatewayclass`) is Programmed at `10.23.134.247`. For this GAIE check, `extensionManager` was restored on Envoy Gateway (`backendResources` for `inference.networking.k8s.io/InferencePool` pointing at `ai-gateway-controller.nai-system.svc.cluster.local:1063`), after it had been removed for the earlier weighted-routing Gateway check.

InferencePool + EPP

Two model pods advertised vLLM-style metrics and served HTTP responses identifying themselves. `model-low-kv` reported `vllm:gpu_cache_usage_perc 0.05` and `vllm:num_requests_waiting 0`. `model-high-kv` reported `0.99` and `50`. EPP `llm-epp` used `queue-scorer`, `kv-cache-utilization-scorer`, and `max-score-picker`.

```
kubectl get inferencepool,httproute,deploy -n gw-inference
kubectl get pods -n gw-inference -l 'pool=llm-models'
kubectl get pods -n gw-inference -l app=llm-epp
```

```
NAME                                                AGE
inferencepool.inference.networking.k8s.io/llm-models    5m39s

NAME                                                HOSTNAMES                  AGE
httproute.gateway.networking.k8s.io/gaie-llm-route     ["gaie-llm.example.com"]   5m38s

NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/llm-epp           1/1     1            1           5m38s
deployment.apps/model-high-kv     1/1     1            1           5m39s
deployment.apps/model-low-kv      1/1     1            1           5m39s

NAME                               READY   STATUS    RESTARTS   AGE
model-high-kv-7bcdb7f874-2mprv     1/1     Running   0          72s
model-low-kv-6964d678bb-zxrp8      1/1     Running   0          72s
llm-epp-55bbb88fbd-7btvn           1/1     Running   0          4m49s
```

`HTTPRoute/gaie-llm-route` parented to `nai-ingress-gateway` is Accepted and ResolvedRefs with backendRef `InferencePool/llm-models`.

End-to-end routing and KV/queue-aware pick

From an in-cluster client, 80 `POST` requests to `http://10.23.134.247/v1/chat/completions` with `Host: gaie-llm.example.com` and an OpenAI-shaped JSON body all returned `model-low-kv`. None returned `model-high-kv`.

```
  80 model-low-kv
```

That matches the advertised metrics: the EPP preferred the endpoint with low KV-cache utilization and an empty queue over the saturated endpoint.

> **TIP:** GAIE is exercised end to end: Gateway → HTTPRoute → InferencePool → NAI EPP → model pods. Advanced routing used queue and KV-cache utilization scorers; 80/80 live requests went to the low-utilization endpoint.

High-Performance Networking

This requirement is a **SHOULD**: *if high-performance pod-to-pod communication is needed, provide well-defined mechanisms for these specialized network resources to be managed and exposed so their characteristics are discoverable and pods can attach to multiple network interfaces. Platforms should use DRA (for example DRANET).*

What this cluster has:

```
kubectl api-resources --api-group=mellanox.com
kubectl get pods -n kommander | grep nvidia-network-operator
kubectl -n kube-system get ds | grep multus
```

```
hostdevicenetworks     mellanox.com/v1alpha1   HostDeviceNetwork
ipoibnetworks          mellanox.com/v1alpha1   IPoIBNetwork
macvlannetworks        mellanox.com/v1alpha1   MacvlanNetwork
nicclusterpolicies     mellanox.com/v1alpha1   NicClusterPolicy

nvidia-network-operator-7c777996d-wnwpl   1/1   Running
multus                                    10   10   10   10   10
```

HelmRelease `nvidia-network-operator` is Ready with chart `network-operator@26.1.0`.

What this cluster does not have: no `NicClusterPolicy` objects, no `NetworkAttachmentDefinition` objects, no RDMA device, and no DRANET `ResourceSlice`. Pods were not attached to a second

interface.

> **NOTE:** The NVIDIA Network Operator CRDs and Multus are installed. A live high-performance fabric and DRA-managed NICs were not demonstrated.

Disaggregated Inference

This requirement is a **SHOULD**: *support the deployment and operation of disaggregated inference architectures. To be conformant, the platform should demonstrate that it can install and run a disaggregated inference solution (for example vLLM, SGLang, llm-d, or Dynamo) with separate instances for distinct phases such as prefill and decode.*

SGLang is not in the 2.19 catalogs used for this page. It was installed manually for this check: separate prefill and decode Deployments (`lmsysorg/sglang:v0.5.15`) with `--disaggregation-mode=prefill` and `--disaggregation-mode=decode` , transfer backend `nixl` (NIXL/UCX), and one `nvidia.com/gpu` each on distinct GPUs. An SGLang PD router (`--pd-disaggregation`) was also run against those workers.

