Developer

Use this section for developer-oriented ACP guidance, including application building, operations, and the consolidated host pages for application-management workflows.

TOC

Start with the consolidated path

For most application delivery and day-2 operations, start with Building Applications. This path is the long-term host for application creation, workload, namespace, configuration, operations, and observability guidance.

Use Overview when you need the high-level lifecycle map, and continue to the task-specific pages under Building Applications.

Overview

Overview

Quick Start

Creating a simple application via image

Building Applications

Images

Registry Image Use

Source to Image

Node Isolation Strategy

FAQ

FAQ

Overview

Alauda Container Platform provides a unified interface to create, edit, delete, and manage cloud-native applications through both a web console and CLI (Command-Line Interface). Applications can be deployed across multiple namespaces with RBAC policies.

TOC

Namespace Management

Namespaces provide logical isolation for Kubernetes resources. Key operations include:

- Creating Namespaces: Define resource quotas and pod security admission policies.
- Importing Namespaces: Importing existing Kubernetes namespaces into Alauda Container Platform provides full platform capabilities parity with natively created namespaces.

Application-management workflows

Use the consolidated pages under Building Applications for ongoing application lifecycle work.

Most migrated web console workflows still follow the same scope model:

- Application dashboards and summary views can aggregate data across the clusters associated with the current project.
- Concrete create, update, inspect, or troubleshooting actions still run in a selected cluster and namespace context.
- The same resource type can appear in both the **Applications** view and the **Clusters** view, but the entry path changes the task framing and operational scope.

Depending on the current project, enabled integrations, and your permissions, the console can show different tabs, actions, warning banners, or monitoring signals for the same task family.

Application Lifecycle Management

Alauda Container Platform supports end-to-end lifecycle management including:

Application Creation Patterns

In Alauda Container Platform, applications can be created in multiple ways. Here are some common methods:

- Create from Images: Create custom applications using pre-built container images. This method supports creating complete application that include `Deployments`, `Services`, `ConfigMaps`, and other Kubernetes resources.
- Create from Catalog: Alauda Container Platform provides application catalogs, allowing users to select predefined application templates (Helm Charts or Operator Backed) for creation.
- Create from YAML: By importing a YAML file, create a custom application with all included resources in one step.
- Create from Code: Build images via Source to Image (S2I).

These pages are also the main landing points for migrated application-creation workflows from the newer console experience.

Application Operations

- Updating Applications: Update an application's image version, environment variables, and other configurations, or import existing Kubernetes resources for centralized management.
- Exporting Applications: Export applications in YAML, Kustomize, or Helm Chart formats, then import them to create new application instances in other namespaces or clusters.
- Version Management: Support automatically or manually creating application versions, and in case of issues, one-click rollback to a specific version is available for quick recovery.
- Deleting Applications: Delete an application, it simultaneously deletes the application itself and all of its directly contained Kubernetes resources. Additionally, this action severs any association the application might have had with other Kubernetes resources that were not directly part of its definition.

Application Observability

For continuous operation management, the platform provides logs, events, monitoring, etc.

- Logs: Supports viewing real-time logs from the currently running Pod, and also provides logs from previous container restarts.
- Events: Supports viewing event information for all resources within a namespace.
- Monitoring Dashboards: Provides monitoring dashboards for Applications, Workloads, and Pods. The **Applications Overview** dashboard is project-scoped, while workload and Pod dashboards remain namespace-scoped. You can also customize dashboards to suit specific operational requirements.

Kubernetes Workload Management

Support for core workload types:

- Deployments: Manage stateless applications with rolling updates.
- StatefulSets: Run stateful apps with stable network IDs.

- DaemonSets: Deploy node-level services (e.g., log collectors).
- CronJobs: Schedule batch jobs with retry policies.

The workload pages under `building_application` are also the primary landing points for migrated workload workflows from the newer console experience.

Quick Start

Creating a simple application via image

> This technical guide demonstrates how to efficiently create, manage, and access containerized applications in Alauda Container Platform using Kubernetes-native methodologies.

TOC

Introduction

Use Cases

- New users seeking to understand fundamental application creation workflows on Kubernetes platforms
- Practical exercise demonstrating core platform capabilities including:
  - Project/Namespace orchestration
  - Deployment creation
  - Service exposure patterns
  - Application accessibility verification

Time Commitment

Estimated completion time: 10-15 minutes

Important Notes

- This technical guide focuses on essential parameters - refer to comprehensive documentation for advanced configurations
- Required permissions:
  - Project/Namespace creation
  - Image repository integration
  - Workload deployment

Prerequisites

- Basic understanding of Kubernetes architecture and Alauda Container Platform platform concepts
- Pre-configured project following platform establishment procedures

Workflow Overview

No.	Operation	Description
1	Create Namespace	Establish resource isolation boundary
2	Configure Image Repository	Set up container image sources
3	Create application via Deployment	Create Deployment workload
4	Expose Service via NodePort	Configure NodePort service
5	Validate Application Accessibility	Test endpoint connectivity

Procedure

Create namespace

> Namespaces provide logical isolation for resource grouping and quota management.

Prerequisites

- Permissions to create, update, and delete namespaces(e.g., Administrator or Project Administrator roles)
- kubectl configured with cluster access

Creation Process

1. Log in, and navigate to **Project Management** > **Namespaces**
2. Select **Create Namespace**
3. Configure essential parameters:

** Parameter **	Description
Cluster	Target cluster from project-associated clusters

** Parameter **	Description
Namespace	Unique identifier (auto-prefixed with project name)

4. Complete creation with default resource constraints

Configure Image Repository

Alauda Container Platform supports multiple image sourcing strategies:

Method 1: Integrated Registry via Toolchain

1. Access **Administrator** > **Toolchain** > **Integration**
2. Initiate new integration:

Parameter	Requirement
Name	Unique integration identifier
API Endpoint	Registry service URL (HTTP/HTTPS)
Secret	Pre-existing or newly created credential

3. Allocate registry to target platform project

Method 2: External Registry Services

- Use publicly accessible registry URLs
- Example: `nginx:latest`

Registry Access and Authentication

For the cluster to successfully pull container images, two key requirements must be met:

- **Network Accessibility**: The cluster nodes must have outbound (egress) network access to the container registry's endpoints. If the connection fails, you may encounter `ImagePullBackOff` or other network-related errors.

- **Authentication**: If the registry requires authentication, create an `ImagePullSecret` and reference it in your Pod or Deployment so the cluster can pull images. For step-by-step instructions, see the Image Pull Secrets documentation.

Create application via Deployment

> Deployments provide declarative updates for Pod replicasets.

Creation Process

1. From **Container Platform** view:
    - Use namespace selector to choose target isolation boundary
2. Navigate to **Workloads** > **Deployments**
3. Click **Create Deployment**
4. Specify image source:
    - Select integrated registry *or*
    - Input external image URL (e.g., `nginx:latest`)
5. Configure workload identity and launch

Management Operations

- Monitor replica status
- View events and logs
- Inspect YAML manifests
- Analyze resource metrics, alerts

Expose Service via NodePort

Services enable network accessibility to Pod groups.

Creation Process

1. Navigate to **Networking** > **Services**

2. Click **Create Service** with parameters:

Parameter	Value
Type	NodePort
Selector	Target Deployment name
Port Mapping	Service Port: Container Port (e.g., 8080:80)

3. Confirm creation.

Critical

- Cluster-visible virtual IP
- NodePort allocation range (30000-32767)

Internal routes enable service discovery for workloads by providing a unified IP address or host port for access.

1. Click on **Network** > **Service**.
2. Click on **Create Service**.
3. Configure the **Details** based on the parameters below, keeping other parameters at their defaults.

Parameter	Description
Name	Enter the name of the Service.
Type	`NodePort`
Workload Name	Select the `Deployment` created previously.

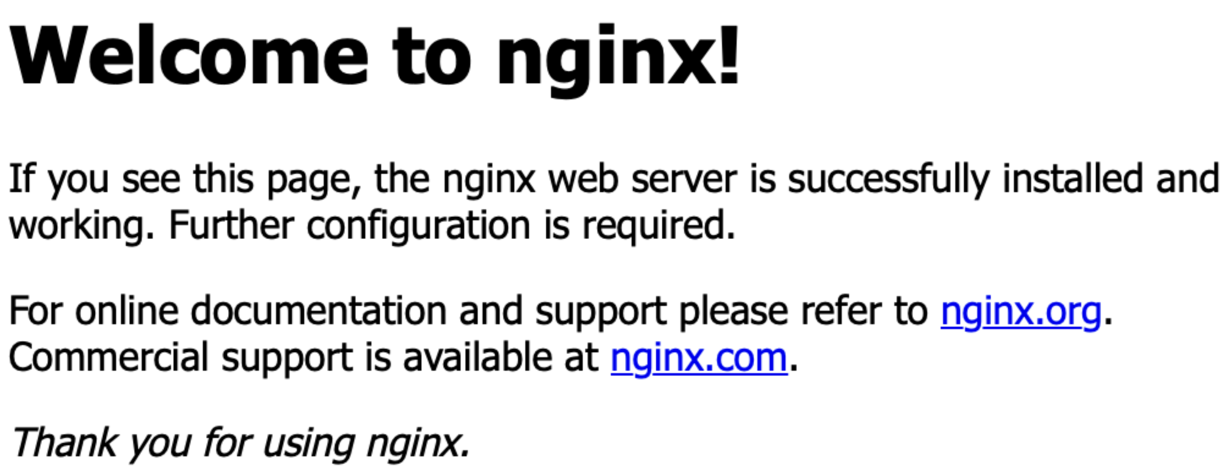
Parameter	Description
Port	**Service Port**: The port number exposed by the Service within the cluster, i.e., Port, e.g., `8080`. **Container Port**: The target port number (or name) mapped by the service port, i.e., targetPort, e.g., `80`.

4. Click on **Create**. At this point, the Service is successfully created.

Validate Application Accessibility

Verification Method

1. Obtain exposed endpoint components:
   - **Node IP**: Worker node public address
   - **NodePort**: Allocated external port
2. Construct access URL: `http://<Node_IP>:<NodePort>`
3. Expected result: Nginx welcome page



Building Applications

This section is the consolidated entry for application-oriented workflows in ACP.

Use this path first when you need to create applications, manage workloads, prepare namespaces, work with ConfigMaps or Secrets, or perform routine application operations and observability tasks.

TOC

Relationship to the migrated web console docs

Many workflows from the newer console experience now map directly to the pages in this section.

In most cases:

- Use `create_applications` for application creation and update paths.
- Use `application_workloads` for Deployments, StatefulSets, DaemonSets, Pods, Jobs, and CronJobs.
- Use `namespace` for namespace preparation and governance.
- Use `configuration` for ConfigMaps and Secrets.
- Use `operation_maintaining` and `application_observability` for status, versioning, logs, events, monitoring, and related operational tasks.

Across these task families, keep the following scope rules in mind:

- Aggregated application dashboards can summarize information across the clusters associated with the current project.
- Concrete resource operations still resolve in a selected cluster and namespace context.
- When a workflow can start from both the **Applications** view and the **Clusters** view, document the scope-specific behavior in the page where the action happens instead of keeping a separate parallel procedure.

Build application architecture

Build application architecture

Concepts

Namespaces

Creating Applications

Operation and Maintaining Applications

Updating and deleting Chart Applications

Workload Rebalancing (Descheduler)

Deleting Applications

Using PodDisruptionBudgets

Handling Out of Resource Errors

Health Checks

Workloads

Working with Helm charts

Working with Helm charts

Configurations

Configuring ConfigMap

Configuring Secrets

Application Observability

How To

Build application architecture

TOC

Introduction to the build application

Alauda Container Platform is a platform for developing and running containerized applications. It is designed to allow applications and the data centers that support them to expand from just a few machines and applications to thousands of machines that serve millions of clients.

Built on Kubernetes, Alauda Container Platform leverages the same robust technology that powers large-scale telecommunications, streaming video, gaming, banking, and other critical applications. This foundation enables you to extend your containerized applications across hybrid environments - from on-premise infrastructure to multi-cloud deployments.

Core components

Archon

Provides advanced APIs for application and resource management operations. As a control plane component, `Archon` exclusively runs on the `global` cluster, serving as the central management interface for cluster-wide operations. Its API layer enables declarative configuration of applications, namespaces, and infrastructure resources across the entire platform.

Metis

Functions as the multi-purpose controller within `business clusters`, delivering critical cluster-level operations:

- **Webhook management**: Admission webhooks for resource validation, including resource-ratio and resource-labeling policy enforcement.
- **Status synchronization**: Maintains consistency across distributed components through:
  - `Helm chart application` status reconciliation
  - `Project quota` synchronization
  - `Application` status updates (writing to `Application.status`)

Captain controller manager

Serves as the `Helm chart` application lifecycle management controller operating exclusively on the `global cluster`. Its responsibilities include:

- **Chart installation**: Orchestrating deployment of `Helm chart` across clusters
- **Version management**: Handling seamless upgrades and rollbacks of `Helm chart` releases
- **Uninstallation**: Complete removal of `Helm chart` application and associated resources
- **Release tracking**: Maintaining state and history of all deployed `Helm chart` releases

Icarus

Provides the centralized web-based management interface for `Container Platform`. As the presentation layer component, `Icarus`:

- Delivers comprehensive dashboard visualizations for cluster health monitoring
- Enables GUI-based application deployment and management workflows
- **Implements Kubernetes RBAC-based multi-tenant management**:
  - Distinguishes tenant accounts through namespace isolation
  - Manages resource access permissions per tenant
  - Provides tenant-specific view isolation
- Exclusively runs on the `global cluster`, serving as the unified control point for multi-cluster operations

Concepts

Application Types

In the platform's **Container Platform > Applications**, the following types of applications can be created:

- **Custom Application**: A Custom Application represents a complete business application composed of one or more interconnected computing components (such as Workloads like Deployments or StatefulSets), internal networking configurations (Services), and other native Kubernetes resources. This type of application offers flexible creation methods, supporting direct UI editing, YAML orchestration, and templated deployments, making it suitable for development, testing, and production environments. To learn more about this application type, refer to Custom Application. Different types of native applications can be created in the following ways:
  - Create from Image: Quickly create applications using existing container images.
  - Create from YAML: Create applications using YAML configuration files.
  - Create from Code: Create applications using source code.
- **Helm Chart Application**: A Helm Chart Application allows you to deploy and manage applications packaged as Helm Charts. Helm Charts are bundles of pre-configured Kubernetes resources that can be deployed as a single unit, simplifying the installation and management of complex applications. To learn more about this application type, refer to Helm Chart Application
- **Operator Backed Application**: An Operator-Backed Application leverages the power of Kubernetes Operators to automate the lifecycle management of complex applications. By deploying an application backed by an Operator, you benefit from automated deployment, scaling, upgrades, and maintenance, as the Operator acts as an intelligent controller

tailored to the specific application. To learn more about this application type, refer to Operator Backed Application.

Custom Applications

TOC

Understanding Custom Applications

A Custom Application is an application paradigm built on native Kubernetes resources (e.g., Deployment, Service, ConfigMap), strictly adhering to Kubernetes declarative API design principles. Users can define and deploy applications through standard YAML files or direct Kubernetes API calls, enabling fine-grained control over the application lifecycle. These are created from sources such as Images, code, and YAML are classified as custom application in Alauda Container Platform. Its design core lies in balancing flexibility and standardization, ideal for scenarios requiring deeply customized management.

Core Capabilities

1. **Declarative API-Driven Management**

- Aggregates distributed resources (e.g., Deployment, Service, Ingress) into a logical application unit through Application CRD, enabling atomic operations.

2. **Application-Level Abstraction & State Aggregation**

- Masks low-level resource details (e.g., Pod replica status). Developers can monitor overall application health (e.g., ready endpoint ratio, version consistency) directly via the Application resource.
- Supports cross-component dependency declarations (e.g., database service must start before application service) to ensure resource initialization order and coordination.

3. **Full Lifecycle Governance**

- Version Control: Tracks historical configurations, enabling one-click rollback to any stable state.
- Dependency Resolution: Automatically identifies and manages version compatibility between components (e.g., matching Service API versions with Ingress controllers).

4. **Enhanced Observability**

- Aggregates status metrics of all associated resources (e.g., Deployment available replicas, Service traffic load), providing a global view through a unified Dashboard.

Design Value

Dimension	Value Proposition
Complexity Management	Encapsulates scattered resources (e.g., Deployment, Service) into a single logical entity, reducing cognitive and operational overhead.
Standardization	Unifies application description standards via Application CRD, eliminating management entropy caused by YAML fragmentation.
Ecosystem Compatibility	Ecosystem Compatibility Seamlessly integrates with native toolchains (e.g., kubectl, Kubernetes Dashboard) and supports Helm Chart extensions.

Dimension	Value Proposition
DevOps Efficiency	Implements declarative delivery through GitOps pipelines (e.g., Argo CD), accelerating CI/CD automation.

Custom Application CRD Architecture Design

The Custom Application module defines two core CRD resources, forming atomic abstraction units for application management:

Dimension	Value Proposition
Application	Describes metadata and component topology of logical application units, aggregating resources like Deployment/Service into a single entity.
ApplicationHistory	Records all application lifecycle operations (create/update/rollback/delete) as versioned snapshots, tightly coupled with the Application CRD to enable end-to-end change traceability.

Application CRD Define

The Application CRD uses the `spec.componentKinds` field to declare Kubernetes resource types (e.g., Deployment, Service), enabling cross-resource lifecycle management.

```yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: applications.app.k8s.io
spec:
  group: app.k8s.io
  names:
    kind: Application
    listKind: ApplicationList
    plural: applications
    singular: application
  scope: Namespaced
  subresources:
    status: {}
  validation:
    openAPIV3Schema:
      properties:
        apiVersion:
          description: 'APIVersion defines the versioned schema of this representation
            of an object. Servers should convert recognized schemas to the latest
            internal value, and may reject unrecognized values. More info: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md#resources'
          type: string
        kind:
          description: 'Kind is a string value representing the REST resource this
            object represents. Servers may infer this from the endpoint the client
            submits requests to. Cannot be updated. In CamelCase. More info: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md#types-kinds'
          type: string
        metadata:
          description: 'Metadata is a object value representing the metadata of the kubernetes resource.
            More info: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md#metadata'
          type: object
        spec:
          properties:
```

```yaml
            assemblyPhase:
              description: |
                The installer can set this field to indicate that the application's components
                are still being deployed ("Pending") or all are deployed already ("Succeeded"). When the
                application cannot be successfully assembled, the installer can set this field to "Failed".'
              type: string
            componentKinds:
              description: |
                This array of GroupKinds is used to indicate the types of resources that the
                application is composed of. As an example an Application that has a service and a deployment
                would set this field to [{"group":"core","kind": "Service"},{"group":"apps","kind":"Deployment"}]
              items:
                description: 'The item of the GroupKinds, with a structure like \"{"group":"core","kind": "Service"}\"'
                type: object
              type: array
            descriptor:
              properties:
                description:
                  description: 'A short, human readable textual description of the Application.'
                  type: string
                icons:
                  description: 'A list of icons for an application. Icon information includes the source, size, and mime type.'
                  items:
                    properties:
                      size:
                        description: 'The size of the icon.'
                        type: string
                      src:
                        description: 'The source of the icon.'
                        type: string
                      type:
                        description: 'The mime type of the icon.'
                        type: string
                    required:
                    - src
```

```
                type: object
              type: array
            keywords:
              description: 'A list of keywords that identify the appl
ication.'
              items:
                type: string
              type: array
            links:
              description: 'Links are a list of descriptive URLs inte
nded to be used to surface additional documentation, dashboards, etc.'
              items:
                properties:
                  description:
                    description: 'The description of the link.'
                    type: string
                  url:
                    description: 'The url of the link.'
                    type: string
                type: object
              type: array
            maintainers:
              description: 'A list of the maintainers of the Applicat
ion. Each maintainer has a
                name, email, and URL. This field is meant for the dis
tributors of the Application
                to indicate their identity and contact information.'
              items:
                properties:
                  email:
                    description: 'The email of the maintainer.'
                    type: string
                  name:
                    description: 'The name of the maintainer.'
                    type: string
                  url:
                    description: 'The url to contact the maintainer.'
                    type: string
                type: object
              type: array
            notes:
              description: 'Notes contain human readable snippets int
ended as a quick start
                for the users of the Application. They may be plain t
```

```
ext or CommonMark markdown.'
                  type: string
                owners:
                  items:
                    properties:
                      email:
                        description: 'The email of the owner.'
                        type: string
                      name:
                        description: 'The name of the owner.'
                        type: string
                      url:
                        description: 'The url to contact the owner.'
                        type: string
                    type: object
                  type: array
                type:
                  description: 'The type of the application (e.g. WordPre
ss, MySQL, Cassandra).
                    You can have many applications of different names in
the same namespace.
                    They type field is used to indicate that they are all
the same type of application.'
                  type: string
                version:
                  description: 'A version indicator for the application
(e.g. 5.7 for MySQL version 5.7).'
                  type: string
              type: object
            info:
              description: 'Info contains human readable key-value pairs
for the Application.'
              items:
                properties:
                  name:
                    description: 'The name of the information.'
                    type: string
                  type:
                    description: 'The type of the information.'
                    type: string
                  value:
                    description: 'The value of the information.'
                    type: string
                  valueFrom:
```

```
                    description: 'The value reference from other resourc
e.'
                    properties:
                      configMapKeyRef:
                        description: 'The config map key reference.'
                        properties:
                          key:
                            type: string
                        type: object
                      ingressRef:
                        description: 'The ingress reference.'
                        properties:
                          host:
                            description: 'The host of the ingress referen
ce.'
                            type: string
                          path:
                            description: 'The path of the ingress referen
ce.'
                            type: string
                        type: object
                      secretKeyRef:
                        description: 'The secret key reference.'
                        properties:
                          key:
                            type: string
                        type: object
                      serviceRef:
                        description: 'The service reference.'
                        properties:
                          path:
                            description: 'The path of the service referen
ce.'
                            type: string
                          port:
                            description: 'The port of the service referen
ce.'
                            format: int32
                            type: integer
                        type: object
                      type:
                        type: string
                    type: object
                  type: object
```

```yaml
              type: array
            selector:
              description: 'The selector is used to match resources that belong to the Application.
                All of the applications resources should have labels such that they match this selector.
                Users should use the app.kubernetes.io/name label on all components of the Application
                and set the selector to match this label. For instance,
                {"matchLabels": [{"app.kubernetes.io/name": "my-cool-app"}]} should be used as the selector
                for an Application named "my-cool-app", and each component should contain a label that matches.'
              type: object
          type: object
        status:
          description: 'The status summarizes the current state of the object.'
          properties:
            observedGeneration:
              description: 'The observedGeneration is the generation most recently observed by the component
                responsible for acting upon changes to the desired state of the resource.'
              format: int64
              type: integer
          type: object
  version: v1beta1
  versions:
  - name: v1beta1
    served: true
    storage: true
```

ApplicationHistory Define

The ApplicationHistory CRD captures all lifecycle operations (e.g., creation, update, rollback) as version-controlled snapshots and is tightly integrated with the Application CRD to deliver end-to-end audit trails.

```yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: applicationhistories.app.k8s.io
spec:
  group: app.k8s.io
  names:
    kind: ApplicationHistory
    listKind: ApplicationHistoryList
    plural: applicationhistories
    singular: applicationhistory
  scope: Namespaced
  validation:
    openAPIV3Schema:
      properties:
        apiVersion:
          description: 'APIVersion defines the versioned schema of this r
epresentation
            of an object. Servers should convert recognized schemas to th
e latest
```

Workload Types

In addition to creating cloud-native applications via the Applications module, workloads can also be directly created in Container Platform > Workloads:

- Deployment: The most commonly used workload controller for deploying stateless applications. It ensures a specified number of Pod replicas are running, supporting rolling updates and rollbacks, ideal for stateless services like web servers and APIs.
- DaemonSet: Ensures a Pod runs on every node (or specific nodes) in the cluster. Pods are automatically created when nodes join and removed when nodes leave. Ideal for node-level tasks such as logging agents and monitoring daemons.
- StatefulSet: A workload controller for managing stateful applications. It provides stable network identities (hostname) and persistent storage for each Pod, ensuring data consistency even during rescheduling. Suitable for databases, distributed caches, and other stateful services.
- CronJob: Manages time-based Jobs using cron expressions. The system automatically creates Jobs at scheduled intervals, ideal for periodic tasks like backups, report generation, and cleanup jobs.
- Job: A workload for running finite tasks. It creates one or more Pods and ensures a specified number of successful completions before terminating. Suitable for batch processing, data migrations, and other one-time operations.

In addition to creating workloads via the web console, Kubernetes also supports direct management of lower-level resources via CLI tools::

- Pod: The smallest deployable unit in Kubernetes. A Pod can contain one or more tightly coupled containers sharing storage, network, and lifecycle. Pods are typically managed by higher-level controllers (e.g., Deployments).

- Container: A standardized unit encapsulating application code and dependencies, ensuring consistent execution across environments. Containers run within Pods and share the Pod's resources.

Understanding Parameters

TOC

Overview

Parameters in Kubernetes refer to command-line arguments passed to containers at runtime. They correspond to the `args` field in Kubernetes Pod specifications and override the default CMD arguments defined in container images. Parameters provide a flexible way to configure application behavior without rebuilding images.

Core Concepts

What are Parameters?

Parameters are runtime arguments that:

- Override the default CMD instruction in container images
- Are passed to the container's main process as command-line arguments
- Allow dynamic configuration of application behavior
- Enable reuse of the same image with different configurations

Relationship with Container Image

In container image terminology:

- **ENTRYPOINT**: Defines the executable (maps to Kubernetes `command`)
- **CMD**: Provides default arguments (maps to Kubernetes `args`)

- **Parameters**: Override CMD arguments while preserving ENTRYPOINT

```
# Dockerfile example
FROM nginx:alpine
ENTRYPOINT ["nginx"]
CMD ["-g", "daemon off;"]
```

```
# Kubernetes override
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: nginx
    image: nginx:alpine
    args: ["-g", "daemon off;", "-c", "/custom/nginx.conf"]
```

Use Cases and Scenarios

1. Application Configuration

Pass configuration options to applications:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  template:
    spec:
      containers:
      - name: app
        image: myapp:latest
        args:
        - "--port=8080"
        - "--log-level=info"
        - "--config=/etc/app/config.yaml"
```

2. Environment-Specific Deployment

Different parameters for different environments:

```
# Development
args: ["--debug", "--reload", "--port=3000"]

# Production
args: ["--optimize", "--port=80", "--workers=4"]
```

3. Database Connection Configuration

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: db-client
    image: postgres:13
    args:
    - "psql"
    - "-h"
    - "postgres.example.com"
    - "-p"
    - "5432"
    - "-U"
    - "myuser"
    - "-d"
    - "mydb"
```

CLI Examples and Practical Usage

Using kubectl run

```
# Basic parameter passing
kubectl run nginx --image=nginx:alpine --restart=Never -- -g "daemon of
f;" -c "/custom/nginx.conf"

# Multiple parameters
kubectl run myapp --image=myapp:latest --restart=Never -- --port=8080 --l
og-level=debug

# Interactive debugging
kubectl run debug --image=ubuntu:20.04 --restart=Never -it -- /bin/bash
```

Using kubectl create

```
# Create deployment with parameters
kubectl create deployment web --image=nginx:alpine --dry-run=client -o ya
ml > deployment.yaml

# Edit the generated YAML to add args:
# spec:
#   template:
#     spec:
#       containers:
#       - name: nginx
#         image: nginx:alpine
#         args: ["-g", "daemon off;", "-c", "/custom/nginx.conf"]

kubectl apply -f deployment.yaml
```

Complex Parameter Examples

Web Server with Custom Configuration

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-custom
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-custom
  template:
    metadata:
      labels:
        app: nginx-custom
    spec:
      containers:
      - name: nginx
        image: nginx:1.21-alpine
        args:
        - "-g"
        - "daemon off;"
        - "-c"
        - "/etc/nginx/custom.conf"
        ports:
        - containerPort: 80
        volumeMounts:
        - name: config
          mountPath: /etc/nginx/custom.conf
          subPath: nginx.conf
      volumes:
      - name: config
        configMap:
          name: nginx-config
```

Application with Multiple Parameters

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp
spec:
  containers:
  - name: app
    image: mycompany/myapp:v1.2.3
    args:
    - "--server-port=8080"
    - "--database-url=postgresql://db:5432/mydb"
    - "--log-level=info"
    - "--metrics-enabled=true"
    - "--cache-size=256MB"
    - "--worker-threads=4"
```

Best Practices

1. Parameter Design Principles

- **Use meaningful parameter names**: `--port=8080` instead of `-p 8080`
- **Provide sensible defaults**: Ensure applications work without parameters
- **Document all parameters**: Include help text and examples
- **Validate input**: Check parameter values and provide error messages

2. Security Considerations

```yaml
# ❌ Avoid sensitive data in parameters
args: ["--api-key=secret123", "--password=mypass"]

# ✅ Use environment variables for secrets
env:
- name: API_KEY
  valueFrom:
    secretKeyRef:
      name: app-secrets
      key: api-key
args: ["--config-from-env"]
```

3. Configuration Management

```yaml
# ✅ Combine parameters with ConfigMaps
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    args:
    - "--config=/etc/config/app.yaml"
    - "--log-level=info"
    volumeMounts:
    - name: config
      mountPath: /etc/config
  volumes:
  - name: config
    configMap:
      name: app-config
```

Troubleshooting Common Issues

1. Parameter Not Recognized

```
# Check container logs
kubectl logs pod-name

# Common error: unknown flag
# Solution: Verify parameter syntax and application documentation
```

2. Parameter Override Not Working

```
# ❌ Incorrect: mixing command and args
command: ["myapp", "--port=8080"]
args: ["--log-level=debug"]

# ✅ Correct: use args only to override CMD
args: ["--port=8080", "--log-level=debug"]
```

3. Debugging Parameter Issues

```
# Run container interactively to test parameters
kubectl run debug --image=myapp:latest -it --rm --restart=Never -- /bin/s
h

# Inside container, test the command manually
/app/myapp --port=8080 --log-level=debug
```

Advanced Usage Patterns

1. Conditional Parameters with Init Containers

```yaml
apiVersion: v1
kind: Pod
spec:
  initContainers:
  - name: config-generator
    image: busybox
    command: ['sh', '-c']
    args:
    - |
      if [ "$ENVIRONMENT" = "production" ]; then
        echo "--optimize --workers=8" > /shared/args
      else
        echo "--debug --reload" > /shared/args
      fi
    volumeMounts:
    - name: shared
      mountPath: /shared
  containers:
  - name: app
    image: myapp:latest
    command: ['sh', '-c']
    args: ['exec myapp $(cat /shared/args)']
    volumeMounts:
    - name: shared
      mountPath: /shared
  volumes:
  - name: shared
    emptyDir: {}
```

2. Parameter Templating with Helm

```yaml
# values.yaml
app:
  parameters:
    port: 8080
    logLevel: info
    workers: 4

# deployment.yaml template
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:
      - name: app
        image: myapp:latest
        args:
        - "--port={{ .Values.app.parameters.port }}"
        - "--log-level={{ .Values.app.parameters.logLevel }}"
        - "--workers={{ .Values.app.parameters.workers }}"
```

Parameters provide a powerful mechanism for configuring containerized applications in Kubernetes. By understanding how to properly use parameters, you can create flexible, reusable, and maintainable deployments that adapt to different environments and requirements.

Understanding Environment Variables

TOC

4. Default Values and Validation

Overview

Environment variables in Kubernetes are key-value pairs that provide configuration data to containers at runtime. They offer a flexible and secure way to inject configuration information, secrets, and runtime parameters into your applications without modifying container images or application code.

Core Concepts

What are Environment Variables?

Environment variables are:

- Key-value pairs available to processes running inside containers
- Runtime configuration mechanism that doesn't require image rebuilds
- Standard way to pass configuration data to applications
- Accessible through standard operating system APIs in any programming language

Environment Variable Sources in Kubernetes

Kubernetes supports multiple sources for environment variables:

Source Type	Description	Use Case
Static Values	Direct key-value pairs	Simple configuration
ConfigMap	Reference to ConfigMap keys	Non-sensitive configuration
Secret	Reference to Secret keys	Sensitive data (passwords, tokens)

Source Type	Description	Use Case
Field Reference	Pod/Container metadata	Dynamic runtime information
Resource Reference	Resource requests/limits	Resource-aware configuration

Environment Variable Precedence

Environment variables override configuration in this order:

1. **Kubernetes env** (highest priority)
2. **Referenced ConfigMaps/Secrets**
3. **Dockerfile ENV instructions**
4. **Application default values** (lowest priority)

Use Cases and Scenarios

1. Application Configuration

Basic application settings:

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: web-app
    image: myapp:latest
    env:
    - name: PORT
      value: "8080"
    - name: LOG_LEVEL
      value: "info"
    - name: ENVIRONMENT
      value: "production"
    - name: MAX_CONNECTIONS
      value: "100"
```

2. Database Configuration

Database connection settings using ConfigMaps and Secrets:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  DB_HOST: "postgres.example.com"
  DB_PORT: "5432"
  DB_NAME: "myapp"
  DB_POOL_SIZE: "10"

---
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  DB_USER: bXl1c2Vy  # base64 encoded "myuser"
  DB_PASSWORD: bXlwYXNzd29yZA==  # base64 encoded "mypassword"

---
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    env:
    # From ConfigMap
    - name: DB_HOST
      valueFrom:
        configMapKeyRef:
          name: db-config
          key: DB_HOST
    - name: DB_PORT
      valueFrom:
        configMapKeyRef:
          name: db-config
          key: DB_PORT
    - name: DB_NAME
      valueFrom:
        configMapKeyRef:
          name: db-config
```

```yaml
          key: DB_NAME
    # From Secret
    - name: DB_USER
      valueFrom:
        secretKeyRef:
          name: db-secret
          key: DB_USER
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: db-secret
          key: DB_PASSWORD
```

3. Dynamic Runtime Information

Access Pod and Node metadata:

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    env:
    # Pod information
    - name: POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: NODE_NAME
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
    # Resource information
    - name: CPU_REQUEST
      valueFrom:
        resourceFieldRef:
          resource: requests.cpu
    - name: MEMORY_LIMIT
      valueFrom:
        resourceFieldRef:
          resource: limits.memory
```

4. Environment-Specific Configuration

Different configurations for different environments:

```yaml
# Development environment
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config-dev
data:
  DEBUG: "true"
  LOG_LEVEL: "debug"
  CACHE_TTL: "60"
  RATE_LIMIT: "1000"

---
# Production environment
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config-prod
data:
  DEBUG: "false"
  LOG_LEVEL: "warn"
  CACHE_TTL: "3600"
  RATE_LIMIT: "100"

---
# Deployment using environment-specific config
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  template:
    spec:
      containers:
      - name: app
        image: myapp:latest
        envFrom:
        - configMapRef:
            name: app-config-prod  # Change to app-config-dev for develop
ment
```

CLI Examples and Practical Usage

Using kubectl run

```
# Set environment variables directly
kubectl run myapp --image=nginx --env="PORT=8080" --env="DEBUG=true"

# Multiple environment variables
kubectl run webapp --image=myapp:latest \
  --env="DATABASE_URL=postgresql://localhost:5432/mydb" \
  --env="REDIS_URL=redis://localhost:6379" \
  --env="LOG_LEVEL=info"

# Interactive pod with environment variables
kubectl run debug --image=ubuntu:20.04 -it --rm \
  --env="TEST_VAR=hello" \
  --env="ANOTHER_VAR=world" \
  -- /bin/bash
```

Using kubectl create

```
# Create ConfigMap from literal values
kubectl create configmap app-config \
  --from-literal=DATABASE_HOST=postgres.example.com \
  --from-literal=DATABASE_PORT=5432 \
  --from-literal=CACHE_SIZE=256MB

# Create ConfigMap from file
echo "DEBUG=true" > app.env
echo "LOG_LEVEL=debug" >> app.env
kubectl create configmap app-env --from-env-file=app.env

# Create Secret for sensitive data
kubectl create secret generic db-secret \
  --from-literal=username=myuser \
  --from-literal=password=mypassword
```

Complex Environment Variable Examples

Microservices with Service Discovery

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: service-config
data:
  USER_SERVICE_URL: "http://user-service:8080"
  ORDER_SERVICE_URL: "http://order-service:8080"
  PAYMENT_SERVICE_URL: "http://payment-service:8080"
  NOTIFICATION_SERVICE_URL: "http://notification-service:8080"

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-gateway
spec:
  template:
    spec:
      containers:
      - name: gateway
        image: api-gateway:latest
        env:
        - name: PORT
          value: "8080"
        - name: ENVIRONMENT
          value: "production"
        envFrom:
        - configMapRef:
            name: service-config
        - secretRef:
            name: api-keys
```

Multi-Container Pod with Shared Configuration

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-app
spec:
  containers:
  # Main application
  - name: app
    image: myapp:latest
    env:
    - name: ROLE
      value: "primary"
    - name: SHARED_SECRET
      valueFrom:
        secretKeyRef:
          name: shared-secret
          key: token
    envFrom:
    - configMapRef:
        name: shared-config

  # Sidecar container
  - name: sidecar
    image: sidecar:latest
    env:
    - name: ROLE
      value: "sidecar"
    - name: MAIN_APP_URL
      value: "http://localhost:8080"
    - name: SHARED_SECRET
      valueFrom:
        secretKeyRef:
          name: shared-secret
          key: token
    envFrom:
    - configMapRef:
        name: shared-config
```

Best Practices

1. Security Best Practices

```yaml
# ✅ Use Secrets for sensitive data
apiVersion: v1
kind: Secret
metadata:
  name: app-secrets
type: Opaque
data:
  api-key: <base64-encoded-value>
  database-password: <base64-encoded-value>

---
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    env:
    # ✅ Reference secrets
    - name: API_KEY
      valueFrom:
        secretKeyRef:
          name: app-secrets
          key: api-key
    # ❌ Avoid hardcoding sensitive data
    # - name: API_KEY
    #   value: "secret-api-key-123"
```

2. Configuration Organization

```yaml
# ✅ Organize configuration by purpose
apiVersion: v1
kind: ConfigMap
metadata:
  name: database-config
data:
  DB_HOST: "postgres.example.com"
  DB_PORT: "5432"
  DB_POOL_SIZE: "10"

---
apiVersion: v1
kind: ConfigMap
metadata:
  name: cache-config
data:
  REDIS_HOST: "redis.example.com"
  REDIS_PORT: "6379"
  CACHE_TTL: "3600"

---
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    envFrom:
    - configMapRef:
        name: database-config
    - configMapRef:
        name: cache-config
```

3. Environment Variable Naming

```yaml
# ✅ Use consistent naming conventions
env:
- name: DATABASE_HOST      # Clear, descriptive names
  value: "postgres.example.com"
- name: DATABASE_PORT      # Use underscores for separation
  value: "5432"
- name: LOG_LEVEL         # Use uppercase for environment variables
  value: "info"
- name: FEATURE_FLAG_NEW_UI  # Prefix related variables
  value: "true"

# ❌ Avoid unclear or inconsistent naming
# - name: db               # Too short
# - name: databaseHost     # Inconsistent casing
# - name: log-level        # Inconsistent separator
```

4. Default Values and Validation

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    env:
    - name: PORT
      value: "8080"         # Provide sensible defaults
    - name: LOG_LEVEL
      value: "info"         # Default to safe values
    - name: TIMEOUT_SECONDS
      value: "30"           # Include units in names
    - name: MAX_RETRIES
      value: "3"            # Limit retry attempts
```

Understanding Startup Commands

TOC

Overview

Startup commands in Kubernetes define the primary executable that runs when a container starts. They correspond to the `command` field in Kubernetes Pod specifications and override the default ENTRYPOINT instruction defined in container images. Startup commands provide complete control over what process runs inside your containers.