Verify Prefill and Decode Are Separate Ready Instances

```
kubectl get pods -n sglang -o wide
kubectl get deploy -n sglang sglang-sglang-prefill sglang-sglang-decode \
  -o jsonpath='{range .items[*]}{.metadata.name}{" "}{.spec.template.spec.containers[0].args}{"\n"}{end}'
```

```
NAME                                   READY   STATUS    NODE
sglang-router-854859db47-lv8n5         1/1     Running   rocky-98-ootb-rocky-starter-fad3-md-0-...
sglang-sglang-decode-658646bfd-pbt6f   1/1     Running   rocky-98-ootb-rocky-starter-fad3-gpu-...-qc4wk
sglang-sglang-prefill-66759cdcc6-fs7n9 1/1     Running   rocky-98-ootb-rocky-starter-fad3-gpu-...-lg5sr

sglang-sglang-prefill [... "--disaggregation-mode=prefill","--disaggregation-transfer-backend=nixl"]
sglang-sglang-decode  [... "--disaggregation-mode=decode","--disaggregation-transfer-backend=nixl"]
```

Prefill and decode each completed PD disaggregation warmup and initialized NIXL with the UCX backend (`NIXL KVManager initialized with backend: UCX` , then `The server is fired up and ready to roll!`).

> **TIP:** Manually installed SGLang was run as separate prefill and decode instances on distinct GPUs, with NIXL/UCX transfer configured and a PD router present.

Gang Scheduling

This requirement is a **MUST**: *the platform must allow for the installation and successful operation of at least one gang scheduling solution that ensures all-or-nothing scheduling for distributed AI workloads (for example Kueue, Volcano).*

NKP ships Kueue in the AI Applications catalog. Kueue admits a distributed workload atomically: it reserves quota for the entire job and unsuspends the job only when the whole gang fits. If the gang does not fit, nothing from that job is scheduled.

The following was verified on the NKP 2.19 cluster. Kueue is installed as shown in AI Job and Inference Service Metrics (`nkp create appdeployment kueue --app kueue-0.18.0 --workspace kommander-workspace`).

Verify Kueue Is Running and Its APIs Are Available

```
kubectl get pods -n kueue-system
kubectl api-resources --api-group=kueue.x-k8s.io
```

```
NAME                                                READY   STATUS
kueue-system-kueue-controller-manager-5db757656c-w5pp9   1/1     Running

NAME                         SHORTNAMES                          APIVERSION               KIND
admissionchecks              ac                                  kueue.x-k8s.io/v1beta2   AdmissionCheck
clusterqueues                cq                                  kueue.x-k8s.io/v1beta2   ClusterQueue
cohorts                      co                                  kueue.x-k8s.io/v1beta2   Cohort
localqueues                  queue,queues,lq                     kueue.x-k8s.io/v1beta2   LocalQueue
multikueueclusters           mkc                                 kueue.x-k8s.io/v1beta2   MultiKueueCluster
multikueueconfigs            mkconf                              kueue.x-k8s.io/v1beta2   MultiKueueConfig
provisioningrequestconfigs   prc                                 kueue.x-k8s.io/v1beta2   ProvisioningRequestC
resourceflavors              flavor,flavors,rf                   kueue.x-k8s.io/v1beta2   ResourceFlavor
topologies                   topo                                kueue.x-k8s.io/v1beta2   Topology
workloadpriorityclasses      wpc                                 kueue.x-k8s.io/v1beta2   WorkloadPriorityClas
workloads                    kwl,kueueworkload,kueueworkloads    kueue.x-k8s.io/v1beta2   Workload
```

Define a Quota Pool and Queue

```
apiVersion: kueue.x-k8s.io/v1beta2
kind: ResourceFlavor
metadata:
  name: default-flavor
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata:
  name: gang-cq
spec:
```

```
  namespaceSelector: {}
  resourceGroups:
  - coveredResources: ["cpu", "memory"]
    flavors:
    - name: default-flavor
      resources:
      - name: cpu
        nominalQuota: "4"
      - name: memory
        nominalQuota: 4Gi
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: LocalQueue
metadata:
  name: gang-lq
  namespace: gang-demo
spec:
  clusterQueue: gang-cq
```

Gang that fits → admitted atomically

```
apiVersion: batch/v1
kind: Job
metadata:
  name: gang-ok
  namespace: gang-demo
  labels:
    kueue.x-k8s.io/queue-name: gang-lq
spec:
  parallelism: 3
  completions: 3
  suspend: true
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: worker
        image: registry.k8s.io/e2e-test-images/agnhost:2.53
        args: ["pause"]
        resources:
          requests:
            cpu: "1"
            memory: 256Mi
```

```
kubectl get workloads -n gang-demo
kubectl get pods -n gang-demo
```

```
kubectl get job gang-ok -n gang-demo -o jsonpath='suspend={.spec.suspend} active={.status.active}{"\n"}'
```

```
NAME                QUEUE     RESERVED IN   ADMITTED   FINISHED
job-gang-ok-53cca   gang-lq   gang-cq       True

NAME            READY   STATUS
gang-ok-8d9zw   1/1     Running
gang-ok-n7dxs   1/1     Running
gang-ok-zvj6w   1/1     Running

suspend=false active=3
```

Oversized gang → refused in full

```
apiVersion: batch/v1
kind: Job
metadata:
  name: gang-toobig
  namespace: gang-demo
  labels:
    kueue.x-k8s.io/queue-name: gang-lq
spec:
  parallelism: 4
  completions: 4
  suspend: true
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: worker
        image: registry.k8s.io/e2e-test-images/agnhost:2.53
        args: ["pause"]
        resources:
          requests:
            cpu: "2"
            memory: 256Mi
```

```
kubectl get workloads -n gang-demo
kubectl get pods -n gang-demo
kubectl get job gang-toobig -n gang-demo -o jsonpath='suspend={.spec.suspend} active={.status.active}{"\n"}'
```

```
NAME                    QUEUE     RESERVED IN   ADMITTED
job-gang-ok-53cca       gang-lq   gang-cq       True
job-gang-toobig-d10eb   gang-lq
```

```
NAME            READY   STATUS
gang-ok-8d9zw   1/1     Running
gang-ok-n7dxs   1/1     Running
gang-ok-zvj6w   1/1     Running

suspend=true active=
```

```
kubectl describe workload job-gang-toobig-d10eb -n gang-demo
```

```
Type:     QuotaReserved
Status:   False
Reason:   Pending
Message:  couldn't assign flavors to pod set main: insufficient quota for cpu in flavor default-flavor, previ
```

```
kubectl get clusterqueue gang-cq -o yaml
```

```
admittedWorkloads: 1
pendingWorkloads: 1
flavorsUsage:
- name: default-flavor
  resources:
  - name: cpu
    total: "3"
  - name: memory
    total: 768Mi
```

> **TIP:** Kueue admitted the 3-pod gang and refused the 8-CPU gang with no pods from that job placed.

Cluster Autoscaling

This requirement is a **MUST**: *if the platform provides a cluster autoscaler or an equivalent mechanism, it must be able to scale up and down node groups containing specific accelerator types based on pending pods requesting those accelerators.*

NKP is built on Cluster API and deploys the upstream Kubernetes Cluster Autoscaler with the `clusterapi` cloud provider. Each NKP node pool is a `MachineDeployment`. The autoscaler treats each pool as its own node group.

The following was verified on the NKP 2.19 cluster. The GPU pool is pinned to one replica, so this capture shows discovery and bounds, not a scale-up event.

The Cluster Autoscaler Is Running

```
kubectl get deploy -n kommander | grep cluster-autoscaler
```

```
cluster-autoscaler-01a0ea04-b191-7836-9b7a-5cc70cba54d9
```

The deployment is `1/1`. At `2026-09-29T01:07:45Z` its logs showed both node groups discovered and the GPU node placed in the GPU group.

Node Pools Are Autoscaling Node Groups

The discovery log lists both MachineDeployments:

```
MachineDeployment/kommander/rocky-98-ootb-rocky-starter-fad3-md-0-d8jj5
min: 6
max: 6
replicas: 6

MachineDeployment/kommander/rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n
min: 1
max: 1
replicas: 1
```

On Nutanix, clusters are managed by a Cluster API ClusterClass (`nkp-nutanix-v2.19.0-dev.34`). The topology controller owns the generated `MachineDeployment`. Set min and max on the `Cluster` topology so the next reconcile does not revert a direct edit of the MachineDeployment annotations. The Rocky worker pool was scaled to 6 replicas, so its min and max are both 6. The GPU pool was left at min = max = 1.

The GPU Node Pool Is a Managed, Accelerator-Specific Node Group

```
kubectl logs -n kommander deploy/cluster-autoscaler-01a0ea04-b191-7836-9b7a-5cc70cba54d9 | grep -iE 'node gro
```

```
discovered node group: MachineDeployment/kommander/rocky-98-ootb-rocky-starter-fad3-md-0-d8jj5 (min: 6, max:
discovered node group: MachineDeployment/kommander/rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n (min: 1, max: 1
node "rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n-bgqqg-25m89" is in nodegroup "MachineDeployment/kommander/ro
Skipping node group MachineDeployment/kommander/rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n - max size reached
```

That GPU node advertises the accelerator resource:

```
kubectl get node rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n-bgqqg-25m89 \
  -o jsonpath='{.status.allocatable.nvidia\.com/gpu}{"\n"}'
```

```
1
```

That allocatable value is from before time-slicing. GPU Sharing then left the node at capacity 4. The GPU group is pinned at min = max = 1, so this capture shows discovery and bounds. A second pod requesting `nvidia.com/gpu` stayed Pending, and the autoscaler logged `max size reached` for the GPU node group. Raising `max` above the current replica count on the Cluster topology is what allows the autoscaler to add GPU nodes.