Core Concepts

What are Startup Commands?

Startup commands are:

- The primary executable that runs when a container starts
- Override the ENTRYPOINT instruction in container images
- Define the main process (PID 1) inside the container
- Work in conjunction with parameters (args) to form the complete command line

Relationship with Dockerfile and Parameters

Understanding the relationship between Dockerfile instructions and Kubernetes fields:

Dockerfile	Kubernetes	Purpose
ENTRYPOINT	`command`	Defines the executable

Dockerfile	Kubernetes	Purpose
CMD	`args`	Provides default arguments

```
# Dockerfile example
FROM ubuntu:20.04
ENTRYPOINT ["/usr/bin/myapp"]
CMD ["--config=/etc/default.conf"]
```

```yaml
# Kubernetes override
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: myapp
    image: myapp:latest
    command: ["/usr/bin/myapp"]
    args: ["--config=/etc/custom.conf", "--debug"]
```

Command vs Args Interaction

Scenario	Container Image	Kubernetes Spec	Resulting Command
Default	ENTRYPOINT + CMD	(none)	ENTRYPOINT + CMD
Override args only	ENTRYPOINT + CMD	`args: ["new-args"]`	ENTRYPOINT + new-args
Override command only	ENTRYPOINT + CMD	`command: ["new-cmd"]`	new-cmd
Override both	ENTRYPOINT + CMD	`command: ["new-cmd"]` `args: ["new-args"]`	new-cmd + new-args

Use Cases and Scenarios

1. Custom Application Startup

Run different applications using the same base image:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-server
spec:
  containers:
  - name: nginx
    image: ubuntu:20.04
    command: ["/usr/sbin/nginx"]
    args: ["-g", "daemon off;", "-c", "/etc/nginx/nginx.conf"]
```

2. Debugging and Troubleshooting

Override the default command to start a shell for debugging:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: debug-pod
spec:
  containers:
  - name: debug
    image: myapp:latest
    command: ["/bin/bash"]
    args: ["-c", "sleep 3600"]
```

3. Initialization Scripts

Run custom initialization before starting the main application:

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    command: ["/bin/sh"]
    args:
    - "-c"
    - |
      echo "Initializing application..."
      /scripts/init.sh
      echo "Starting main application..."
      exec /usr/bin/myapp --config=/etc/app.conf
```

4. Multi-Purpose Images

Use the same image for different purposes:

```yaml
# Web server
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  template:
    spec:
      containers:
      - name: web
        image: myapp:latest
        command: ["/usr/bin/myapp"]
        args: ["server", "--port=8080"]

---
# Background worker
apiVersion: apps/v1
kind: Deployment
metadata:
  name: worker
spec:
  template:
    spec:
      containers:
      - name: worker
        image: myapp:latest
        command: ["/usr/bin/myapp"]
        args: ["worker", "--queue=tasks"]

---
# Database migration
apiVersion: batch/v1
kind: Job
metadata:
  name: migrate
spec:
  template:
    spec:
      containers:
      - name: migrate
        image: myapp:latest
        command: ["/usr/bin/myapp"]
        args: ["migrate", "--up"]
```

```
    restartPolicy: Never
```

CLI Examples and Practical Usage

Using kubectl run

```
# Override command completely
kubectl run debug --image=nginx:alpine --command -- /bin/sh -c "sleep 3600"

# Run interactive shell
kubectl run -it debug --image=ubuntu:20.04 --restart=Never --command -- /bin/bash

# Custom application startup
kubectl run myapp --image=myapp:latest --command -- /usr/local/bin/start.sh --config=/etc/app.conf

# One-time task
kubectl run task --image=busybox --restart=Never --command -- /bin/sh -c "echo 'Task completed'"
```

Using kubectl create job

```
# Create a job with custom command
kubectl create job backup --image=postgres:13 --dry-run=client -o yaml -- pg_dump -h db.example.com mydb > backup.yaml

# Apply the job
kubectl apply -f backup.yaml
```

Complex Startup Command Examples

Multi-Step Initialization

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: complex-init
spec:
  containers:
  - name: app
    image: myapp:latest
    command: ["/bin/bash"]
    args:
    - "-c"
    - |
      set -e
      echo "Step 1: Checking dependencies..."
      /scripts/check-deps.sh

      echo "Step 2: Setting up configuration..."
      /scripts/setup-config.sh

      echo "Step 3: Running database migrations..."
      /scripts/migrate.sh

      echo "Step 4: Starting application..."
      exec /usr/bin/myapp --config=/etc/app/config.yaml
    volumeMounts:
    - name: scripts
      mountPath: /scripts
    - name: config
      mountPath: /etc/app
  volumes:
  - name: scripts
    configMap:
      name: init-scripts
      defaultMode: 0755
  - name: config
    configMap:
      name: app-config
```

Conditional Startup Logic

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: conditional-app
spec:
  template:
    spec:
      containers:
      - name: app
        image: myapp:latest
        command: ["/bin/sh"]
        args:
        - "-c"
        - |
          if [ "$APP_MODE" = "worker" ]; then
            exec /usr/bin/myapp worker --queue=$QUEUE_NAME
          elif [ "$APP_MODE" = "scheduler" ]; then
            exec /usr/bin/myapp scheduler --interval=60
          else
            exec /usr/bin/myapp server --port=8080
          fi
        env:
        - name: APP_MODE
          value: "server"
        - name: QUEUE_NAME
          value: "default"
```

Best Practices

1. Signal Handling and Graceful Shutdown

```yaml
# ✅ Proper signal handling
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    command: ["/bin/bash"]
    args:
    - "-c"
    - |
      # Trap SIGTERM for graceful shutdown
      trap 'echo "Received SIGTERM, shutting down gracefully..."; kill -T
ERM $PID; wait $PID' TERM

      # Start the main application in background
      /usr/bin/myapp --config=/etc/app.conf &
      PID=$!

      # Wait for the process
      wait $PID
```

2. Error Handling and Logging

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    command: ["/bin/bash"]
    args:
    - "-c"
    - |
      set -euo pipefail  # Exit on error, undefined vars, pipe failures

      log() {
        echo "[$(date '+%Y-%m-%d %H:%M:%S')] $*" >&2
      }

      log "Starting application initialization..."

      if ! /scripts/health-check.sh; then
        log "ERROR: Health check failed"
        exit 1
      fi

      log "Starting main application..."
      exec /usr/bin/myapp --config=/etc/app.conf
```

3. Security Considerations

```yaml
# ✅ Run as non-root user
apiVersion: v1
kind: Pod
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    runAsGroup: 1000
  containers:
  - name: app
    image: myapp:latest
    command: ["/usr/bin/myapp"]
    args: ["--config=/etc/app.conf"]
    securityContext:
      allowPrivilegeEscalation: false
      readOnlyRootFilesystem: true
      capabilities:
        drop:
        - ALL
```

4. Resource Management

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    command: ["/usr/bin/myapp"]
    args: ["--config=/etc/app.conf"]
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Advanced Usage Patterns

1. Init Containers with Custom Commands

```yaml
apiVersion: v1
kind: Pod
spec:
  initContainers:
  - name: setup
    image: busybox
    command: ["/bin/sh"]
    args:
    - "-c"
    - |
      echo "Setting up shared data..."
      mkdir -p /shared/data
      echo "Setup complete" > /shared/data/status
    volumeMounts:
    - name: shared-data
      mountPath: /shared
  containers:
  - name: app
    image: myapp:latest
    command: ["/bin/sh"]
    args:
    - "-c"
    - |
      while [ ! -f /shared/data/status ]; do
        echo "Waiting for setup to complete..."
        sleep 1
      done
      echo "Starting application..."
      exec /usr/bin/myapp
    volumeMounts:
    - name: shared-data
      mountPath: /shared
  volumes:
  - name: shared-data
    emptyDir: {}
```

2. Sidecar Containers with Different Commands

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  # Main application
  - name: app
    image: myapp:latest
    command: ["/usr/bin/myapp"]
    args: ["--config=/etc/app.conf"]

  # Log shipper sidecar
  - name: log-shipper
    image: fluent/fluent-bit:latest
    command: ["/fluent-bit/bin/fluent-bit"]
    args: ["--config=/fluent-bit/etc/fluent-bit.conf"]

  # Metrics exporter sidecar
  - name: metrics
    image: prom/node-exporter:latest
    command: ["/bin/node_exporter"]
    args: ["--path.rootfs=/host"]
```

3. Job Patterns with Custom Commands

```yaml
# Backup job
apiVersion: batch/v1
kind: Job
metadata:
  name: database-backup
spec:
  template:
    spec:
      containers:
      - name: backup
        image: postgres:13
        command: ["/bin/bash"]
        args:
        - "-c"
        - |
          set -e
          echo "Starting backup at $(date)"
          pg_dump -h $DB_HOST -U $DB_USER $DB_NAME > /backup/dump-$(date
+%Y%m%d-%H%M%S).sql
          echo "Backup completed at $(date)"
        env:
        - name: DB_HOST
          value: "postgres.example.com"
        - name: DB_USER
          value: "backup_user"
        - name: DB_NAME
          value: "myapp"
        volumeMounts:
        - name: backup-storage
          mountPath: /backup
      restartPolicy: Never
      volumes:
      - name: backup-storage
        persistentVolumeClaim:
          claimName: backup-pvc
```

Startup commands provide complete control over container execution in Kubernetes. By understanding how to properly configure and use startup commands, you can create flexible, maintainable, and robust containerized applications that meet your specific requirements.

Resource Unit Description

- CPU: Optional units are: core, m (millicore). Where 1 core = 1000 m.
- Memory: Optional units are: Mi (1 MiB = 2^20 bytes), Gi (1 GiB = 2^30 bytes). Where 1 Gi = 1024 Mi.
- Virtual GPU (optional): This parameter is only effective when there are GPU resources under the cluster. The number of virtual GPU cores; 100 virtual cores equal 1 physical GPU core. It supports positive integers.
- Video Memory (optional): This parameter is only effective when there are GPU resources under the cluster. Virtual GPU video memory; 1 unit of video memory equals 256 Mi. It supports positive integers.

Namespaces

The **Namespaces** section covers namespace lifecycle and governance operations for application workloads.

Use these pages when you need to create a namespace, import an existing namespace into a project, update namespace governance settings, or remove or delete a namespace.

TOC

Choose the correct entry view

Namespace management can start from different views in the new web console, and the entry path determines the operational scope.

- Use the **Applications** view when the goal is project-scoped namespace management in a cluster already associated with the current project.
- Use the **Clusters** view when the goal is direct cluster-side namespace governance for a concrete cluster resource.

The **Clusters** view does not cover project-association workflows such as importing a namespace into a project or removing it from a project.

Main tasks

- Creating Namespaces
- Importing Namespaces
- Updating Namespaces
- Deleting/Removing Namespaces

When you choose the target page, place the workflow guidance where the user performs the action:

- create guidance belongs in the creation page together with quota, limit range, and policy review
- import guidance belongs in the import page because it changes project governance expectations
- update guidance belongs in the update page beside quota, limit range, and security operations
- remove or delete guidance belongs in the removal page because project disassociation and resource deletion are different actions

---

Creating Namespaces

Understanding namespaces

Creating namespaces by using web consc

Creating namespaces by using new web c

Creating namespace in a project by **ac** CL

Creating namespace by **kubectl**

Importing Namespaces

Overview

Use Cases

Prerequisites

Procedure

Importing namespaces by using new web console

Related guidance

Resource Q

Understanding

Quotas

Hardware acce

Limit Range

UID/GID Assignment

Enable UID/GID Assignment

Verify UID/GID Assignment

Overcommi

Updating Namespaces

Updating Quotas

Updating Container LimitRanges

Updating Pod Security Admission

Updating a namespace from the Clusters view in the new web console

le

Deleting/Removing Namespaces

Deleting Namespaces

Removing Namespaces

How to choose between remove and delete

Removing or deleting namespaces by using new web console

Creating Namespaces

TOC

Understanding namespaces

Refer to the official Kubernetes documentation: Namespaces

> In Kubernetes, namespaces provide a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. Namespace-based scoping is applicable only for namespaced objects (e.g.

> Deployments, Services, etc.) and not for cluster-wide objects (e.g. StorageClass, Nodes, PersistentVolumes, etc.).

Creating namespaces by using web console

> Within the cluster associated with the project, create a new namespace aligned with the project's available resource quotas. The new namespace operates within the resource quotas allocated to the project (e.g., CPU, memory), and all resources in the namespace must reside within the associated cluster.

1. In the **Project Management** view, click on the ***Project Name*** for which you want to create a namespace.
2. In the left navigation bar, click on **Namespaces** > **Namespaces**.
3. Click on **Create Namespace**.
4. Configure **Basic Information**.

Parameter	Description
Cluster	Select the cluster linked to the project to host the namespace.
Namespace	The namespace name must include a mandatory prefix, which is the project name.

5. (Optional) Configure Resource Quota.
Every time a resource limit (limits) for computational or storage resources is specified for a container within the namespace, or each time a new Pod or PVC is added, it will consume the quota set here.
NOTICE:

- The namespace's resource quota is inherited from the project's allocated quota in the cluster. The maximum allowable quota for a resource type cannot exceed the remaining available quota of the project. If any resource's available quota reaches 0, namespace creation will be blocked. Contact your platform administrator for quota adjustments.
- **GPU Quota Configuration Requirements**:

  - GPU quotas (vGPU or pGPU) can only be configured if GPU resources are provisioned in the cluster.
  - When using vGPU, memory quotas can also be set.

  GPU Unit Definitions:

  - **vGPU Units**: 100 virtual GPU units (vGPU) = 1 physical GPU core (pGPU).
    - Note: pGPU units are counted in whole numbers only (e.g., 1 pGPU = 1 core = 100 vGPU).
  - **Memory Units**:
    - 1 memory unit = 256 MiB.
    - 1 GiB = 4 memory units (1024 MiB = 4 × 256 MiB).

- **Default Quota Behavior**:
  - If no quota is specified for a resource type, the default is unbounded.
  - This means the namespace can consume **all available resources of that type allocated to the project** without explicit limits.

Quota Parameter Description

Category	Quota Type	Value and Unit	Description
Storage Resource Quota	All	Gi	The total requested storage capacity of all Persistent Volume Claims (PVCs) in this namespace cannot exceed this value.
	Storage Class		The total requested storage capacity of all Persistent Volume Claims (PVCs) associated with the selected StorageClass in this namespace cannot exceed this value.

Category	Quota Type	Value and Unit	Description
			Note: Please allocate StorageClass to the project that the namespace belongs to in advance.
Extended Resources	Obtained from the configuration dictionary (ConfigMap); please refer to Extended Resources Quotas description for details.	-	This category will not be displayed if there is no corresponding configuration dictionary.
Other Quotas	Enter custom quotas; for specific input rules, please refer to Other Quota description.	-	To avoid problems of resource duplication, the following values are not allowed as quota types: • limits.cpu • limits.memory • requests.cpu • requests.memory • pods • cpu • memory

6. (Optional) Configure **Container Limit Range**; please refer to Limit Range for more details.
7. (Optional) Configure **Pod Security Admission**; please refer to Pod Security Admission for specific details.

8. (Optional) In the **More Configuration** area, add labels and annotations for the current namespace.
   Tip: You can define the attributes of the namespace through labels or supplement the namespace with additional information through annotations; both can be used to filter and sort namespaces.
9. Click on **Create**.

Creating namespaces by using new web console

This page mainly describes the legacy console workflow, but namespace creation in the new web console can start from different views.

Create from the Applications view

Use the **Applications** view when you are creating a namespace for the current project.

Entry path and action order:

- Open the current project, then go to **Applications** > **Namespaces**.
- Click **Create Namespace**.
- Select the target **Cluster** first, then continue with namespace name, quota, limit range, Pod Security Admission, labels, and annotations.

Before submitting the form, confirm that:

- the target cluster is already associated with the current project
- cluster selection happens before namespace submission because the namespace is created as part of the current project's cluster-scoped governance workflow
- the namespace naming rule matches current project governance expectations
- quota, limit range, security policy, labels, and annotations reflect the intended workload baseline
- the namespace appears afterward in the current project and is associated with the expected cluster

Create from the Clusters view

Use the **Clusters** view when you are creating a namespace as a cluster resource in a concrete cluster.

Entry path and action order:

- Open the target cluster first.
- Go to **Namespaces** in that cluster context.
- Click **Create Namespace** and continue with namespace name, labels, and annotations.

Creating namespace in a project by ac CLI

To use `ac` command-line tool to create a namespace in a project, run the following command:

```
# Create a namespace in project with specific clusters
ac adm new-project-namespace <namespace-name> --project <project-name> --cluster <cluster-name>
```

Creating namespace by kubectl

YAML file examples

```
example-namespace.yaml
```

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: example
  labels:
    pod-security.kubernetes.io/audit: baseline # Option, to ensure securi
ty, it is recommended to choose the baseline or restricted mode.
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/warn: baseline
```

example-resourcequota.yaml

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: example-resourcequota
  namespace: example
spec:
  hard:
    limits.cpu: '20'
    limits.memory: 20Gi
    pods: '500'
    requests.cpu: '2'
    requests.memory: 2Gi
```

example-limitrange.yaml

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: example-limitrange
  namespace: example
spec:
  limits:
    - default:
        cpu: 100m
        memory: 100Mi
      defaultRequest:
        cpu: 50m
        memory: 50Mi
      max:
        cpu: 1000m
        memory: 1000Mi
      type: Container
```

Create via YAML file

```
kubectl apply -f example-namespace.yaml
kubectl apply -f example-resourcequota.yaml
kubectl apply -f example-limitrange.yaml
```

Create via command line directly

```
kubectl create namespace example
kubectl create resourcequota example-resourcequota --namespace=example --hard=limits.cpu=20,limits.memory=20Gi,pods=500
kubectl create limitrange example-limitrange --namespace=example --default='cpu=100m,memory=100Mi' --default-request='cpu=50m,memory=50Mi' --max='cpu=1000m,memory=1000Mi'
```

Importing Namespaces

TOC

Overview

Namespace Lifecycle Management Capabilities:

- Cross-Cluster Namespace Import: Importing Namespaces into a Project centralizes their management across all Kubernetes Clusters provisioned by the platform. This provides administrators with unified resource governance and monitoring capabilities across distributed environments.

Namespace Disassociation:

- The Disassociate Namespace feature enables you to unlink a Namespace from its current Project, resetting its association for subsequent reassignment or cleanup.

- Importing a Namespace into a Project grants it capabilities equivalent to those of natively created Namespaces on the platform. This includes inherited Project-level Policies (e.g., Resource Quotas), unified monitoring, and centralized governance controls.

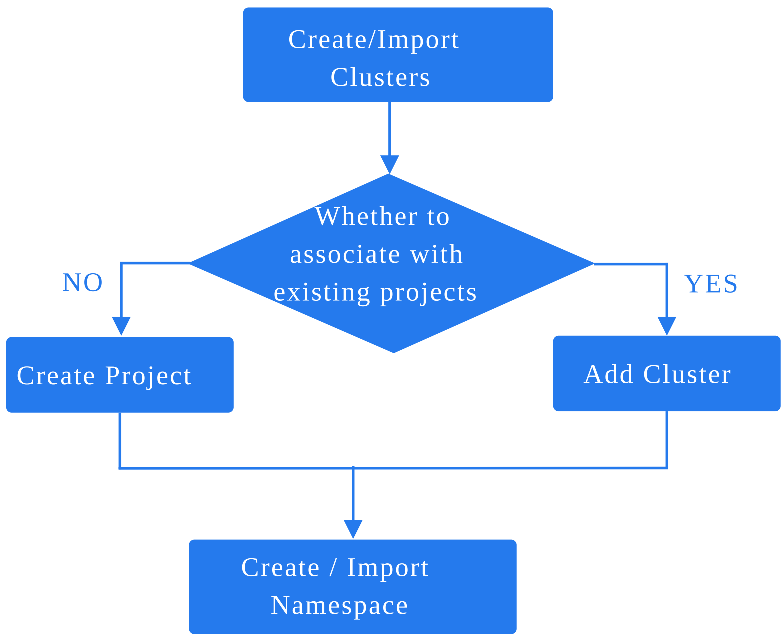
Important Notes:

- A Namespace can only be associated with one Project at any given time.
- If a Namespace is already linked to a Project, it cannot be imported into or reassigned to another Project without first disassociating it from its original Project.

Use Cases

Common use cases for **Namespace** management include:

- Upon connecting a new **Kubernetes cluster** to the platform, you can utilize the **Import Namespace** feature to associate its existing **Kubernetes Namespaces** with a Project. Simply select the target Project and Cluster to initiate the import. This action grants the **project** governance over these **namespace**, encompassing **Resource Quotas**, monitoring, and policy enforcement.



- A **namespace** that has been disassociated from one **project** can be seamlessly re-associated with another project via the **Import Namespace** feature for continued centralized governance.

- Namespaces not currently managed by any **project** (e.g., those created via cluster-level scripts) must be linked to a target **project** using the **Import Namespace** feature to enable platform-level governance, including visibility and centralized management.

Prerequisites

- The Namespace is not currently managed by any existing Project within the platform.
- Namespaces can only be imported into a Project that is already associated with their target Kubernetes Cluster. If no such Project exists, you must first provision a Project linked to that Cluster.

Procedure

1. **Project Management**, click on the ***Project name*** where the namespace is to be imported.
2. Navigate to **Namespaces** > **Namespaces**.
3. Click on the **Dropdown** button next to **Create Namespace**, then select **Import Namespace**.
4. Refer to the Creating Namespaces documentation for parameter configuration details.
5. Click **Import**.

Importing namespaces by using new web console

The new web console keeps namespace import in the **Applications** view because import is a project-scoped governance action.

When you use that workflow:

- Start from the target project and open **Namespaces**.
- Use **Import Namespace** only for namespaces in clusters already associated with the current project.
- Review the namespace state before import so that monitoring, quota governance, and policy expectations still make sense after association.

- Verify after import that the namespace appears in the current project and remains associated with the expected cluster.

Related guidance

For project-scoped namespace governance, continue with the namespace pages in this section:

- Creating Namespaces
- Updating Namespaces
- Deleting/Removing Namespaces

If you need background information about the product UI entry and scope, see Web Console.

Resource Quota

Refer to the official Kubernetes documentation: Resource Quotas

TOC

Understanding Resource Requests & Limits

Used to restrict resources available to a specific namespace. The total resource usage by all Pods in the namespace (excluding those in a `Terminating` state) must not exceed the quota.

Resource Requests: Define the minimum resources (e.g., CPU, memory) required by a container, guiding the Kubernetes Scheduler to place the Pod on a node with sufficient capacity.

Resource Limits: Define the maximum resources a container can consume, preventing resource exhaustion and ensuring cluster stability.

Quotas

Resource Quotas

> If a resource is marked as `Unlimited`, no explicit quota is enforced, but usage cannot exceed the cluster's available capacity.

Resource Quotas track the cumulative resource consumption (e.g., container limits, new Pods, or PVCs) within a namespace.

Supported Quota Types

Field	Description
Resource Requests	Total requested resources for all Pods in the namespace: • CPU • Memory
Resource Limits	Total limit resources for all Pods in the namespace: • CPU • Memory
Number of Pods	Maximum number of Pods allowed in the namespace.

Note:

- Namespace quotas are derived from the project's allocated cluster resources. If any resource's available quota is 0, namespace creation will fail. Contact the administrator.

- `Unlimited` implies the namespace can consume the project's remaining cluster resources for that resource type.

YAML file example

```yaml
# example-resourcequota.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: example-resourcequota
  namespace: <example>
spec:
  hard:
    limits.cpu: "20"
    limits.memory: 20Gi
    pods: "500"
    requests.cpu: "2"
    requests.memory: 2Gi
```

Create resouce quota by using CLI

Create via YAML file

```
kubectl apply -f example-resourcequota.yaml
```

Create via command line directly

```
kubectl create resourcequota example-resourcequota --namespace=<example>
--hard=limits.cpu=20,limits.memory=20Gi,pods=500
```

Storage Quotas

Quota Type:

- **All**: Total PVC storage capacity in the namespace.
- **Storage Class**: Total PVC storage capacity for a specific storage class.

Note: Ensure the storage class is pre-assigned to the project containing the namespace.

Hardware accelerator Resources Quotas

When an accelerator implementation such as HAMi, NVIDIA GPU Device Plugin, or NPU Operator is installed in the target cluster, ACP can expose corresponding accelerator quota fields through Kubernetes extended resources.

This page owns namespace quota operations. For accelerator resource names, implementation mapping, and product documentation entry points, refer to Accelerator Resource Quota.

Other Quotas

The format for custom quota names must comply with the following specifications:

- If the custom quota name does not contain a slash (/): It must start and end with a letter or number, and can contain letters, numbers, hyphens (-), underscores (_), or periods (.), forming a qualified name with a maximum length of 63 characters.
- If the custom quota name contains a slash (/): The name is divided into two parts: prefix and name, in the form of: prefix/name. The prefix must be a valid DNS subdomain, while the name must comply with the rules for a qualified name.
- DNS Subdomain:
  - Label: Must start and end with lowercase letters or numbers, may contain hyphens (-), but cannot be exclusively composed of hyphens, with a maximum length of 63 characters.
  - Subdomain: Extends the rules of the label, allowing multiple labels to be connected by periods (.) to form a subdomain, with a maximum length of 253 characters.

Limit Range

TOC

Understanding Limit Range

Refer to the official Kubernetes documentation: Limit Ranges

Using Kubernetes LimitRange as an admission controller is **resource limitations at the container or Pod level**. It sets default request values, limit values, and maximum values for containers or Pods created after the LimitRange is created or updated, while continuously monitoring container usage to ensure that no resources exceed the defined maximum values within the namespace.

> The resource request of a container is the ratio between resource limits and cluster overcommitment. Resource request values serve as a reference and criterion for the scheduler when scheduling containers. The scheduler will check the available resources for each node (total resources - sum of resource requests of containers within Pods scheduled

> on the node). If the total resource requests of the new Pod container exceed the remaining available resources of the node, that Pod will not be scheduled on that node.

LimitRange is an admission controller:

- It applies default request and limit values for all Containers that do not set compute resource requirements.
- It tracks usage to ensure it does not exceed resource maximum and ratio defined in any LimitRange present in the namespace.

Includes the following configurations

Resource	Field
CPU	• Default Request • Limit • Max
Memory	• Default Request • Limit • Max

Create Limit Range by using CLI

YAML file examples

```yaml
# example-limitrange.yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: example-limitrange
  namespace: example
spec:
  limits:
    - default:
        cpu: 100m
        memory: 100Mi
      defaultRequest:
        cpu: 50m
        memory: 50Mi
      max:
        cpu: 1000m
        memory: 1000Mi
      type: Container
```

Create via YAML file

```
kubectl apply -f example-limitrange.yaml
```

Create via command line directly

```
kubectl create limitrange example-limitrange --namespace=example --defaul
t='cpu=100m,memory=100Mi' --default-request='cpu=50m,memory=50Mi' --max
='cpu=1000m,memory=1000Mi'
```

Pod Security Policies

ACP supports Kubernetes Pod Security Admission (PSA) and Kyverno Policy to help enforce security standards for Pods running in your clusters.

TOC

Pod Security Admission

Refer to the official Kubernetes documentation: Pod Security Admission ↗

Pod Security Admission (PSA) is a Kubernetes admission controller that enforces security policies at the namespace level by validating Pod specifications against predefined standards.

Security Modes

PSA defines three modes to control how policy violations are handled:

Mode	Behavior	Use Case
Enforce	Denies creation/modification of non-compliant Pods.	Production environments requiring strict security enforcement.
Audit	Allows Pod creation but logs violations in audit logs.	Monitoring and analyzing security incidents without blocking workloads.
Warn	Allows Pod creation but returns client warnings for violations.	Testing environments or transitional phases for policy adjustments.

Key Notes:

- **Enforce** acts on Pods only (e.g., rejects Pods but allows non-Pod resources like Deployments).
- **Audit** and **Warn** apply to both Pods and their controllers (e.g., Deployments).

Security Standards

PSA defines three security standards to restrict Pod privileges:

Standard	Description	Key Restrictions
Privileged	Unrestricted access. Suitable for trusted workloads (e.g., system components).	No validation of `securityContext` fields.
Baseline	Minimal restrictions to prevent known privilege escalations.	Blocks `hostNetwork` , `hostPID` , privileged containers, and unrestricted `hostPath` volumes.

Standard	Description	Key Restrictions
Restricted	Strictest policy enforcing security best practices.	Requires: - `runAsNonRoot: true` - `seccompProfile.type: RuntimeDefault` - Dropped Linux capabilities.

Configuration

Namespace Labels

Apply labels to namespaces to define PSA policies.

YAML file example

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: example-namespace
  labels:
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/audit: baseline
    pod-security.kubernetes.io/warn: baseline
```

CLI command

```
# Step 1: Update Pod Admission labels
kubectl label namespace <namespace-name> \
  pod-security.kubernetes.io/enforce=baseline \
  pod-security.kubernetes.io/audit=restricted \
  --overwrite

# Step 2: Verify labels
kubectl get namespace <namespace-name> --show-labels
```

Exemptions

Exempt specific users, namespaces, or runtime classes from PSA checks.

Example Configuration:

```yaml
apiVersion: pod-security.admission.config.k8s.io/v1
kind: PodSecurityConfiguration
exemptions:
  usernames: ['admin']
  runtimeClasses: ['nvidia']
  namespaces: ['kube-system']
```

Kyverno Policy

ACP provides several samples to create the Kyverno policies for Pod security. The samples includes:

- **Restricted**: Restricted denies access to all host features and requires pods to be run with a UID, and SELinux context that are allocated to the namespace.
- **Restricted-v2**: Restricted-v2 denies access to all host features and requires pods to be run with a UID, and SELinux context that are allocated to the namespace. This is the most restrictive policy and it is used by default for authenticated users. On top of the legacy 'restricted', it also requires to drop ALL capabilities and does not allow privilege escalation binaries. It will also default the seccomp profile to runtime/default if unset, otherwise this seccomp profile is required.
- **Anyuid**: Anyuid provides all features of the restricted policy but allows users to run with any UID and any GID.
- **Hostaccess**: Hostaccess allows access to all host namespaces but still requires pods to be run with a UID and SELinux context that are allocated to the namespace. WARNING: this policy allows host access to namespaces, file systems, and PIDS. It should only be used by trusted pods. Grant with caution.
- **Hostmount-anyuid**: Hostmount-anyuid provides all the features of the restricted policy but allows host mounts and any UID by a pod. This is primarily used by the persistent volume recycler. WARNING: this policy allows host file system access as any UID, including UID 0. Grant with caution.

- **Hostnetwork**: Hostnetwork allows using host networking and host ports but still requires pods to be run with a UID and SELinux context that are allocated to the namespace.
- **Hostnetwork-v2**: Hostnetwork-v2 allows using host networking and host ports but still requires pods to be run with a UID and SELinux context that are allocated to the namespace. On top of the legacy 'hostnetwork' policy, it also requires to drop ALL capabilities and does not allow privilege escalation binaries. It will also default the seccomp profile to runtime/default if unset, otherwise this seccomp profile is required.
- **Node-exporter**: Node-exporter policy is used for the Prometheus node exporter.
- **Nonroot**: Nonroot provides all features of the restricted policy but allows users to run with any non-root UID. The user must specify the UID or it must be specified on the by the manifest of the container runtime.
- **Nonroot-v2**: Nonroot-v2 provides all features of the restricted policy but allows users to run with any non-root UID. The user must specify the UID or it must be specified on the by the manifest of the container runtime. On top of the legacy 'nonroot' policy, it also requires to drop ALL capabilities and does not allow privilege escalation binaries. It will also default the seccomp profile to runtime/default if unset, otherwise this seccomp profile is required.
- **Privileged**: Privileged allows access to all privileged and host features and the ability to run as any user, any group, any fsGroup, and with any SELinux context. WARNING: this is the most relaxed and should be used only for cluster administration. Grant with caution.

> **NOTICE**
>
> The **Restricted** policy is not equal to the Kubernetes Pod Security Admission 'restricted' standard. You may need to change your pod security configuration if you want to use the kyverno **Restricted** policy instead of Kubernetes Pod Security Admission 'restricted' standard.

Prerequisites

- Install Alauda Container Platform Compliance for Kyverno, refer to the document.
- Enable feature gate `namespace-resource-manage` in ACP featuregate settings.

Apply Kyverno Policies

Web Console

1. In the ACP console, navigate to **Container Platform**, choose the namespace where you want to apply the security policy.
2. Go to **Advanced** > **Resources**.
3. Search the resource type with name **Policy** for resource **Policy** of group **kyverno.io**.
4. Select the version "v2beta1", Click **Create** to create a new Kyverno Policy.
5. In the **Create Resource** dialog, select the **Samples** tab.
6. Choose the desired Pod security policy sample (e.g., `Restricted`), then click **Try**.
7. Review and modify the policy YAML as needed, then click **Update** to apply the policy.

CLI

1. Log in to the kubernetes cluster where you want to apply the security policy.
2. Run the following command to create a Kyverno Policy from a sample resource:

```
$ kubectl get consoleyamlsamples.console.alauda.io restricted-policy -o
template --template={{.spec.yaml}}|kubectl apply -f -
$ kubectl get policies.kyverno.io
NAME          ADMISSION   BACKGROUND   READY   AGE   MESSAGE
restricted    true        true         True    1m    Ready
```

The sample resources available are:

- restricted-policy
- restrictedv2-policy
- anyuid-policy
- hostaccess-policy
- hostmount-anyuid-policy
- hostnetwork-policy
- hostnetwork-v2-policy
- node-exporter-policy
- nonroot-policy

- nonroot-v2-policy
- privileged-policy

UID/GID Assignment

In Kubernetes, each Pod runs with a specific User ID (UID) and Group ID (GID) to ensure security and proper access control. By default, Pods may run as the root user (UID 0), which can pose security risks. To enhance security, it's recommended to assign non-root UIDs and GIDs to Pods.

ACP allows to auto assign a namespace with specific UID and GID ranges to ensure that all Pods within the namespace run with the designated user and group IDs.

TOC

Enable UID/GID Assignment

To enable UID/GID assignment for a namespace, follow these steps:

1. Enter **Project Management**.
2. In the left navigation bar, click **Namespace**.
3. Click on the target namespace.

4. Click **Actions** > **Upate Pod Security Policy**.
5. Change the **Enforce** option value to **Restricted**, click **Update**.
6. Click edit icon next to **Labels**, add a label with key `security.cpaas.io/enabled` and value `true`, click **Update**. (To disable, remove this label or set the value to `false`.)
7. Click **Save**.

Verify UID/GID Assignment

The UID/GID Range

In the namespace details page, you can view the assigned UID and GID ranges in the **Annotations**.

The **security.cpaas.io/uid-range** annotation specifies the range of UID/GIDs that can be assigned to Pods in the namespace, e.g. **security.cpaas.io/uid-range=1000002000-1000011999**, means the uid/gid range is between 1000002000 to 1000011999.

Verify the Pod UID/GID

If the pod does not specify `runAsUser` and `fsGroup` in the `securityContext`, the platform will automatically assign the first value from the assigned uid range.

1. Create a Pod in the namespace with the following YAML configuration:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: uid-gid-test-pod
spec:
  containers:
  - name: test-container
    image: busybox
    command: ["sleep", "3600"]
```

2. After the Pod is created, get the Pod yaml to check the assigned UID and GID:

```
kubectl get pod uid-gid-test-pod -n <namespace-name> -o yaml
```

the Pod YAML will show the assigned UID and GID in the `securityContext` section:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: uid-gid-test-pod
spec:
  containers:
  - name: test-container
    image: busybox
    command: ["sleep", "3600"]
    securityContext:
      runAsUser: 1000000
  securityContext:
    fsGroup: 1000000
```

If the pod specifies runAsUser and fsGroup in the securityContext, the platform will validate if the specified UID/GID are within the assigned range. If they are not, the Pod creation will fail.

1. Create a Pod in the namespace with the following YAML configuration:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: uid-gid-test-pod-invalid
spec:
  containers:
  - name: test-container
    image: busybox
    command: ["sleep", "3600"]
    securityContext:
      runAsUser: 2000000  # Invalid UID, outside the assigned range
  securityContext:
    fsGroup: 2000000  # Invalid GID, outside the assigned range
```

2. After applying the YAML, the Pod creation will fail with an error message indicating that the specified UID/GID are outside the assigned range.

Overcommit Ratio

TOC

Understanding Namespace Resource Overcommit Ratio

Alauda Container Platform allows you to set a resource overcommit ratio (CPU and memory) per namespace. This manages the relationship between container limits (maximum usage) and requests (guaranteed minimum) within that namespace, optimizing resource utilization.

By configuring this ratio, you ensure user-defined container limits and requests remain within reasonable bounds, improving overall cluster resource efficiency.

Key Concepts

- Limits: The maximum resource a container can use. Exceeding limits can lead to throttling (CPU) or termination (memory).
- Requests: The guaranteed minimum resource a container needs. Kubernetes schedules containers based on these requests.
- Overcommit Ratio: Limits / Requests. This setting defines the acceptable range for this ratio within the namespace, balancing resource guarantees and preventing over-consumption.

Core Capabilities

- Enhance resource density and application stability within the namespace by setting an appropriate overcommit ratio to manage the balance between resource limits and requests.

Example

> Assuming the namespace overcommit ratio is set to 2, when creating an application and specifies a CPU limit of 4c, the corresponding CPU request value is calculated as:

CPU Request = CPU Limit / Overcommit ratio. Thus, the CPU request becomes 4c / 2 = 2c.

CRD Define

```yaml
# example-namespace-overcommit.yaml
apiVersion: resource.alauda.io/v1
kind: NamespaceResourceRatio
metadata:
  namespace: example
  name: example-namespace-overcommit
spec:
  cpu: 3 # Absence of this field indicates inheritance of the cluster overcommit ratio; 0 means no limitation.
  memory: 4 # Absence of this field indicates inheritance of the cluster overcommit ratio; 0 means no limitation.
status:
  clusterCPU: 2 # Cluster Overcommit Ratio
  clusterMemory: 3
```

Creating overcommit ratio by using CLI

```
kubectl apply -f example-namespace-overcommit.yaml
```

Creating/Updating Overcommit Ratio by using web console

Allows adjusting the **overcommit ratio** for a namespace to manage the ratio between resource limits and requests. This ensures container's resource allocations remain within defined bounds, improving cluster resource utilization.

Precautions

If the cluster uses node virtualization (e.g., virtual nodes), disable oversubscription at the cluster/namespace level before configuring it for virtual machines.

Procedure

1. Enter the **Project Management** and navigation to **Namespaces** > **Namespace** List.
2. Click the target ***Namespace name***.
3. Click **Actions** > **Update Overcommit**.
4. Select the appropriate overcommit ratio **configuration method** to set the CPU or memory overcommit ratio for the namespace.

Parameter	Description
Inherit from Cluster	• Namespace inherits the cluster's oversubscription ratio. • Example: If cluster CPU/memory ratio is 4, namespace defaults to 4. • Container requests = limit / cluster ratio. • If no limit is set, use the namespace's default container quota.
Custom	• Set a namespace-specific ratio (integer > 1). • Example: Cluster ratio = 4, namespace ratio = 2 → requests = limit / 2. • Leave empty to disable oversubscription for the namespace.

5. Click **Update**.

Note: Changes apply only to newly created Pods. Existing Pods retain their original requests until rebuilt.

Updating overcommit ratio by using new web console

This setting belongs to namespace governance, so the entry depends on whether you are operating from a project context or a concrete cluster context.

Update from the Applications view

Use the **Applications** view when the namespace is already associated with the current project and you want to adjust request-to-limit governance in that project scope.

Before you update the ratio, confirm that:

- the namespace belongs to the current project
- the new CPU and memory ratios still match the workloads that will be created or recreated later in that namespace
- the change is evaluated together with quota, default limits, and scheduling expectations in the current project

Managing Namespace Members

TOC

Importing Members

The platform supports bulk importing members into a namespace and assigning roles such as Namespace Administrator or Developer to grant corresponding permissions.