> **TIP:** Cluster Autoscaler discovers the GPU MachineDeployment as its own node group, with its own min and max, separate from the Rocky worker pool.

Horizontal Pod Autoscaler

This requirement is a **MUST**: *if the platform supports the HorizontalPodAutoscaler, it must function correctly for pods utilizing accelerators. This includes the ability to scale these Pods based on custom metrics relevant to AI/ML workloads.*

The HorizontalPodAutoscaler (`autoscaling/v2`) is a core Kubernetes API. NKP ships the Prometheus Adapter, which serves both `metrics.k8s.io` and `custom.metrics.k8s.io` , including the DCGM GPU series.

The following was verified on the NKP 2.19 cluster, which has a single NVIDIA A40.

Verify Both Metrics APIs Are Served

```
kubectl get apiservices | grep -E 'metrics.k8s.io'
```

```
v1beta1.custom.metrics.k8s.io   kommander/prometheus-adapter   True
v1beta1.metrics.k8s.io          kommander/prometheus-adapter   True
```

GPU Utilization Is Available as a Custom Metric

```
kubectl get --raw '/apis/custom.metrics.k8s.io/v1beta1' | jq -r '.resources[].name' | grep DCGM_FI_DEV_GPU_UT
```

```
jobs.batch/DCGM_FI_DEV_GPU_UTIL
namespaces/DCGM_FI_DEV_GPU_UTIL
```

```
pods/DCGM_FI_DEV_GPU_UTIL
services/DCGM_FI_DEV_GPU_UTIL
```

While the GPU was idle:

```
kubectl get --raw '/apis/custom.metrics.k8s.io/v1beta1/namespaces/kommander/services/nvidia-dcgm-exporter/DCG
```


```
{"kind":"MetricValueList","apiVersion":"custom.metrics.k8s.io/v1beta1",
 "items":[{"describedObject":{"kind":"Service","namespace":"kommander","name":"nvidia-dcgm-exporter"},
 "metricName":"DCGM_FI_DEV_GPU_UTIL","timestamp":"2026-09-29T01:07:58Z","value":"0"}]}
```


Under the nbody workload in the next section the same metric was `100`.

HPA Scales a GPU Workload on the Custom GPU Metric

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gpu-inference
  namespace: kommander
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gpu-inference
  template:
    metadata:
      labels:
        app: gpu-inference
    spec:
      runtimeClassName: nvidia
      tolerations:
      - key: nvidia.com/gpu
        operator: Exists
        effect: NoSchedule
      containers:
      - name: nbody
        image: nvcr.io/nvidia/k8s/cuda-sample:nbody-cuda11.7.1
        args: ["nbody", "-benchmark", "-numbodies=4096000", "-numdevices=1"]
        resources:
          limits:
            nvidia.com/gpu: 1
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
```

```
  name: gpu-inference-hpa
  namespace: kommander
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: gpu-inference
  minReplicas: 1
  maxReplicas: 3
  metrics:
  - type: Object
    object:
      describedObject:
        apiVersion: v1
        kind: Service
        name: nvidia-dcgm-exporter
      metric:
        name: DCGM_FI_DEV_GPU_UTIL
      target:
        type: Value
        value: "70"
```

```
kubectl get hpa gpu-inference-hpa -n kommander
```

```
NAME                REFERENCE                   TARGETS   MINPODS   MAXPODS   REPLICAS
gpu-inference-hpa   Deployment/gpu-inference   100/70    1         3         2
```

```
Metrics:                                                                    ( current / target )
  "DCGM_FI_DEV_GPU_UTIL" on Service/nvidia-dcgm-exporter (target value):  100 / 70
Conditions:
  Type            Status  Reason              Message
  ----            ------  ------              -------
  AbleToScale     True    ReadyForNewScale    recommended size matches current size
  ScalingActive   True    ValidMetricFound    the HPA was able to successfully calculate a replica count from
  ScalingLimited  False   DesiredWithinRange  the desired count is within the acceptable range
Events:
  Type    Reason             From                       Message
  ----    ------             ----                       -------
  Normal  SuccessfulRescale  horizontal-pod-autoscaler  New size: 2; reason: Service metric DCGM_FI_DEV_GPU_U
```

```
NAME                             READY   STATUS
gpu-inference-545b9d7d58-sd82t   1/1     Running
gpu-inference-545b9d7d58-mf4s6   0/1     Pending
```

The second replica stayed Pending because the node has a single `nvidia.com/gpu`.

> **TIP:** HPA scaled an accelerator Deployment using `DCGM_FI_DEV_GPU_UTIL` from the Prometheus Adapter. The extra replica could not start on this single-GPU node.

Accelerator Performance Metrics

This requirement is a **MUST**: *for supported accelerator types, the platform must allow installation and successful operation of at least one accelerator metrics solution that exposes fine-grained performance metrics via a standardized, machine-readable metrics endpoint. This must include per-accelerator utilization and memory usage. Temperature, power draw, and interconnect bandwidth should be exposed when the hardware provides them.*

The NVIDIA GPU Operator includes the DCGM Exporter. It serves Prometheus text on port `9400`. The exporter image on this cluster is `nvcr.io/nvidia/k8s/dcgm-exporter:4.5.3-4.8.2-distroless`.

Verify the DCGM Exporter Is Running

```
kubectl get pods -n kommander | grep dcgm
```

```
nvidia-dcgm-tgdrx            1/1   Running
nvidia-dcgm-exporter-5999b   1/1   Running
```

Verify the Metrics Service

```
kubectl get svc -n kommander | grep dcgm
```

```
nvidia-dcgm            ClusterIP   10.105.151.99   <none>   5555/TCP
nvidia-dcgm-exporter   ClusterIP   10.106.13.156   <none>   9400/TCP
```

Confirm a Standardized, Machine-Readable Endpoint

```
curl -sL http://nvidia-dcgm-exporter.kommander.svc.cluster.local:9400/metrics \
  | grep -E '^# (HELP|TYPE) DCGM_FI_DEV_(GPU_UTIL|FB_USED|GPU_TEMP|POWER_USAGE)'
```

```
### HELP DCGM_FI_DEV_GPU_TEMP GPU temperature (in C).
### TYPE DCGM_FI_DEV_GPU_TEMP gauge
### HELP DCGM_FI_DEV_POWER_USAGE Power draw (in W).
### TYPE DCGM_FI_DEV_POWER_USAGE gauge
### HELP DCGM_FI_DEV_GPU_UTIL GPU utilization (in %).
```

```
### TYPE DCGM_FI_DEV_GPU_UTIL gauge
### HELP DCGM_FI_DEV_FB_USED Framebuffer memory used (in MiB).
### TYPE DCGM_FI_DEV_FB_USED gauge
```

Metrics Reflect Actual GPU Activity

Idle GPU:

```
DCGM_FI_DEV_GPU_TEMP{gpu="0",UUID="GPU-0384e187-141b-b2ef-e56e-c92a783f6fd8",modelName="NVIDIA A40",Hostname=
DCGM_FI_DEV_POWER_USAGE{gpu="0",UUID="GPU-0384e187-141b-b2ef-e56e-c92a783f6fd8",modelName="NVIDIA A40",...} 3
DCGM_FI_DEV_GPU_UTIL{gpu="0",UUID="GPU-0384e187-141b-b2ef-e56e-c92a783f6fd8",modelName="NVIDIA A40",...} 0
DCGM_FI_DEV_FB_USED{gpu="0",UUID="GPU-0384e187-141b-b2ef-e56e-c92a783f6fd8",modelName="NVIDIA A40",...} 0
```

The load was the `gpu-inference` Deployment from Horizontal Pod Autoscaler (`nbody` , `numbodies=4096000`), not a separate `gpu-load` pod.

Under load:

```
DCGM_FI_DEV_GPU_TEMP{gpu="0",UUID="GPU-0384e187-141b-b2ef-e56e-c92a783f6fd8",modelName="NVIDIA A40",DCGM_FI_D
DCGM_FI_DEV_POWER_USAGE{...,container="nbody",namespace="kommander",pod="gpu-inference-545b9d7d58-sd82t"} 292
DCGM_FI_DEV_GPU_UTIL{...,container="nbody",namespace="kommander",pod="gpu-inference-545b9d7d58-sd82t"} 100
DCGM_FI_DEV_FB_USED{...,container="nbody",namespace="kommander",pod="gpu-inference-545b9d7d58-sd82t"} 842
```

Automatic Collection by the Platform Prometheus

Prometheus Adapter exposes the scraped DCGM series on the custom metrics API. That is the collection path HPA used. While idle, `services/nvidia-dcgm-exporter/DCGM_FI_DEV_GPU_UTIL` was `0` . While nbody ran, the HPA read `100/70` from that same Service metric. A separate Prometheus `query` API response and a `ServiceMonitor` listing were not captured in this run.

> **TIP:** DCGM exporter publishes utilization, framebuffer, temperature, and power for the A40 in Prometheus text format. The series move under load, and Prometheus Adapter serves `DCGM_FI_DEV_GPU_UTIL` to HPA.

AI Job and Inference Service Metrics

This requirement is a **MUST**: *provide a monitoring system capable of discovering and collecting metrics from workloads that expose them in a standard format (for example Prometheus exposition format).*

NKP ships kube-prometheus-stack, Grafana, Thanos, and Prometheus Adapter. Prometheus discovers workloads that expose the Prometheus text format. This section shows Kueue job metrics scraped from

the controller, and the KServe install. The sklearn sample did not become Ready, so it is not used as inference-metrics evidence.

Verify the Monitoring Stack