Constraints and Limitations

- Members can only be imported into the namespace from the **Project Members** of the namespace's project.

- The platform does not support importing default system-created admin users or the active user.

Prerequisites

To import users as namespace members, they must first be added to the namespace's project.

Procedure

1. **Project Management**, click on ***Project Name*** where the members to be imported are located.
2. Navigation to **Namespaces** > **Namespaces**.
3. Click on ***Namespace Name*** of the members to be imported.
4. In the **Namespace Members** tab, click **Import Members**.
5. Follow the procedures below to import all or some users from the list into the namespace.

   > **TIP**
   >
   > You can select a user group using the dropdown box at the top right of the dialog and perform a fuzzy search by entering the username in the username search box.

   - Import all users in the list as namespace members and assign roles to users in bulk.
     1. Click the dropdown on the right side of the **Set Role** item at the bottom of the dialog, and select the role name to assign.
     2. Click **Import All**.
   - Import one or more users from the list as namespace members.
     1. Click the checkbox in front of the username/display name to select one or more users.
     2. Click the dropdown on the right side of the **Set Role** item at the bottom of the dialog, and select the role name to assign to the selected users.
     3. Click **Import**.

Adding Members

When the platform has added an OICD type IDP, OIDC users can be added as namespace members.

You can add valid OIDC accounts that meet the input requirements as namespace roles and assign the corresponding namespace roles to the user.

Note: When adding members, the system does not verify the validity of the accounts. Please ensure that the accounts you add are valid; otherwise, these accounts will not be able to log in to the platform successfully.

Valid OIDC accounts include: Valid accounts in the OIDC identity authentication service configured via IDP for the platform, including those that have successfully logged in to the platform and those that have not logged in to the platform.

Prerequisites

The platform has added an **OICD** type IDP.

Procedure

1. **Project Management**, click on ***Project Name*** where the member to be added is located.
2. Navigation to **Namespaces** > **Namespaces**.
3. Click on ***Namespace Name*** of the member to be added.
4. In the **Namespace Members** tab, click **Add Member**.
5. In the **Username** input box, enter a username for an existing third-party platform account supported by the platform.
   Note: Please confirm that the entered username corresponds to an existing account on the third-party platform; otherwise, that account will not be able to log in to this platform successfully.
6. In the **Role** dropdown, select the role name to configure for this user.
7. Click **Add**. After a successful addition, you can view the member in the namespace member list. At the same time, in the user list (**Platform Management > User Management**), you can view that user. Before the user successfully logs in or is synchronized to this platform, the source will be `-`, and it can be deleted; when the

account successfully logs in or synchronizes to the platform, the platform will record the account's source information and display it in the user list.

Removing Members

Remove specified namespace members and delete their associated roles to revoke their namespace permissions.

Procedure

1. **Project Management**, click on ***Project Name*** where the member to be removed is located.
2. Navigation to **Namespaces** > **Namespaces**.
3. Click on ***Namespace Name*** of the member to be removed.
4. In the **Namespace Members** tab, click ⋮ on the right side of the record of the member to be removed > **Remove**.
5. Click **Remove**.

Managing namespace members by using new web console

This page mainly describes the legacy console workflow, but namespace member management in the new web console is still a project-scoped action.

Use the **Applications** view when you need to manage member access for a namespace that already belongs to the current project.

When you manage namespace members from that view, confirm that:

- the namespace is already associated with the current project
- the member source still matches the project-level ownership model because imports come from project members
- the role change affects namespace access only and does not replace project-level membership governance

- the resulting namespace member list still matches the intended operational ownership of the workloads in that namespace

Updating Namespaces

TOC

Updating Quotas

> Resource Quota

Updating a Resource Quota by using web console

1. **Project Management**, and navigate to **Namespaces** > **Namespace** List in the left sidebar.
2. Click the target ***namespace name***.
3. Click **Actions** > **Update Quota**.
4. Adjust resource quotas (CPU, Memory, Pods, etc.) and click **Update**.

Updating a Resource Quota by using new web console

Use the **Applications** view when you are adjusting namespace governance inside the current project scope.

Entry path and action order:

- Open the current project, then go to **Applications** > **Namespaces**.
- Click the target namespace.
- Use **Actions** to choose **Update Quota**.

When you update the namespace quota from this view, confirm that:

- the change belongs to a namespace already associated with the current project
- the action order is namespace selection first, then the specific governance action, because cluster and namespace identity stay fixed during update
- the cluster and namespace identity remain fixed while quota values are updated
- the change still fits the project's intended resource governance model

Updating a Resource Quota by using CLI

> Resource Quota YAML file example

```
# Step 1: Edit the namespace quota
kubectl edit resourcequota <quota-name> -n <namespace-name>

# Step 2: Verify changes
kubectl get resourcequota <quota-name> -n <namespace-name> -o yaml
```

When you update namespace quota, review whether the remaining project or cluster capacity still supports later workload creation, scheduling, and scaling. Tighter quotas often affect the next rollout or scale-out attempt before they affect already running Pods.

Updating Container LimitRanges

> Limit Range

Updating a LimitRange by using web console

1. **Project Management** view, and navigate to **Namespaces** > **Namespace** List in the left sidebar.
2. Click the target ***namespace name***.
3. Click **Actions** > **Update Container LimitRange**.
4. Adjust container limit range (`defaultRequest` , `default` , `max`) and click **Update**.

Updating a LimitRange by using new web console

Use the **Applications** view when you are adjusting namespace governance inside the current project scope.

Entry path and action order:

- Open the current project, then go to **Applications** > **Namespaces**.
- Click the target namespace.
- Use **Actions** to choose **Update Container LimitRange**.

When you update the namespace limit range from this view, confirm that:

- the change belongs to a namespace already associated with the current project
- the action order is namespace selection first, then the specific governance action, because cluster and namespace identity stay fixed during update
- the cluster and namespace identity remain fixed while limit range values are updated
- the change still fits the project's intended resource governance model

Updating a LimitRange by using CLI

> Limit Range YAML file example

```
# Step 1: Edit the LimitRange
kubectl edit limitrange <limitrange-name> -n <namespace-name>

# Step 2: Verify changes
kubectl get limitrange <limitrange-name> -n <namespace-name> -o yaml
```

Container limit range changes usually matter most for workloads created or recreated after the update. Review how new default requests, default limits, and maximum values will affect later workload forms and admission behavior.

Updating Pod Security Admission

> Pod Security Admission

Updating a Pod Security Admission by using web console

1. **Project Management** view, and navigate to **Namespaces** > **Namespace** List in the left sidebar.
2. Click the target ***namespace name***.
3. Click **Actions** > **Update Pod Security Admission**.
4. Adjust security standard (`enforce` , `audit` , `warn`) and click **Update**.

Updating a Pod Security Admission by using new web console

Use the **Applications** view when you are adjusting namespace governance inside the current project scope.

Entry path and action order:

- Open the current project, then go to **Applications** > **Namespaces**.
- Click the target namespace.
- Use **Actions** to choose **Update Pod Security Admission**.

When you update the namespace policy from this view, confirm that:

- the change belongs to a namespace already associated with the current project
- the action order is namespace selection first, then the specific governance action, because cluster and namespace identity stay fixed during update
- the cluster and namespace identity remain fixed while policy values are updated
- the change still fits the project's intended resource and security governance model

Updating a Pod Security Admission by using CLI

> Update Pod Security Admission CLI command

Stricter namespace-level security settings can block Pods that rely on elevated privileges, host access, or incompatible security contexts. Review expected workload requirements before tightening the policy.

Updating a namespace from the Clusters view in the new web console

This page mainly describes the legacy console workflow, but namespace updates in the new web console can start from different views.

Update from the Clusters view

Use the **Clusters** view when you are adjusting namespace governance from a concrete cluster.

Entry path and action order:

- Open the target cluster first.
- Go to **Namespaces** in that cluster context and open the target namespace.
- Use **Actions** to choose **Update**, **Delete**

You can manage the corresponding resources under the namespace in the **Resource Quota** Tab and **LimitRange** Tab on the details page.

Deleting/Removing Namespaces

You can either delete a namespace permanently or remove it from the current project.

TOC

Deleting Namespaces

Delete Namespace: Permanently deletes a namespace and all resources within it (e.g., Pods, Services, ConfigMaps). This action cannot be undone and releases allocated resource quotas.

```
kubectl delete namespace <namespace-name>
```

Removing Namespaces

Remove Namespace: Removing a namespace from the current project without deleting its resources. The namespace remains in the cluster and can be imported into other projects via Import Namespace.

> **NOTE**
>
> - This feature is exclusive to the Alauda Container Platform.
> - Kubernetes does not natively support "removing" namespaces from projects.

```
kubectl label namespace <namespace-name> cpaas.io/project- --overwrite
```

How to choose between remove and delete

Use **Remove Namespace** when the namespace resource should remain in the cluster but should no longer belong to the current project.

Use **Delete Namespace** when the namespace resource itself is no longer needed and the workloads and namespaced resources inside it can be removed.

Before deleting a namespace, verify that workloads, ConfigMaps, Secrets, storage usage, and any remaining troubleshooting needs are no longer required in that namespace.

Removing or deleting namespaces by using new web console

This page mainly describes the legacy console workflow, but namespace removal or deletion in the new web console can start from different views.

Remove from the Applications view

Use the **Applications** view when you need to remove the namespace from the current project without necessarily deleting the namespace resource itself.

Entry path and action order:

- Open the current project, then go to **Applications** > **Namespaces**.
- Open the target namespace or use its **Actions** menu.
- Choose **Remove Namespace** when the intent is to end project association while keeping the namespace resource in the cluster.
- Choose **Delete Namespace** only when the namespace resource itself must be deleted and the environment exposes that action in the current view.

Delete from the Clusters view

Use the **Clusters** view when you need to delete the namespace resource from a concrete cluster.

Entry path and action order:

- Open the target cluster first.
- Go to **Namespaces** in that cluster context and select the target namespace.
- Choose **Delete Namespace** from the available action entry and confirm the deletion.

Creating Applications

Creating applications from Image

TOC

Prerequisites

Obtain the image address. The source of the images can be from the image repository integrated by the platform administrator through the toolchain or from third-party platforms' image repositories.

- For the former, the Administrator typically assigns the image repository to your project, and you can use the images within it. If the required image repository is not found, please contact the Administrator for allocation.
- If it is a third-party platform's image repository, ensure that images can be pulled directly from it in the current cluster.
- When selecting an image through an OCI Connector, also select an image pull Secret for the Connector proxy, or confirm that the workload ServiceAccount already references that Secret. See Use an ImagePullSecret with an OCI Connector.

Procedure 1 - Workloads

1. **Container Platform**, navigate to **Applications** > **Applications** in the left sidebar.
2. Click **Create**.
3. Choose **Create from Image** as the creation approach.
4. **Select** or **Input** an image, and click **Confirm**.

> **INFO**
>
> **Note**: When using images from the image repository integrated into web console, you can filter images by **Already Integrated**. The **Integration Project Name**, for example, images (registry-projectname), which includes the project name projectname in this web console and the project name containers in the image repository.
>
> When using images from a private registry, you need to configure the corresponding image pull secret. For more information, see Add ImagePullSecrets to ServiceAccount.
>
> When using an OCI Connector, use a Connector proxy image pull Secret directly from the Pod template or through the workload ServiceAccount. The Secret must authenticate to the proxy, not the original registry.

6. Refer to the following instructions to configure the related parameters.

Workload 1 - Configure Basic Info

In the **Workload** > **Basic Info** section, configure declarative parameters for workloads

Parameters	Description
Model	Select a workload as needed: • **Deployment**: For detailed parameter descriptions, please refer to Creating Deployment. • **DaemonSet**: For detailed parameter descriptions, please refer to Creating DaemonSet. • **StatefulSet**: For detailed parameter descriptions, please refer to Creating StatefulSet.
Replicas	Defines the desired number of Pod replicas in the Deployment (default: `1`). Adjust based on workload requirements.
More > **Update Strategy**	Configures the `rollingUpdate` strategy for zero-downtime deployments: **Max surge** (`maxSurge`): • Maximum number of Pods that can exceed the desired replica count during an update. • Accepts absolute values (e.g., `2`) or percentages (e.g., `20%`). • Percentage calculation: `ceil(current_replicas × percentage)` . • Example: 4.1 → `5` when calculated from 10 replicas. **Max unavailable** (`maxUnavailable`): • Maximum number of Pods that can be temporarily unavailable during an update. • Percentage values cannot exceed `100%` . • Percentage calculation: `floor(current_replicas × percentage)` . • Example: 4.9 → `4` when calculated from 10 replicas. **Notes**: 1. **Default values**: `maxSurge=1` , `maxUnavailable=1` if not explicitly set. 2. **Non-running Pods** (e.g., in `Pending` / `CrashLoopBackOff` states) are considered unavailable. 3. **Simultaneous constraints**: • `maxSurge` and `maxUnavailable` cannot both be `0` or `0%` .

Parameters	Description
	• If percentage values resolve to `0` for both parameters, Kubernetes forces `maxUnavailable=1` to ensure update progress. **Example**: For a Deployment with 10 replicas: • `maxSurge=2` → Total Pods during update: `10 + 2 = 12`. • `maxUnavailable=3` → Minimum available Pods: `10 - 3 = 7`. • This ensures availability while allowing controlled rollout.

Workload 2 - Configure Pod

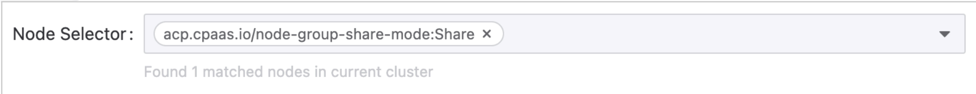
Note: In mixed-architecture clusters deploying single-architecture images, ensure proper Node Affinity Rules are configured for Pod scheduling.

1. **Pod** section, configure container runtime parameters and lifecycle management:

Parameters	Description
Volumes	Mount persistent volumes to containers. Supported volume types include `PVC`, `ConfigMap`, `Secret`, `emptyDir`, `hostPath`, and so on. For implementation details, see Storage Volume Mounting Instructions.
Image Credential	For a private registry image, use the Secret that contains credentials for that registry. For an OCI Connector image, select the Connector proxy image pull Secret or use a ServiceAccount that already references it. Images from the platform's integrated registry continue to inherit associated credentials. See Use an ImagePullSecret with an OCI Connector.
More > **Close Grace Period**	Duration (default: `30s`) allowed for a Pod to complete graceful shutdown after receiving termination signal. - During this period, the Pod completes inflight requests and releases resources.

Parameters	Description
	- Setting `0` forces immediate deletion (SIGKILL), which may cause request interruptions.

2. **Node Affinity Rules**

Parameters	Description
More > **Node Selector**	Constrain Pods to nodes with specific labels (e.g., `kubernetes.io/os: linux`). 
More > **Affinity**	Define fine-grained scheduling rules based on existing Pods. **Pod Affinity Types**: • **Inter-Pod Affinity**: Schedule new Pods to nodes hosting specific Pods (same topology domain). • **Inter-Pod Anti-affinity**: Prevent co-location of new Pods with specific Pods. **Enforcement Modes**: • **RequiredDuringSchedulingIgnoredDuringExecution**: Pods are scheduled *only* if rules are satisfied. • **PreferredDuringSchedulingIgnoredDuringExecution**: Prioritize nodes meeting rules, but allow exceptions. **Configuration Fields**: • `topologyKey` : Node label defining topology domains (default: `kubernetes.io/hostname`). • `labelSelector` : Filters target Pods using label queries.

3. **Network Configuration**

- Kube-OVN

Parameters	Description
Bandwidth Limits	Enforce QoS for Pod network traffic: • **Egress rate limit**: Maximum outbound traffic rate (e.g., `10Mbps`). • **Ingress rate limit**: Maximum inbound traffic rate.
Subnet	Assign IPs from a predefined subnet pool. If unspecified, uses the namespace's default subnet.
Static IP Address	Bind persistent IP addresses to Pods: • Multiple Pods across Deployments can claim the same IP, but only one Pod can use it concurrently. • **Critical**: Number of static IPs must ≥ Pod replica count.

- Calico

Parameters	Description
Static IP Address	Assign fixed IPs with strict uniqueness: • Each IP can be bound to **only one Pod** in the cluster. • **Critical**: Static IP count must ≥ Pod replica count.

Workload 3 - Configure Containers

1. **Container** section, refer to the following instructions to configure the relevant information.

Parameters	Description
Resource Requests & Limits	• **Requests**: Minimum CPU/memory required for container operation.

Parameters	Description
	• **Limits**: Maximum CPU/memory allowed during container execution. For unit definitions, see Resource Units. **Namespace overcommit ratio**: • **Without overcommit ratio**: If namespace resource quotas exist: Container requests/limits inherit namespace defaults (modifiable). No namespace quotas: No defaults; custom Request. • **With overcommit ratio**: Requests auto-calculated as `Limits / Overcommit ratio` (immutable). **Constraints**: • Request ≤ Limit ≤ Namespace quota maximum. • Overcommit ratio changes require pod recreation to take effect. • Overcommit ratio disables manual request configuration. • No namespace quotas → no container resource constraints.
Extended Resources	Configure cluster-available extended resources (e.g., vGPU, pGPU).
Volume Mount	Persistent storage configuration. See Storage Volume Mounting Instructions. **Operations**: • Existing pod volumes: Click **Add** • No pod volumes: Click **Add & Mount** **Parameters**: • `mountPath`: Container filesystem path (e.g., `/data`) • `subPath`: Relative file/directory path within volume. For `ConfigMap` / `Secret`: Select specific key • `readOnly`: Mount as read-only (default: read-write)

Parameters	Description
	See Kubernetes Volumes.
Port	Expose container ports. **Example**: Expose TCP port `6379` with name `redis`. **Fields**: • `protocol`: TCP/UDP • `Port`: Exposed port (e.g., `6379`) • `name`: DNS-compliant identifier (e.g., `redis`)
Startup Commands & Arguments	Override default ENTRYPOINT/CMD: **Example 1**: Execute `top -b` - **Command**: `["top", "-b"]` - **OR** Command: `["top"]`, Args: `["-b"]` **Example 2**: Output `$MESSAGE`: `/bin/sh -c "while true; do echo $(MESSAGE); sleep 10; done"` See Defining Commands.
More > **Environment Variables**	• Static values: Direct key-value pairs • Dynamic values: Reference ConfigMap/Secret keys, pod fields (`fieldRef`), resource metrics (`resourceFieldRef`) **Note**: Env variables override image/configuration file settings.
More > **Referenced ConfigMap**	Inject entire ConfigMap/Secret as env variables. Supported Secret types: `Opaque`, `kubernetes.io/basic-auth`.

Parameters	Description
More > **Health Checks**	• **Liveness Probe**: Detect container health (restart if failing) • **Readiness Probe**: Detect service availability (remove from endpoints if failing) See Health Check Parameters.
More > **Log File**	Configure log paths: - Default: Collect `stdout` - File patterns: e.g., `/var/log/*.log` **Requirements**: • Storage driver `overlay2`: Supported by default • `devicemapper`: Manually mount EmptyDir to log directory • Windows nodes: Ensure parent directory is mounted (e.g., `c:/a` for `c:/a/b/c/*.log`)
More > **Exclude Log File**	Exclude specific logs from collection (e.g., `/var/log/aaa.log`).
More > **Execute before Stopping**	Execute commands before container termination. **Example**: `echo "stop"` **Note**: Command execution time must be shorter than pod's `terminationGracePeriodSeconds`.

2. Click **Add Container** (upper right) OR **Add Init Container**.
   See Init Containers. Init Container:

   1. Start before app containers (sequential execution).
   2. Release resources after completion.
   3. Deletion allowed when:

      - Pod has >1 app container AND ≥1 init container.
      - Not allowed for single-app-container pods.

3. Click **Create**.

Procedure 2 - Services

Parameters	Description
Service	Kubernetes **Service**, expose an application running in your cluster behind a single outward-facing endpoint, even when the workload is split across multiple backends.. For specific parameter explanations, please refer to Creating Services. **Note** The default name prefix for the internal routing created under the application is the name of the compute component. If the compute component type (deployment mode) is StatefulSet, it is advisable not to change the default name of the internal routing (the name of the workload); otherwise, it may lead to accessibility issues for the workload.

Procedure 3 - Ingress

Parameters	Description
Ingress	Kubernetes **Ingress**, make your HTTP (or HTTPS) network service available using a protocol-aware configuration mechanism, that understands web concepts like URIs, hostnames, paths, and more. The Ingress concept lets you map traffic to different backends based on rules you define via the Kubernetes API. For detailed parameter descriptions, please refer to Creating Ingresses. **Note**: The **Service** used when creating **Ingress** under the application must be resources created under the current application. However, ensure that the **Service** is associated with the workload under the application; otherwise, service discovery and access for workload will fail.

7. Click **Create**.

Application Management Operations

To modify application configurations, use one of the following methods:

1. Click the vertical ellipsis (⋮) on the right side of the application list.
2. Select **Actions** from the upper-right corner of the application details page.

<table>
<tr><th>Operation</th><th>Description</th></tr>
<tr><td>Update</td><td>

Update: Modifies only the target workload using its defined update strategy (Deployment strategy shown as example). Preserves existing replica count and rollout configuration.
Force Update: Triggers full application rollout using each component's update strategy.
1. Use cases:

Batch configuration changes requiring immediate cluster-wide propagation (e.g., ConfigMap/Secret updates referenced as environment variables).
Coordinated component restarts for critical security.

2. Warning Caution:

May cause temporary service degradation during mass restarts.
Not recommended for production environments without business continuity validation.

Network Implications:

Ingress Rule Deletion: External access remains available via <code>LB_IP:NodePort</code> if:
1) LoadBalancer Service uses default ports.
2) Surviving routing rules reference application components.
Full external access termination requires Service deletion.
Service Deletion: Irreversible loss of network connectivity to application components. Associated Ingress rules become non-functional despite API object persistence.

</td></tr>
</table>

Operation	Description
Delete	• **Cascading Deletion**: 1. Removes all child resources including Deployments, Services, and Ingress rules. 2. Persistent Volume Claims (PVCs) follow retention policy defined in StorageClass • **Pre-deletion Checklist**: 1. Verify no active traffic through associated Services. 2. Confirm data backup completion for stateful components. 3. Check dependent resource relationships using `kubectl describe ownerReferences`.

When updating a component image or OCI Connector, also review that component Pod template's image pull Secret or ServiceAccount and the token validity. Verify the Connector annotation, actual rewritten proxy image, Pod rollout status, and Events after the update.

Reference Information

Storage Volume Mounting Instructions

Type	Purpose
Persistent Volume Claim	Binds an existing PVC to request persistent storage. **Note**: Only bound PVCs (with associated PV) are selectable. Unbound PVCs will cause pod creation failures.
ConfigMap	Mounts full/partial ConfigMap data as files: • Full ConfigMap: Creates files named after keys under mount path • Subpath selection: Mount specific key (e.g., `my.cnf`)

Type	Purpose
Secret	Mounts full/partial Secret data as files: • Full Secret: Creates files named after keys under mount path • Subpath selection: Mount specific key (e.g., `tls.crt`)
Ephemeral Volumes	Cluster-provisioned temporary volume with features: • Dynamic provisioning • Lifecycle tied to pod • Supports declarative configuration **Use Case**: Temporary data storage. See Ephemeral Volumes
Empty Directory	Ephemeral storage sharing between containers in same pod: - Created on node when pod starts - Deleted with pod removal **Use Case**: Inter-container file sharing, temporary data storage. See EmptyDir
Host Path	Mounts host machine directory (must start with `/`, e.g., `/volumepath`).

Health Check Parameters

Common Parameters

Parameters	Description
Initial Delay	Grace period (seconds) before starting probes. Default: `300`.
Period	Probe interval (1-120s). Default: `60`.
Timeout	Probe timeout duration (1-300s). Default: `30`.

Parameters	Description
Success Threshold	Minimum consecutive successes to mark healthy. Default: `0`.
Failure Threshold	Maximum consecutive failures to trigger action: - `0`: Disables failure-based actions - Default: `5` failures → container restart.

Protocol-Specific Parameters

Parameter	Applicable Protocols	Description
Protocol	HTTP/HTTPS	Health check protocol
Port	HTTP/HTTPS/TCP	Target container port for probing.
Path	HTTP/HTTPS	Endpoint path (e.g., `/healthz`).
HTTP Headers	HTTP/HTTPS	Custom headers (Add key-value pairs).
Command	EXEC	Container-executable check command (e.g., `sh -c "curl -I localhost:8080 | grep OK"`). **Note**: Escape special characters and test command viability.

Creating applications from Chart

Based on Helm Chart represents a native application deployment pattern. A Helm Chart is a collection of files that define Kubernetes resources, designed to package applications and facilitate application distribution with version control capabilities. This enables seamless environment transitions, such as migrations from development to production environments.

TOC

Precautions

When a cluster contains both Linux and Windows nodes, explicit node selection MUST be configured to prevent scheduling conflicts. Example:

```
spec:
    spec:
      nodeSelector:
        kubernetes.io/os: linux
```

Prerequisites

If the template is from a application and references relevant resources (e.g., secret dictionaries), ensure the to-be-referenced resources already exist in the current namespace before application deployment.

Procedure

1. **Container Platform**, navigate to **Applications** > **Applications** in the left sidebar.
2. Click **Create**.
3. Choose **Create from Catalog** as the creation approach.
4. Select a Chart and configure parameters, pick a Chart and configure the required parameters, such as `resources.requests`, `resources.limits`, and other parameters that are closely related to the chart.
5. Click **Create**.

The web console will redirect you to the **Application** > [**Native Applications**] details page. The process will take some time, so please be patient. In case of operation failure, follow the interface prompts to complete the operation.

Status Analysis Reference

Click on ***Application Name*** to display detailed status analysis of the Chart in the details information.

Type	Reason
Initialized	Indicates the status of Chart template download. • **True**: It indicates that the Chart template has been successfully downloaded. • **False**: It indicates that the Chart template download has failed; you can check the specific failure reason in the message column. ◦ `ChartLoadFailed`: Chart template download failed.

<table>
<tr><th>Type</th><th>Reason</th></tr>
<tr><td></td><td><code>InitializeFailed</code> : There was an exception in the initialization process before the Chart was downloaded.</td></tr>
<tr><td>Validated</td><td>Indicates the status of user permissions, dependencies, and other validations for the Chart template.
True: It indicates that all validation checks have passed.
False: It indicates that there are validation checks that have not passed; you can check the specific failure reason in the message column.
<code>DependenciesCheckFailed</code> : Chart dependency check failed.
<code>PermissionCheckFailed</code> : The current user lacks permission to perform operations on certain resources.
<code>ConsistentNamespaceCheckFailed</code> : When deploying applications through templates in native applications, the Chart contains resources that require cross-namespace deployment.</td></tr>
<tr><td>Synced</td><td>Indicates the deployment status of the Chart template.
True: It indicates that the Chart template has been successfully deployed.
False: It indicates that the Chart template deployment has failed; the reason column shows <code>ChartSyncFailed</code> , and you can check the specific failure reason in the message column.</td></tr>
</table>

WARNING

- If the template references cross - namespace resources, contact the Administrator for help with creation. Afterward, you can normally Updating and deleting Chart Applications on web console.
- If the template references cluster - level resources (e.g., StorageClasses), it's recommended to contact the Administrator for assistance with creation.

Creating applications from YAML

If you are proficient in YAML syntax and prefer to define configurations outside of forms or pre-defined templates, you can create applications from YAML. This approach provides more flexible control over application configuration and referenced resources.

TOC

Precautions

When both Linux and Windows nodes exist in the cluster, consider configuring node selection so that the workload is scheduled only onto compatible nodes. For example:

```yaml
spec:
    spec:
      nodeSelector:
        kubernetes.io/os: linux
```

Prerequisites

Ensure the images defined in the YAML can be pulled within the current cluster. You can verify this using the `podman pull` command.

Procedure

1. **Container Platform**, and navigate to **Application** > **Applications**.
2. Click **Create**.
3. Select **Create from YAML**.
4. Complete the configuration and click **Create**.
5. View the corresponding **Deployment** on the details page.

Creation entry differences between the Applications and Clusters views

The new web console can expose YAML-based creation from different views. Choose the entry based on whether you are working from project scope or from a concrete cluster.

- **Applications view**: Start from the current project. Open **Applications** > **Applications**, click **Create from YAML**, select the target **Cluster** first, review or confirm the target **Namespace** from the YAML content, and then submit the resources.
- **Clusters view**: Start from a concrete cluster. Open the target cluster first, enter the target namespace, and then navigate to **Applications** or the related resource list to open the YAML creation workflow in that fixed cluster context.

Before proceeding, confirm the following differences:

- In the **Applications** view, cluster selection is part of the project-scoped application workflow.
- In the **Clusters** view, the cluster is already fixed because you start inside a concrete cluster.
- In the **Applications** view, the main order is to select the cluster first and then confirm the namespace declared in the YAML.
- In the **Clusters** view, the main order is to enter the concrete cluster first and then work in the target namespace context.

Additional guidance for the Applications view

If your environment exposes YAML creation in the **Applications** view, use the same YAML creation path and review the following Application-view-specific behavior.

1. Open the application page in the **Applications** view.
2. Click **Create from YAML**.
3. Select the target **Cluster**.
4. Review the namespace declared in the YAML metadata.
5. Paste or edit the YAML content.
6. Review the visible metadata and validation result.
7. Click **Create**.

Understand YAML validation and consistency

When you create an application from YAML in the **Applications** view:

- The selected cluster determines the project context used by the workflow.
- The workflow does not forcibly overwrite the namespace declared in the YAML.
- Namespace validation follows the namespace declared in the YAML and standard Kubernetes rules for namespaced resources.

- YAML validation confirms that the content can be parsed and accepted by the workflow before submission.
- If the declared namespace does not exist or you do not have permission to use it, the submission returns an explicit error.
- Successful validation does not guarantee that runtime dependencies are already satisfied.

If your YAML contains multiple resources, make sure each resource declares the intended namespace explicitly when the resource is namespace-scoped.

Review points before submission

Before clicking **Create**, confirm that:

- The namespace declared in the YAML is the namespace where you intend to create the resource.
- The declared namespace already exists, and your current identity has permission to create resources in that namespace.
- The image source is reachable from the selected cluster.
- Referenced ConfigMaps, Secrets, services, PVCs, and other dependencies either already exist or are included consistently in the submitted YAML.
- Labels, selectors, ports, and environment settings remain internally consistent across all submitted objects.
- The YAML does not unintentionally conflict with an existing object that has the same identity in the declared namespace.
- Any scheduling constraints still match the target cluster, especially in mixed-architecture environments.

When you submit multiple related resources together, review not only each object individually but also the relationships between workloads, Services, configuration objects, and storage references.

Creating applications from Code

Creating application from code is implemented using Source to Image(S2I) technology. S2I is an automated framework for building container images directly from source code. This approach standardizes and automates the application build process, allowing developers to focus on source code development without worrying about containerization details.

TOC

Prerequisites

- Complete the installation of Alauda Container Platform Builds

Procedure

1. **Container Platform**, and navigate to **Application** > **Applications**.
2. Click **Create**.
3. Select the **Create from Code**.

4. For detailed parameter descriptions, please refer to Managing applications created from Code
5. After completing the parameter input, click **Create**.
6. The corresponding deployment can be viewed on the **Detail Information** page.

Creating applications from Operator Backed

TOC

Understanding Operator Backed Application

An **Operator** is an extension mechanism built upon Kubernetes Custom Controllers and Custom Resource Definitions (CRDs), designed to automate the complete lifecycle management of complex applications. Within Alauda Container Platform, an Operator Backed Application refers to an application instance provisioned through pre-integrated or user-defined Operators, with its operational workflows managed by the Operator Lifecycle Manager (OLM). This encompasses critical processes such as installation, upgrades, dependency resolution, and access control.

Core Capabilities

1. **Automation of Complex Operations**: Operators overcome the inherent limitations of native Kubernetes resources (e.g., Deployment, StatefulSet) to address the complexities of managing stateful applications, including distributed coordination, persistent storage, and versioned rolling updates. Example: Operator-encoded logic enables autonomous operations for database cluster failover, cross-node data consistency, and backup recovery.
2. **Declarative, State-Driven Architecture**: Operators utilize YAML-based declarative APIs to define desired application states (e.g., spec.replicas: 5). Operators continuously reconcile the actual state with the declared state, providing self-healing capabilities. Deep integration with GitOps tools (e.g., Argo CD) ensures consistent environment configurations.
3. **Intelligent Lifecycle Management**:

   - Rolling Updates & Rollback: OLM's Subscription object subscribes to update channels (e.g., stable, alpha), triggering automated version iterations for both Operators and their managed applications.
   - Dependency Resolution: Operators dynamically identify runtime dependencies (e.g., specific storage drivers, CNI plugins) to ensure successful deployment.

4. **Standardized Ecosystem Integration**: OLM standardizes Operator packaging (Bundle) and distribution channels, enabling one-click deployment of production-grade applications (e.g., Etcd) from OperatorHub or private registries. Enterprise Enhancements: Alauda Container Platform extends RBAC policies and multi-cluster distribution capabilities to meet enterprise compliance requirements.

Operator Backed Application CRD

This Operator is designed and implemented by fully embracing open-source community standards and solutions. Its Custom Resource Definition (CRD) design thoughtfully incorporates established best practices and architectural patterns prevalent within the Kubernetes ecosystem. CRD design reference materials:

1. CatalogSource ↗: Defines the source of Operator packages available to the cluster, such as OperatorHub or custom Operator repositories.
2. ClusterServiceVersion (CSV) ↗: The core metadata definition for an Operator, containing its name, version, provided APIs, required permissions, installation strategy, and detailed lifecycle management information.

3. InstallPlan ↗: The actual execution plan for installing an Operator, automatically generated by OLM based on the Subscription and CSV, detailing the specific steps to create the Operator and its dependent resources.
4. OperatorGroup ↗: Defines a set of target namespaces where an Operator will provide its services and reconcile resources, while also limiting the scope of the Operator's RBAC permissions.
5. Subscription ↗: Used to declare the specific Operator that a user wants to install and track in the cluster, including the Operator's name, target channel (e.g., stable, alpha), and update strategy. OLM uses the Subscription to create and manage the Operator's installation and upgrades.

Creating a Operator Backed Application by using web console

1. **Container Platform**, navigate to **Applications** > **Applications** in the left sidebar.
2. Click **Create**.
3. Choose **Create from Catalog** as the creation approach.
4. Select an Operator-Backed Instance and Configure **Custom Resource Parameters**. Select an Operator-managed application instance and configure its Custom Resource (CR) specifications in the CR manifest, including:
   - `spec.resources.limits` (container-level resource constraints).
   - `spec.resourceQuota` (Operator-defined quota policies). Other CR-specific parameters such as `spec.replicas`, `spec.storage.className`, etc.
5. Click **Create**.

The web console will navigate to **Applications** > **Operator Backed Apps** page.

> **INFO**
>
> **Note**: The Kubernetes resource creation process requires asynchronous reconciliation. Completion may take several minutes depending on cluster conditions.

Troubleshooting

If resource creation fails:

1. Inspect controller reconciliation errors:

```
kubectl get events --field-selector involvedObject.kind=<Your-Custom-Re
source> --sort-by=.metadata.creationTimestamp
```

2. Verify API resource availability:

```
kubectl api-resources | grep <Your-Resource-Type>
```

3. Retry creation after verifying CRD/Operator readiness:

```
kubectl apply -f your-resource-manifest.yaml
```

Creating applications by using CLI

`kubectl` is the primary command-line interface (CLI) for interacting with Kubernetes clusters. It functions as a client for the Kubernetes API Server - a RESTful HTTP API that serves as the control plane's programmatic interface. All Kubernetes operations are exposed through API endpoints, and `kubectl` essentially translates CLI commands into corresponding API requests to manage cluster resources and application workloads (Deployments, StatefulSets, etc.).

The CLI tools facilitates application deployment by intelligently interpreting input artifacts (images, or Chart, etc.) and generating corresponding Kubernetes API objects. The generated resources vary based on input types:

- **Image**: Directly creates Deployment.
- **Chart**: Instantiates all objects defined in the Helm Chart.

TOC

Prerequisites

The **Alauda Container Platform Web Terminal** plugin is installed, and the web-cli switch is enabled.

Procedure

1. **Contianer Platform**, click the terminal icon in the lower-right corner.
2. Wait for session initialization (1-3 sec).
3. Execute kubectl commands in the interactive shell:

   ```
   kubectl get pods -n ${CURRENT_NAMESPACE}
   ```

4. View real-time command output

Example

YAML

```yaml
# webapp.yaml
apiVersion: app.k8s.io/v1beta1
kind: Application
metadata:
  name: webapp
spec:
  componentKinds:
    - group: apps
      kind: Deployment
    - group: ""
      kind: Service
  descriptor: {}

# webapp-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
  labels:
    app: webapp
    env: prod
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
        tier: frontend
    spec:
      containers:
      - name: webapp
        image: nginx:1.25-alpine
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: "100m"
            memory: "128Mi"
          limits:
            cpu: "250m"
```

```yaml
            memory: "256Mi"
---
# webapp-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  selector:
    app: webapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

kubectl commands

```
kubectl apply -f webapp.yaml -n {CURRENT_NAMESPACE}
kubectl apply -f webapp-deployment.yaml -n {CURRENT_NAMESPACE}
kubectl apply -f webapp-service.yaml -n {CURRENT_NAMESPACE}
```

Reference

- **Conceptual Guide**: kubectl Overview
- **Syntax Reference**: kubectl Cheat Sheet
- **Command Manual**: kubectl Commands

Operation and Maintaining Applications

This section covers the ongoing management work that usually starts after an application has been created and deployed.

Typical tasks include:

- Reviewing application status before deciding whether to troubleshoot, update, or roll back
- Updating application configuration and managing the resources associated with the application
- Creating version snapshots, comparing historical revisions, and rolling back when needed
- Exporting the current version or a historical version for reuse in another environment

If you mainly need logs, events, metrics, or alert rules, continue with Application Observability.

TOC

Workload Availability

Use these pages to manage high availability and constraints during maintenance for your workloads:

- PodDisruptionBudgets

Application Rollout

Status Description

Configuring HPA

Starting and Stopping Applications

Configuring VerticalPodAutoscaler (VPA)

Configuring CronHPA

Updating Applications

Exporting Applications

Updating and deleting Chart Applications

Workload Rebalancing (Descheduler)

Version Management for Applications

Deleting Applications

Using PodDisruptionBudgets

Dynamically Inject PSA Security Contexts for Workloads

Dynamically Inject PSA Security Contexts for Workloads

Handling Out of Resource Errors

Handling Out of Resource Errors

Health Checks

Health Checks

Application Rollout

Installing Alauda Container Platform Argo Rollouts

TOC

Prerequisites

1. **Download** the **Alauda Container Platform Argo Rollouts** cluster plugin installation package corresponding to your platform architecture.
2. **Upload** the installation package using the Upload Packages mechanism.
3. **Install** the installation package to the cluster using the cluster plugins mechanism.

> **INFO**
>
> Upload Packages: **Administrator** > **Marketplace** > **Upload Packages** page. Click **Help Document** on the right to get instructions on how to publish the cluster plugin to cluster. For more details, please refer to CLI.

Installing Alauda Container Platform Argo Rollouts

Procedure

2. Click **Marketplace** > **Cluster Plugins** to enter the **Cluster Plugins** list page.
3. Find the **Alauda Container Platform Argo Rollouts** cluster plugin, click **Install**, and navigate to the **Install Alauda Container Platform Argo Rollouts Plugin** page.
4. Simply click **Install** to complete the **Alauda Container Platform Argo Rollouts** cluster plugin installation.

Application Blue Green Deployment

In the modern world of software development, deploying new versions of applications is a crucial part of the development cycle. However, rolling out updates to production environments can be a risky proposition, as even small issues can result in significant downtime and lost revenue. Blue-Green Deployments are a deployment strategy that mitigates this risk by ensuring that new versions of applications can be deployed with zero downtime.

A Blue-Green Deployment is a deployment strategy where two identical environments, the “blue” environment and the “green” environment, are set up. The blue environment is the production environment, where the live version of the application is currently running, and the green environment is the non-production environment, where new versions of the application are deployed.

When a new version of the application is ready to be deployed, it is deployed to the green environment. Once the new version is deployed and tested, traffic is switched to the green environment, making it the new production environment. The blue environment then becomes the non-production environment, where future versions of the application can be deployed.

TOC

Benefits of Blue Green Deployments

- Zero Downtime: Blue-Green Deployments allow new versions of applications to be deployed with zero downtime, as traffic is switched from the blue environment to the green environment seamlessly.
- Easy Rollback: If a new version of the application has issues, rolling back to the previous version is easy, as the blue environment is still available.
- Reduced Risk: By using Blue-Green Deployments, the risk of deploying new versions of applications is reduced significantly. This is because the new version can be deployed and tested in the green environment before traffic is switched over from the blue environment. This allows for thorough testing and reduces the chance of issues arising in production.
- Increased Reliability: By using Blue-Green Deployments, the reliability of the application is increased. This is because the blue environment is always available, and any issues with the green environment can be quickly identified and resolved without affecting users.
- Flexibility: Blue-Green Deployments provide flexibility in the deployment process. Multiple versions of an application can be deployed side-by-side, allowing for easy testing and experimentation.

Blue Green Deployment with Argo Rollouts

Argo Rollouts is a Kubernetes controller and set of CRDs which provide advanced deployment capabilities such as blue-green, canary, canary analysis, experimentation, and progressive delivery features to Kubernetes.

Argo Rollouts (optionally) integrates with ingress controllers and service meshes, leveraging their traffic shaping abilities to gradually shift traffic to the new version during an update. Additionally, Rollouts can query and interpret metrics from various providers to verify key KPIs and drive automated promotion or rollback during an update.

With Argo Rollouts, you can automate blue green deployments on Alauda Container Platform (ACP) clusters. The typical process includes:

1. Defining Rollout resources to manage different application versions.
2. Configuring Kubernetes services to route traffic between blue (current) and green (new) environments.
3. Deploying the new version to the green environment.
4. Verifying and testing the new version.
5. Promoting the green environment to production by switching traffic.

This approach minimizes downtime and enables controlled, safe deployments.

> **Key Concepts:**
>
> - **Rollout**: A custom resource definition (CRD) in Kubernetes that replaces standard Deployment resources, enabling advanced deployment control such as blue-green, canary deployment.

Prerequisites

1. ACP (Alauda Container Platform).
2. Kubernetes Cluster managed by ACP.
3. Argo Rollouts installed in the cluster.
4. Argo Rollouts kubectl plugin.
5. A project to create a namespace in it.
6. A namespace in the cluster where the application will be deployed.

Procedure

1 Creating the Deployment

Start by defining the "blue" version of your application. This is the current version that users will access. Create a Kubernetes deployment with the appropriate number of replicas, container image version (e.g., `hello:1.23.1`), and proper labels such as `app=web`.

Use the following YAML:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: hello:1.23.1
          ports:
            - containerPort: 80
```

Explanation of YAML fields:

- `apiVersion`: The version of the Kubernetes API used to create the resource.
- `kind`: Specifies that this is a Deployment resource.
- `metadata.name`: The name of the deployment.
- `spec.replicas`: Number of desired pod replicas.
- `spec.selector.matchLabels`: Defines how the Deployment finds which pods to manage.

- `template.metadata.labels` : Labels applied to pods, used by Services to select them.
- `spec.containers` : The containers to run in each pod.
- `containers.name` : Name of the container.
- `containers.image` : Container image to run.
- `containers.ports.containerPort` : Port exposed by the container.

Apply the configuration using `kubectl` :

```
kubectl apply -f deployment.yaml
```

This sets up the production environment.

Alternative, you could use helm chart to create the deployments and services.

2 Creating the Blue Service

Create a Kubernetes `Service` that exposes the blue deployment. This service will forward traffic to the blue pods based on matching labels. Initially, the service selector targets pods labeled with `app=web` .

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  selector:
    app: web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Explanation of YAML fields:

- `apiVersion` : The version of the Kubernetes API used to create the Service.

- `kind`: Specifies this resource is a Service.
- `metadata.name`: Name of the Service.
- `spec.selector`: Identifies pods to route traffic to, based on labels.
- `ports.protocol`: The protocol used (TCP).
- `ports.port`: Port exposed by the Service.
- `ports.targetPort`: The port on the container to which the traffic is directed.

Apply it using:

```
kubectl apply -f web-service.yaml
```

This allows external access to the blue deployment.

3 Verify the Blue Deployment

Confirm that the `blue` deployment is running correctly by listing the pods:

```
kubectl get pods -l app=web
```

Check that all expected replicas (2) are in the `Running` state. This ensures the application is ready to serve traffic.

4 Verify Traffic Routing to Blue

Ensure that the `web` service is correctly forwarding traffic to the blue deployment. Use this command:

```
kubectl describe service web | grep Endpoints
```

The output should list the IP addresses of the blue pods. These are the endpoints receiving traffic.

5 Creating the Rollout

Next, creating the `Rollout` resource from Argo Rollouts with `BlueGreen` strategy.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: rollout-bluegreen
spec:
  replicas: 2
  revisionHistoryLimit: 2
  selector:
    matchLabels:
      app: web
  workloadRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web
    scaleDown: onsuccess
  strategy:
    blueGreen:
      activeService: web
      autoPromotionEnabled: false
```

Explanation of YAML fields:

- `spec.selector`: Label selector for pods. Existing ReplicaSets whose pods are selected by this will be the ones affected by this rollout. It must match the pod template's labels.
- `workloadRef`: Specify the workload reference and scale down strategy to apply the rollouts.
  - `scaleDown`: Specifies if the workload (Deployment) is scaled down after migrating to Rollout. The possible options are:
    - "never": the Deployment is not scaled down.
    - "onsuccess": the Deployment is scaled down after the Rollout becomes healthy.
    - "progressively": as the Rollout is scaled up the Deployment is scaled down. If the Rollout fails the Deployment will be scaled back up.
- `strategy`: The rollout strategy, support `BlueGreen` and `Canary` strategy.

- `blueGreen`: The `BlueGreen` rollout strategy definition.
  - `activeService`: Specifies the service to update with the new template hash at time of promotion. This field is mandatory for the blueGreen update strategy.
  - `autoPromotionEnabled`: autoPromotionEnabled disables automated promotion of the new stack by pausing the rollout immediately before the promotion. If omitted, the default behavior is to promote the new stack as soon as the ReplicaSet are completely ready/available. Rollouts can be resumed using: `kubectl argo rollouts promote ROLLOUT`

Apply it with:

```
kubectl apply -f rollout.yaml
```

This sets up the rollouts for the deployment with `BlueGreen` strategy.

6 Verify the Rollouts

After the `Rollout` was created, the Argo Rollouts will create a new ReplicaSet with same template of the deployment. While the pods of new ReplicaSet is healthy, the deployment is scaled down to 0.

Use the following command to ensure the pods are running properly:

```
kubectl argo rollouts get rollout rollout-bluegreen
Name:              rollout-bluegreen
Namespace:         default
Status:            ✔ Healthy
Strategy:          BlueGreen
Images:            hello:1.23.1 (stable, active)
Replicas:
  Desired:         2
  Current:         2
  Updated:         2
  Ready:           2
  Available:       2

NAME                                         KIND         STATUS
AGE   INFO
⟳ rollout-bluegreen                           Rollout      ✔ Health
y   95s
└──# revision:1
   └──⧉ rollout-bluegreen-595d4567cc          ReplicaSet  ✔ Healthy
18s   stable,active
      ├──□ rollout-bluegreen-595d4567cc-mc769  Pod         ✔ Running
8s    ready:1/1
      └──□ rollout-bluegreen-595d4567cc-zdc5x  Pod         ✔ Running
8s    ready:1/1
```

The service `web` will forward traffic to the pods created by rollouts. Use this command:

```
kubectl describe service web | grep Endpoints
```

7 Preparing Green Deployment

Next, prepare the new version of the application as the green deployment. Update the deployment `web` with the new image version (e.g., `hello:1.23.2`).

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: hello:1.23.2
          ports:
            - containerPort: 80
```

Explanation of YAML fields:

- Identical to the original deployment, with the exception of:
  - `containers.image`: Updated to new image version.

Apply it with:

```
kubectl apply -f deployment.yaml
```

This sets up the new application version for testing.

The rollouts will create a new Replicaset to manage the green pods, and the traffic still forward to the blue pods. Use the following command to verify:

```
kubectl argo rollouts get rollout rollout-bluegreen
Name:            rollout-bluegreen
Namespace:       default
Status:          ॥ Paused
Message:         BlueGreenPause
Strategy:        BlueGreen
Images:          hello:1.23.1 (stable, active)
                 hello:1.23.2
Replicas:
  Desired:       2
  Current:       4
  Updated:       2
  Ready:         2
  Available:     2

NAME                                           KIND         STATUS
AGE  INFO
⟳ rollout-bluegreen                            Rollout      ॥ Paused
14m
├──# revision:2
│  └──⧉ rollout-bluegreen-776b688d57           ReplicaSet  ✔ Healthy
24s
│     ├──□ rollout-bluegreen-776b688d57-kxr66  Pod         ✔ Running
23s  ready:1/1
│     └──□ rollout-bluegreen-776b688d57-vv7t7  Pod         ✔ Running
23s  ready:1/1
└──# revision:1
   └──⧉ rollout-bluegreen-595d4567cc           ReplicaSet  ✔ Healthy
12m  stable,active
      ├──□ rollout-bluegreen-595d4567cc-mc769  Pod         ✔ Running
12m  ready:1/1
      └──□ rollout-bluegreen-595d4567cc-zdc5x  Pod         ✔ Running
12m  ready:1/1
```

Currently, there are 4 pods running, with blue and green version. And the active service is the blue version, the rollout process is paused.

If you use helm chart to deploy the application, use helm tool to upgrade the application to the green version.

8 Promoting the Rollout to Green

When the green version is ready, promote the rollout to switch traffic to the green pods. Use the following command:

```
kubectl argo rollouts promote rollout-bluegreen
```

To Verify if the rollout is completed:

```
kubectl argo rollouts get rollout rollout-bluegreen
Name:            rollout-bluegreen
Namespace:       default
Status:          ✔ Healthy
Strategy:        BlueGreen
Images:          hello:1.23.2 (stable, active)
Replicas:
  Desired:       2
  Current:       2
  Updated:       2
  Ready:         2
  Available:     2

NAME                                           KIND        STATUS
AGE   INFO
⟳ rollout-bluegreen                             Rollout     ✔ Health
y       3h2m
├──# revision:2
│  └──⧉ rollout-bluegreen-776b688d57           ReplicaSet  ✔ Healthy
168m  stable,active
│     ├──□ rollout-bluegreen-776b688d57-kxr66  Pod         ✔ Running
168m  ready:1/1
│     └──□ rollout-bluegreen-776b688d57-vv7t7  Pod         ✔ Running
168m  ready:1/1
└──# revision:1
   └──⧉ rollout-bluegreen-595d4567cc           ReplicaSet  • ScaledD
own   3h1m
      ├──□ rollout-bluegreen-595d4567cc-mc769  Pod         ◌ Terminat
ing  3h     ready:1/1
      └──□ rollout-bluegreen-595d4567cc-zdc5x  Pod         ◌ Terminat
ing  3h     ready:1/1
```

> If the active `Images` is updated to `hello:1.23.2`, and the blue ReplicaSet is scaled down to 0, that means the rollout is completed.

Application Canary Deployment

Canary Deployment is a progressive release strategy where a new application version is gradually introduced to a small subset of users or traffic. This incremental rollout allows teams to monitor system behavior, collect metrics, and ensure stability before a full-scale deployment. The approach significantly reduces risk, especially in production environments.

Argo Rollouts is a Kubernetes-native progressive delivery controller that facilitates advanced deployment strategies. It extends Kubernetes capabilities by offering features like Canary, Blue-Green Deployments, Analysis Runs, Experimentation, and Automated Rollbacks. It integrates with observability stacks for metric-based health checks and provides CLI and dashboard-based control over application delivery.

> **Key Concepts:**
>
> - **Rollout**: A custom resource definition (CRD) in Kubernetes that replaces standard Deployment resources, enabling advanced deployment control such as blue-green, canary deployment.
> - **Canary Steps**: A series of incremental traffic shifting actions, such as directing 25%, then 50% of traffic to the new version.
> - **Pause Steps**: Introduce wait intervals for manual or automatic validation before progressing to the next canary step.

TOC

Benefits of Canary Deployments

- **Risk mitigation**: By deploying changes to a small subset of servers initially, you can find issues and address them before the full rollout, minimizing the impact on users.
- **Incremental rollouts**: This approach allows gradual exposure to new features, which helps you effectively monitor performance and user feedback.
- **Real-time feedback**: Canary deployments provide immediate insights into the performance and stability of new releases under real-world conditions.
- **Flexibility**: You can adjust the deployment process based on performance metrics. This allows for a dynamic rollout that you can pause or roll back as needed.
- **Cost-effectiveness**: Unlike blue/green deployments, canary deployments don't require a separate environment, making them more resource-efficient.

Canary Deployments with Argo Rollouts

Argo Rollouts supports canary deployment strategy to rollout a deployment, and control the traffic through Gateway API Plugin. In ACP, you could use ALB to act as a Gateway API Provider to implement the traffic control for Argo Rollouts.

Prerequisites

1. Argo Rollouts with Gateway API plugin installed in the cluster.
2. Argo Rollouts kubectl plugin (Install from here).
3. A project to create a namespace in it.
4. ALB deployed in the cluster and allocated to the project.
5. A namespace in the cluster where the application will be deployed.

Procedure

1 Creating the Deployment

Start by defining the "stable" version of your application. This is the current version that users will access. Create a Kubernetes deployment with the appropriate number of replicas, container image version (e.g., `hello:1.23.1`), and proper labels such as `app=web`.

Use the following YAML:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: hello:1.23.1
          ports:
            - containerPort: 80
```

Explanation of YAML fields:

- `apiVersion`: The version of the Kubernetes API used to create the resource.
- `kind`: Specifies that this is a Deployment resource.
- `metadata.name`: The name of the deployment.
- `spec.replicas`: Number of desired pod replicas.
- `spec.selector.matchLabels`: Defines how the Deployment finds which pods to manage.
- `template.metadata.labels`: Labels applied to pods, used by Services to select them.
- `spec.containers`: The containers to run in each pod.
- `containers.name`: Name of the container.
- `containers.image`: Container image to run.
- `containers.ports.containerPort`: Port exposed by the container.

Apply the configuration using `kubectl`:

```
kubectl apply -f deployment.yaml
```

This sets up the production environment.

Alternative, you could use helm chart to create the deployments and services.

2 Creating the Stable Service

Create a Kubernetes `Service` that exposes the stable deployment. This service will forward traffic to the pods of stable version based on matching labels. Initially, the service selector targets pods labeled with `app=web`.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-stable
spec:
  selector:
    app: web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Explanation of YAML fields:

- `apiVersion`: The version of the Kubernetes API used to create the Service.
- `kind`: Specifies this resource is a Service.
- `metadata.name`: Name of the Service.
- `spec.selector`: Identifies pods to route traffic to, based on labels.
- `ports.protocol`: The protocol used (TCP).
- `ports.port`: Port exposed by the Service.
- `ports.targetPort`: The port on the container to which the traffic is directed.

Apply it using:

```
kubectl apply -f web-stable-service.yaml
```

This allows external access to the stable deployment.

3 Creating the Canary Service

Create a Kubernetes `Service` that exposes the canary deployment. This service will forward traffic to the pods of canary version based on matching labels. Initially, the service selector targets pods labeled with `app=web`.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-canary
spec:
  selector:
    app: web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Explanation of YAML fields:

- `apiVersion`: The version of the Kubernetes API used to create the Service.
- `kind`: Specifies this resource is a Service.
- `metadata.name`: Name of the Service.
- `spec.selector`: Identifies pods to route traffic to, based on labels.
- `ports.protocol`: The protocol used (TCP).
- `ports.port`: Port exposed by the Service.
- `ports.targetPort`: The port on the container to which the traffic is directed.

Apply it using:

```
kubectl apply -f web-canary-service.yaml
```

This allows external access to the canary deployment.

4 Creating the Gateway

Use `example.com` as the domain to access the service, create the gateway to expose the service with the domain:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: default
spec:
  gatewayClassName: exclusive-gateway
  listeners:
  - allowedRoutes:
      namespaces:
        from: All
    name: gateway-metric
    port: 11782
    protocol: TCP
  - allowedRoutes:
      namespaces:
        from: All
    hostname: example.com
    name: web
    port: 80
    protocol: HTTP
```

Use the command:

```
kubectl apply -f gateway.yaml
```

The gateway will be allocated an external IP address, get the IP address from the `status.addresses` of type `IPAddress` in the gateway resource.

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: default
...
status:
  addresses:
  - type: IPAddress
    value: 192.168.134.30
```

5 DNS Configuration

Configure the domain in your dns server to resolve the domain to the IP address of the gateway. Verify the dns resolve with the command:

```
nslookup example.com
Server:         192.168.16.19
Address:        192.168.16.19#53

Non-authoritative answer:
Name:   example.com
Address: 192.168.134.30
```

It should return the address of the gateway.

6 Creating the HTTPRoute

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: web
spec:
  hostnames:
  - example.com
  parentRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: default
    namespace: default
    sectionName: web
  rules:
  - backendRefs:
    - group: ""
      kind: Service
      name: web-canary
      namespace: default
      port: 80
      weight: 0
    - group: ""
      kind: Service
      name: web-stable
      namespace: default
      port: 80
      weight: 100
    matches:
    - path:
        type: PathPrefix
        value: /
```

Use the command:

```
kubectl apply -f httproute.yaml
```

7 Accessing the Stable service

Outside the cluster, use the command to access the service from the domain:

```
curl http://example.com
```

Or you can access `http://example.com` in the browser.

8 Creating the Rollout

Next, creating the `Rollout` resource from Argo Rollouts with `Canary` strategy.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: rollout-canary
spec:
  minReadySeconds: 30
  replicas: 2
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: web
  strategy:
    canary:
      canaryService: web-canary
      maxSurge: 25%
      maxUnavailable: 0
      stableService: web-stable
      steps:
      - setWeight: 50
      - pause: {}
      - setWeight: 100
      trafficRouting:
        plugins:
          argoproj-labs/gatewayAPI:
            httpRoute: web
            namespace: default
  workloadRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web
    scaleDown: onsuccess
```

Explanation of YAML fields:

- `spec.selector`: Label selector for pods. Existing ReplicaSets whose pods are selected by this will be the ones affected by this rollout. It must match the pod template's labels.
- `workloadRef`: Specify the workload reference and scale down strategy to apply the rollouts.
- `scaleDown`: Specifies if the workload (Deployment) is scaled down after migrating to Rollout. The possible options are:
  - "never": the Deployment is not scaled down.
  - "onsuccess": the Deployment is scaled down after the Rollout becomes healthy.
  - "progressively": as the Rollout is scaled up the Deployment is scaled down. If the Rollout fails the Deployment will be scaled back up.
- `strategy`: The rollout strategy, support `BlueGreen` and `Canary` strategy.
- `canary`: The `Canary` rollout strategy definition.
  - `canaryService`: Reference to a service which the controller will update to select canary pods. Required for traffic routing.
  - `stableService`: Reference to a service which the controller will update to select stable pods. Required for traffic routing.
  - `steps`: Steps define sequence of steps to take during an update of the canary. Skipped upon initial deploy of a rollout.
    - `setWeight`: Sets the ratio of canary ReplicaSet.
    - `pause`: Pauses the rollout indefinitely or for a time. Supported units: s, m, h. `{}` means indefinitely.
    - `plugin`: executes the configured plugin, here we configure it with the `gatewayAPI` plugin.

Apply it with:

```
kubectl apply -f rollout.yaml
```

This sets up the rollouts for the deployment with `Canary` strategy. It will set weight to 50 initially, and wait for the promoting. The 50% of the traffic will forward to the canary service. After promoting the rollout, the weight will be set to 100, and 100% of the traffic will forward to the canary service. Finally, the canary service will become the stable service.

9 Verify the Rollouts

After the `Rollout` was created, the Argo Rollouts will create a new ReplicaSet with same template of the deployment. While the pods of new ReplicaSet is healthy, the deployment is scaled down to 0.

Use the following command to ensure the pods are running properly:

```
kubectl argo rollouts get rollout rollout-canary
Name:            rollout-canary
Namespace:       default
Status:          ✔ Healthy
Strategy:        Canary
Step:          9/9
SetWeight:     100
ActualWeight:  100
Images:          hello:1.23.1 (stable)
Replicas:
Desired:       2
Current:       2
Updated:       2
Ready:         2
Available:     2

NAME                                          KIND        STATUS     AGE
INFO
⟳ rollout-canary                                 Rollout     ✔ Healthy
32s
└──# revision:1
  └──⧉ rollout-canary-5c9d79697b            ReplicaSet  ✔ Healthy  32
s  stable
     ├──□ rollout-canary-5c9d79697b-fh78d  Pod         ✔ Running  32s
ready:1/1
     └──□ rollout-canary-5c9d79697b-rrbtj  Pod         ✔ Running  32s
ready:1/1
```

10 Preparing Canary Deployment

Next, prepare the new version of the application as the green deployment. Update the deployment `web` with the new image version (e.g., `hello:1.23.2`). Use the command:

```
kubectl patch deployment web -p '{"spec":{"template":{"spec":{"contai
ners":[{"name":"web","image":"hello:1.23.2"}]}}}}'
```

This sets up the new application version for testing.

The rollouts will create a new Replicaset to manage the canary pods, and the 50% traffic will forward to the canary pods. Use the following command to verify:

```
kubectl argo rollouts get rollout rollout-canary
Name:                rollout-canary
Namespace:           default
Status:              ॥ Paused
Message:             CanaryPauseStep
Strategy:            Canary
Step:              1/3
SetWeight:         50
ActualWeight:      50
Images:              hello:1.23.1 (stable)
                   hello:1.23.2 (canary)
Replicas:
Desired:           2
Current:           3
Updated:           1
Ready:             3
Available:         3

NAME                                       KIND         STATUS     AGE
INFO
⟳ rollout-canary                              Rollout      ॥ Paused    9
5s
├──# revision:2
│  └──⧉ rollout-canary-5898765588           ReplicaSet  ✔ Healthy  4
6s  canary
│     └──□ rollout-canary-5898765588-ls5jk  Pod         ✔ Running  4
5s  ready:1/1
└──# revision:1
   └──⧉ rollout-canary-5c9d79697b           ReplicaSet  ✔ Healthy  95
s  stable
      ├──□ rollout-canary-5c9d79697b-fk269  Pod         ✔ Running  94s
ready:1/1
      └──□ rollout-canary-5c9d79697b-wkmcn  Pod         ✔ Running  94s
ready:1/1
```

Currently, there are 3 pods running, with stable and canary version. And the weight is 50, 50% of the traffic will forward to the canary service. The rollout process is paused to wait for the promoting.

If you use helm chart to deploy the application, use helm tool to upgrade the application to the canary version.

Accessing `http://example.com`, the 50% traffic will forward to the canary service. You should have different response from the URL.

11 Promoting the Rollout

When the canary version is tested ok, you could promote the rollout to switch all traffic to the canary pods. Use the following command:

```
kubectl argo rollouts promote rollout-canary
```

To Verify if the rollout is completed:

```
kubectl argo rollouts get rollout rollout-canary
Name:            rollout-canary
Namespace:       default
Status:          ✔ Healthy
Strategy:        Canary
Step:          3/3
SetWeight:     100
ActualWeight:  100
Images:          hello:1.23.2 (stable)
Replicas:
Desired:       2
Current:       2
Updated:       2
Ready:         2
Available:     2

NAME                                          KIND        STATUS
AGE    INFO
⟳ rollout-canary                               Rollout     ✔ Healthy
8m42s
├──# revision:2
│  └──⧉ rollout-canary-5898765588           ReplicaSet  ✔ Healthy
7m53s  stable
│     ├──□ rollout-canary-5898765588-ls5jk  Pod         ✔ Running
7m52s  ready:1/1
│     └──□ rollout-canary-5898765588-dkfwg  Pod         ✔ Running
68s    ready:1/1
└──# revision:1
  └──⧉ rollout-canary-5c9d79697b            ReplicaSet  • ScaledDown
8m42s
    ├──□ rollout-canary-5c9d79697b-fk269   Pod         ◌ Terminating
8m41s  ready:1/1
    └──□ rollout-canary-5c9d79697b-wkmcn   Pod         ◌ Terminating
8m41s  ready:1/1
```

If the stable `Images` is updated to `hello:1.23.2`, and the ReplicaSet of revision 1 is scaled down to 0, that means the rollout is completed.

Accessing `http://example.com`, the 100% traffic will forward to the canary service.

12 Aborting the Rollout (Optional)

If you found the canary version has some problems during rollout process, you can abort the process to switch all traffic to the stable service. Use the command:

```
kubectl argo rollouts abort rollout-canary
```

To verify the results:

```
kubectl argo rollouts get rollout rollout-canary
Name:              rollout-demo
Namespace:         default
Status:            ✖ Degraded
Message:           RolloutAborted: Rollout aborted update to revision 3
Strategy:          Canary
Step:            0/3
SetWeight:       0
ActualWeight:    0
Images:            hello:1.23.1 (stable)
Replicas:
Desired:         2
Current:         2
Updated:         0
Ready:           2
Available:       2

NAME                                        KIND        STATUS        A
GE  INFO
⟳ rollout-canary                               Rollout     ✖ Degraded
18m
├──# revision:3
│  └──⧉ rollout-canary-5c9d79697b           ReplicaSet  • ScaledDown
18m  canary,delay:passed
└──# revision:2
  └──⧉ rollout-canary-5898765588           ReplicaSet  ✔ Healthy
17m  stable
    ├──□ rollout-canary-5898765588-ls5jk  Pod         ✔ Running
17m  ready:1/1
    └──□ rollout-canary-5898765588-dkfwg  Pod         ✔ Running
10m  ready:1/1
```

Accessing `http://example.com`, the 100% traffic will forward to the stable service.

Status Description

TOC

Applications

The status of native applications and their corresponding meanings are as follows. The numbers following the status indicate the number of computing components.

Status	Meaning
Running	All computing components are in normal operation.
Partially Running	Some computing components are running, while others have stopped.
Stopped	All computing components have stopped.
Processing	At least one computing component is in a pending state.
No Computing Components	There are no computing components under the application.

Status	Meaning
Failed	Deployment has failed.

Note: Similarly, the numbers in the computing component status indicate the number of container groups.

Use these statuses together with the application overview and application details page:

- **Running**: All managed workloads are healthy and the application can usually be treated as operational.
- **Partially Running**: Some workloads are healthy while others are stopped, failing, or not yet ready. This often indicates a partial rollout result or a dependency issue limited to part of the application boundary.
- **Processing**: At least one workload is still being created, scheduled, updated, or reconciled.
- **Stopped**: All managed computing components are stopped.
- **No Computing Components**: The application currently has no workload resources associated with it.
- **Failed**: The application contains one or more failed workloads or an unsuccessful deployment result that requires investigation.

When the application is not in the expected state, continue by checking associated Pods, events, logs, and monitoring data from the application details page.

Deployment

- Running: All Pods are in normal operation.
- Processing: There are Pods that are not in a running state.
- Stopped: All Pods have stopped.
- Failed: Deployment has failed.

Configuring HPA

HPA (Horizontal Pod Autoscaler) automatically scales the number of pods up or down based on preset policies and metrics, enabling applications to handle sudden spikes in business load while optimizing resource utilization during low-traffic periods.

TOC

Understanding Horizontal Pod Autoscalers

You can create a horizontal pod autoscaler to specify the minimum and maximum number of pods you want to run, as well as the CPU utilization or memory utilization your pods should target.

After you create a horizontal pod autoscaler, the platform begins to query the CPU and/or memory resource metrics on the pods. When these metrics are available, the horizontal pod autoscaler computes the ratio of the current metric utilization with the desired metric utilization, and scales up or down accordingly. The query and scaling occurs at a regular interval, but can take one to two minutes before metrics become available.

For replication controllers, this scaling corresponds directly to the replicas of the replication controller. For deployment configurations, scaling corresponds directly to the replica count of the deployment configuration. Note that autoscaling applies only to the latest deployment in the Complete phase.

The platform automatically accounts for resources and prevents unnecessary autoscaling during resource spikes, such as during start up. Pods in the unready state have 0 CPU usage when scaling up and the autoscaler ignores the pods when scaling down. Pods without known metrics have 0% CPU usage when scaling up and 100% CPU when scaling down. This allows for more stability during the HPA decision. To use this feature, you must configure readiness checks to determine if a new pod is ready for use.

How Does the HPA Work?

The horizontal pod autoscaler (HPA) extends the concept of pod auto-scaling. The HPA lets you create and manage a group of load-balanced nodes. The HPA automatically increases or decreases the number of pods when a given CPU or memory threshold is crossed.

The HPA works as a control loop with a default of 15 seconds for the sync period. During this period, the controller manager queries the CPU, memory utilization, or both, against what is defined in the configuration for the HPA. The controller manager obtains the utilization metrics from the resource metrics API for per-pod resource metrics like CPU or memory, for each pod that is targeted by the HPA.

If a utilization value target is set, the controller calculates the utilization value as a percentage of the equivalent resource request on the containers in each pod. The controller then takes the average of utilization across all targeted pods and produces a ratio that is used to scale the number of desired replicas.

Supported Metrics

The following metrics are supported by horizontal pod autoscalers:

Metric	Description
CPU Utilization	Number of CPU cores used. Can be used to calculate a percentage of the pod's requested CPU.
Memory Utilization	Amount of memory used. Can be used to calculate a percentage of the pod's requested memory.
Network Inbound Traffic	Amount of network traffic coming into the pod, measured in KiB/s.
Network Outbound Traffic	Amount of network traffic going out from the pod, measured in KiB/s.
Storage Read Traffic	Amount of data read from storage, measured in KiB/s.
Storage Write Traffic	Amount of data written to storage, measured in KiB/s.

> **Important**: For memory-based autoscaling, memory usage must increase and decrease proportionally to the replica count. On average:
>
> - An increase in replica count must lead to an overall decrease in memory (working set) usage per-pod.
> - A decrease in replica count must lead to an overall increase in per-pod memory usage.
> - Use the platform to check the memory behavior of your application and ensure that your application meets these requirements before using memory-based autoscaling.

Prerequisites

Please ensure that the monitoring components are deployed in the current cluster and are functioning properly. You can check the deployment and health status of the monitoring components by clicking on the top right corner of the platform ? > **Platform Health Status**..

Creating a Horizontal Pod Autoscaler

Using the CLI

You can create a horizontal pod autoscaler using the command line interface by defining a YAML file and using the `kubectl create` command. The following example shows autoscaling for a Deployment object. The initial deployment requires 3 pods. The HPA object increases the minimum to 5. If CPU usage on the pods reaches 75%, the pods increase to 7:

1. Create a YAML file named `hpa.yaml` with the following content:

```
apiVersion: autoscaling/v2 1
kind: HorizontalPodAutoscaler 2
metadata:
  name: hpa-demo 3
  namespace: default
spec:
  maxReplicas: 7 4
  minReplicas: 3 5
  scaleTargetRef:
    apiVersion: apps/v1 6
    kind: Deployment 7
    name: deployment-demo 8
  metrics: 9
    - type: Resource 10
      resource: 11
        name: cpu 12
        target: 13
          type: Utilization 14
          averageUtilization: 75 15
```

1 Use the autoscaling/v2 API.

2 The name of the HPA resource.

3. The name of the deployment to scale.

4. The maximum number of replicas to scale up to.

5. The minimum number of replicas to maintain.

6. Specify the API version of the object to scale.

7. Specify the type of object. The object must be a Deployment, ReplicaSet, or StatefulSet.

8. The target resource to which the HPA applies.

9. The target CPU utilization percentage that triggers scaling.

2. Apply the YAML file to create the HPA:

```
kubectl create -f hpa.yaml
```

Example output:

```
horizontalpodautoscaler.autoscaling/hpa-demo created
```

3. After you create the HPA, you can view the new state of the deployment by running the following command:

```
kubectl get deployment deployment-demo
```

Example output:

```
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment-demo   5/5     5            5           3m
```

4. You can also check the status of your HPA:

```
kubectl get hpa hpa-demo
```

Example output:

```
NAME        REFERENCE                         TARGETS    MINPODS    MAXPODS
REPLICAS    AGE
hpa-demo    Deployment/deployment-demo        0%/75%     3          7
3           2m
```

Using the Web Console

1. Enter **Container Platform**.
2. In the left navigation bar, click **Workloads** > **Deployments**.
3. Click on ***Deployment Name***.
4. Scroll down to the **Elastic Scaling** area and click on **Update** on the right.
5. Select **Horizontal Scaling** and complete the policy configuration.

Parameter	Description
Pod Count	After a deployment is successfully created, you need to evaluate the **Minimum Pod Count** corresponding to known and regular business volume changes, as well as the **Maximum Pod Count** that can be supported by the namespace quota under high business pressure. The maximum or minimum pod counts can be changed after setting, and it is recommended to first derive a more accurate value through performance testing and to continuously adjust during usage to meet business needs.
Trigger Policy	List the **Metrics** that are sensitive to business changes and their **Target Thresholds** to trigger scale-up or scale-down actions. For example, if you set *CPU Utilization = 60%*, once the CPU utilization deviates from 60%, the platform will start to automatically adjust the number of pods based on the current deployment's insufficient or excessive resource allocation. **Note**: Metric types include built-in metrics and custom metrics. Custom metrics only apply to deployments in native applications, and you must first add custom metrics.
Scale Up/Down Step (Alpha)	For businesses with specific scaling rate requirements, you can gradually adapt to changes in business volume by specifying **Scale-Up Step** or

Parameter	Description
	Scale-Down Step. For the scale-down step, you can customize the **Stability Window**, which defaults to 300 seconds, meaning that you must wait 300 seconds before executing scale-down actions.

6. Click **Update**.

Using Custom Metrics for HPA

Custom metrics HPA extends the original HorizontalPodAutoscaler by supporting additional metrics beyond CPU and memory utilization.

Requirements

- kube-controller-manager: horizontal-pod-autoscaler-use-rest-clients=true
- Pre-installed metrics-server
- Prometheus
- custom-metrics-api

Traditional (Core Metrics) HPA

Traditional HPA supports CPU utilization and memory metrics to dynamically adjust the number of Pod instances, as shown in the example below:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-app-nginx
  namespace: test-namespace
spec:
  maxReplicas: 1
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-app-nginx
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

In this YAML, `scaleTargetRef` specifies the workload object for scaling, and `targetCPUUtilizationPercentage` specifies the CPU utilization trigger metric.

Custom Metrics HPA

To use custom metrics, you need to install prometheus-operator and custom-metrics-api. After installation, custom-metrics-api provides a large number of custom metric resources:

```
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "custom.metrics.k8s.io/v1beta1",
  "resources": [
    {
      "name": "namespaces/go_memstats_heap_sys_bytes",
      "singularName": "",
      "namespaced": false,
      "kind": "MetricValueList",
      "verbs": ["get"]
    },
    {
      "name": "jobs.batch/go_memstats_last_gc_time_seconds",
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": ["get"]
    },
    {
      "name": "pods/go_memstats_frees",
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": ["get"]
    }
  ]
}
```


These resources are all sub-resources under MetricValueList. You can create rules through Prometheus to create or maintain sub-resources. The HPA YAML format for custom metrics differs from traditional HPA:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: demo
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: demo
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Pods
      pods:
        metric:
          name: metric-demo
        target:
          type: AverageValue
          averageValue: 10
```

In this example, `scaleTargetRef` specifies the workload.

Trigger Condition Definition

- `metrics` is an array type, supporting multiple metrics
- `metric type` can be: Object (describing k8s resources), Pods (describing metrics for each Pod), Resources (built-in k8s metrics: CPU, memory), or External (typically metrics external to the cluster)
- If the custom metric is not provided by Prometheus, you need to create a new metric through a series of operations such as creating rules in Prometheus

The main structure of a metric is as follows:

```
{
      "describedObject": {  # Described object (Pod)
        "kind": "Pod",
        "namespace": "monitoring",
        "name": "nginx-788f78d959-fd6n9",
        "apiVersion": "/v1"
      },
      "metricName": "metric-demo",
      "timestamp": "2020-02-5T04:26:01Z",
      "value": "50"
}
```

This metric data is collected and updated by Prometheus.