```
kubectl get helmreleases -n kommander | grep -E 'kube-prometheus-stack|centralized-grafana|prometheus-adapter
```

```
centralized-grafana     True     Helm install succeeded for release kommander/centralized-grafana.v1 with cha
kube-prometheus-stack   True     Helm install succeeded for release kommander/kube-prometheus-stack.v1 with c
prometheus-adapter      True     Helm install succeeded for release kommander/prometheus-adapter.v1 with char
thanos                  True     Helm install succeeded for release kommander/thanos.v1 with chart thanos@17.
```

The ClusterApp name for Thanos is `thanos-17.3.2`. The installed chart is `thanos@17.3.1`. `grafana-loki-v3` was not installed.

AI Job Metrics — Kueue

Kueue serves Prometheus metrics on port `8443`.

```
nkp create appdeployment kueue \
  --app kueue-0.18.0 \
  --workspace kommander-workspace
```

```
kueue   True   Helm install succeeded for release kueue-system/kueue-system-kueue.v1 with chart kueue@0.18.0
```

```
kubectl get pods -n kueue-system
```

```
NAME                                                  READY   STATUS
kueue-system-kueue-controller-manager-5db757656c-w5pp9   1/1     Running
```

The gang jobs in Gang Scheduling are the workload behind these series. The controller ServiceAccount cannot scrape its own metrics endpoint. A ServiceAccount bound to ClusterRole `kueue-system-kueue-metrics-reader` can:

```
kubectl create serviceaccount kueue-metrics-scraper -n kueue-system
kubectl create clusterrolebinding kueue-metrics-scraper \
  --clusterrole=kueue-system-kueue-metrics-reader \
  --serviceaccount=kueue-system:kueue-metrics-scraper
TOKEN=$(kubectl create token kueue-metrics-scraper -n kueue-system)
curl -sk -H "Authorization: Bearer $TOKEN" \
```

```
  https://kueue-system-kueue-controller-manager-metrics-service.kueue-system.svc:8443/metrics \
  | grep -E '^kueue_(admitted_workloads_total|pending_workloads|reserving_active_workloads|admitted_active_wo
```

```
kueue_admitted_workloads_total{cluster_queue="gang-cq",priority_class="",replica_role="leader"} 1
kueue_admitted_active_workloads{cluster_queue="gang-cq",replica_role="leader"} 1
kueue_reserving_active_workloads{cluster_queue="gang-cq",replica_role="leader"} 1
kueue_pending_workloads{cluster_queue="gang-cq",replica_role="leader",status="inadmissible"} 1
kueue_cluster_queue_status{cluster_queue="gang-cq",replica_role="leader",status="active"} 1
```

```
### HELP kueue_admitted_workloads_total The total number of admitted workloads per 'cluster_queue'
### HELP kueue_pending_workloads The number of pending workloads, per 'cluster_queue' and 'status'.
```

One admitted workload and one inadmissible pending workload match `gang-ok` and `gang-toobig`.

Inference Service Metrics — KServe

KServe 0.19.0 is the catalog app for this release.

```
nkp create appdeployment kserve \
  --app kserve-0.19.0 \
  --workspace kommander-workspace
```

```
kserve                      True   Helm install succeeded ... chart kserve-resources@v0.19.0
kserve-crd                  True   Helm install succeeded ... chart kserve-crd@v0.19.0
kserve-llmisvc-crd          True   Helm install succeeded ... chart kserve-llmisvc-crd@v0.19.0
kserve-llmisvc-resources    True   Helm install succeeded ... chart kserve-llmisvc-resources@v0.19.0
```

```
kubectl get pods -n kserve
```

```
NAME                                        READY   STATUS
kserve-controller-manager-598584c48f-n7bv6   2/2     Running
```

Served inference APIs:

```
inferenceservices            serving.kserve.io/v1beta1   InferenceService
llminferenceservices         serving.kserve.io/v1alpha2  LLMInferenceService
llminferenceserviceconfigs   serving.kserve.io/v1alpha2  LLMInferenceServiceConfig
inferencegraphs              serving.kserve.io/v1alpha1  InferenceGraph
```

`kubectl get clusterservingruntimes,servingruntimes -A` returned no runtimes. A sample InferenceService was created anyway:

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: sklearn-iris
  namespace: kserve-test
spec:
  predictor:
    model:
      modelFormat:
        name: sklearn
      storageUri: gs://kfserving-examples/models/sklearn/1.0/model
```

```
kubectl get inferenceservice -n kserve-test
kubectl get events -n kserve-test
```

```
NAME           READY
sklearn-iris   Unknown

Warning  InternalError  inferenceservice/sklearn-iris  no runtime found to support predictor with model type:
```

No predictor pod was created. There is no `/metrics` endpoint and no prediction response from this sample.

The Platform Prometheus Discovers and Collects the Metrics

The Kueue scrape above is a Prometheus text endpoint collected with the platform's metrics reader role. Prometheus Adapter collecting `DCGM_FI_DEV_GPU_UTIL` is the other closed loop on this cluster, shown under Accelerator Performance Metrics and Horizontal Pod Autoscaler. The KServe sample never served metrics, so this page does not show a predictor `ServiceMonitor` or a `request_predict_seconds` series.

> **TIP:** Prometheus-format metrics from Kueue match the live queue: one admitted workload and one inadmissible pending workload. KServe 0.19.0 is installed. The sklearn InferenceService had no serving runtime, so it is not metrics evidence.

Secure Accelerator Access

This requirement is a **MUST**: *ensure that access to accelerators from within containers is properly isolated and mediated by the Kubernetes resource management framework (device plugin or DRA) and the container runtime, preventing unauthorized access or interference between workloads.*

Both mediation paths are installed. The NVIDIA device plugin advertises `nvidia.com/gpu`. The NVIDIA DRA driver publishes `gpu.nvidia.com` (see Support Dynamic Resource Allocation (DRA)). The check below uses the device plugin. The node has one GPU.

The following was verified on the NKP 2.19 cluster.

Authorized → GPU access

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-authorized
  namespace: saa-test
spec:
  restartPolicy: Never
  runtimeClassName: nvidia
  containers:
  - name: cuda
    image: nvcr.io/nvidia/k8s/cuda-sample:vectoradd-cuda12.5.0-ubuntu22.04
    command: ["sleep", "3600"]
    resources:
      limits:
        nvidia.com/gpu: 1
```

```
kubectl exec gpu-authorized -n saa-test -- nvidia-smi -L
```

```
GPU 0: NVIDIA A40 (UUID: GPU-0384e187-141b-b2ef-e56e-c92a783f6fd8)
```

The pod ran on `rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n-bgqqg-25m89`.

Capacity enforced → second pod Pending

```
kubectl get pod gpu-second -n saa-test
```

```
NAME              READY   STATUS    RESTARTS
gpu-second        0/1     Pending   0
```

```
kubectl describe pod gpu-second -n saa-test
```

```
Events:
  Type     Reason            From               Message
  ----     ------            ----               -------
```

```
  Warning  FailedScheduling   default-scheduler   0/10 nodes are available: 1 Insufficient nvidia.com/gpu, 3
  Normal   NotTriggerScaleUp  cluster-autoscaler  pod didn't trigger scale-up: 2 max node group size reached
```

Unauthorized → no GPU access

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-unauthorized
  namespace: saa-test
spec:
  restartPolicy: Never
  containers:
  - name: cuda
    image: nvcr.io/nvidia/cuda:12.5.0-base-ubuntu22.04
    command: ["sh","-c","nvidia-smi -L; ls -l /dev/nvidia* || true"]
```

The pod ran on the GPU node, `rocky-98-ootb-rocky-starter-fad3-gpu-6nz8n-bgqqg-25m89`. It did not request `nvidia.com/gpu`.

```
kubectl logs gpu-unauthorized -n saa-test
```

```
sh: 1: nvidia-smi: not found
ls: cannot access '/dev/nvidia*': No such file or directory
```

> **TIP:** A pod that requests `nvidia.com/gpu` sees the A40. A second request stays Pending. A pod that does not request a GPU has no NVIDIA device files. DRA publishes the same GPU as `gpu.nvidia.com` for claim-based allocation.

Robust CRD and Controller Operation

This requirement is a **MUST**: *the platform must prove that at least one complex AI operator with a CRD (for example Ray or Kubeflow) can be installed and functions reliably. This includes verifying that the operator's pods run correctly, its webhooks are operational, and its custom resources can be reconciled.*

NKP demonstrates this with the Slurm Operator (the Slinky project) from the AI Applications catalog.