Custom Metrics HPA Compatibility

Custom metrics HPA YAML is actually compatible with the original core metrics (CPU). Here's how to write it:

```yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: nginx
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        targetAverageUtilization: 80
    - type: Resource
      resource:
        name: memory
        targetAverageValue: 200Mi
```

- `targetAverageValue` is the average value obtained for each Pod
- `targetAverageUtilization` is the utilization calculated from the direct value

The algorithm reference is:

```
replicas = ceil(sum(CurrentPodsCPUUtilization) / Target)
```

Updates in autoscaling/v2

autoscaling/v2 supports memory utilization:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
  namespace: default
spec:
  minReplicas: 1
  maxReplicas: 3
  metrics:
    - resource:
        name: cpu
        target:
          averageUtilization: 70
          type: Utilization
      type: Resource
    - resource:
        name: memory
        target:
          averageUtilization: 80
          type: Utilization
      type: Resource
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
```

Changes: `targetAverageUtilization` and `targetAverageValue` have been changed to `target` and converted to a combination of `xxxValue` and `type`:

- `xxxValue`: AverageValue (average value), AverageUtilization (average utilization), Value (direct value)
- `type`: Utilization (utilization), AverageValue (average value)

Notes:

- For **CPU Utilization** and **Memory Utilization** metrics, auto-scaling will only be triggered when the actual value fluctuates outside the range of ±10% of the target threshold.
- Scale-down may impact ongoing business operations; please proceed with caution.

Calculation Rules

When business metrics change, the platform will automatically calculate the target pod count that matches the business volume according to the following rules and adjust accordingly. If the business metrics continue to fluctuate, the value will be adjusted to the set **Minimum Pod Count** or **Maximum Pod Count**.

- Single Policy Target Pod Count: ceil[(sum(actual metric values)/metric threshold)] . This means that the sum of the actual metric values of all pods divided by the metric threshold, rounded up to the smallest integer that is greater than or equal to the result. For example: If there are currently 3 pods with CPU utilizations of 80%, 80%, and 90%, and the set CPU utilization threshold is 60%. According to the formula, the number of pods will be automatically adjusted to: ceil[(80%+80%+90%)/60%] = ceil 4.1 = 5 pods.
  Note:
  - If the calculated target pod count exceeds the set **Maximum Pod Count** (for example *4*), the platform will only scale up to 4 pods. If after changing the maximum pod count the metrics remain persistently high, you may need to use alternate scaling methods, such as increasing the namespace pod quota or adding hardware resources.
  - If the calculated target pod count (in the previous example *5*) is less than the pod count adjusted according to the **Scale-Up Step** (for example *10*), the platform will only scale up to 5 pods.
- Multiple Policy Target Pod Count: Take the maximum value among the results of each policy calculation.

Starting and Stopping Applications

TOC

Starting the Application

1. Access the **Container Platform**.
2. In the left navigation bar, click **Application** > **Applications**.
3. Click on the application name.
4. Click **Start**.

Stopping the Application

1. Access the **Container Platform**.
2. In the left navigation bar, click **Application** > **Applications**.
3. Click on the application name.
4. Click **Stop**.

5. Read the prompt message, and after confirming that everything is correct, click **Stop**.

Configuring VerticalPodAutoscaler (VPA)

TOC

Overview

The **Alauda Container Platform Vertical Pod Autoscaler** is based on the open-source Vertical Pod Autoscaler component. It analyzes historical resource usage data of Pods and provides quota recommendations to improve resource utilization.

For both stateless and stateful applications, the VerticalPodAutoscaler (VPA) automatically recommends—and optionally applies—more appropriate CPU and memory resource limits based on your application's needs. This helps ensure Pods have sufficient resources while improving overall cluster resource utilization.

Understanding VerticalPodAutoscalers

A VerticalPodAutoscaler (VPA) is used to recommend or automatically update CPU and memory resource requests and limits for your Pods based on their historical usage patterns.

After creating a VPA, the platform begins monitoring the resource usage of the target Pods. Once sufficient data is collected, the VPA calculates recommended resource values. Depending on its configured update mode, the VPA can either apply these recommendations automatically or simply provide them for manual review and application.

By analyzing resource usage over time, VPA helps ensure your Pods are allocated the resources they need without over-provisioning, leading to more efficient cluster-wide resource utilization.

How Does the VPA Work?

The VerticalPodAutoscaler (VPA) extends the concept of pod resource optimization. The VPA monitors the resource usage of your pods and provides recommendations for CPU and memory requests based on the observed usage patterns.

The VPA works by continuously monitoring the resource usage of your pods and updating its recommendations as new data becomes available. The VPA can operate in the following modes:

- **Off**: VPA only provides recommendations without automatically applying them.
- **Manual Adjustment**: You can manually adjust resource configurations based on VPA recommendations.

> **Important**: Elastic scaling (horizontal or vertical) performs best when cluster resources are sufficient. When resources are scarce, scaling actions may cause Pods to become stuck in a `Pending` state. Ensure your cluster has adequate resources, set reasonable quotas, and configure alerts to monitor scaling events.

Supported Features

The VPA provides resource recommendations based on historical usage patterns, allowing you to optimize your pod's CPU and memory configurations.

> **Important**: When manually applying VPA recommendations, pod recreation will occur, which can cause temporary disruption to your application. Consider applying recommendations during maintenance windows for production workloads.

> **Info**
>
> VPA recommendations and in-place Pod resizing solve different parts of resource management. VPA recommends or applies resource values based on observed usage. In-place Pod resizing is a Kubernetes mechanism for updating CPU or memory resources of a running Pod through the Pod `resize` subresource. Some VPA versions support `InPlaceOrRecreate`, where VPA can try to apply recommendations through in-place Pod resizing and fall back to Pod recreation when an in-place update is not possible. This mode requires Kubernetes in-place Pod resizing support, VPA components that enable the `InPlaceOrRecreate` feature, and a VPA CRD that accepts this `updateMode` value. For manual runtime adjustments and `resizePolicy` behavior, see Adjust Pod Resource Levels Without Pod Disruption.

Prerequisites

Before using VPA, ensure the following:

- The **Alauda Container Platform Vertical Pod Autoscaler** cluster plugin is installed in your cluster.
  - Download the latest plugin package compatible with your platform version.

  - Utilize the `violet` CLI tool to upload **Alauda Container Platform Vertical Pod Autoscalers** and **Alauda DevOps Pipelines** packages to your target cluster. For detailed instructions on using `violet`, please refer to the CLI.

- Please ensure that the monitoring components are deployed in the current cluster and are functioning properly. You can check the deployment and health status of the monitoring components by clicking on the top right corner of the platform ? > **Platform Health Status**..

Installing the Vertical Pod Autoscaler Plugin

1. Log in and navigate to the **Administrators** page.
2. Click **Marketplace** > **Cluster Plugins** to access the **Cluster Plugins** list page.
3. Locate the Alauda Container Platform Vertical Pod Autoscaler cluster plugin, click **Install**, then proceed to the installation page.

Upgrading the Vertical Pod Autoscaler Plugin

1. Log in and navigate to the **Administrators** page.
2. Click **Marketplace** > **Cluster Plugins** to access the **Cluster Plugins** list page.
3. Locate the Alauda Container Platform Vertical Pod Autoscaler cluster plugin, click **Upgrade**, then proceed to the installation page.

Creating a VerticalPodAutoscaler

Using the CLI

You can create a VerticalPodAutoscaler using the command line interface by defining a YAML file and using the `kubectl create` command. The following example shows vertical pod autoscaling for a Deployment object:

1. Create a YAML file named `vpa.yaml` with the following content:

```yaml
apiVersion: autoscaling.k8s.io/v1 1
kind: VerticalPodAutoscaler 2
metadata:
  name: my-deployment-vpa 3
  namespace: default
spec:
  targetRef:
    apiVersion: apps/v1 4
    kind: Deployment 5
    name: my-deployment 6
  updatePolicy:
    updateMode: 'Off' 7
  resourcePolicy: 8
    containerPolicies:
      - containerName: '*' 9
        mode: 'Auto' 10
```

1 Use the autoscaling.k8s.io/v1 API.

2 The name of the VPA

3 Specify the target workload object. VPA uses the workload's selector to find pods that need resource adjustment. Supported workload types include DaemonSet, Deployment, ReplicaSet, StatefulSet, ReplicationController, Job, and CronJob.

4 Specify the API version of the object to scale.

5 Specify the type of object.

6 The target resource to which the VPA applies

7 Update policy that defines how VPA applies recommendations. The updateMode can be:

- Off: Does not automatically modify pod resource requests, only provides recommendations in the VPA object.
- Initial: Only sets resource requests when creating pods, no modifications afterward.
- Recreate: Automatically sets resource requests when creating pods and evicts current pods to update to recommended resource requests. Will not use in-place updates.
- Auto: Automatically sets resource requests when creating pods and updates current pods to recommended resource requests. Currently equivalent to "Recreate" and

may cause application downtime.

- InPlaceOrRecreate: Available only when the cluster supports in-place Pod resizing, the VPA components enable the `InPlaceOrRecreate` feature, and the VPA CRD accepts this value. VPA first attempts to apply recommendations by using in-place Pod resizing, then falls back to Pod recreation when the in-place update cannot be applied. Container `resizePolicy` settings can determine whether a container restarts during the resize.

8 Resource policy that can set specific strategies for different containers. For example, setting a container's mode to "Auto" means it will calculate recommendations for that container, while "Off" means it won't calculate recommendations.

9 Apply policy to all containers in the pod.

10 Set the mode to Auto or Off. Auto means recommendations will be generated for this container, Off means no recommendations will be generated.

2. Apply the YAML file to create the VPA:

```
kubectl create -f vpa.yaml
```

Example output:

```
verticalpodautoscaler.autoscaling.k8s.io/my-deployment-vpa created
```

3. After you create the VPA, you can view the recommendations by running the following command:

```
kubectl describe vpa my-deployment-vpa
```

Example output (partial):

```
Status:
  Recommendation:
    Container Recommendations:
      Container Name:  my-container
      Lower Bound:
        Cpu:     100m
        Memory:  262144k
      Target:
        Cpu:     200m
        Memory:  524288k
      Upper Bound:
        Cpu:     300m
        Memory:  786432k
```

Using the Web Console

1. Log in and navigate to the **Container Platform**.
2. In the left navigation bar, click **Workloads** > **Deployments**.
3. Click on ***Deployment Name***.
4. Scroll down to the **Elastic Scaling** area and click **Update** on the right.
5. Select **Vertical Scaling** and configure the scaling rules.

Parameter	Description
Scaling Mode	Currently supports **Manual Scaling** mode, which provides recommended resource configurations by analyzing past resource usage. You can manually adjust according to the recommended values. Adjustments will cause pods to be recreated and restarted, so please choose an appropriate time to avoid impacting running applications. Typically, after pods have been running for more than 8 days, the recommended values will become accurate. Note that when cluster resources are insufficient, scaling may cause Pods to be in a Pending state. Please ensure that the cluster has sufficient resources or reasonable quotas, or configure alerts to monitor scaling conditions.

Parameter	Description
Target Container	Defaults to the first container of the workload. You can choose to enable resource limit recommendations for one or more containers as needed.

6. Click **Update**.

Advanced VPA Configuration

Update Policy Options

- `updateMode: "Off"` - VPA only provides recommendations without automatically applying them. You can manually apply these recommendations as needed.
- `updateMode: "Initial"` - Only sets resource requests when creating pods, no modifications afterward.
- `updateMode: "Recreate"` - Automatically sets resource requests when creating pods and evicts current pods to update to recommended values.
- `updateMode: "Auto"` - Automatically sets resource requests when creating pods and updates current pods to recommended values. Currently equivalent to "Recreate".
- `updateMode: "InPlaceOrRecreate"` - Available only when the cluster supports in-place Pod resizing, the VPA components enable the `InPlaceOrRecreate` feature, and the VPA CRD accepts this value. VPA first attempts to apply recommendations by using in-place Pod resizing, then falls back to Pod recreation when the in-place update cannot be applied. Container `resizePolicy` settings can determine whether a container restarts during the resize.
- `minReplicas: <number>` - Minimum number of replicas. Ensures this minimum number of pods remain available when the Updater evicts pods. Must be greater than 0.

Container Policy Options

- `containerName: "*"` - Apply policy to all containers in the pod.
- `mode: "Auto"` - Automatically generate recommendations for the container.
- `mode: "Off"` - Do not generate recommendations for the container.

Notes:

- VPA recommendations are based on historical usage data, so it may take several days of pod operation before recommendations become accurate.
- Pod recreation will occur when VPA recommendations are applied in Auto or Recreate mode, which can cause temporary disruption to your application. If your VPA plugin supports InPlaceOrRecreate mode, in-place updates can still fall back to Pod recreation when the requested resize cannot be applied in place.

Follow-Up Actions

After configuring VPA, the recommended values for CPU and memory resource limits of the target container can be viewed in the **Elastic Scaling** area. In the **Containers** area, select the target container tab and click the icon on the right side of **Resource Limits** to update the resource limits according to the recommended values.

Configuring CronHPA

For stateless applications with periodic fluctuations in business usage, CronHPA (Cron Horizontal Pod Autoscaler) supports adjusting the number of pods based on the time policies you set, allowing you to optimize resource usage according to predictable business patterns.

TOC

Understanding Cron Horizontal Pod Autoscalers

You can create a cron horizontal pod autoscaler to specify the number of pods you want to run at specific times according to a schedule, allowing you to prepare for predictable traffic patterns or reduce resource usage during off-peak hours.

After you create a cron horizontal pod autoscaler, the platform begins to monitor the schedule and automatically adjusts the number of pods at the specified times. This time-based scaling occurs independently of resource utilization metrics, making it ideal for applications with known usage patterns.

The CronHPA works by defining one or more schedule rules, each specifying a time (using crontab format) and a target number of replicas. When a scheduled time is reached, the CronHPA adjusts the pod count to match the specified target, regardless of the current resource utilization.

How Does the CronHPA Work?

The cron horizontal pod autoscaler (CronHPA) extends the concept of pod auto-scaling with time-based controls. The CronHPA lets you define specific times when the number of pods should change, allowing you to prepare for predictable traffic patterns or reduce resource usage during off-peak hours.

The CronHPA works by continuously checking the current time against the defined schedules. When a scheduled time is reached, the controller adjusts the number of pods to match the target replica count specified for that schedule. If multiple schedules trigger at the same time, the platform will use the rule with higher priority (the one defined earlier in the configuration).

Prerequisites

Please ensure that the monitoring components are deployed in the current cluster and are functioning properly. You can check the deployment and health status of the monitoring components by clicking on the top right corner of the platform ? > **Platform Health Status**..

Creating a Cron Horizontal Pod Autoscaler

Using the CLI

You can create a cron horizontal pod autoscaler using the command line interface by defining a YAML file and using the `kubectl create` command. The following example shows scheduled scaling for a Deployment object:

1. Create a YAML file named `cronhpa.yaml` with the following content:

   ```yaml
   apiVersion: tkestack.io/v1
   kind: CronHPA
   metadata:
     name: my-deployment-cronhpa
     namespace: default
   spec:
     scaleTargetRef:
       apiVersion: apps/v1
       kind: Deployment
       name: my-deployment
     crons:
       - schedule: '0 0 * * *'
         targetReplicas: 0
       - schedule: '0 8 * * 1-5'
         targetReplicas: 3
       - schedule: '0 18 * * 1-5'
         targetReplicas: 1
   ```

   (1) Use the tkestack.io/v1 API.

   (2) The name of the CronHPA resource.

   (3) The name of the deployment to scale.

   (4) Specify the API version of the object to scale.

   (5) Specify the type of object. The object must be a Deployment, ReplicaSet, or StatefulSet.

   (6) The target resource to which the CronHPA applies.

   (7) The cron schedule in standard crontab format (minute hour day month weekday).

   (8) The target number of replicas to scale to when the schedule is triggered.

   This example configures the deployment to:

   - Scale down to 0 replicas at midnight every day
   - Scale up to 3 replicas at 8:00 AM on weekdays (Monday-Friday)
   - Scale down to 1 replica at 6:00 PM on weekdays

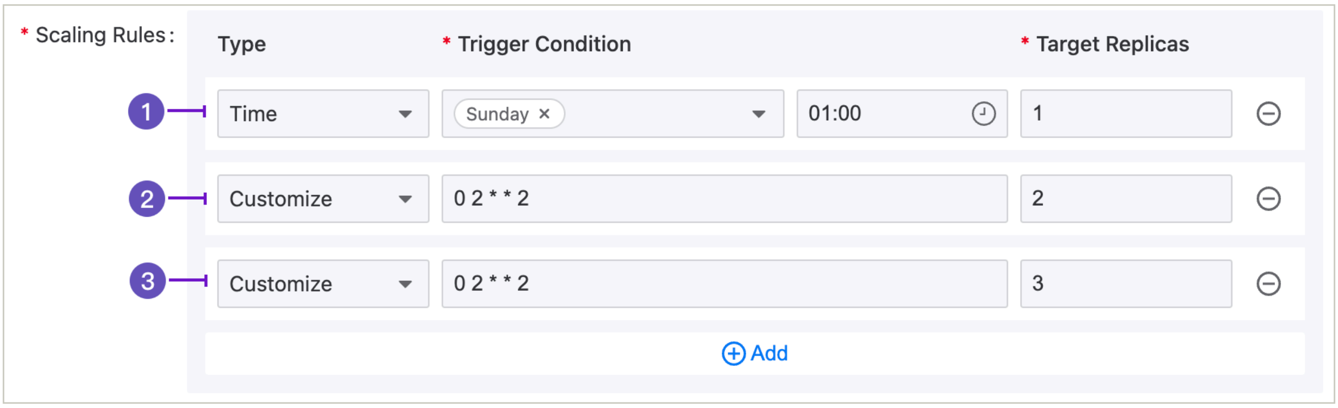
2. Apply the YAML file to create the CronHPA:

```
kubectl create -f cronhpa.yaml
```

Using the Web Console

1. Enter **Container Platform**.
2. In the left navigation bar, click **Workloads** > **Deployments**.
3. Click on ***Deployment Name***.
4. Scroll down to the **Elastic Scaling** section and click **Update** on the right.
5. Select **Scheduled Scaling**, and configure the scaling rules. When the type is **Custom**, you must provide a Crontab expression for the trigger condition, formatted as `minute hour day month week`. For detailed introduction, please refer to Writing Crontab Expressions.
6. Click **Update**.

Schedule Rule Explanation



1. Indicates that starting from 01:00 AM every Monday, only 1 pod will be retained.
2. Indicates that starting from 02:00 AM every Tuesday, only 2 pods will be retained.
3. Indicates that starting from 02:00 AM every Tuesday, only 3 pods will be retained.

Important Notes:

- When multiple rules have the same trigger time (Examples 2 and 3), the platform will execute automatic scaling based only on the rule that is higher in priority (Example 2).
- CronHPA operates independently of HPA. If both are configured for the same workload, they may conflict with each other. Consider your scaling strategy carefully.

- The schedule uses the crontab format (`minute hour day month week`) and follows the same rules as Kubernetes CronJobs.
- Time is based on the cluster's timezone setting.
- For workloads with critical availability requirements, ensure that your scheduled scaling doesn't unexpectedly reduce capacity during high-traffic periods.

Updating Applications

Custom Applications greatly facilitate the unified management of workloads, networks, storage, and configurations, but not all resources belong to the application.

- Resources added during the application creation process, or added through application updates, are by default associated with the application and do not require additional importing.
- Resources created outside the application do not belong to the application and cannot be found in the application's details. However, as long as the resource definitions meet business requirements, the business can operate normally. In this case, it is recommended that you import the resources into the application for unified management.
- **Image Management**
  - Rollout new container images with tag/patch version control
  - Configure imagePullPolicy (Always/IfNotPresent/Never)
- **Runtime Configuration**
  - Modify environment variables via ConfigMaps/Secrets
  - Update resource requests/limits (CPU/Memory)
- **Resource Orchestration**
  - Import existing Kubernetes resources (Deployments/Services/Ingresses)
  - Synchronize configurations across namespaces using `kubectl apply -f`

Resources imported into the application can benefit from the following features:

Feature	Description
Version Snapshot	When creating a version snapshot for the application, a snapshot will also be generated for the resources within the application. • If the application is rolled back, the resources will also roll back to the state in the snapshot. • If a specific version of the application is distributed, the platform will automatically create the resources recorded in the snapshot upon redeploying the application.
Deleted with Application	If an application is no longer needed, deleting the application will automatically remove all resources associated with the application, including computing components, internal routes, and inbound rules.
Easier to Find	In the application detail information, you can quickly view the resources associated with the application. For example: External traffic can access *Deployment D* through *Service S*, which belongs to *Application A*, but the corresponding access address can only be quickly found in the application details if *Service S* also belongs to *Application A*.

TOC

Importing Resources

> Batch import related resources under the namespace where the application resides; a resource can belong to only one application.

1. Enter **Container Platform**.
2. In the left navigation bar, click **Application Management** > **Native Applications**.
3. Click on ***Application Name***.
4. Click **Actions** > **Manage Resources**.
5. In the **Resource Type** at the bottom, select the type of resources to be imported.
   Note: Common resource types include Deployment, DaemonSet, StatefulSet, Job, CronJob, Service, Ingress, PVC, ConfigMap, Secret, and HorizontalPodAutoscaler, which are displayed at the top; other resources are arranged in alphabetical order, and you can quickly query specific resource types by searching keywords.
6. In the **Resources** section, select the resources to be imported.
   Attention: For **Job** type resources, only tasks created through YAML are supported for import.
7. Click **Import Resources**.

Removing/Batch Removing Resources

> Removing / batch removing resources from an application only disassociates the application from the resources and does not delete the resources.

If there are interconnections between resources under an application, removing any resource from the application will not change the associations between the resources. For example, even if *Service S* is removed from *Application A*, external traffic can still access *Deployment D* through *Service S*.

1. Enter **Container Platform**.
2. In the left navigation bar, click **Application Management** > **Native Applications**.
3. Click on ***Application Name***.
4. Click **Actions** > **Manage Resources**.
5. Click **Remove** on the right side of a resource to remove it; or select multiple resources at once, and click **Remove** at the top of the table to batch remove resources.

Exporting Applications

To standardize the export process of applications between development, testing, and production environments, and to facilitate the rapid migration of business to new environments, you can export native applications as application templates (Charts) or export simplified YAML files that can be used directly for deployment. This allows the native application to run in different environments or namespaces. You can also export YAML files to a code repository to deploy applications across clusters quickly using GitOps functionality.

TOC

Exporting Helm Charts

Procedure

1. Access the **Container Platform**.
2. In the left navigation bar, click on **Application Management** > **Native Applications**.
3. Click on the ***application name*** of the type `Custom Application`.
4. Click on **Actions** > **Export**; you can also export a specific version from the application detail page.
5. Choose one export method as needed and refer to the following instructions to configure the relevant information.

   - Exporting Helm Charts to a template repository with management permissions
     Note: The template repository is added by the platform administrator. Please contact the platform administrator to obtain a valid template repository of type **Chart** or **OCI Chart** with **Management** permissions.

Parameter	Description
Target Location	Select **Template Repository** to directly sync the template to a template repository of type **Chart** or **OCI Chart** with **Management** permissions. The project owner assigned to this **Template Repository** can directly use the template.
Template Directory	When the selected template repository type is OCI Chart, you need to select or manually input the directory for storing the Helm Chart. **Note**: When manually entering a new template directory, the platform will create this directory in the template repository, but there is a risk of the creation failing.
Version	The version number of the application template. The format should be `v<Major>.<Minor>.<Patch>`. The default value is the current application version or the current snapshot version.

Parameter	Description
Icon	Supports JPG, PNG, and GIF image formats, with a file size of no more than 500KB. Suggested dimensions are 80*60 pixels.
Description	The description will be displayed in the list of application templates within the application directory.
README	Description file. Supports editing in Markdown format and will be displayed on the details page of the application template.
NOTES	Template help file. Supports standard plaintext editing; after the deployment template is completed, it will be displayed on the template application details page.

- Exporting Helm Charts to local for manual upload to the template repository: Select **Local** as the target location and choose **Helm Chart** as the file format to generate a Helm Chart package which will be downloaded locally for offline transmission.

6. Click **Export**.

Follow-Up Actions

- If you export the Helm Chart to local, you will need to add the template to a template repository with management permissions.
- Regardless of the export method chosen, you can refer to Creating Native Applications - Template Method to create a `Template Application` type of native application in a **non-current** namespace.

Exporting YAML to Local

Steps

Method 1

1. Access the **Container Platform**.
2. In the left navigation bar, click on **Application Management** > **Native Applications**.
3. Click on ***application name***.
4. Click on **Actions** > **Export**; you can also export a specific version from the application detail page.
5. Select **Local** as the target location and **YAML** as the file format, at which point you can export a simplified YAML file that can be deployed directly in other environments.
6. Click **Export**.

Method 2

1. Access the **Container Platform**.
2. In the left navigation bar, click on **Application Management** > **Native Applications**.
3. Click on ***application name***.
4. Click on the **YAML** tab, configure settings as needed, and preview the YAML file.

Type	Description
Full YAML	By default, **Preview Simplified YAML** is not selected, displaying the YAML file with the **managedFields fields hidden**. You can preview it and export directly; you may also uncheck **Hide managedFields fields** to export the full YAML file. **Note**: Full YAML is primarily used for operations and troubleshooting and cannot be used to quickly create native applications on the platform.
Simplified YAML	Check **Preview Simplified YAML**, at which point you can preview and export a simplified YAML file that can be deployed directly in other environments.

5. Click **Export**.

Follow-Up Actions

After exporting the simplified YAML, you can refer to Creating Native Applications - YAML Method to create a `Custom Application` type of native application in a **non-current** namespace.

Exporting YAML to Code Repository (Alpha)

Precautions

- Only platform administrators and project administrators can directly export native application YAML files to the code repository.
- `Template Applications` do not support exporting Kustomize formatted application configuration files or directly exporting YAML files to the code repository; you can first **detach from the template** and convert it to a `Custom Application`.

Steps

1. Access the **Container Platform**.
2. In the left navigation bar, click on **Application Management** > **Native Applications**.
3. Click on the ***application name*** of type `Custom`.
4. Click on **Actions** > **Export**; you can also export a specific version from the application detail page.
5. Choose one export method as needed and refer to the following instructions to configure the relevant information.

   - Exporting YAML to a code repository:

Parameter	Description
Target Location	Select **Code Repository** to directly sync the YAML file to the specified Git code repository. The project owner assigned to this **Code Repository** can directly use the YAML file.
Integration Project Name	The name of the integration tool project assigned or associated with your project by the platform administrator.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>Repository Address</td><td>The repository address assigned for your use under the integrated tool project.</td></tr>
<tr><td>Export Method</td><td>Existing Branch: Export the application YAML to the selected branch.
New Branch: Create a new branch based on the selected Branch/Tag/Commit ID and export the application YAML to the new branch.
If Submit PR (Pull Request) is checked, the platform will create a new branch and submit a Pull Request.
If Automatically delete source branch after merging PR is checked, the source branch will be automatically deleted after you merge the PR in the Git code repository.</td></tr>
<tr><td>File Path</td><td>The specific location where the file should be saved in the code repository; you can also input a file path, and the platform will create a new path in the code repository based on the input.</td></tr>
<tr><td>Commit Message</td><td>Fill in commit information to identify the content of this submission.</td></tr>
<tr><td>Preview</td><td>Preview the YAML file to be submitted and compare differences with the existing YAML in the code repository, displayed with color differentiation.</td></tr>
</table>

- Exporting Kustomize-type files to local for manual upload to the code repository: Select **Local** as the target location and choose **Kustomize** as the file format to export the Kustomize-type application configuration file locally. This file supports differentiated configurations and is suitable for cross-cluster application deployments.

6. Click **Export**.

Follow-Up Actions

After exporting the YAML to a Git code repository, you can refer to Creating GitOps Applications ↗ to create a `Custom Application` type of GitOps application across clusters.

Updating and deleting Chart Applications

Due to overlapping functionality between the current template applications and native applications, and the enhanced operational capabilities available under native applications, independent management of template applications will no longer be offered in future versions. Please upgrade your currently successfully deployed template applications to native applications as soon as possible.

TOC

Important Notes

This feature is **going to be discontinued**. Please upgrade your currently successfully deployed template applications to native applications as soon as possible.

Prerequisites

Please contact the platform administrator to enable template application-related features.

Status Analysis Description

Click on ***Template Application Name*** to display detailed deployment status analysis of the Chart in the detail information.

<table>
<tr><th>Type</th><th>Reason</th></tr>
<tr><td>Initialized</td><td>Indicates the state of the Chart template download.

When the status is True, it indicates that the Chart template download was successful.
When the status is False, it indicates that the Chart template download has failed, and the reason for failure can be viewed in the message column.

ChartLoadFailed: Chart template download failed.
InitializeFailed: An exception occurred during initialization before downloading the Chart.

</td></tr>
<tr><td>Validated</td><td>Indicates the state of user permissions and dependencies verification for the Chart template.

When the status is True, it indicates that all validation checks have passed.
When the status is False, it indicates that there are validation checks that have failed, and the reason for failure can be viewed in the message column.

DependenciesCheckFailed: Chart dependency check failed.
PermissionCheckFailed: The current user lacks permissions for certain resource operations.
ConsistentNamespaceCheckFailed: When deploying the template application as a native application, the Chart contains resources that require cross-namespace deployment.

</td></tr>
<tr><td>Synced</td><td>Indicates the state of the Chart template deployment.

When the status is True, it indicates that the Chart template deployment was successful.
</td></tr>
</table>

Type	Reason
	• When the status is False, it indicates that the Chart template deployment has failed, with the reason displayed as ChartSyncFailed, and the specific reason for failure can be viewed in the message column.

Workload Rebalancing (Descheduler)

Kubernetes scheduling is a point-in-time decision. When a Pod is created, the kube-scheduler selects the most appropriate node based on the cluster state at that moment. As the cluster changes, that original decision can become stale: nodes can become over- or under-utilized, taints or labels can change, affinity rules can be updated, failed nodes can recover, and new nodes can be added.

The **Alauda Build of Descheduler** plugin helps rebalance your cluster by identifying running Pods that violate scheduling policy or are placed on less suitable nodes, evicting those Pods, and allowing the default scheduler to place the replacement Pods on more appropriate nodes.

The descheduler does not schedule replacement Pods itself. It only calls the Kubernetes eviction API. The default scheduler reschedules Pods that are managed by controllers such as Deployments, ReplicaSets, StatefulSets, or Jobs.

Typical scenarios for using the descheduler include:

- Nodes are underutilized or overutilized.
- Node labels, node taints, Pod affinity, or Pod anti-affinity requirements changed after Pods were scheduled.
- Node failures moved Pods onto a smaller set of nodes, and recovered nodes should receive workload again.
- New nodes were added and existing workloads should be redistributed.
- Pods have restarted too many times or have been running for longer than the expected lifecycle.

TOC

Pod Eviction Rules and Eligibility

To maintain cluster stability, the descheduler must be conservative about which Pods it evicts. Keep these protections in place unless you have verified that the workload will be recreated safely and that eviction is operationally acceptable.

- **Protected Pods**:
  - Pods in system or platform namespaces, such as `kube-system` and ACP platform namespaces. If you add namespace include or exclude rules, keep platform namespaces excluded.
  - Critical Pods with `priorityClassName` set to `system-cluster-critical` or `system-node-critical`.

  - Static Pods, mirrored Pods, or standalone Pods that are not managed by a controller, because these Pods will not be recreated automatically.
  - Pods associated with DaemonSets.
  - Pods with local storage, unless the policy explicitly disables the `PodsWithLocalStorage` protection.

- **Pod Disruption Budgets (PDBs)**: The descheduler uses the eviction subresource. If evicting a Pod would violate its PDB, the Pod is not evicted.
- **Eviction Order**: When multiple Pods are eligible, lower-priority Pods are selected before higher-priority Pods. For Pods with the same priority, BestEffort Pods are evicted before Burstable and Guaranteed Pods.
- **Explicit Eviction Override**: A Pod annotated with `descheduler.alpha.kubernetes.io/evict` is eligible for eviction even when some internal descheduler checks would normally skip it. This annotation does not bypass PDB protection. Use it only when you know how the Pod will be recreated.
- **Eviction Preference**: A Pod annotated with `descheduler.alpha.kubernetes.io/prefer-no-eviction` asks the descheduler to avoid evicting it. Whether this is advisory or mandatory depends on the `DefaultEvictor` `noEvictionPolicy` setting.

Understanding Descheduler Strategies

The descheduler policy enables strategy plugins. Use the strategy that matches the operational goal; avoid enabling strategies that pull the cluster in opposite directions, such as spreading workloads with `LowNodeUtilization` and compacting workloads with `HighNodeUtilization` in the same policy profile.

Goal	Strategy	Description
Remove duplicate replicas	`RemoveDuplicates`	Evicts duplicate Pods fron more than one replica is ru helps redistribute Pods aft recovery.

Goal	Strategy	Description
Spread from overloaded nodes	`LowNodeUtilization`	Finds underutilized nodes overutilized nodes so the : workloads more evenly. B calculated from Pod resou allocatable resources, not usage.
Compact workloads	`HighNodeUtilization`	Evicts Pods from underuti Pods can be scheduled or with a scheduler or autosc allocated nodes, such as Pods can be spread out a
Recycle old Pods	`PodLifeTime`	Evicts Pods older than a c configured lifecycle states running Pods or cleanup p
Move repeatedly failing Pods	`RemovePodsHavingTooManyRestarts`	Evicts Pods whose contai the configured threshold. I included.
Enforce updated taints	`RemovePodsViolatingNodeTaints`	Evicts Pods that no longe on their current nodes.
Enforce required node affinity	`RemovePodsViolatingNodeAffinity`	Evicts Pods whose curren required node affinity, suc `requiredDuringSchedul`
Enforce inter-Pod anti-affinity	`RemovePodsViolatingInterPodAntiAffinity`	Evicts Pods that violate in the cluster state or worklo
Rebalance topology domains	`RemovePodsViolatingTopologySpreadConstraint`	Evicts Pods from larger to spread constraints are vio constraints such as `DoNo` `ScheduleAnyway` only w topology constraints consi

Common strategy groups:

Group	Use When	Strategies
Affinity and taints	You changed taints, labels, node affinity, or inter-Pod anti-affinity and want running Pods to follow the new rules.	`RemovePodsViolatingNodeTaints` , `RemovePodsViolatingNodeAffinity` , `RemovePodsViolatingInterPodAntiAffinity`
Topology and duplicates	You need similar Pods spread across nodes or topology domains.	`RemoveDuplicates` , `RemovePodsViolatingTopologySpreadConstraint`
Lifecycle and utilization	You need to remove stale or unhealthy Pods and spread workloads away from overloaded nodes.	`RemovePodsHavingTooManyRestarts` , `PodLifeTime` , `LowNodeUtilization`
Compact and scale	You want to pack workloads onto fewer nodes to help scale down unused capacity.	`HighNodeUtilization`

Installing Descheduler in ACP

The Alauda Build of Descheduler is packaged and managed as a **Cluster Plugin** in ACP.

1. **Upload the Plugin Package**:

    - Obtain the `Alauda Build of Descheduler` plugin package from the Alauda Customer Portal.

- Publish it to the platform using the `violet` tool. For detailed CLI instructions, refer to CLI Tools.
- Navigate to **Administrator** > **Marketplace** > **Upload Packages** and verify the package is present under the **Cluster Plugin** tab.

2. **Install the Plugin**:

- Navigate to **Administrator** > **Marketplace** > **Cluster Plugins**.
- Select the target cluster, find the **Alauda Build of Descheduler** plugin, and click **Install**.
- Adjust installation-time configuration options in the dynamic form if required, then confirm the installation.

Configuring Scheduling Policies

After the plugin is installed, do not edit Helm values directly. The installed plugin renders Kubernetes resources, and the runtime descheduler policy is stored as YAML in the descheduler ConfigMap.

Runtime Mode

- **CronJob (Recommended)**: The descheduler runs periodically as a Job. This mode avoids running a persistent agent when the cluster state is stable. Updated policy YAML is loaded by the next scheduled Job.
- **Deployment**: The descheduler runs as a continuous Pod and reconciles the cluster at the interval configured during plugin installation. After updating the policy YAML, restart the descheduler Pod so the running process reloads the configuration.

Editing the Policy YAML

Locate the descheduler policy ConfigMap:

```
kubectl get configmap -n kube-system -l app.kubernetes.io/name=descheduler
```

Back up the current policy before editing it:

```
kubectl get configmap -n kube-system <descheduler-configmap-name> -o yaml
> descheduler-configmap.backup.yaml
```

Edit the ConfigMap returned by the previous command:

```
kubectl edit configmap -n kube-system <descheduler-configmap-name>
```

In `data.policy.yaml`, keep the existing policy and merge only the fields you need to change. Do not replace the whole policy with an example, because doing so can remove existing protections, enabled strategies, namespace filters, or eviction limits.

Example: Rebalancing Overutilized Nodes

This example enables `LowNodeUtilization` and sets the same threshold pattern commonly used for spread-style descheduling: nodes below `20%` CPU, memory, and Pod capacity are underutilized; nodes above `50%` for any of those resources are overutilized.

Relevant policy fragment:

```yaml
apiVersion: "descheduler/v1alpha2"
kind: "DeschedulerPolicy"
maxNoOfPodsToEvictTotal: 20
profiles:
  - name: default
    pluginConfig:
      - name: DefaultEvictor
        args:
          nodeFit: true
      - name: LowNodeUtilization
        args:
          thresholds:
            cpu: 20
            memory: 20
            pods: 20
          targetThresholds:
            cpu: 50
            memory: 50
            pods: 50
          evictionLimits:
            node: 5
    plugins:
      balance:
        enabled:
          - LowNodeUtilization
```

Notes:

- `thresholds` and `targetThresholds` must define the same resource keys.
- The valid percentage range is `0` to `100`.
- `thresholds` must not be greater than `targetThresholds` for the same resource.
- The strategy only runs when at least one underutilized node and one overutilized node exist.
- By default, node utilization is calculated from Pod resource requests and node allocatable resources. If you need actual-usage-based descheduling, configure supported metrics providers and `metricsUtilization` in the policy.

Common Policy Customizations

Requirement	Policy Location	Notes
Limit total disruption per run	Top-level `maxNoOfPodsToEvictTotal`, `maxNoOfPodsToEvictPerNode`, or `maxNoOfPodsToEvictPerNamespace`	Start increa evictio
Check whether replacement Pods can fit before eviction	`DefaultEvictor.args.nodeFit: true`	Preve canno accor select resou schec
Exclude or include namespaces	Strategy `args.namespaces` or `LowNodeUtilization` / `HighNodeUtilization` `args.evictableNamespaces`	Use e `excl` ACP a name
Restrict by Pod priority	`DefaultEvictor.args.priorityThreshold.name` or `DefaultEvictor.args.priorityThreshold.value`	Do no `valu` must
Evict old Pods	`PodLifeTime.args.maxPodLifeTimeSeconds`	The v secon `8640`
Evict Pods with too many restarts	`RemovePodsHavingTooManyRestarts.args.podRestartThreshold`	Use `incl` `true` shoul
Consider soft topology spread constraints	`RemovePodsViolatingTopologySpreadConstraint.args.constraints`	Add when trigge
Allow local-storage Pods to be evicted	`DefaultEvictor.args.podProtections.defaultDisabled` with `PodsWithLocalStorage`	This c protec workl tolera

Requirement	Policy Location	Notes
Protect PVC-backed Pods	`DefaultEvictor.args.podProtections.extraEnabled` with `PodsWithPVC`	This a Pods

Example: Lifecycle and Health Cleanup

This fragment evicts Pods older than 24 hours and Pods whose containers have restarted more than 100 times:

```yaml
profiles:
  - name: default
    pluginConfig:
      - name: PodLifeTime
        args:
          maxPodLifeTimeSeconds: 86400
      - name: RemovePodsHavingTooManyRestarts
        args:
          podRestartThreshold: 100
          includingInitContainers: true
    plugins:
      deschedule:
        enabled:
          - PodLifeTime
          - RemovePodsHavingTooManyRestarts
```

Example: Affinity, Taints, and Topology Drift

This fragment evicts Pods that no longer match updated node taints, required node affinity, inter-Pod anti-affinity, or hard topology spread constraints:

```yaml
profiles:
  - name: default
    pluginConfig:
      - name: RemovePodsViolatingNodeTaints
      - name: RemovePodsViolatingNodeAffinity
        args:
          nodeAffinityType:
            - requiredDuringSchedulingIgnoredDuringExecution
      - name: RemovePodsViolatingInterPodAntiAffinity
      - name: RemovePodsViolatingTopologySpreadConstraint
        args:
          constraints:
            - DoNotSchedule
    plugins:
      deschedule:
        enabled:
          - RemovePodsViolatingNodeTaints
          - RemovePodsViolatingNodeAffinity
          - RemovePodsViolatingInterPodAntiAffinity
      balance:
        enabled:
          - RemovePodsViolatingTopologySpreadConstraint
```

Verifying Installation and Evictions

1. Check Plugin Installation Status

Verify that the `ModuleInfo` has transitioned to the `Running` state:

```
kubectl get moduleinfo -l cpaas.io/module-name=descheduler
```

2. Verify the Policy ConfigMap

Check that the policy YAML contains the expected strategies and protections:

```
kubectl get configmap -n kube-system <descheduler-configmap-name> -o yaml
```

3. View Descheduler Runs and Logs

If running as a `CronJob`, list the completed or running Jobs:

```
kubectl get jobs -n kube-system -l app.kubernetes.io/name=descheduler
```

If running as a `Deployment`, confirm the running Pod and restart it after policy changes:

```
kubectl get pods -n kube-system -l app.kubernetes.io/name=descheduler
kubectl rollout restart deployment -n kube-system -l app.kubernetes.io/na
me=descheduler
```

Retrieve descheduler logs to check node evaluation, skipped Pods, and eviction actions:

```
kubectl logs -n kube-system -l app.kubernetes.io/name=descheduler --tail=
100
```

Example eviction log:

```
I0601 15:30:15.123456       1 evictions.go:160] "Evicting pod" pod="defau
lt/my-app-67d7f8d68c-xxxxx" reason="RemoveDuplicates"
```

4. Check Pod Events

When a Pod is evicted by the descheduler, inspect Pod events and the Pod description. The event reason can vary by strategy, so do not rely on a single `reason=Descheduled` filter.

```
kubectl get events -n <namespace> --field-selector involvedObject.kind=Po
d --sort-by=.lastTimestamp
kubectl describe pod -n <namespace> <pod-name>
```

Operational Guidance

- Start in a narrow scope: limit namespaces, set conservative eviction limits, and verify logs before broadening the policy.
- Ensure workloads have controllers and sufficient replicas before enabling strategies that may evict many Pods.
- Keep PDBs current for critical applications. The descheduler respects PDBs, but missing PDBs provide no disruption budget.
- Use `HighNodeUtilization` only when the scheduler or autoscaler is configured to compact Pods. Otherwise, evicted Pods may be spread again.
- Do not disable local-storage, DaemonSet, system-critical, or standalone-Pod protections unless the workload has an explicit, tested recovery path.
- For actual-usage-based decisions, confirm that metrics providers are configured and that the selected strategy consumes those metrics; otherwise utilization strategies are based on requests.

Version Management for Applications

After updating the application through the platform interface, a historical version record is automatically generated. For application updates triggered by non-interface operations, such as updating the application via API calls, you can manually create a version snapshot to record the changes.

Note: When the number of version snapshot entries exceeds 6, the platform retains only the latest 6 entries and automatically deletes the others, prioritizing the removal of the oldest version snapshot entries.

TOC

Creating a Version Snapshot

Procedure

1. Access **Container Platform**.

2. In the left navigation bar, click **Application Management** > **Native Applications**.

3. Click on ***Application Name***.

4. In the **Version Snapshot** tab, click **Create Version Snapshot**.

5. Configure the information and click **Confirm**.

   Note: You can also Distribute the Application, which allows you to distribute the version snapshot of the application as a Chart, facilitating the rapid deployment of the same application across multiple clusters and namespaces on the platform.

Rolling Back to a Historical Version

Roll back the current application's configuration to a historical version.

Procedure

1. Access **Container Platform**.

2. In the left navigation bar, click **Application Management** > **Native Applications**.

3. Click on ***Application Name***.

4. In the **Historical Versions** tab, click on ***Version Number***.

5. Click ⋮ > **Roll Back to This Version**.

6. Click **Roll Back**.

Deleting Applications

Delete an application, it simultaneously deletes the application itself and all of its directly contained Kubernetes resources. Additionally, this action severs any association the application might have had with other Kubernetes resources that were not directly part of its definition.

Using PodDisruptionBudgets

TOC

Understanding PodDisruptionBudgets

Refer to the official Kubernetes documentation: Specifying a Disruption Budget for your Application ↗

A **PodDisruptionBudget** (PDB) allows you to limit the disruption to your application when its pods need to be rescheduled for some reason such as upgrades or routine maintenance work on the Kubernetes nodes.

As a cluster administrator or application owner, you can create a PodDisruptionBudget for each application. A PDB limits the number of Pods of a replicated application that are down simultaneously from voluntary disruptions. For example, a quorum-based application would like to ensure that the number of replicas running is never brought below the number needed for a quorum. A web front end might want to ensure that the number of replicas serving load never falls below a certain percentage of the total.

By using PodDisruptionBudgets, you can:

- Ensure High Availability: Maintain a minimum number of available replicas during node maintenance, cluster upgrades, or other voluntary disruptions.
- Control Disruption Impact: Define exactly how much disruption an application can tolerate by specifying `minAvailable` or `maxUnavailable`.
- Safely Drain Nodes: During operations like `kubectl drain`, the system respects PDBs and waits until Pods can be safely evicted without violating the disruption budget.

How it works:

1. You define a PodDisruptionBudget resource targeting a specific set of Pods using a label selector.
2. You specify either `minAvailable` (minimum number or percentage of Pods that must remain available) or `maxUnavailable` (maximum number or percentage of Pods that can be concurrently unavailable).
3. When a voluntary disruption occurs (like draining a node for an OS upgrade), the eviction API checks the PDB. If evicting the Pod would violate the budget, the eviction is blocked or delayed until enough Pods are available again.

Creating PodDisruptionBudgets

Creating a PDB by using CLI

Prerequisites

- Ensure you have `kubectl` configured and connected to your cluster.
- You must have an application running with multiple replicas (like a Deployment or StatefulSet).

YAML file example

```yaml
# example-pdb.yaml
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: nginx-pdb
spec:
  minAvailable: 2 # Alternatively, you can use maxUnavailable: 1
  selector:
    matchLabels:
      app: nginx
```

Creating a PDB via YAML

```bash
# Step 1: Create the PodDisruptionBudget
kubectl apply -f example-pdb.yaml

# Step 2: Verify the PDB status
kubectl get pdb nginx-pdb
```

The output will show the allowed disruptions, minimum available, and current healthy pods.

Managing a PDB by using CLI

Viewing a PDB

- List all PDBs in a namespace:

  ```bash
  kubectl get poddisruptionbudgets
  ```

- Get detailed information about a specific PDB:

```
kubectl describe poddisruptionbudget <pdb-name>
```

Updating a PDB

You can update a PDB by editing its configuration. For instance, to change `minAvailable`:

```
kubectl edit poddisruptionbudget <pdb-name>
```

Modify the `minAvailable` or `maxUnavailable` fields, save, and exit.

Deleting a PDB

If you no longer need to protect the workload with a PDB, you can safely delete it:

```
kubectl delete poddisruptionbudget <pdb-name>
```

Deleting the PDB will not affect the actual running application, it only removes the disruption budget constraints for node maintenance and eviction processes.

When to use a PodDisruptionBudget

You should consider using a PDB when:

- Your application requires a strict minimum number of replicas to handle traffic or maintain quorum (e.g., ZooKeeper, Elasticsearch).
- You are running workloads on a cluster where cluster administrators frequently perform node upgrades or scaling operations.
- You want to ensure zero-downtime during voluntary disruptions such as node drains.

A PDB is NOT helpful for involuntary disruptions (e.g., node hardware failure, kernel panic), as those cannot be delayed or prevented by Kubernetes.

Dynamically Inject PSA Security Contexts for Workloads

When Pod Security Admission (PSA) is enabled on a namespace, any Workload (such as a Deployment or StatefulSet) created within it must comply with the specified security profile. If the `securityContext` of the Workload does not meet the requirements, the Pods generated by the Workload will be rejected during Pod admission by the Kubernetes API server, even if the Workload resource itself is accepted.

To simplify the user experience and ensure continuous compliance, you can use **Kyverno** along with the **Alauda Container Platform Compliance with Kyverno** plugin to dynamically inject the required `securityContext` fields based on the namespace's PSA policy.

TOC

Prerequisites

Before proceeding, ensure that:

- The **Alauda Container Platform Compliance with Kyverno** plugin is installed and enabled in your cluster. For installation details, please refer to Install Compliance Plugin.
- Your target namespace has a PSA label applied, such as `pod-security.kubernetes.io/enforce: restricted` or `baseline`.

How It Works

1. **Namespace Labeling**: A namespace is assigned a specific PSA level (e.g., `restricted`).
2. **Kyverno Interception**: When a user submits a Workload creation request, Kyverno intercepts the request during the admission phase.
3. **Compliance Plugin Mutation**: Kyverno works together with the Compliance plugin to read the target namespace's PSA configuration.
4. **Dynamic Patching**: Based on the PSA level, Kyverno dynamically patches the Workload's Pod template with the appropriate `securityContext` (e.g., adding `runAsNonRoot: true`, dropping specific capabilities, or setting the `seccompProfile`).

PSA Enforcement Configurations

Once the Compliance plugin is installed, you can apply the following `ClusterPolicy` configurations. These policies use context variables to query the namespace's `pod-security.kubernetes.io/enforce` label and apply the exact `securityContext` mutations required by the respective PSA profile across all container types (including `initContainers` and `ephemeralContainers`).

1. Restricted Policy Mutation

When a namespace enforces the `restricted` profile, Workloads must drop all privileges, run as non-root, and use the default seccomp profile.

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: mutate-psa-restricted
  annotations:
    policies.kyverno.io/title: Mutate Workload for PSA Restricted
    policies.kyverno.io/description: >-
      Dynamically injects the Restricted PSA securityContext into Workloads based on the target Namespace label.
spec:
  rules:
    - name: inject-restricted-security-context-controllers
      match:
        any:
          - resources:
              kinds:
                - Deployment
                - StatefulSet
                - DaemonSet
                - Job
                - CronJob
      context:
        - name: namespacePSA
          apiCall:
            urlPath: "/api/v1/namespaces/{{request.namespace}}"
            jmesPath: "metadata.labels.\"pod-security.kubernetes.io/enforce\" || 'none'"
      preconditions:
        all:
        - key: "{{namespacePSA}}"
          operator: Equals
          value: "restricted"
      mutate:
        patchStrategicMerge:
          spec:
            template:
              spec:
                securityContext:
                  runAsNonRoot: true
                  seccompProfile:
                    type: RuntimeDefault
                containers:
                  - (name): "?*"
```

```yaml
                    securityContext:
                      allowPrivilegeEscalation: false
                      capabilities:
                        drop:
                          - ALL
                initContainers:
                  - (name): "?*"
                    securityContext:
                      allowPrivilegeEscalation: false
                      capabilities:
                        drop:
                          - ALL
                ephemeralContainers:
                  - (name): "?*"
                    securityContext:
                      allowPrivilegeEscalation: false
                      capabilities:
                        drop:
                          - ALL
    - name: inject-restricted-security-context-pods
      match:
        any:
          - resources:
              kinds:
                - Pod
      context:
        - name: namespacePSA
          apiCall:
            urlPath: "/api/v1/namespaces/{{request.namespace}}"
            jmesPath: "metadata.labels.\"pod-security.kubernetes.io/enforce\" || 'none'"
      preconditions:
        all:
        - key: "{{namespacePSA}}"
          operator: Equals
          value: "restricted"
      mutate:
        patchStrategicMerge:
          spec:
            securityContext:
              runAsNonRoot: true
              seccompProfile:
                type: RuntimeDefault
            containers:
```

```
          - (name): "?*"
            securityContext:
              allowPrivilegeEscalation: false
              capabilities:
                drop:
                  - ALL
        initContainers:
          - (name): "?*"
            securityContext:
              allowPrivilegeEscalation: false
              capabilities:
                drop:
                  - ALL
        ephemeralContainers:
          - (name): "?*"
            securityContext:
              allowPrivilegeEscalation: false
              capabilities:
                drop:
                  - ALL
```

2. Baseline Policy Mutation

When a namespace enforces the `baseline` profile, Workloads are prohibited from using host namespaces and privileged containers.

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: mutate-psa-baseline
  annotations:
    policies.kyverno.io/title: Mutate Workload for PSA Baseline
    policies.kyverno.io/description: >-
      Dynamically injects the Baseline PSA securityContext into Workloads based on the target Namespace label.
spec:
  rules:
    - name: inject-baseline-security-context-controllers
      match:
        any:
          - resources:
              kinds:
                - Deployment
                - StatefulSet
                - DaemonSet
                - Job
                - CronJob
      context:
        - name: namespacePSA
          apiCall:
            urlPath: "/api/v1/namespaces/{{request.namespace}}"
            jmesPath: "metadata.labels.\"pod-security.kubernetes.io/enforce\" || 'none'"
      preconditions:
        all:
        - key: "{{namespacePSA}}"
          operator: Equals
          value: "baseline"
      mutate:
        patchStrategicMerge:
          spec:
            template:
              spec:
                hostNetwork: false
                hostIPC: false
                hostPID: false
                containers:
                  - (name): "?*"
                    securityContext:
```

```yaml
                      privileged: false
                initContainers:
                  - (name): "?*"
                    securityContext:
                      privileged: false
                ephemeralContainers:
                  - (name): "?*"
                    securityContext:
                      privileged: false
    - name: inject-baseline-security-context-pods
      match:
        any:
          - resources:
              kinds:
                - Pod
      context:
        - name: namespacePSA
          apiCall:
            urlPath: "/api/v1/namespaces/{{request.namespace}}"
            jmesPath: "metadata.labels.\"pod-security.kubernetes.io/enforce\" || 'none'"
      preconditions:
        all:
        - key: "{{namespacePSA}}"
          operator: Equals
          value: "baseline"
      mutate:
        patchStrategicMerge:
          spec:
            hostNetwork: false
            hostIPC: false
            hostPID: false
            containers:
              - (name): "?*"
                securityContext:
                  privileged: false
            initContainers:
              - (name): "?*"
                securityContext:
                  privileged: false
            ephemeralContainers:
              - (name): "?*"
                securityContext:
                  privileged: false
```

> **Note**: For the `privileged` PSA profile, no mutation is required as it allows unrestricted access.

Verification

To verify that the dynamic configuration works:

1. Create a namespace with the `restricted` PSA label:

```
kubectl create namespace test-psa
kubectl label namespace test-psa pod-security.kubernetes.io/enforce=restricted
```

2. Deploy a simple application without specifying a `securityContext`:

```
kubectl create deployment nginx --image=nginxinc/nginx-unprivileged -n test-psa
```

3. Check the created Pod's YAML definition:

```
kubectl get pod -n test-psa -l app=nginx -o yaml
```

You should see the `securityContext` fields automatically injected to meet the `restricted` profile requirements.

Handling Out of Resource Errors

TOC

Overview

This guide describes how to prevent Alauda Container Platform nodes from running out of memory (OOM) or disk space. Stable node operation is critical, especially for non-compressible resources like memory and disk. Resource exhaustion can lead to node instability.

Administrators can configure eviction policies to monitor nodes and reclaim resources before stability is compromised.

This document covers how Alauda Container Platform handles out-of-resource scenarios, including resource reclamation, pod eviction, pod scheduling, and the Out of Memory Killer. Example configurations and best practices are also provided.

> **NOTE**
>
> If swap memory is enabled on a node, memory pressure cannot be detected. Disable swap to enable memory-based evictions.

Configuring Eviction Policies

Eviction policies allow nodes to terminate pods when resources are low, reclaiming needed resources. Policies combine eviction signals and threshold values, set in the node configuration or via command line. Evictions can be:

- **Hard**: Immediate action when a threshold is exceeded.
- **Soft**: Grace period before action is taken.

Properly configured eviction policies help nodes proactively prevent resource exhaustion.

> **NOTE**
>
> When a pod is evicted, all containers in the pod are terminated, and the PodPhase transitions to Failed.

For disk pressure, nodes monitor both `nodefs` (root filesystem) and `imagefs` (container image storage).

- **nodefs/rootfs**: Used for local disk volumes, logs, and other storage (e.g., `/var/lib/kubelet`).
- **imagefs**: Used by the container runtime for images and writable layers.

> **NOTE**
>
> Without local storage isolation (ephemeral storage) or XFS quota (volumeConfig), pod disk usage cannot be limited.

Creating Eviction Policies in Node Configuration

To set eviction thresholds, edit the node configuration map under `eviction-hard` or `eviction-soft`.

Hard Eviction Example:

```
kubeletArguments:
  eviction-hard: 1
    - memory.available<100Mi 2
    - nodefs.available<10%
    - nodefs.inodesFree<5%
    - imagefs.available<15%
    - imagefs.inodesFree<10%
```

1. The type of eviction: use `eviction-hard` for hard eviction thresholds.
2. Each eviction threshold is defined as `<eviction_signal><operator><quantity>`, such as `memory.available<500Mi` or `nodefs.available<10%`.

> **NOTE**
>
> Use percentage values for `inodesFree`. Other parameters accept percentages or numeric values.

Soft Eviction Example:

```
kubeletArguments:
  eviction-soft: 1
    - memory.available<100Mi 2
    - nodefs.available<10%
    - nodefs.inodesFree<5%
    - imagefs.available<15%
    - imagefs.inodesFree<10%
  eviction-soft-grace-period: 3
    - memory.available=1m30s
    - nodefs.available=1m30s
    - nodefs.inodesFree=1m30s
    - imagefs.available=1m30s
    - imagefs.inodesFree=1m30s
```

1 The type of eviction: use `eviction-soft` for soft eviction thresholds.

2 Each eviction threshold is defined as `<eviction_signal><operator><quantity>`, such as `memory.available<500Mi` or `nodefs.available<10%`.

3 The grace period for the soft eviction. Leave the default values for optimal performance.

Restart the kubelet service for changes to take effect:

```
$ systemctl restart kubelet
```

Eviction Signals

Nodes can trigger evictions based on the following signals:

Node Condition	Eviction Signal	Description
MemoryPressure	memory.available	Available memory below threshold
DiskPressure	nodefs.available	Node root filesystem space below threshold
	nodefs.inodesFree	Free inodes below threshold

	imagefs.available	Image filesystem space below threshold
	imagefs.inodesFree	Free inodes in imagefs below threshold

- `inodesFree` must be specified as a percentage.
- Memory calculations exclude reclaimable inactive file memory.
- Do not use `free -m` in containers.

Nodes monitor these filesystems every 10 seconds. Dedicated filesystems for volumes/logs are not monitored.

> **NOTE**
>
> Before evicting pods due to disk pressure, nodes perform container and image garbage collection.

Eviction Thresholds

Eviction thresholds trigger resource reclamation. When a threshold is met, the node reports a pressure condition, preventing new pods from being scheduled until resources are reclaimed.

- **Hard thresholds**: Immediate action.
- **Soft thresholds**: Action after a grace period.

Thresholds are configured as:

```
<eviction_signal><operator><quantity>
```

Example:

- `memory.available<1Gi`
- `memory.available<10%`

Nodes evaluate thresholds every 10 seconds.

Hard Eviction Thresholds

No grace period; immediate action is taken.

Example:

```
kubeletArguments:
  eviction-hard:
    - memory.available<500Mi
    - nodefs.available<500Mi
    - nodefs.inodesFree<5%
    - imagefs.available<100Mi
    - imagefs.inodesFree<10%
```

Default Hard Eviction Thresholds

```
kubeletArguments:
  eviction-hard:
    - memory.available<100Mi
    - nodefs.available<10%
    - nodefs.inodesFree<5%
    - imagefs.available<15%
```

Soft Eviction Thresholds

Soft thresholds require a grace period. Optionally, set a maximum pod termination grace period (`eviction-max-pod-grace-period`).

Example:

```yaml
kubeletArguments:
  eviction-soft:
    - memory.available<500Mi
    - nodefs.available<500Mi
    - nodefs.inodesFree<5%
    - imagefs.available<100Mi
    - imagefs.inodesFree<10%
  eviction-soft-grace-period:
    - memory.available=1m30s
    - nodefs.available=1m30s
    - nodefs.inodesFree=1m30s
    - imagefs.available=1m30s
    - imagefs.inodesFree=1m30s
```

Configuring Allocatable Resources for Scheduling

Control how much node resource is available for scheduling by setting `system-reserved` for system daemons. Evictions occur only if pods exceed their requested resources.

- **Capacity**: Total resource on the node.
- **Allocatable**: Resource available for scheduling.

Example:

```yaml
kubeletArguments:
  eviction-hard:
    - "memory.available<500Mi"
  system-reserved:
    - "memory=1.5Gi"
```

Determine appropriate values using the node summary API.

Restart the kubelet for changes:

```
$ systemctl restart kubelet
```

Preventing Node Condition Oscillation

To avoid oscillation above/below soft eviction thresholds, set `eviction-pressure-transition-period`:

Example:

```
kubeletArguments:
  eviction-pressure-transition-period:
    - 5m
```

Default is 5 minutes. Restart services for changes.

Reclaiming Node-level Resources

When eviction criteria are met, nodes reclaim resources before evicting user pods.

- **With imagefs**:
  - If `nodefs` threshold is met: Delete dead pods/containers.
  - If `imagefs` threshold is met: Delete unused images.
- **Without imagefs**:
  - If `nodefs` threshold is met: Delete dead pods/containers, then unused images.

Pod Eviction

If a threshold and grace period are met, pods are evicted until the signal is below the threshold.

Pods are ranked for eviction by quality of service (QoS) and resource consumption.

QoS Level	Description
Guaranteed	Highest resource consumers evicted first.
Burstable	Highest resource consumers relative to request evicted first.
BestEffort	Highest resource consumers evicted first.

Guaranteed pods are only evicted if system daemons exceed reserved resources or only guaranteed pods remain.

Disk is a best-effort resource; pods are evicted one at a time to reclaim disk space, ranked by QoS and disk usage.

Quality of Service and Out of Memory Killer

If a system OOM event occurs before memory can be reclaimed, the OOM killer responds.

OOM scores are set based on QoS:

QoS Level	oom_score_adj Value
Guaranteed	-998
Burstable	min(max(2, 1000 - (1000 * memoryRequestBytes) / machineMemoryCapacityBytes), 999)
BestEffort	1000

OOM killer ends the container with the highest score. Containers with lowest QoS and highest memory usage are ended first. Containers may be restarted per node policy.

Scheduler and Out of Resource Conditions

Scheduler considers node conditions when placing pods.

Node Condition	Scheduler Behavior
MemoryPressure	BestEffort pods not scheduled.
DiskPressure	No additional pods scheduled.

Example Scenario

Operator wants:

- Node with 10Gi memory.
- Reserve 10% for system daemons.
- Evict pods at 95% utilization.

Calculation:

- `capacity = 10Gi`
- `system-reserved = 1Gi`
- `allocatable = 9Gi`

To trigger eviction below 10% available memory for 30s, or immediately below 5%:

- `system-reserved = 2Gi`
- `allocatable = 8Gi`

Configuration:

```
kubeletArguments:
  system-reserved:
    - "memory=2Gi"
  eviction-hard:
    - "memory.available<.5Gi"
  eviction-soft:
    - "memory.available<1Gi"
  eviction-soft-grace-period:
    - "memory.available=30s"
```

This prevents immediate memory pressure and eviction after scheduling.

Recommended Practices

Daemon Sets and Out of Resource Handling

Pods created by daemon sets are immediately recreated if evicted. Daemon sets should avoid best-effort pods and use guaranteed QoS to reduce eviction risk.

Health Checks

TOC

Understanding Health Checks

Refer to the official Kubernetes documentation:

- Liveness, Readiness, and Startup Probes ↗
- Configure Liveness, Readiness and Startup Probes ↗

> In Kubernetes, health checks, also known as probes, are a critical mechanism to ensure the high availability and resilience of your applications. Kubernetes uses these probes to determine the health and readiness of your Pods, allowing the system to take appropriate actions, such as restarting containers or routing traffic. Without proper health checks, Kubernetes cannot reliably manage your application's lifecycle, potentially leading to service degradation or outages.

Kubernetes offers three types of probes:

- `livenessProbe`: Detects if the container is still running. If a liveness probe fails, Kubernetes will terminate the Pod and restart it according to its restart policy.
- `readinessProbe`: Detects if the container is ready to serve traffic. If a readiness probe fails, the Endpoint Controller removes the Pod from the Service's Endpoint list until the probe succeeds.
- `startupProbe`: Specifically checks if the application has successfully started. Liveness and readiness probes will not execute until the startup probe succeeds. This is very useful for applications with long startup times.

Properly configuring these probes is essential for building robust and self-healing applications on Kubernetes.

Probe Types

Kubernetes supports three mechanisms for implementing probes:

HTTP `GET` Action

> Executes an HTTP `GET` request against the Pod's IP address on a specified port and path. The probe is considered successful if the response code is between 200 and 399.

- **Use Cases**: Web servers, REST APIs, or any application exposing an HTTP endpoint.
- **Example**:

```yaml
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 15
  periodSeconds: 20
```

`exec` Action

> Executes a specified command inside the container. The probe is successful if the command exits with status code 0.

- **Use Cases**: Applications without HTTP endpoints, checking internal application state, or performing complex health checks that require specific tools.
- **Example**:

```yaml
readinessProbe:
  exec:
    command:
      - cat
      - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

TCP `Socket` Action

> Attempts to open a TCP socket on the container's IP address and a specified port. The probe is successful if the TCP connection can be established.

- **Use Cases**: Databases, message queues, or any application that communicates over a TCP port but might not have an HTTP endpoint.
- **Example**:

```yaml
startupProbe:
  tcpSocket:
    port: 3306
  initialDelaySeconds: 5
  periodSeconds: 10
  failureThreshold: 30
```

Best Practices

- **Liveness** vs. **Readiness**:
  - **Liveness**: If your application is unresponsive, it's better to restart it. If it fails, Kubernetes will restart it.
  - **Readiness**: If your application is temporarily unable to serve traffic (e.g., connecting to a database), but might recover without a restart, use a Readiness Probe. This prevents traffic from being routed to an unhealthy instance.
- **Startup Probes for Slow Applications**: Use Startup Probes for applications that take a significant amount of time to initialize. This prevents premature restarts due to Liveness Probe failures or traffic routing issues due to Readiness Probe failures during startup.
- **Lightweight Probes**: Ensure your probe endpoints are lightweight and perform quickly. They should not involve heavy computation or external dependencies (like database calls) that could make the probe itself unreliable.
- **Meaningful Checks**: Probe checks should genuinely reflect the health and readiness of your application, not just whether the process is running. For example, for a web server, check if it can serve a basic page, not just if the port is open.
- **Adjust initialDelaySeconds**: Set initialDelaySeconds appropriately to give your application enough time to start before the first probe.
- **Tune periodSeconds and failureThreshold**: Balance the need for quick detection of failures with avoiding false positives. Too frequent probes or too low a failureThreshold can lead to unnecessary restarts or unready states.
- **Logs for Debugging**: Ensure your application logs clear messages related to health check endpoint calls and internal state to aid in debugging probe failures.
- **Combine Probes**: Often, all three probes (Liveness, Readiness, Startup) are used together to manage application lifecycle effectively.

YAML file example

```yaml
spec:
  template:
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2 # Container image
          ports:
            - containerPort: 80 # Container exposed port
          startupProbe:
            httpGet:
              path: /startup-check
              port: 8080
            initialDelaySeconds: 0 # Usually 0 for startup probes, or very small
            periodSeconds: 5
            failureThreshold: 60 # Allows 60 * 5 = 300 seconds (5 minutes) for startup
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8080
            initialDelaySeconds: 5 # Delay 5 seconds after Pod starts before checking
            periodSeconds: 10 # Check every 10 seconds
            timeoutSeconds: 5 # Timeout after 5 seconds
            failureThreshold: 3 # Consider unhealthy after 3 consecutive failures
          readinessProbe:
            httpGet:
              path: /ready
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 10
            timeoutSeconds: 5
            failureThreshold: 3
```

Health Checks configuration parameters by using web console

Common parameters

Parameters	Description
Initial Delay	`initialDelaySeconds`: Grace period (seconds) before starting probes. Default: `300`.
Period	`periodSeconds`: Probe interval (1-120s). Default: `60`.
Timeout	`timeoutSeconds`: Probe timeout duration (1-300s). Default: `30`.
Success Threshold	`successThreshold`: Minimum consecutive successes to mark healthy. Default: `0`.
Failure Threshold	`failureThreshold`: Maximum consecutive failures to trigger action: - `0`: Disables failure-based actions - Default: `5` failures → container restart.

Protocol specific parameters

Parameter	Applicable Protocols	Description
Protocol	HTTP/HTTPS	Health check protocol
Port	HTTP/HTTPS/TCP	Target container port for probing.
Path	HTTP/HTTPS	Endpoint path (e.g., `/healthz`).
HTTP Headers	HTTP/HTTPS	Custom headers (Add key-value pairs).
Command	EXEC	Container-executable check command (e.g., `sh -c "curl -I localhost:8080 | grep OK"`).

Parameter	Applicable Protocols	Description
		Note: Escape special characters and test command viability.

Troubleshooting probe failures

When a Pod's status indicates issues related to probes, here's how to troubleshoot:

Check pod events

```
kubectl describe pod <pod-name>
```

Look for events related to LivenessProbe failed, ReadinessProbe failed, or StartupProbe failed. These events often provide specific error messages (e.g., connection refused, HTTP 500 error, command exit code).

View container logs

```
kubectl logs <pod-name> -c <container-name>
```

Examine application logs to see if there are errors or warnings around the time the probe failed. Your application might be logging why its health endpoint isn't responding correctly.

Test probe endpoint manually

- **HTTP**: If possible, `kubectl exec -it <pod-name> -- curl <probe-path>:<probe-port>` or `wget` from within the container to see the actual response.
- **Exec**: Run the probe command manually: `kubectl exec -it <pod-name> -- <command-from-probe>` and check its exit code and output.
- **TCP**: Use `nc` (netcat) or `telnet` from another Pod in the same network or from the host if allowed, to test TCP connectivity: `kubectl exec -it <another-pod> -- nc -vz <pod-`

`ip> <probe-port>`.

Review probe configuration

- Double-check the probe parameters (path, port, command, delays, thresholds) in your Deployment/Pod YAML. A common mistake is an incorrect port or path.

Check application code

- Ensure your application's health check endpoint is correctly implemented and truly reflects the application's readiness/liveness. Sometimes, the endpoint might return success even when the application itself is broken.

Resource constraints

- Insufficient CPU or memory resources could cause your application to become unresponsive, leading to probe failures. Check Pod resource usage (`kubectl top pod <pod-name>`) and consider adjusting `resources` limits/requests.

Network issues

- In rare cases, network policies or CNI issues might prevent probes from reaching the container. Verify network connectivity within the cluster.

Workloads

The workload pages in this section cover project-scoped runtime resources used to run application code, finite tasks, and scheduled tasks.

Use these pages when you need to create, inspect, update, troubleshoot, or remove workloads in a selected project, cluster, and namespace context.

Some workload types can also be reached from other console views, but this section is organized around workload intent so you can choose the right operational path more quickly.

TOC

Long-running application workloads

Use these pages for workloads that stay running and represent the main runtime units of an application:

When you manage these workloads from the Applications view, treat the selected cluster and namespace as fixed runtime scope. Update operations change the workload definition inside the current scope, but they do not move the workload to a different cluster or namespace.

- Deployments
- StatefulSets
- DaemonSets
- Pods

Scheduled or finite execution workloads

Use these pages for workloads that stop after completion or run on a schedule:

- CronJobs
- Jobs

Choose the right workload page

In most cases:

- Start with **Deployments** for common stateless application workloads.
- Use **StatefulSets** when stable identity, ordered rollout, or per-replica storage matters.
- Use **DaemonSets** when the workload should run broadly on matching nodes.
- Use **CronJobs** or **Jobs** when the main question is scheduled or finite execution.
- Use **Pods** when you need to inspect the actual runtime instances behind higher-level workloads.

Deployments

Understanding Deployments

Creating Deployments

When to use a Deployment in the Applicat

Deployment review points before submissio

Managing Deployments

DaemonSets

Understanding DaemonSets

Creating DaemonSets

Managing DaemonSets

StatefulSets

Understanding

Creating Statef

When to use a

Stateful workloa

CronJobs

Pods

Containers

Adjust Pod

Deployments

TOC

Understanding Deployments

Refer to the official Kubernetes documentation: Deployments

> **Deployment** is a Kubernetes higher-level workload resource used to declaratively manage and update Pod replicas for your applications. It provides a robust and flexible way to define how your application should run, including how many replicas to maintain and how to safely perform rolling updates.

A **Deployment** is an object in the Kubernetes API that manages Pods and ReplicaSets. When you create a Deployment, Kubernetes automatically creates a ReplicaSet, which is then responsible for maintaining the specified number of Pod replicas.

By using Deployments, you can:

- Declarative Management: Define the desired state of your application, and Kubernetes automatically ensures the cluster's actual state matches the desired state.

- Version Control and Rollback: Track each revision of a Deployment and easily roll back to a previous stable version if issues arise.
- Zero-Downtime Updates: Gradually update your application using a rolling update strategy without service interruption.
- Self-Healing: Deployments automatically replace Pod instances if they crash, are terminated, or are removed from a node, ensuring the specified number of Pods are always available.

How it works:

1. You define the desired state of your application through a Deployment (e.g., which image to use, how many replicas to run).
2. The Deployment creates a ReplicaSet to ensure the specified number of Pods are running.
3. The ReplicaSet creates and manages the actual Pod instances.
4. When you update a Deployment (e.g., change the image version), the Deployment creates a new ReplicaSet and gradually replaces the old Pods with new ones according to the predefined rolling update strategy until all new Pods are running, then it removes the old ReplicaSet.

Creating Deployments

Creating a Deployment by using CLI

Prerequisites

- Ensure you have `kubectl` configured and connected to your cluster.

YAML file example

```yaml
# example-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment # Name of the Deployment
  labels:
    app: nginx # Labels for identification and selection
spec:
  replicas: 3 # Desired number of Pod replicas
  selector:
    matchLabels:
      app: nginx # Selector to match Pods managed by this Deployment
  template:
    metadata:
      labels:
        app: nginx # Pod's labels, must match selector.matchLabels
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2 # Container image
          ports:
            - containerPort: 80 # Container exposed port
          resources: # Resource limits and requests
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 200m
              memory: 256Mi
```

Creating a Deployment via YAML

```bash
# Step 1: Create Deployment via yaml
kubectl apply -f example-deployment.yaml

# Step 2: Check the Deployment status
kubectl get deployment nginx-deployment # View Deployment
kubectl get pod -l app=nginx # View Pods created by this Deployment
```

Creating a Deployment by using web console

Prerequisites

Obtain the image address. The source of the images can be from the image repository integrated by the platform administrator through the toolchain or from third-party platforms' image repositories.

- For the former, the Administrator typically assigns the image repository to your project, and you can use the images within it. If the required image repository is not found, please contact the Administrator for allocation.
- If it is a third-party platform's image repository, ensure that images can be pulled directly from it in the current cluster.
- If the image registry requires authentication, you need to configure the corresponding image pull secret. For more information, see Add ImagePullSecrets to ServiceAccount.
- When selecting an image through an OCI Connector, also select an image pull Secret for the Connector proxy, or confirm that the workload ServiceAccount already references that Secret. The two reference methods are alternatives. See Use an ImagePullSecret with an OCI Connector.

Creating a Deployment by using new web console

The new web console can expose Deployment creation from different views. Choose the entry based on whether you are working from project scope or from a concrete cluster.

- **Applications view**: Start from the current project. Open **Applications** > **Applications**, select the target **Cluster** and **Namespace**, and then choose the workload model or creation flow that results in a Deployment.
- **Clusters view**: Start from a concrete cluster. Open the target cluster first, enter the target namespace, and then navigate to **Workloads** > **Deployments** to click **Create Deployment**.

Before proceeding, confirm the following differences:

- In the **Applications** view, cluster selection is part of the project-scoped application workflow.

- In the **Clusters** view, the cluster is already fixed because you start inside a concrete cluster.
- In the **Applications** view, review whether the namespace belongs to the intended project scope.
- In the **Clusters** view, review whether the namespace-side resources in that cluster are ready for the Deployment.

Procedure - Configure Basic Info

1. **Container Platform**, navigate to **Workloads** > **Deployments** in the left sidebar.
2. Click on **Create Deployment**.
3. **Select** or **Input** an image, and click **Confirm**.

> **INFO**
>
> **Note**: When using images from the image repository integrated into web console, you can filter images by **Already Integrated**. The **Integration Project Name**, for example, images (registry-projectname), which includes the project name projectname in this web console and the project name containers in the image repository.

4. In the **Basic Info** section, configure declarative parameters for Deployment workloads:

Parameters	Description
Replicas	Defines the desired number of Pod replicas in the Deployment (default: `1`). Adjust based on workload requirements.
More > **Update Strategy**	Configures the `rollingUpdate` strategy for zero-downtime deployments: **Max surge** (`maxSurge`): • Maximum number of Pods that can exceed the desired replica count during an update. • Accepts absolute values (e.g., `2`) or percentages (e.g., `20%`). • Percentage calculation: `ceil(current_replicas × percentage)` .

Parameters	Description
	• Example: 4.1 → `5` when calculated from 10 replicas. **Max unavailable** (`maxUnavailable`): • Maximum number of Pods that can be temporarily unavailable during an update. • Percentage values cannot exceed `100%` . • Percentage calculation: `floor(current_replicas × percentage)` . • Example: 4.9 → `4` when calculated from 10 replicas. **Notes**: 1. **Default values**: `maxSurge=1` , `maxUnavailable=1` if not explicitly set. 2. **Non-running Pods** (e.g., in `Pending` / `CrashLoopBackOff` states) are considered unavailable. 3. **Simultaneous constraints**: • `maxSurge` and `maxUnavailable` cannot both be `0` or `0%` . • If percentage values resolve to `0` for both parameters, Kubernetes forces `maxUnavailable=1` to ensure update progress. **Example**: For a Deployment with 10 replicas: • `maxSurge=2` → Total Pods during update: `10 + 2 = 12` . • `maxUnavailable=3` → Minimum available Pods: `10 - 3 = 7` . • This ensures availability while allowing controlled rollout.

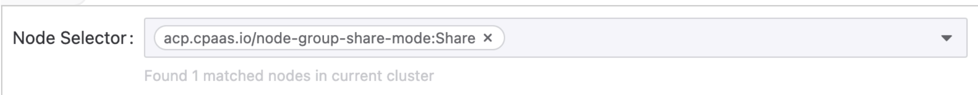
Procedure - Configure Pod

Note: In mixed-architecture clusters deploying single-architecture images, ensure proper Node Affinity Rules are configured for Pod scheduling.

1. **Pod** section, configure container runtime parameters and lifecycle management:

Parameters	Description
Volumes	Mount persistent volumes to containers. Supported volume types include `PVC` , `ConfigMap` , `Secret` , `emptyDir` , `hostPath` , and so on. For implementation details, see Volume Mounting Guide.
Pull Secret	For a private registry image, use the Secret that contains credentials for that registry. For an OCI Connector image, select the Connector proxy image pull Secret or use a ServiceAccount that already references it. Images from a platform Integration continue to use their associated credentials. See Use an ImagePullSecret with an OCI Connector.
Close Grace Period	Duration (default: `30s`) allowed for a Pod to complete graceful shutdown after receiving termination signal. - During this period, the Pod completes inflight requests and releases resources. - Setting `0` forces immediate deletion (SIGKILL), which may cause request interruptions.

2. **Node Affinity Rules**

Parameters	Description
More > **Node Selector**	Constrain Pods to nodes with specific labels (e.g. `kubernetes.io/os: linux`). 
More > **Affinity**	Define fine-grained scheduling rules based on existing. **Affinity Types**: • **Pod Affinity**: Schedule new Pods to nodes hosting specific Pods(same topology domain). • **Pod Anti-affinity**: Prevent co-location of new Pods with specific Pods. **Enforcement Modes**:

Parameters	Description
	• `requiredDuringSchedulingIgnoredDuringExecution` : Pods are scheduled *only* if rules are satisfied. • `preferredDuringSchedulingIgnoredDuringExecution` : Prioritize nodes meeting rules, but allow exceptions. **Configuration Fields**: • `topologyKey` : Node label defining topology domains (default: `kubernetes.io/hostname`). • `labelSelector` : Filters target Pods using label queries.

3. **Network Configuration**

- Kube-OVN

Parameters	Description
Bandwidth Limits	Enforce QoS for Pod network traffic: • **Egress rate limit**: Maximum outbound traffic rate (e.g., `10Mbps`). • **Ingress rate limit**: Maximum inbound traffic rate.
Subnet	Assign IPs from a predefined subnet pool. If unspecified, uses the namespace's default subnet.
Static IP Address	Bind persistent IP addresses to Pods: • Multiple Pods across Deployments can claim the same IP, but only one Pod can use it concurrently. • **Critical**: Number of static IPs must ≥ Pod replica count.

- Calico

Parameters	Description
Static IP Address	Assign fixed IPs with strict uniqueness: • Each IP can be bound to **only one Pod** in the cluster. • **Critical**: Static IP count must ≥ Pod replica count.

Procedure - Configure Containers

1. **Container** section, refer to the following instructions to configure the relevant information.

Parameters	Description
Resource Requests & Limits	• **Requests**: Minimum CPU/memory required for container operation. • **Limits**: Maximum CPU/memory allowed during container execution. For unit definitions, see Resource Units. **Namespace overcommit ratio**: • **Without overcommit ratio**: If namespace resource quotas exist: Container requests/limits inherit namespace defaults (modifiable). No namespace quotas: No defaults; custom Request. • **With overcommit ratio**: Requests auto-calculated as `Limits / Overcommit ratio` (immutable). **Constraints**: • Request ≤ Limit ≤ Namespace quota maximum. • Overcommit ratio changes require pod recreation to take effect. • Overcommit ratio disables manual request configuration. • No namespace quotas → no container resource constraints.