Install the Slurm Operator

```
nkp create appdeployment slurm-operator \
  --app slurm-operator-1.1.1 \
  --workspace kommander-workspace
```

```
slurm-operator              True   Helm install succeeded ... chart slurm-operator@1.1.1
slurm-operator-1.1.1-crds   True   Helm install succeeded ... chart slurm-operator-crds@1.1.1
```

Verify the Operator Pods Are Running

```
kubectl get pods -n slurm-operator
```

```
NAME                                  READY   STATUS
slurm-operator-7897b56bf9-2f6qz       1/1     Running
slurm-operator-webhook-665c95c89d-n7jzb   1/1     Running
```

Verify the CRDs Are Registered

```
kubectl api-resources --api-group=slinky.slurm.net
```

```
NAME          SHORTNAMES            APIVERSION                 NAMESPACED   KIND
accountings   slurmdbd              slinky.slurm.net/v1beta1   true         Accounting
controllers   slurmctld             slinky.slurm.net/v1beta1   true         Controller
loginsets     loginsets,lss,sackd   slinky.slurm.net/v1beta1   true         LoginSet
nodesets      nodesets,nss,slurmd   slinky.slurm.net/v1beta1   true         NodeSet
restapis      slurmrestd            slinky.slurm.net/v1beta1   true         RestApi
tokens        tokens,jwt            slinky.slurm.net/v1beta1   true         Token
```

Verify the Admission Webhooks Are Operational

```
kubectl get validatingwebhookconfigurations,mutatingwebhookconfigurations | grep slurm-operator
```

```
validatingwebhookconfiguration.admissionregistration.k8s.io/slurm-operator-webhook   6
mutatingwebhookconfiguration.admissionregistration.k8s.io/slurm-operator-webhook     1
```

```
kubectl get certificate,issuer -n slurm-operator
```

```
NAME                                                    READY   SECRET
certificate.cert-manager.io/slurm-operator-webhook-ca        True    slurm-operator-webhook-ca
certificate.cert-manager.io/slurm-operator-webhook-root-ca   True    slurm-operator-webhook-root-ca
```

```
NAME                                                    READY
issuer.cert-manager.io/slurm-operator-webhook-root-issuer   True
issuer.cert-manager.io/slurm-operator-webhook-self-issuer   True
```

A NodeSet submitted without `spec.controllerRef` was rejected by the API server:

```
kubectl apply -f - <<'EOF'
apiVersion: slinky.slurm.net/v1beta1
kind: NodeSet
metadata:
  name: invalid-nodeset
  namespace: slurm-operator
spec:
  replicas: 1
EOF
```

```
The NodeSet "invalid-nodeset" is invalid: spec.controllerRef: Required value
```

The message is the CRD's required-field check. The object was not stored. The webhook configurations and their cert-manager certificates are Ready, and the smoke test below shows the operator reconciling valid custom resources.

Verify Custom Resource Reconciliation

```
kubectl create namespace slurm-smoke
helm upgrade --install slurm-smoke oci://ghcr.io/slinkyproject/charts/slurm \
  --version 1.1.1 \
  --namespace slurm-smoke \
  --set controller.persistence.enabled=false \
  --set nodesets.slinky.replicas=1 \
  --wait --timeout 12m
```

```
kubectl get controllers,nodesets -n slurm-smoke
kubectl get pods -n slurm-smoke
```

```
NAME                                    AGE
controller.slinky.slurm.net/slurm-smoke   ...

NAME
nodeset.slinky.slurm.net/slurm-smoke-worker-slinky

NAME                                  READY   STATUS
```

```
slurm-smoke-controller-0               3/3     Running
slurm-smoke-worker-slinky-0            2/2     Running
slurm-smoke-restapi-5f64b8cfcf-t5m2n   1/1     Running
```

```
kubectl exec -n slurm-smoke slurm-smoke-controller-0 -c slurmctld -- sinfo
kubectl exec -n slurm-smoke slurm-smoke-controller-0 -c slurmctld -- srun -N1 -n1 hostname
```

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
all*         up   infinite      1   idle slinky-0

slinky-0
```

```
kubectl exec -n slurm-smoke slurm-smoke-controller-0 -c slurmctld -- \
  scontrol show job 2
```

```
JobState=COMPLETED Reason=None
ExitCode=0:0
```

`sbatch /tmp/job.sh` was job 2 (`JobState=COMPLETED`, `ExitCode=0:0`, `NodeList=slinky-0`). `sacct` printed `Slurm accounting storage is disabled`, which matches this chart install (`accounting.enabled` is false).

> **TIP:** The Slurm operator's pods, CRDs, and webhook configuration are in place. A Controller and NodeSet reconciled into a Slurm cluster that ran `srun` and completed a batch job.

templates