Parameters	Description
Extended Resources	Configure cluster-available extended resources (e.g., vGPU, pGPU).
Volume Mounts	Persistent storage configuration. See Storage Volume Mounting Instructions. **Operations**: • Existing pod volumes: Click **Add** • No pod volumes: Click **Add & Mount** **Parameters**: • `mountPath`: Container filesystem path (e.g., `/data`) • `subPath`: Relative file/directory path within volume. For `ConfigMap` / `Secret`: Select specific key • `readOnly`: Mount as read-only (default: read-write) See Kubernetes Volumes.
Ports	Expose container ports. **Example**: Expose TCP port `6379` with name `redis`. **Fields**: • `protocol`: TCP/UDP • `Port`: Exposed port (e.g., `6379`) • `name`: DNS-compliant identifier (e.g., `redis`)

Parameters	Description
Startup Commands & Arguments	Override default ENTRYPOINT/CMD: **Example 1**: Execute `top -b` - **Command**: `["top", "-b"]` - **OR** Command: `["top"]`, Args: `["-b"]` **Example 2**: Output `$MESSAGE`: `/bin/sh -c "while true; do echo $(MESSAGE); sleep 10; done"` See Defining Commands.
More > **Environment Variables**	• Static values: Direct key-value pairs • Dynamic values: Reference ConfigMap/Secret keys, pod fields (`fieldRef`), resource metrics (`resourceFieldRef`) **Note**: Env variables override image/configuration file settings.
More > **Referenced ConfigMaps**	Inject entire ConfigMap/Secret as env variables. Supported Secret types: `Opaque`, `kubernetes.io/basic-auth`.
More > **Health Checks**	• **Liveness Probe**: Detect container health (restart if failing) • **Readiness Probe**: Detect service availability (remove from endpoints if failing) See Health Check Parameters.

Parameters	Description
More > **Log Files**	Configure log paths: - Default: Collect `stdout` - File patterns: e.g., `/var/log/*.log` **Requirements**: • Storage driver `overlay2`: Supported by default • `devicemapper`: Manually mount EmptyDir to log directory • Windows nodes: Ensure parent directory is mounted (e.g., `c:/a` for `c:/a/b/c/*.log`)
More > **Exclude Log Files**	Exclude specific logs from collection (e.g., `/var/log/aaa.log`).
More > **Execute before Stopping**	Execute commands before container termination. **Example**: `echo "stop"` **Note**: Command execution time must be shorter than pod's `terminationGracePeriodSeconds`.

2. Click **Add Container** (upper right) OR **Add Init Container**.
   See Init Containers. Init Container:

   1. Start before app containers (sequential execution).
   2. Release resources after completion.
   3. Deletion allowed when:
      - Pod has >1 app container AND ≥1 init container.
      - Not allowed for single-app-container pods.

3. Click **Create**.

Reference Information

Storage Volume Mounting instructions

Type	Purpose
Persistent Volume Claim	Binds an existing PVC to request persistent storage. **Note**: Only bound PVCs (with associated PV) are selectable. Unbound PVCs will cause pod creation failures.
ConfigMap	Mounts full/partial ConfigMap data as files: • Full ConfigMap: Creates files named after keys under mount path • Subpath selection: Mount specific key (e.g., `my.cnf`)
Secret	Mounts full/partial Secret data as files: • Full Secret: Creates files named after keys under mount path • Subpath selection: Mount specific key (e.g., `tls.crt`)
Ephemeral Volumes	Cluster-provisioned temporary volume with features: • Dynamic provisioning • Lifecycle tied to pod • Supports declarative configuration **Use Case**: Temporary data storage. See Ephemeral Volumes
Empty Directory	Ephemeral storage sharing between containers in same pod: • Created on node when pod starts • Deleted with pod removal **Use Case**: Inter-container file sharing, temporary data storage. See EmptyDir
Host Path	Mounts host machine directory (must start with `/`, e.g., `/volumepath`).

Heath Checks

- Health checks YAML file example

- Health checks configuration parameters in web console

When to use a Deployment in the Applications view

Use a Deployment when the workload is primarily stateless and should be updated through replicated Pod replacement rather than stable identity or per-replica persistent state.

A Deployment is usually the better fit when:

- replica identity does not need to remain stable across replacement
- the workload should scale horizontally through interchangeable Pods
- rolling update behavior is preferred for application delivery

If the workload depends on stable ordinal identity or per-replica persistent storage, review StatefulSets instead.

Deployment review points before submission

Before creating or updating a Deployment in the Applications view, review the following carefully:

- Whether the selected project, cluster, and namespace are the intended runtime scope
- Whether the replica count matches the intended availability target
- Whether the selected image and pull credentials are valid in the target cluster
- Whether referenced ConfigMaps, Secrets, PVCs, or ServiceAccounts already exist in the same namespace
- Whether resource requests, limits, and probes match the real startup and health behavior of the application
- Whether the rollout strategy is safe for the workload and traffic pattern

Managing Deployments

Managing a Deployment by using CLI

Viewing a Deployment

- Check the Deployment was created.

```
kubectl get deployments
```

- Get details of your Deployment.

```
kubectl describe deployments
```

Updating a Deployment

Follow the steps given below to update your Deployment:

1. Let's update the nginx Pods to use the nginx:1 .16.1 image.

   ```
   kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1
   ```

   or use the following command:

   ```
   kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1
   ```

   Alternatively, you can edit the Deployment and change `.spec.template.spec.containers[0].image` from `nginx:1.14.2` to `nginx:1.16.1`:

   ```
   kubectl edit deployment/nginx-deployment
   ```

2. To see the rollout status, run:

   ```
   kubectl rollout status deployment/nginx-deployment
   ```

   Run kubectl get rs to see that the Deployment updated the Pods by creating a new ReplicaSet and scaling it up to 3 replicas, as well as scaling down the old ReplicaSet to 0 replicas.

```
kubectl get rs
```

Running get pods should now show only the new Pods:

```
kubectl get pods
```

When changing an image or OCI Connector, also review the Pod template's **Pull Secret**, `serviceAccountName`, and token validity. After the update, verify the new ReplicaSet and Pods: check the Connector annotation, confirm that the actual Pod image is rewritten to the Connector proxy, wait for the Pods to become `Running`, and review Events.

Scaling a Deployment

You can scale a Deployment by using the following command:

```
kubectl scale deployment/nginx-deployment --replicas=10
```

Rolling Back a Deployment

- Suppose that you made a typo while updating the Deployment, by putting the image name as `nginx:1.161` instead of `nginx:1.16.1`:

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.161
```

- The rollout gets stuck. You can verify it by checking the rollout status:

```
kubectl rollout status deployment/nginx-deployment
```

Deleting a Deployment

Deleting a Deployment will also delete its managed ReplicaSet and all associated Pods.

```
kubectl delete deployment <deployment-name>
```

Managing a Deployment by using web console

Viewing a Deployment

You can view a deployment to get information of your application.

1. **Container Platform**, and navigate to **Workloads** > **Deployments**.
2. Locate the Deployment you wish to view.
3. Click the deployment name to see the **Details**, **Topology**, **Logs**, **Events**, **Monitoring**, etc.

Updating a Deployment

1. **Container Platform**, and navigate to **Workloads** > **Deployments**.
2. Locate the Deployment you wish to update.
3. In the **Actions** drop-down menu, select **Update** to view the Edit Deployment page.

The update operation keeps the Deployment in its current cluster and namespace. If the workload must run in a different cluster or namespace, create or apply a separate Deployment in that target scope instead of treating update as a move operation.

Before submitting an update, confirm that changes to images, probes, environment variables, ports, volume mounts, or resource settings will not break the running workload.

Changes to the Pod template can trigger a new rollout. A successful update request does not by itself mean the rollout is healthy. The real effect appears only after the new Pod revision schedules correctly, becomes ready, and replaces the previous revision without leaving the workload unavailable.

When changing an image or OCI Connector in the web console, also review the **Pull Secret**, workload ServiceAccount, and token validity. Confirm the resulting Pods use the Connector proxy image and complete the rollout; a successful update request alone does not verify image pulling.

Deleting a Deployment

1. **Container Platform**, and navigate to **Workloads** > **Deployments**.
2. Locate the Deployment you wish to delete.

3. In the **Actions** drop-down menu, Click the **Delete** button in the operations column and confirm.

After deletion, verify that the Deployment no longer appears in the list and that any separate Service, ConfigMap, Secret, or PVC intended to remain is still present.

Troubleshooting by using CLI

When a Deployment encounters issues, here are some common troubleshooting methods.

For an OCI Connector image pull failure, verify the path in this order:

1. Check the Pod template's `connectors.<baseDomain>/connectors` annotation.
2. Check `spec.imagePullSecrets` and the `imagePullSecrets` of the workload ServiceAccount.
3. Check that the actual Pod image was rewritten to the Connector proxy.
4. Check Pod status and Events.

If Events show an HTTPS error for the original HTTP registry, the workload might not have entered the Connector proxy path. Check the Connector proxy Secret, ServiceAccount, and token before changing registry runtime settings. See Use an ImagePullSecret with an OCI Connector.

Check Deployment status

```
kubectl get deployment nginx-deployment
kubectl describe deployment nginx-deployment # View detailed events and status
```

Check ReplicaSet status

```
kubectl get rs -l app=nginx
kubectl describe rs <replicaset-name>
```

Check Pod status

```
kubectl get pods -l app=nginx
kubectl describe pod <pod-name>
```

View Logs

```
kubectl logs <pod-name> -c <container-name> # View logs for a specific container
kubectl logs <pod-name> --previous         # View logs for the previously terminated container
```

Enter Pod for debugging

```
kubectl exec -it <pod-name> -- /bin/bash # Enter the container shell
```

Check Health configuration

Ensure livenessProbe and readinessProbe are correctly configured, and your application's health check endpoints are responding properly. Troubleshooting probe failures

Check Resource Limits

Ensure container resource requests and limits are reasonable and that containers are not being killed due to insufficient resources.

Alauda Container Platform

DaemonSets

TOC

Understanding DaemonSets

Refer to the official Kubernetes documentation: DaemonSets

A **DaemonSet** is a Kubernetes controller that ensures all (or a subset of) cluster nodes run exactly one replica of a specified Pod. Unlike Deployments, DaemonSets are node-centric rather than application-centric, making them ideal for deploying cluster-wide infrastructure services such as log collectors, monitoring agents, or storage daemons.

> **WARNING**
>
> **DaemonSet Operational Notes**
>
> 1. **Behavior Characteristics**
>
>   - **Pod Distribution**: A DaemonSet deploys exactly one **Pod** replica per schedulable **Node** that matches its criteria:
>   - Deploys exactly **one Pod replica per schedulable node** matching:
>   - Matches `nodeSelector` or `nodeAffinity` criteria (if specified).
>   - Is not in the `NotReady` state.
>   - Does not have `NoSchedule` or `NoExecute` **Taints** unless corresponding **Tolerations** are configured in the **Pod Template**.
>   - **Pod Count Formula**: The **number of Pods** managed by a DaemonSet **equals** the **number of qualified Nodes**.
>   - **Dual-Role Node Handling**: Nodes serving both **Control Plane** and **Worker Node** roles will only run one **Pod** instance of the DaemonSet, regardless of their role labels, provided they are schedulable.
>
> 2. **Key Constraints** (Excluded Nodes)

- Nodes explicitly marked `Unschedulable: true` (e.g., via `kubectl cordon`).
- Nodes with a `NotReady` status.
- Nodes having incompatible **Taints** without matching Tolerations configured in the DaemonSet's **Pod Template**.

Creating DaemonSets

Creating a DaemonSet by using CLI

Prerequisites

- Ensure you have `kubectl` configured and connected to your cluster.

YAML file example

```yaml
# example-daemonSet.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  labels:
    k8s-app: fluentd-logging
spec:
  selector: # defines how the DaemonSet identifies its managed Pods. Must match `template.metadata.label`s.
    matchLabels:
      name: fluentd-elasticsearch
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template: # defines the Pod Template for the DaemonSet. Each Pod created by this DaemonSet will conform to this template
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations: # these tolerations are to have the daemonset runnable on control plane nodes, remove them if your control plane nodes should not run pods
        - key: node-role.kubernetes.io/control-plane
          operator: Exists
          effect: NoSchedule
        - key: node-role.kubernetes.io/master
          operator: Exists
          effect: NoSchedule
      containers:
        - name: fluentd-elasticsearch
          image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
          resources:
            limits:
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          volumeMounts:
            - name: varlog
              mountPath: /var/log
```

```yaml
      # it may be desirable to set a high priority class to ensure that a DaemonSet Pod
      # preempts running Pods
      # priorityClassName: important
      terminationGracePeriodSeconds: 30
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
```

Creating a DaemonSet via YAML

```
# Step 1: To create the DaemonSet defined in *example-daemonSet.yaml*, execute the following command
kubectl apply -f example-daemonSet.yaml

# Step 2: To verify the creation and status of your DaemonSet and its associated Pods:
kubectl get daemonset fluentd-elasticsearch # View DaemonSet
kubectl get pods -l name=fluentd-elasticsearch -o wide # Check Pods managed by this DaemonSet on specific nodes
```

Creating a DaemonSet by using web console

Prerequisites

Obtain the image address. The source of the images can be from the image repository integrated by the platform administrator through the toolchain or from third-party platforms' image repositories.

- For the former, the Administrator typically assigns the image repository to your project, and you can use the images within it. If the required image repository is not found, please contact the Administrator for allocation.
- If it is a third-party platform's image repository, ensure that images can be pulled directly from it in the current cluster.
- If the image registry requires authentication, you need to configure the corresponding image pull secret. For more information, see Add ImagePullSecrets to ServiceAccount.

- When selecting an image through an OCI Connector, also select an image pull Secret for the Connector proxy, or confirm that the workload ServiceAccount already references that Secret. See Use an ImagePullSecret with an OCI Connector.

Procedure - Configure Basic Info

1. **Container Platform**, navigate to **Workloads** > **DaemonSets** in the left sidebar.
2. Click **Create DaemonSet**.
3. **Select** or **Input** an image, and click **Confirm**.

> **INFO**
>
> **Note**: When using images from the image repository integrated into web console, you can filter images by **Already Integrated**. The **Integration Project Name**, for example, images (registry-projectname), which includes the project name projectname in this web console and the project name containers in the image repository.

In the **Basic Info** section, configure declarative parameters for DaemonSet workloads:

Parameters	Description
More > **Update Strategy**	Configures the `rollingUpdate` strategy for zero-downtime updates of DaemonSet Pods. **Max unavailable** (`maxUnavailable`): The maximum number of Pods that can be temporarily unavailable during an update. Accepts absolute values (e.g., 1) or percentages (e.g., 10%). **Example**: If there are 10 nodes and maxUnavailable is 10%, then floor(10 * 0.1) = 1 Pod can be unavailable. **Notes:** • **Default Values**: If not explicitly set, `maxSurge` defaults to 0 and `maxUnavailable` defaults to 1 (or 10% if `maxUnavailable` is specified as a percentage). • **Non-running Pods**: Pods in states like `Pending` or `CrashLoopBackOff` are considered unavailable.

Parameters	Description
	• **Simultaneous Constraints**: `maxSurge` and `maxUnavailable` cannot both be 0 or 0%. If percentage values resolve to 0 for both parameters, Kubernetes forces `maxUnavailable=1` to ensure update progress.

Procedure - Configure Pod

Pod section, please refer to Deployment - Configure Pod

Procedure - Configure Containers

Containers section, please refer to Deployment - Configure Containers

Procedure - Create

Click **Create**.

After clicking **Create**, the DaemonSet will:

- ✅ Automatically deploy Pod replicas to all eligible Nodes meeting:
  - `nodeSelector` criteria (if defined).
  - `tolerations` configuration (allowing scheduling on tainted nodes).
  - Node is in `Ready` state and `Schedulable: true`.
- ❌ Excluded Nodes:
  - Nodes with a `NoSchedule` taint (unless explicitly tolerated).
  - Manually cordoned Nodes (`kubectl cordon`).
  - Nodes in `NotReady` or `Unschedulable` states.

Creating a DaemonSet by using new web console

The new web console can expose DaemonSet creation from different views. Choose the entry based on whether you are working from project scope or from a concrete cluster.

- **Applications view**: Start from the current project. Open **Workloads** > **DaemonSets**, select the target **Cluster** and **Namespace**, and then choose the workload model or creation flow that results in a DaemonSet.
- **Clusters view**: Start from a concrete cluster. Open the target cluster first, enter the target namespace, and then navigate to **Workloads** > **DaemonSets** to click **Create DaemonSet**.

Managing DaemonSets

Managing a DaemonSet by using CLI

Viewing a DaemonSet

- To get a summary of all DaemonSets in a namespace.

```
kubectl get daemonsets -n <namespace>
```

- To get detailed information about a specific DaemonSet, including its events and Pod status

```
kubectl describe daemonset <daemonset-name>
```

Updating a DaemonSet

When you modify the **Pod Template** of a DaemonSet (e.g., changing the container image or adding a volume mount), Kubernetes automatically performs a rolling update by default (if `updateStrategy.type` is `RollingUpdate`, which is the default).

Before changing an image or OCI Connector, confirm that the Connector proxy image pull Secret is available directly to the Pod or through its ServiceAccount. Every target node's container runtime must be able to access the Connector proxy address. During the update, verify the Connector annotation, rewritten proxy image, status, and Events for Pods on each node.

- First, edit the YAML file (e.g., `example-daemonset.yaml`) with the desired changes, then apply it:

```
kubectl apply -f example-daemonset.yaml
```

- You can monitor the progress of the rolling update:

```
kubectl rollout status daemonset/<daemonset-name>
```

Deleting a DaemonSet

To delete a DaemonSet and all the Pods it manages:

```
kubectl delete daemonset <daemonset-name>
```

Managing a DaemonSet by using web console

Viewing a DaemonSet

1. **Container Platform**, and navigate to **Workloads** > **DaemonSets**.
2. Locate the DaemonSet you wish to view.
3. Click the DaemonSet name to see the **Details**, **Topology**, **Logs**, **Events**, **Monitoring**, etc.

Updating a DaemonSet

1. **Container Platform**, and navigate to **Workloads** > **DaemonSets**.
2. Locate the DaemonSet you wish to update.
3. In the **Actions** drop-down menu, select **Update** to view the Edit DaemonSet page, you can update `Replicas`, `image`, `updateStrategy`, etc.

When changing an image or OCI Connector, also review the Connector proxy image pull Secret, workload ServiceAccount, and token validity. Verify the actual Pod image and Events on every target node during the rollout.

Deleting a DaemonSet

1. **Container Platform**, and navigate to **Workloads** > **DaemonSets**.

2. Locate the DaemonSet you wish to delete.
3. In the **Actions** drop-down menu, Click the **Delete** button in the operations column and confirm.

StatefulSets

TOC

Understanding StatefulSets

Refer to the official Kubernetes documentation: StatefulSets ↗

StatefulSet is a Kubernetes workload API object designed to manage stateful applications by providing:

- **Stable network identity**: DNS hostname `<statefulset-name>-<ordinal>.<service-name>.ns.svc.cluster.local`.
- **Stable persistent storage**: via `volumeClaimTemplates`.
- **Ordered deployment/scaling**: sequential Pod creation/deletion: Pod-0 → Pod-1 → Pod-N.
- **Ordered rolling updates**: reverse-ordinal Pod updates: Pod-N → Pod-0.

In distributed systems, multiple StatefulSets can be deployed as discrete components to deliver specialized stateful services (e.g., *Kafka brokers*, *MongoDB shards*).

Creating StatefulSets

Creating a StatefulSet by using CLI

Prerequisites

- Ensure you have `kubectl` configured and connected to your cluster.

YAML file example

```yaml
# example-statefulset.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: 'nginx' # this headless Service is responsible for the network identity of the Pods
  replicas: 3 # defines the desired number of Pod replicas (default: 1)
  minReadySeconds: 10 # by default is 0
  template: # defines the Pod template for the StatefulSet
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: nginx
          image: registry.k8s.io/nginx-slim:0.24
          ports:
            - containerPort: 80
              name: web
          volumeMounts:
            - name: www
              mountPath: /usr/share/nginx/html
  volumeClaimTemplates: # defines PersistentVolumeClaim (PVC) templates. Each Pod gets a unique PersistentVolume (PV) dynamically provisioned based on these templates.
    - metadata:
        name: www
      spec:
        accessModes: ['ReadWriteOnce']
        storageClassName: 'my-storage-class'
        resources:
          requests:
            storage: 1Gi
---
# example-service.yaml
apiVersion: v1
kind: Service
```

```yaml
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - port: 80
      name: web
  clusterIP: None
  selector:
    app: nginx
```

Creating a StatefulSet via YAML

```
# Step 1: To create the StatefulSet defined in *example-statefulset.yaml
*, execute the following command
kubectl apply -f example-statefulset.yaml

# Step 2: To verify the creation and status of your StatefulSet and its a
ssociated Pods and PVCs:
kubectl get statefulset web # View StatefulSet
kubectl get pods -l app=nginx # Check Pods managed by this StatefulSet
kubectl get pvc -l app=nginx # Check PVCs created by volumeClaimTemplates
```

Creating a StatefulSet by using web console

Prerequisites

Obtain the image address. The source of the images can be from the image repository integrated by the platform administrator through the toolchain or from third-party platforms' image repositories.

- For the former, the Administrator typically assigns the image repository to your project, and you can use the images within it. If the required image repository is not found, please contact the Administrator for allocation.
- If it is a third-party platform's image repository, ensure that images can be pulled directly from it in the current cluster.
- If the image registry requires authentication, you need to configure the corresponding image pull secret. For more information, see Add ImagePullSecrets to ServiceAccount.

- When selecting an image through an OCI Connector, also select an image pull Secret for the Connector proxy, or confirm that the workload ServiceAccount already references that Secret. See Use an ImagePullSecret with an OCI Connector.

Creating a StatefulSet by using new web console

The new web console can expose StatefulSet creation from different views. Choose the entry based on whether you are working from project scope or from a concrete cluster.

- **Applications view**: Start from the current project. Open **Workloads** > **StatefulSets**, select the target **Cluster** and **Namespace**, and then choose the workload model or creation flow that results in a StatefulSet.
- **Clusters view**: Start from a concrete cluster. Open the target cluster first, enter the target namespace, and then navigate to **Workloads** > **StatefulSets** to click **Create StatefulSet**.

Procedure - Configure Basic Info

1. **Container Platform**, navigate to **Workloads** > **StatefulSets** in the left sidebar.
2. Click **Create StatefulSet**.
3. **Select** or **Input** an image, and click **Confirm**.

> **INFO**
>
> **Note**: When using images from the image repository integrated into web console, you can filter images by **Already Integrated**. The **Integration Project Name**, for example, images (registry-projectname), which includes the project name projectname in this web console and the project name containers in the image repository.

In the **Basic Info** section, configure declarative parameters for StatefulSet workloads:

Parameters	Description
Replicas	Defines the desired number of Pod replicas in the StatefulSet (default: 1). Adjust based on workload requirements and expected request volume.
Update Strategy	Controls phased updates during StatefulSet rolling updates. The `RollingUpdate` strategy is default and recommended.

Parameters	Description
	Partition value: Ordinal threshold for Pod updates. • Pods with index ≥ `partition` update immediately. • Pods with index < `partition` retain previous spec. **Example**: • `Replicas=5` (Pods: web-0 ~ web-4) • `Partition=3` (Updates web-3 & web-4 only)

Volume Claim Templates	`volumeClaimTemplates` is a critical feature of StatefulSets that enables dynamic per-Pod persistent storage provisioning. Each Pod replica in a StatefulSet automatically gets its own dedicated PersistentVolumeClaim (PVC) based on predefined templates.
	• 1. **Dynamic PVC Creation**: Automatically creates unique PVCs for each Pod with a naming pattern: `<statefulset-name>-<claim-template-name>-<pod-ordinal>`. **Example**: `web-www-web-0`, `web-www-web-1`.
	• 2. **Access Modes**: Supports all Kubernetes access modes.
	  ◦ ReadWriteOnce (RWO - single-node read/write)
	  ◦ ReadOnlyMany (ROX - multi-node read-only)
	  ◦ ReadWriteMany (RWX - multi-node read/write).
	• 3. **Storage Class**: Specify the storage backend via storageClassName. It uses the cluster's default StorageClass if unspecified. Supports various cloud/on-prem storage types (e.g., SSD, HDD).
	• 4. **Capacity**: Configure storage capacity through resources.requests.storage. **Example**: 1Gi. Supports dynamic volume expansion if enabled by the StorageClass.

Procedure - Configure Pod

Pod section, please refer to Deployment - Configure Pod

Procedure - Configure Containers

Containers section, please refer to Deployment - Configure Containers

Procedure - Create

Click **Create**.

When to use a StatefulSet in the Applications view

Use a StatefulSet when the workload depends on per-replica identity, stable naming, ordered lifecycle behavior, or persistent storage that should stay associated with a specific replica.

A StatefulSet is usually the better fit when:

- each replica should keep a stable ordinal identity
- the workload depends on stable network naming or a related headless Service
- each replica needs its own persistent volume claim
- startup, shutdown, or update order matters to the application

If the workload can be treated as interchangeable stateless replicas, review Deployments instead.

Stateful workload review points

Before creating or updating a StatefulSet, review the following carefully:

- Whether the selected project, cluster, and namespace are the intended runtime scope
- Whether each replica requires stable identity
- Whether persistent storage is required for each replica
- Whether the workload depends on ordered startup, shutdown, or scaling behavior
- Whether the referenced storage settings are available in the target environment

- Whether configuration changes could affect data continuity or replica replacement behavior
- Whether the workload also expects a stable network identity through a related Service

Heath Checks

- Health checks YAML file example
- Health checks configuration parameters in web console

Managing StatefulSets

Managing a StatefulSet by using CLI

Viewing a StatefulSet

You can view a StatefulSet to get information of your application.

- Check the StatefulSet was created.

  ```
  kubectl get statefulsets
  ```

- Get details of your StatefulSet.

  ```
  kubectl describe statefulsets
  ```

Scaling a StatefulSet

- To change the number of replicas for an existing StatefulSet:

  ```
  kubectl scale statefulset <statefulset-name> --replicas=<new-replica-co
  unt>
  ```

- Example:

```
kubectl scale statefulset web --replicas=5
```

Updating a StatefulSet (Rolling Update)

When you modify the Pod template of a StatefulSet (e.g., changing the container image), Kubernetes performs a rolling update by default (if updateStrategy is set to RollingUpdate, which is the default).

Before changing an image or OCI Connector, confirm that each new Pod revision can obtain a valid Connector proxy image pull Secret directly or through its ServiceAccount. During the ordered update, verify each Pod's Connector annotation, rewritten proxy image, status, and Events. If an ordinal Pod enters `ImagePullBackOff`, stop the rollout and check the Secret, ServiceAccount, and token validity before continuing.

- First, edit the YAML file (e.g., example-statefulset.yaml) with the desired changes, then apply it:

  ```
  kubectl apply -f example-statefulset.yaml
  ```

- Then, you can monitor the progress of the rolling update:

  ```
  kubectl rollout status statefulset/<statefulset-name>
  ```

Deleting a StatefulSet

To delete a StatefulSet and its associated Pods:

```
kubectl delete statefulset <statefulset-name>
```

By default, deleting a StatefulSet does not delete its associated PersistentVolumeClaims (PVCs) or PersistentVolumes (PVs) to prevent data loss. To also delete the PVCs, you must do so explicitly:

```
kubectl delete pvc -l app=<label-selector-for-your-statefulset> # Exampl
e: kubectl delete pvc -l app=nginx
```

Alternatively, if your `volumeClaimTemplates` use a `StorageClass` with a `reclaimPolicy` of `Delete`, the PVs and underlying storage will be deleted automatically when the PVCs are deleted.

Managing a StatefulSet by using web console

Viewing a StatefulSet

1. **Container Platform**, and navigate to **Workloads** > **StatefulSets**.
2. Locate the StatefulSet you wish to view.
3. Click the statefulSet name to see the **Details**, **Topology**, **Logs**, **Events**, **Monitoring**, etc.

Updating a StatefulSet

1. **Container Platform**, and navigate to **Workloads** > **StatefulSets**.
2. Locate the StatefulSet you wish to update.
3. In the **Actions** drop-down menu, select **Update** to view the Edit StatefulSet page, you can update `Replicas`, `image`, `updateStrategy`, etc.

When changing an image or OCI Connector, also review the Connector proxy image pull Secret, workload ServiceAccount, and token validity. Observe the ordered Pod replacement and verify the actual image rewrite and Events for each new Pod.

Deleting a StatefulSet

1. **Container Platform**, and navigate to **Workloads** > **StatefulSets**.
2. Locate the StatefulSet you wish to delete.
3. In the **Actions** drop-down menu, Click the **Delete** button in the operations column and confirm.

StatefulSet deletion and scaling can have storage consequences that differ from stateless workloads. Before deleting or scaling down, confirm that the removed replicas do not still hold required data or cluster membership.

CronJobs

TOC

Understanding CronJobs

Refer to the official Kubernetes documentation:

- CronJobs
- Running Automated Tasks with a CronJob

CronJob define tasks that run to completion and then stop. They allow you to run the same Job multiple times according to a schedule.

A **CronJob** is a type of workload controller in Kubernetes. You can create a CronJob through the web console or CLI to periodically or repeatedly run a non-persistent program, such as scheduled backups, scheduled clean-ups, or scheduled email dispatches.

Creating CronJobs

Creating a CronJob by using CLI

Prerequisites

- Ensure you have `kubectl` configured and connected to your cluster.

YAML file example

```yaml
# example-cronjob.yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox:1.28
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

Creating a CronJobs via YAML

```
kubectl apply -f example-cronjob.yaml
```

Creating CronJobs by using web console

Prerequisites

Obtain the image address. Images can be sourced from an image registry integrated by the platform administrator via a toolchain, or from third-party image registries.

- For images from an integrated registry, the Administrator typically assigns the image registry to your project, allowing you to use the images within it. If the required image registry is not found, please contact the Administrator for allocation.
- If using a third-party image registry, ensure that images can be pulled directly from it within the current cluster.

- If the image registry requires authentication, you need to configure the corresponding image pull secret. For more information, see Add ImagePullSecrets to ServiceAccount.
- When selecting an image through an OCI Connector, also select an image pull Secret for the Connector proxy, or confirm that the workload ServiceAccount already references that Secret. See Use an ImagePullSecret with an OCI Connector.

Creating CronJobs by using new web console

Creation entry differences between the Applications and Clusters views

The new web console can expose CronJob creation from different views. Choose the entry based on whether you are working from project scope or from a concrete cluster.

- **Applications view**: Start from the current project. Open **Workloads** > **CronJobs**, select the target **Cluster** and **Namespace**, and then choose the creation flow that results in a CronJob.
- **Clusters view**: Start from a concrete cluster. Open the target cluster first, enter the target namespace, and then navigate to **Workloads** > **CronJobs** to click **Create CronJob**.

Procedure - Configure basic info

1. **Container Platform**, navigate to **Workloads** > **CronJobs** in the left sidebar.
2. Click on **Create CronJob**.
3. **Select** or **Input** an image, and click **Confirm**.
   Note: Image filtering is available only when using images from the platform's integrated image registry. For example, an integrated project name like containers (registry-projectname) indicates the platform's project name projectname and the image registry's project name containers.
   For an OCI Connector image, confirm that the Connector proxy image pull Secret is selected by the Pod template or inherited from its ServiceAccount. Creating the CronJob object does not prove that a later Job can pull the image; the token must still be valid when the Job Pod starts.
4. In the **Cron Configuration** section, configure the task execution method and associated parameters.

Execute Type:

- **Manual**: Manual execution requires explicit manual triggering for each task run.
- **Scheduled**: Scheduled execution requires configuring the following scheduling parameters:

Parameter	Description
Schedule	Define the cron schedule using Crontab syntax. The CronJob controller calculates the next execution time based on the selected timezone. **Notes**: • For Kubernetes clusters < v1.25: Timezone selection is unsupported; schedules MUST use UTC. • For Kubernetes clusters ≥ v1.25: Timezone-aware scheduling is supported (default: user's local timezone).
Concurrency Policy	Specify how concurrent Job executions are handled (`Allow` , `Forbid` , or `Replace` per K8s spec).

Job History Retention:

- Set retention limits for completed Jobs:
  - **History Limits**: Successful jobs history limit (default: 20)
  - **Failed Jobs**: Failed jobs history limit** (default: 20)
- When retention limits are exceeded, the oldest jobs are garbage-collected first.

5. In the **Job Configuration** section, select the job type. A CronJob manages Jobs composed of Pods. Configure the Job template based on your workload type:

Parameter	Description
Job Type	Select Job completion mode (`Non-parallel` , `Parallel with fixed completion count` , or `Indexed Job` per K8s Job patterns).
Backoff Limit	Set the maximum number of retry attempts before marking a Job as failed.

Procedure - Configure Pod

- **Pod** section, please refer to Deployment - Configure Pod

Procedure - Configure Containers

- **Container** section, please refer to Deployment - Configure Containers

Create

- Click **Create**.

Execute Immediately

Locate the CronJob resource

- **web console**: **Container Platform**, and navigate to **Workloads** > **CronJobs** in the left sidebar.
- **CLI**:

```
kubectl get cronjobs -n <namespace>
```

Initiate ad-hoc execution

- **web console**: **Execute Immediately**

  1. Click the vertical ellipsis (⋮) on the right side of the cronjob list.

2. Click **Execute Immediately**. (Alternatively, from the CronJob details page, click Actions in the upper-right corner and select **Execute Immediately**).

- **CLI**:

```
kubectl create job --from=cronjob/<cronjob-name> <job-name> -n <namespace>
```

Verify Job details:

```
kubectl describe job/<job-name> -n <namespace>
kubectl logs job/<job-name> -n <namespace>
```

For a Job that uses an OCI Connector, inspect the generated Pod rather than relying only on the CronJob or Job status. Verify the Connector annotation, `imagePullSecrets` or ServiceAccount inheritance, the rewritten Connector proxy image, Pod `Running` or `Completed` status, and Events. Treat an expired token as an authentication-material issue and rotate or recreate the Secret before the next execution.

Monitor execution status

Status	Description
Pending	The Job has been created but not yet scheduled.
Running	The Job Pod(s) are actively executing.
Succeeded	All Pods associated with the Job completed successfully (exit code 0).
Failed	At least one Pod associated with the Job terminated unsuccessfully (non-zero exit code).

When updating a CronJob image or OCI Connector, review the Pod template's Connector proxy image pull Secret or ServiceAccount and token validity. Verify these values again on the next generated Job Pod, because an update accepted by the API does not prove that the scheduled Pod can pull its image.

Deleting CronJobs

Deleting CronJobs by using web console

1. **Container Platform**, and navigate to **Workloads** > **CronJobs**.
2. Locate the CronJobs you wish to delete.
3. In the **Actions** drop-down menu, Click the **Delete** button and confirm.

Deleting CronJobs by using CLI

```
kubectl delete cronjob <cronjob-name>
```

Jobs

TOC

Understanding Jobs

Refer to the official Kubernetes documentation: Jobs

A **Job** provide different ways to define tasks that run to completion and then stop. You can use a Job to define a task that runs to completion, just once.

- **Atomic Execution Unit**: Each Job manages one or more Pods until successful completion.
- **Retry Mechanism**: Controlled by `spec.backoffLimit` (default: 6).
- **Completion Tracking**: Use `spec.completions` to define required success count.

YAML file example

```yaml
# example-job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: data-processing-job
spec:
  completions: 1 # Number of successful completions required
  parallelism: 1 # Maximum parallel Pods
  backoffLimit: 3 # Maximum retry attempts
  template:
    spec:
      restartPolicy: Never # Job-specific policy (Never/OnFailure)
      containers:
        - name: processor
          image: alpine:3.14
          command: ['/bin/sh', '-c']
          args:
            - echo "Processing data..."; sleep 30; echo "Job completed"
```

Execution Overview

Each Job execution in Kubernetes creates a dedicated Job object, enabling users to:

- **Creating a job via**

```
kubectl apply -f example-job.yaml
```

- **Track job lifecycle via**

```
kubectl get jobs
```

- **Inspect execution details via**

```
kubectl describe job/<job-name>
```

- **View Pod logs via**

```
kubectl logs <pod-name>
```

Pods

TOC

Understanding Pods

Refer to the official Kubernetes website documentation: Pod ↗

A **Pod** is the smallest deployable unit of computing that you can create and manage in Kubernetes. A **Pod** (as in a pod of whales or a pea pod) is a group of one or more containers (such as containers), with shared storage and network resources, and a specification for how to run the containers. **Pods** are the fundamental building blocks on which all higher-level controllers (like **Deployments**, **StatefulSets**, **DaemonSets**) are built.

YAML file example

pod-example.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: nginx
spec:
  containers:
    - name: nginx
      image: nginx:latest # The container image to use.
      ports:
        - containerPort: 80 # Container ports exposed.
      resources: # Defines CPU and memory requests and limits for the con
tainer.
        requests:
          cpu: '100m'
          memory: '128Mi'
        limits:
          cpu: '200m'
          memory: '256Mi'
```

Managing a Pod by using CLI

While Pods are often managed by higher-level controllers, direct kubectl operations on Pods are useful for troubleshooting, inspection, and ad-hoc tasks.

Adjusting Pod Resources

To update CPU or memory requests and limits for a running Pod through the Kubernetes `resize` subresource, see Adjust Pod Resource Levels Without Pod Disruption.

Viewing a Pod

- To list all Pods in the current namespace:

```
kubectl get pods
```

- To list all Pods across all namespaces:

```
kubectl get pods --all-namespaces
# Or a shorter version:
kubectl get pods -A
```

- To get detailed information about a specific Pod:

```
kubectl describe pod <pod-name> -n <namespace>

# Example
kubectl describe pod my-nginx-pod -n default
```

Viewing a Pod Logs

- To stream logs from a container within a Pod (useful for debugging):

```
kubectl logs <pod-name> -n <namespace>
```

- If a Pod has multiple containers, you must specify the container name:

```
kubectl logs <pod-name> -c <container-name> -n <namespace>
```

- To follow the logs (stream new logs as they appear):

```
kubectl logs -f <pod-name> -n <namespace>
```

Executing Commands in a Pod

To execute a command inside a specific container within a Pod (useful for debugging, like accessing a shell):

```
kubectl exec -it <pod-name> -n <namespace> -- <command>

# Example (to get a shell):
kubectl exec -it my-nginx-pod -n default -- /bin/bash
```

Port Forwarding to a Pod

To forward a local port to a port on a Pod, allowing direct access to a service running inside the Pod from your local machine (useful for testing or direct access without exposing the service externally):

```
kubectl port-forward <pod-name> <local-port>:<pod-port> -n <namespace>

#Example
kubectl port-forward my-nginx-pod 8080:80 -n default
```

After running this command, you can access the Nginx web server running in my-nginx-pod by visiting localhost:8080 in your web browser.

Deleting a Pod

- To delete a specific Pod:

```
kubectl delete pod <pod-name> -n <namespace>

# Example
kubectl delete pod my-nginx-pod -n default
```

- To delete multiple Pods by their names:

```
kubectl delete pod <pod-name-1> <pod-name-2> -n <namespace>
```

- To delete Pods based on a label selector (e.g., delete all Pods with the label app=nginx):

```
kubectl delete pods -l app=nginx -n <namespace>
```

Managing a Pod by using web console

Viewing a Pod

The platform interface provides various information about the pods for quick reference.

Procedure

1. **Container Platform**, navigate to **Workloads** > **Pods** in the left sidebar.
2. Locate the Pod you wish to view.
3. Click the deployment name to see the **Details**, **YAML**, **Configuration**, **Logs**, **Events**, **Monitoring**, etc.

Pod Parameters

Below are some parameter explanations:

Parameter	Description
Resource Requests & Limits	**Resource Requests** and **Limits** define the CPU and memory consumption boundaries for Containers within a Pod, which then aggregate to form the Pod's overall resource profile. These values are crucial for Kubernetes' scheduler to efficiently place Pods on Nodes and for the kubelet to enforce resource governance. • **Requests**: The minimum guaranteed CPU/memory required for a container to be **scheduled** and run. This value is used by the Kubernetes

Parameter	Description
	scheduler to decide which **Node** a Pod can run on. • **Limits**: The maximum CPU/memory a container is allowed to consume during its execution. Exceeding CPU limits results in throttling, while exceeding memory limits leads to the container being terminated (Out Of Memory - OOM Killed). For detailed unit definitions (e.g., `m` for milliCPU, `Mi` for mebibytes), refer to Resource Units. **Pod-Level Resource Calculation Logic** The effective CPU and memory Requests and Limits values for a Pod are derived from the sum and maximum of its individual container specifications. The calculation method for Pod-level Requests and Limits is analogous; this document illustrates the logic using Limit values as an example. When a Pod contains only standard containers (business containers): The Pod's effective CPU/Memory Limit value is the sum of the CPU/Memory Limit values of all containers within the Pod. **Example**: If a Pod includes two containers with CPU/Memory Limits of 100m/100Mi and 50m/200Mi respectively, the Pod's aggregated CPU/Memory Limit will be 150m/300Mi. When a Pod contains both initContainers and standard containers: The calculation steps for the Pod's CPU/Memory Limit values are as follows: • 1. Determine the maximum CPU/Memory Limit value among all initContainers. • 2. Calculate the sum of CPU/Memory Limit values of all standard containers. • 3. Compare the results from step 1 and step 2. The Pod's comprehensive CPU/Memory Limit will be the maximum of the CPU values (from initContainers max and containers sum) and the maximum of the Memory values (from initContainers max and containers sum). **Calculation Example**: If a Pod contains two initContainers with CPU/Memory Limits of 100m/200Mi and 200m/100Mi, the maximum effective CPU/Memory Limit for the initContainers would be 200m/200Mi. Simultaneously, if the Pod also contains two standard containers with CPU/Memory Limits of 100m/100Mi and 50m/200Mi, the total aggregated Limit for the standard containers will be 150m/300Mi. Therefore, the Pod's comprehensive

Parameter	Description
	CPU/Memory Limit would be max(200m, 150m) for CPU and max(200Mi, 300Mi) for Memory, resulting in 200m/300Mi.
Source	The Kubernetes workload controller that manages this Pod's life cycle. This includes **Deployments**, **StatefulSets**, **DaemonSets**, **Jobs**.
Restart	The number of times the Container within the **Pod** has restarted since the **Pod** was started. A high restart count often indicates an issue with the application or its environment.
Node	The name of the Kubernetes Node where the Pod is currently scheduled and running.
Service Account	A Service Account is a Kubernetes object that provides an identity for processes and services running inside a Pod, allowing them to authenticate and access the Kubernetes APIServer. This field is typically visible only when the currently logged-in user has the platform administrator role or the platform auditor role, enabling the viewing of the Service Account's YAML definition.

Deleting a Pod

Deleting pods may affect the operation of computing components; please proceed with caution.

Use Cases

- Restore the pods to its desired state promptly: If a pods remains in a state that affects business operations, such as `Pending` or `CrashLoopBackOff`, manually deleting the pods after addressing the error message can help it quickly return to its desired state, such as `Running`. At this time, the deleted pods will be rebuilt on the current node or rescheduled.
- Resource cleanup for operations management: Some podss reach a designated stage where they no longer change, and these groups often accumulate in large numbers, complicating the management of other podss. The podss to be cleaned up may include those in the `Evicted` status due to insufficient node resources or those in the

`Completed` status triggered by recurring scheduled tasks. In this case, the deleted podss will no longer exist.

Note: For scheduled tasks, if you need to check the logs of each task execution, it is not recommended to delete the corresponding `Completed` status podss.

Procedure

1. Go to **Container Platform**.
2. In the left navigation bar, click **Workloads** > **Pods**.
3. (Delete individually) Click the ⋮ on the right side of the pods to be deleted > **Delete**, and confirm.
4. (Delete in bulk) Select the podss to be deleted, click **Delete** above the list, and confirm.

Alauda Container Platform

Containers

TOC

Understanding Containers

Refer to the official Kubernetes website documentation: Containers.

A **container** is a lightweight, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings. While Pods are the smallest deployable units, containers are the core components within Pods.

Understanding Ephemeral Containers

Debugging Containers with Refer to the official Kubernetes website documentation: Ephemeral Containers ↗

The Kubernetes Ephemeral Containers feature provides a robust way to debug running containers by injecting specialized debugging tools (system, network, and disk utilities) into an existing Pod.

While you can often execute commands directly within a running container using kubectl exec, many production container images are intentionally minimal and may lack essential debugging utilities (e.g., bash, net-tools, tcpdump) to reduce image size and attack surface. Ephemeral Containers address this limitation by providing a pre-configured environment with a rich set of debugging tools, making them ideal for the following scenarios:

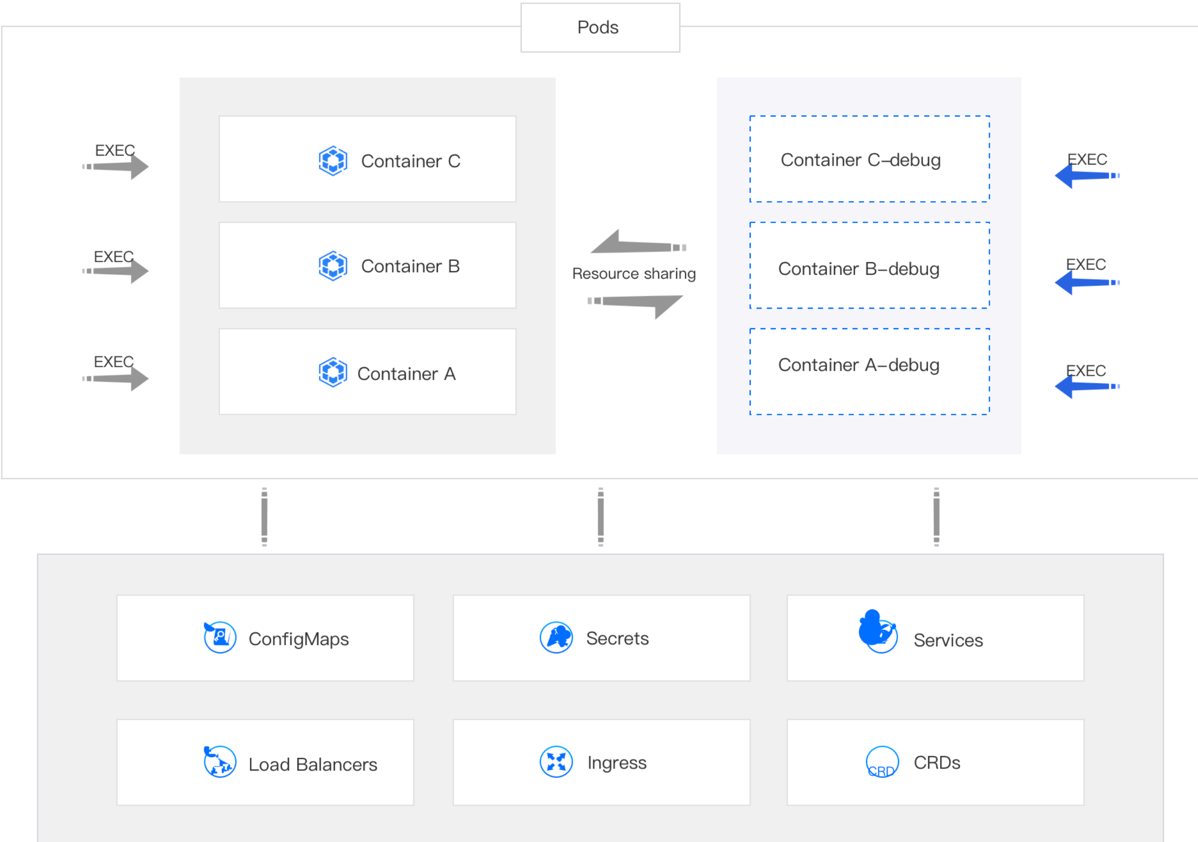
- **Fault Diagnosis**: When a primary application container experiences issues (e.g., unexpected crashes, performance degradation, network connectivity problems), beyond checking standard Pod events and logs, you often need to perform deeper, interactive troubleshooting directly within the Pod's runtime environment.
- **Configuration Tuning and Experimentation**: If the current application configuration exhibits suboptimal behavior, you might want to temporarily adjust component settings or test new configurations directly within the running container to observe immediate effects and devise improved solutions.

Implementation Principle: Leveraging Ephemeral Containers

The debugging functionality is implemented using **Ephemeral Containers**. An Ephemeral Container is a special type of container designed for introspection and debugging. It shares the Pod's network namespace and process namespace (if enabled) with the existing primary `containers`, allowing it to directly interact with and observe the application processes.

You can dynamically add an Ephemeral Container (e.g., `my-app-debug`) to a running Pod and utilize its pre-installed debugging tools. The diagnostic results from this Ephemeral

Container are directly relevant to the behavior and state of the primary application `containers` within the same Pod.



:::Notes * You cannot add an Ephemeral Container by directly modifying a Pod's static manifest (PodSpec). The Ephemeral Containers feature is designed for dynamic injection into running Pods, typically via API calls (like kubectl debug). * **Ephemeral Containers** created via the debug feature do not have resource (CPU/Memory) or scheduling guarantees (i.e., they don't block Pod startup or get their own QoS class) and will not automatically restart if they exit. Therefore, avoid running persistent business applications within them; they are strictly for debugging purposes. * Exercise caution when using the debug feature if the Node where the Pod is located is experiencing high resource utilization or is nearing resource exhaustion. Injecting an Ephemeral Container, even with minimal resource footprint, could potentially contribute to Pod eviction under severe resource pressure. :::

Debugging Ephemeral Containers by using CLI

Kubernetes 1.25+ offers the `kubectl debug` command for creating ephemeral containers. This method provides a powerful command-line alternative for debugging.

Command

```
kubectl debug -it <pod-name> --image=<debug-image> --target=<target-conta
iner-name> -n <namespace>
# --image: Specifies the debug image (e.g., busybox, ubuntu, nicolaka/net
shoot) containing necessary tools.
# --target: (Optional) Specifies the name of the container in the Pod to
target. If omitted, and there's only one container, it defaults to that.
If multiple, it defaults to the first.
# -n: Specifies the namespace.
```

Pod YAML file example

Example: Debugging `nginx` in `my-nginx-pod`

- First, ensure you have a Pod running:

  ```
  kubectl apply -f pod-example.yaml
  ```

- Now, create an ephemeral debug container named `debugger` inside `my-nginx-pod`, targeting my-nginx-container, using the `busybox` image:

  ```
  kubectl debug -it my-nginx-pod --image=busybox --target=nginx -- /bin/s
  h
  ```

  This command will attach you to a shell inside the debugger ephemeral container. You can now use busybox tools to debug my-nginx-container.

- To view the ephemeral containers attached to a Pod:

  ```
  kubectl describe pod my-nginx-pod
  ```

  Look for the `Ephemeral Containers` section in the output.

Debugging Ephemeral Containers by using web console

1. **Container Platform**, and navigate to **Workloads** > **Pods** in the left sidebar.
2. Locate the Pod you wish to view, and click ⋮ > **Debug**.
3. Choose the specific container within the Pod you wish to debug.

4. (Optional) If the interface prompts that **initialization is required** (e.g., for setting up necessary debug environment), click **Initialize**.

> **INFO**
>
> After initializing the Debug feature, as long as the pod is not recreated, you can directly enter the Ephemeral Container (for example, *Container A-debug*) for debugging.

5. Wait for the debugging terminal window to become ready, then begin your debugging operations. Tip: Click the "Command Query" option in the upper right corner of the terminal to view a list of common debugging tools and their usage examples.

> **INFO**
>
> Click the command query in the upper right corner to view common tools and their usage.

6. Once debugging is complete, close the terminal window.

Interacting with Containers

You can directly interact with the internal instance of a running container using the `kubectl exec` command, allowing you to execute arbitrary command-line operations. Additionally, Kubernetes provides convenient features for uploading and downloading files to and from containers.

Interacting with Containers by using CLI

Exec

To execute a command inside a specific container within a Pod (useful for getting a shell, running diagnostic commands, etc.):

```
kubectl exec -it <pod-name> -c <container-name> -n <namespace> -- <comman
d>
# -it: Ensures interactive mode and a TTY (pseudo-terminal) for a shell s
ession.
# -c: Specifies the target container name within the Pod. Omit if the Pod
has only one container.
# --: Separates kubectl arguments from the command to be executed in the
container.
```

- **Example**: Getting a Bash shell in the `nginx` of `my-nginx-pod`

  ```
  kubectl exec -it my-nginx-pod -c nginx -n default -- /bin/bash
  ```

- **Example**: Listing files in `/tmp` of a container

  ```
  kubectl exec my-nginx-pod -c nginx -n default -- ls /tmp
  ```

Transfer Files

- To copy files from your local machine to a container within a Pod:

  ```
  kubectl cp <local-file-path> <namespace>/<pod-name>:<container-file-pat
  h> -c <container-name>
  # -c: (Optional) Specifies the target container name if the Pod has mul
  tiple containers.

  # Example: Uploading `my-config.txt` to Nginx's HTML directory
  kubectl cp my-config.txt default/my-nginx-pod:/usr/share/nginx/html/my-
  config.txt -c nginx
  ```

- To copy files from a container within a Pod to your local machine:

  ```
  kubectl cp <namespace>/<pod-name>:<container-file-path> <local-file-pat
  h> -c <container-name>

  # Example: Downloading Nginx access logs
  kubectl cp default/my-nginx-pod:/var/log/nginx/access.log ./nginx_acces
  s.log -c nginx
  ```

Interacting with Containers by using web console

Entering the Container through Applications

You can enter the internal instance of the container using the `kubectl exec` command, allowing you to execute command-line operations in the Web console window. Additionally, you can easily upload and download files within the container using the file transfer feature.

1. **Container Platform**, and navigate to **Application** > **Applications** in the life sidebar.
2. Click on ***Application Name***.
3. Locate the associated workload (e.g., Deployment, StatefulSet), click **EXEC**, and then select the specific *Pod Name* you wish to enter. **EXEC** > ***Contianer Name***.
4. Enter the command you wish to execute.
5. Click **OK** to enter the Web console window and execute command-line operations.
6. Click **File Transfer**.

- Enter an **Upload Path** to upload local files into the container (e.g., configuration files for testing).
- Enter a **Download Path** to download logs, diagnostic data, or other files from the container to your local machine for analysis.

Entering the Container through the Pod

1. **Container Platform**, and navigate to **Workloads** > **Pods**.
2. Locate the target Pod, click the vertical ellipsis (⋮) next to it, select EXEC, and then choose the specific Container Name within that Pod you wish to enter.
3. Enter the command you wish to execute.
4. Click **OK** to enter the Web console window and execute command-line operations.
5. Click **File Transfer**.

- Enter an **Upload Path** to upload local files into the container (e.g., configuration files for testing).
- Enter a **Download Path** to download logs, diagnostic data, or other files from the container to your local machine for analysis.

Adjust Pod Resource Levels Without Pod Disruption

Use in-place Pod resizing when you need to adjust CPU or memory requests and limits for a running Pod without recreating the Pod. This task uses the Kubernetes Pod `resize` subresource.

TOC

Prerequisites

Ensure that the target Kubernetes cluster supports the Pod `resize` subresource. Use Kubernetes v1.33 or later with the `InPlacePodVerticalScaling` feature gate enabled on the control plane and nodes. In Kubernetes v1.35, this feature is stable and enabled by default.

About In-place Pod Resizing

In-place Pod resizing lets you change CPU and memory resource requests and limits for containers in a running Pod. Standard resource changes usually require the Pod to be recreated, which can interrupt applications or lose runtime state. In-place Pod resizing applies supported resource changes through the Kubernetes Pod `resize` subresource.

You can control whether a container restarts during the resize by configuring `resizePolicy` in the Pod specification. For example, you can allow CPU changes without a restart and require a container restart for memory changes.

Configure In-place Pod Resizing

Configure `resizePolicy` before you resize a running Pod. You cannot add or modify `resizePolicy` on an existing Pod. If the Pod is managed by an owner object, such as a Deployment or StatefulSet, add or update `resizePolicy` in the owner object's Pod template.

deployment-resize-policy.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: resize-demo
spec:
  selector:
    matchLabels:
      app: resize-demo
  template:
    metadata:
      labels:
        app: resize-demo
    spec:
      containers:
        - name: app
          image: nginx:latest
          resizePolicy:
            - resourceName: cpu
              restartPolicy: NotRequired
            - resourceName: memory
              restartPolicy: RestartContainer
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 500m
              memory: 512Mi
```

For CPU and memory resources, use one of the following `restartPolicy` values:

Value	Description
`NotRequired`	Apply supported resource changes without restarting the container.
`RestartContainer`	Apply supported resource changes and restart the container.

When decreasing memory limits with `NotRequired` or no memory `resizePolicy`, the kubelet applies the change on a best-effort basis and does not guarantee that the container avoids an out-of-memory kill. If current memory usage is higher than the requested limit, the resize can be skipped and remain in progress. Use `RestartContainer` for memory changes when you need more predictable behavior.

Resize a Running Pod

Use the `resize` subresource when editing Pod resources. Run the commands from a workstation that can access the target cluster.

> **Tip**
>
> Use `kubectl` v1.32.0 or later, or another API client that supports the Pod `resize` subresource. Older `kubectl` clients might reject `--subresource=resize` with errors such as `invalid subresource value: "resize"` or might not provide the `--subresource` flag for some commands, even when the Kubernetes API server supports `pods/resize`.

Edit the Pod resources interactively:

```
kubectl edit pod <pod_name> -n <namespace> --subresource=resize
```

Apply an updated Pod manifest through the `resize` subresource by using server-side apply:

```
kubectl apply --server-side -f <file_name>.yaml --subresource=resize
```

Patch the target container resources directly:

```
kubectl patch pod <pod_name> \
  -n <namespace> \
  --subresource=resize \
  --patch '{"spec":{"containers":[{"name":"<container_name>","resources":
{"requests":{"cpu":"800m"},"limits":{"cpu":"800m"}}}]}}'
```

Replace `<pod_name>`, `<namespace>`, and `<container_name>` with the target Pod, namespace, and container names.

Because this operation requires the Pod `resize` subresource, use `kubectl` or an API client version that supports the subresource.

Verify Restart Behavior

Check whether the Pod restarted after the resize:

```
kubectl get pods -n <namespace>
```

If `restartPolicy` is `NotRequired` for the changed resource, the Pod should remain running without a container restart.

```
NAME          READY    STATUS     RESTARTS    AGE
resize-pod    1/1      Running    0           5s
```

If `restartPolicy` is `RestartContainer` for the changed resource, the container restarts while the Pod remains running.

```
NAME          READY    STATUS     RESTARTS       AGE
resize-pod    1/1      Running    1 (5s ago)     5s
```

Check Resize Status

Check the resources currently applied to the running container:

```
kubectl get pod <pod_name> \
  -n <namespace> \
  -o jsonpath='{.status.containerStatuses[?(@.name=="<container_name>")].
resources}'
```

Check resize conditions and events:

```
kubectl describe pod <pod_name> -n <namespace>
```

Resize conditions include:

- `PodResizeInProgress`: kubelet can allocate the requested resources and is applying the change.
- `PodResizePending`: kubelet cannot apply the change immediately.
  - `Infeasible`: The requested resize cannot be executed on the current node.
  - `Deferred`: The requested resize is not currently possible, but might become possible later.
  - `Error`: kubelet reports an error during resource allocation.

Limitations

- Only CPU and memory requests and limits can be resized.
- In-place Pod resizing is not supported for non-restartable init containers or ephemeral containers.
- The resize must not violate other Pod mutability constraints, such as changing the Pod QoS class.
- Pods managed by static `cpuManagerPolicy` or `memoryManagerPolicy` cannot be resized in place.
- Pods that use swap memory must use `RestartContainer` for memory requests.
- ResourceQuota, LimitRange, and admission policies can still reject the resize request.

> **Note**
>
> Because this operation requires the Pod `resize` subresource, it might not be available through web console resource editing. Use `kubectl` or an API client version that supports the `resize` subresource.

Additional Resources

- Pods
- Resource Units
- Configuring VerticalPodAutoscaler (VPA)
- Pod API Reference

Working with Helm charts

TOC

---

1. Understanding Helm

Helm is a package manager that simplifies the deployment of applications and services on Alauda Container Platform clusters. Helm uses a packaging format called *charts*. A Helm chart is a collection of files that describe Kubernetes resources. Creating a chart in a cluster generates a chart running instance called a *release*. Each time a chart is created, or a release is upgraded or rolled back, an incremental revision is created.

1.1. Key features

Helm provides the ability to:

- Search for a large collection of charts in chart repositories
- Modify existing charts
- Create your own charts using Kubernetes resources
- Package applications and share them as charts

1.2. Catalog

The Catalog is built on Helm and provides a comprehensive Chart distribution management platform, extending the limitations of the Helm CLI tool. The platform enables developers to more conveniently manage, deploy, and use charts through a user-friendly interface.

Terminology Definitions

Term	Definition	Notes
Application Catalog	A one-stop management platform for Helm Charts	
Helm Charts	An application packaging format	
HelmRequest	CRD. Defines the configuration needed to deploy a Helm Chart	Template Application
ChartRepo	CRD. Corresponds to a Helm charts repository	Template Repository
Chart	CRD. Corresponds to Helm Charts	Template

1.3 Understanding HelmRequest

In Alauda Container Platform, Helm deployments are primarily managed through a custom resource called **HelmRequest**. This approach extends standard Helm functionality and integrates it seamlessly into the Kubernetes native resource model.

Differences Between HelmRequest and Helm

Standard Helm uses CLI commands to manage releases, while Alauda Container Platform uses HelmRequest resources to define, deploy, and manage Helm charts. Key differences include:

1. **Declarative vs Imperative**: HelmRequest provides a declarative approach to Helm deployments, while traditional Helm CLI is imperative.
2. **Kubernetes Native**: HelmRequest is a custom resource directly integrated with the Kubernetes API.
3. **Continuous Reconciliation**: Captain continuously monitors and reconciles HelmRequest resources with their desired state.
4. **Multi-cluster Support**: HelmRequest supports deployments across multiple clusters through the platform.
5. **Platform Feature Integration**: HelmRequest can be integrated with other platform features, such as Application resources.

HelmRequest and Application Integration

HelmRequest and Application resources have conceptual similarities, and users may want to view them uniformly. The platform provides a mechanism to synchronize HelmRequest as Application resources.

Users can mark a HelmRequest to be deployed as an Application by adding the following annotation:

```
alauda.io/create-app: "true"
```

When this feature is enabled, the platform UI displays additional fields and links to the corresponding Application page.

Deployment Workflow

The workflow for deploying charts via HelmRequest includes:

1. **User** creates or updates a HelmRequest resource
2. **HelmRequest** contains chart references and values to apply
3. **Captain** processes the HelmRequest and creates a Helm Release
4. **Release** contains the deployed resources
5. **Metis** monitors HelmRequests with application annotations and synchronizes them to Applications
6. **Application** provides a unified view of deployed resources

Component Definitions

- **HelmRequest**: Custom resource definition that describes the desired Helm chart deployment
- **Captain**: Controller that processes HelmRequest resources and manages Helm releases (source code available at https://github.com/alauda/captain ↗)
- **Release**: Deployed instance of a Helm chart
- **Charon**: Component that monitors HelmRequests and creates corresponding Application resources

- **Application**: Unified representation of deployed resources, providing additional management capabilities
- **Archon-api**: Component responsible for specific advanced API functions within the platform

2 Deploying Helm Charts as Applications via CLI

2.1 Workflow Overview

Prepare chart → Package chart → Obtain API token → Create chart repository → Upload chart → Upload related images → Deploy application → Update application → Uninstall application → Delete chart repository

2.2 Preparing the Chart

Helm uses a packaging format called charts. A chart is a collection of files that describe Kubernetes resources. A single chart can be used to deploy anything from a simple pod to a complex application stack.

Refer to the official documentation: Helm Charts Documentation ↗

Example chart directory structure:

```
nginx/
├── Chart.lock
├── Chart.yaml
├── README.md
├── charts/
│   └── common/
│       ├── Chart.yaml
│       ├── README.md
│       ├── templates/
│       │   ├── _affinities.tpl
│       │   ├── _capabilities.tpl
│       │   ├── _errors.tpl
│       │   ├── _images.tpl
│       │   ├── _ingress.tpl
│       │   ├── _labels.tpl
│       │   ├── _names.tpl
│       │   ├── _secrets.tpl
│       │   ├── _storage.tpl
│       │   ├── _tplvalues.tpl
│       │   ├── _utils.tpl
│       │   ├── _warnings.tpl
│       │   └── validations/
│       │       ├── _cassandra.tpl
│       │       ├── _mariadb.tpl
│       │       ├── _mongodb.tpl
│       │       ├── _postgresql.tpl
│       │       ├── _redis.tpl
│       │       └── _validations.tpl
│       └── values.yaml
├── ci/
│   ├── ct-values.yaml
│   └── values-with-ingress-metrics-and-serverblock.yaml
├── templates/
│   ├── NOTES.txt
│   ├── _helpers.tpl
│   ├── deployment.yaml
│   ├── extra-list.yaml
│   ├── health-ingress.yaml
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── ldap-daemon-secrets.yaml
│   ├── pdb.yaml
│   ├── server-block-configmap.yaml
```

```
│   ├── serviceaccount.yaml
│   ├── servicemonitor.yaml
│   ├── svc.yaml
│   └── tls-secrets.yaml
├── values.descriptor.yaml
├── values.schema.json
└── values.yaml
```

Key file descriptions:

- `values.descriptor.yaml` (optional): Works with ACP UI to display user-friendly forms
- `values.schema.json` (optional): Validates values.yaml content and renders a simple UI
- `values.yaml` (required): Defines chart deployment parameters

2.3 Packaging the Chart

Use the `helm package` command to package the chart:

```
helm package nginx
# 输出: Successfully packaged chart and saved it to: /charts/nginx-8.8.0.t
gz
```

2.4 Obtaining an API Token

1. In **Alauda Container Platform**, click the avatar in the top-right corner => **Profile**
2. Click **Add Api Token**
3. Enter appropriate Description & Remaining Validity
4. Save the displayed token information (only shown once)

2.5 Creating a Chart Repository

Create a local chart repository via API:

```
curl -k --request POST \
--url https://$ACP_DOMAIN/catalog/v1/chartrepos \
--header 'Authorization:Bearer $API_TOKEN' \
--header 'Content-Type: application/json' \
--data '{
  "apiVersion": "v1",
  "kind": "ChartRepoCreate",
  "metadata": {
    "name": "test",
    "namespace": "cpaas-system"
  },
  "spec": {
    "chartRepo": {
      "apiVersion": "app.alauda.io/v1beta1",
      "kind": "ChartRepo",
      "metadata": {
        "name": "test",
        "namespace": "cpaas-system",
        "labels": {
          "project.cpaas.io/catalog": "true"
        }
      },
      "spec": {
        "type": "Local",
        "url": null,
        "source": null
      }
    }
  }
}'
```

2.6 Uploading the Chart

Upload the packaged chart to the repository:

```
curl -k --request POST \
--url https://$ACP_DOMAIN/catalog/v1/chartrepos/cpaas-system/test/charts
\
--header 'Authorization:Bearer $API_TOKEN' \
--data-binary @"/root/charts/nginx-8.8.0.tgz"
```

2.7 Uploading Related Images

1. Pull the image: `podman pull nginx`
2. Save as tar package: `podman save nginx > nginx.latest.tar`
3. Load and push to private registry:

```
podman load -i nginx.latest.tar
podman tag nginx:latest 192.168.80.8:30050/nginx:latest
podman push 192.168.80.8:30050/nginx:latest
```

2.8 Deploying the Application

Create Application resource via API:

```
curl -k --request POST \
--url https://$ACP_DOMAIN/acp/v1/kubernetes/$CLUSTER_NAME/namespaces/$NAM
ESPACE/applications \
--header 'Authorization:Bearer $API_TOKEN' \
--header 'Content-Type: application/json' \
--data '{
  "apiVersion": "app.k8s.io/v1beta1",
  "kind": "Application",
  "metadata": {
    "name": "test",
    "namespace": "catalog-ns",
    "annotations": {
      "app.cpaas.io/chart.source": "test/nginx",
      "app.cpaas.io/chart.version": "8.8.0",
      "app.cpaas.io/chart.values": "{\"image\":{\"pullPolicy\":\"IfNotPre
sent\"}}"
    },
    "labels": {
      "sync-from-helmrequest": "true"
    }
  }
}'
```

2.9 Updating the Application

Update the application using PATCH request:

```
curl -k --request PATCH \
--url https://$ACP_DOMAIN/acp/v1/kubernetes/$CLUSTER_NAME/namespaces/$NAM
ESPACE/applications/test \
--header 'Authorization:Bearer $API_TOKEN' \
--header 'Content-Type: application/merge-patch+json' \
--data '{
  "apiVersion": "app.k8s.io/v1beta1",
  "kind": "Application",
  "metadata": {
    "annotations": {
      "app.cpaas.io/chart.values": "{\"image\":{\"pullPolicy\":\"Always
\"}}"
    }
  }
}'
```

2.10 Uninstalling the Application

Delete the Application resource:

```
curl -k --request DELETE \
--url https://$ACP_DOMAIN/acp/v1/kubernetes/$CLUSTER_NAME/namespaces/$NAM
ESPACE/applications/test \
--header 'Authorization:Bearer $API_TOKEN'
```

2.11 Deleting the Chart Repository

```
curl -k --request DELETE \
--url https://$ACP_DOMAIN/apis/app.alauda.io/v1beta1/namespaces/cpaas-sys
tem/chartrepos/test \
--header 'Authorization:Bearer $API_TOKEN'
```

3. Deploying Helm Charts as Applications via UI

3.1 Workflow Overview

Add templates to manageable repositories → Upload templates → Manage template versions

3.2 Prerequisites

Template repositories are added by platform administrators. Please contact the platform administrator to obtain the available Chart or OCI Chart type template repository names with **Management** permissions.

3.3 Adding Templates to Manageable Repositories

1. Go to **Catalog**.
2. In the left navigation bar, click **Helm Charts**.
3. Click **Add Template** in the upper right corner of the page, and select the template repository based on the parameters below.

Parameter	Description
Template Repository	Synchronize the template directly to a Chart or OCI Chart type template repository with **Management** permissions. Project owners assigned to this **Template Repository** can directly use the template.
Template Directory	When the selected template repository type is OCI Chart, a directory to store the Helm Chart must be selected or manually entered. **Note**: When manually entering a new template directory, the platform will create this directory in the template repository, but there is a risk of creation failure.

4. Click **Upload Template** and upload the local template to the repository.
5. Click **Confirm**. The template upload process may take a few minutes, please be patient.

Note: When the template status changes from `Uploading` to `Upload Successful`, it indicates that the template has been uploaded successfully.

6. If the upload fails, please troubleshoot according to the following prompts.
   Note: An illegal file format means there is an issue with the files in the uploaded compressed package, such as missing content or incorrect formatting.

3.4 Deleting Specific Versions of Templates

If a version of a template is no longer applicable, it can be deleted.

Steps to Operate

1. Go to **Catalog**.
2. In the left navigation bar, click **Helm Charts**.
3. Click on the Chart card to view details.
4. Click **Manage Versions**.
5. Find the template that is no longer applicable, click **Delete**, and confirm.
   After deleting the version, the corresponding application will not be able to be updated.

Configurations

The **Configurations** section covers namespace-scoped configuration resources that workloads consume at runtime.

Use these pages when you need to create, review, update, or delete configuration data for applications, workloads, or cluster-side namespace resources.

TOC

Choose the correct entry view

ConfigMaps and Secrets can be managed from different views in the new web console, but the resource itself still belongs to a concrete cluster and namespace.

- Use the **Applications** view when you are managing configuration for workloads in the current project scope.
- Use the **Clusters** view when you start from a concrete cluster and manage configuration as part of cluster-side namespace operations.

In both views, workloads can only reference ConfigMaps or Secrets from the same namespace.

Non-sensitive configuration

Use **ConfigMaps** for runtime configuration that does not contain credentials or private material:

- Configuring ConfigMap

Sensitive configuration

Use **Secrets** for credentials, tokens, keys, certificates, or image-pull authentication:

- Configuring Secrets

Operational reminder

When you update or delete ConfigMaps or Secrets, review the downstream workload impact. Depending on how workloads consume the data, the change can require Pod recreation, rollout, restart, or image-pull revalidation before it fully takes effect.

Place operational guidance in the page where the action happens:

- creation and entry-scope checks belong beside the create steps
- workload-consumption checks belong beside update and delete behavior
- cluster-view differences belong next to the same capability rather than in a single appendix

---

Configuring ConfigMap

Understanding Config Maps

Config Map Restrictions

Configuring Secrets

Understanding Secrets

Creating an Opaque type Secret

Configuring ConfigMap

Config maps allow you to decouple configuration artifacts from image content to keep containerized applications portable. The following sections define config maps and how to create and use them.

TOC

Understanding Config Maps

Many applications require configuration by using some combination of configuration files, command-line arguments, and environment variables. In OpenShift Container Platform, these configuration artifacts are decoupled from image content to keep containerized applications portable.

The `ConfigMap` object provides mechanisms to inject containers with configuration data while keeping containers agnostic of OpenShift Container Platform. A config map can be used to store fine-grained information like individual properties or coarse-grained information like entire configuration files or JSON blobs.

The `ConfigMap` object holds key-value pairs of configuration data that can be consumed in pods or used to store configuration data for system components such as controllers. For example:

```yaml
# my-app-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-app-config
data:
  app_mode: "development"
  feature_flags: "true"
  database.properties: |-
    jdbc.url=jdbc:mysql://localhost:3306/mydb
    jdbc.username=user
    jdbc.password=password
  log_settings.json: |-
    {
      "level": "INFO",
      "format": "json"
    }
```

Note: You can use the `binaryData` field when you create a config map from a binary file, such as an image.

Configuration data can be consumed in pods in a variety of ways. A config map can be used to:

- Populate environment variable values in containers
- Set command-line arguments in a container
- Populate configuration files in a volume

Users and system components can store configuration data in a config map. A config map is similar to a secret, but designed to more conveniently support working with strings that do not contain sensitive information.

Config Map Restrictions

- A config map must be created before its contents can be consumed in pods.
- Controllers can be written to tolerate missing configuration data. Consult individual components configured by using config maps on a case-by-case basis.
- `ConfigMap` objects reside in a project.
- They can only be referenced by pods in the same project.
- The Kubectl only supports the use of a config map for pods it gets from the API server. This includes any pods created by using the CLI, or indirectly from a replication controller. It does not include pods created by using the OpenShift Container Platform node's `--manifest-url` flag, its `--config` flag, or its REST API because these are not common ways to create pods.

> **NOTE**
>
> A Pod can only use ConfigMaps within the same namespace.

Example ConfigMap

You can now use app-config in a `Pod`.

```yaml
# app-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_ENV: "production"
  LOG_LEVEL: "debug"
```

Creating a ConfigMap by using the web console

1. Go to **Container Platform**.
2. In the left sidebar, click **Configuration** > **ConfigMap**.
3. Click **Create ConfigMap**.
4. Refer to the instructions below to configure the relevant parameters.

Parameter	Description
Entries	Refers to `key:value` pairs, supporting both **Add** and **Import** methods. • **Add**: You can add configuration items one by one, or you can paste one or multiple lines of key=value pairs in the Key input area to bulk add configuration items. • **Import**: Import a text file not larger than 1M. The file name will be used as the key, and the file content will be used as the value, filled into a configuration item.
Binary Entries	Refers to binary files not larger than 1M. The file name will be used as the key, and the file content will be used as the value, filled into a configuration item. **Note**: After creating a ConfigMap, the imported files cannot be modified.

Example of Bulk Add Format:

```
# One key=value pair per line, multiple pairs must be on separate line
s, otherwise they will not be recognized correctly after pasting.
key1=value1
key2=value2
key3=value3
```

5. Click **Create**.

Creating a ConfigMap by using new web console

This page mainly describes the legacy console workflow, but the same capability can also be reached from different views in the new web console.

Create or update from the Applications view

Use the **Applications** view when the ConfigMap is managed as part of the current project scope.

Before creating or updating the ConfigMap from this view, confirm that:

- the selected cluster is already associated with the current project
- the target namespace is the same namespace used by the consuming workloads
- the keys, file names, and value formats match the way the workloads read the configuration
- the ConfigMap appears afterward in the selected project and cluster context

Create or update from the Clusters view

Use the **Clusters** view when you start from a concrete cluster and manage ConfigMaps as namespace resources in that cluster.

Before creating or updating the ConfigMap from this view, confirm that:

- you have opened the target cluster first
- you have selected or confirmed the target namespace in that cluster

- you are reviewing namespace-local workload impact rather than project membership
- the ConfigMap appears afterward in the expected namespace

Creating a ConfigMap by using the CLI

```
kubectl create configmap app-config \
  --from-literal=APP_ENV=production \
  --from-literal=LOG_LEVEL=debug
```

Or from a file:

```
kubectl apply -f app-config.yaml
```

Operations

You can click the (⋮) on the right side of the list page or click **Actions** in the upper right corner of the detail page to update or delete the ConfigMap as needed.

Changes to the ConfigMap will affect the workloads that reference the configuration, so please read the operation instructions in advance.

Operations	Description
Update	• After adding or updating a ConfigMap, any workloads that have referenced this ConfigMap (or its configuration items) through environment variables need to rebuild their Pods for the new configuration to take effect. • For imported binary configuration items, only key updates are supported, not value updates.

Operations	Description
Delete	After deleting a ConfigMap, workloads that have referenced this ConfigMap (or its configuration items) through environment variables may be adversely affected during Pod creation if they are rebuilt and cannot find the reference source.

View, Edit and Delete by using the CLI

```
kubectl get configmap app-config -o yaml
kubectl edit configmap app-config
kubectl delete configmap app-config
```

Ways to Use a ConfigMap in a Pod

As Environment Variables

```
envFrom:
  - configMapRef:
      name: app-config
```

Each key becomes an environment variable in the container.

As Files in a Volume

```yaml
volumes:
  - name: config-volume
    configMap:
      name: app-config

volumeMounts:
  - name: config-volume
    mountPath: /etc/config
```

Each key is a file under `/etc/config`, and the file content is the value.

As Individual Environment Variables

```yaml
env:
  - name: APP_ENV
    valueFrom:
      configMapKeyRef:
        name: app-config
        key: APP_ENV
```

ConfigMap vs Secret

Feature	ConfigMap	Secret
Data Type	Non-sensitive	Sensitive (e.g., passwords)
Encoding	Plaintext	Base64-encoded
Use Cases	Configs, flags	Passwords, tokens

Configuring Secrets

TOC

Understanding Secrets

In Kubernetes (k8s), a Secret is a fundamental object designed to store and manage sensitive information, such as passwords, OAuth tokens, SSH keys, TLS certificates, and API keys. Its primary purpose is to prevent sensitive data from being directly embedded in Pod definitions or container images, thereby enhancing security and portability.

Secrets are similar to ConfigMaps but are specifically intended for confidential data. They are typically base64-encoded for storage and can be consumed by pods in various ways, including being mounted as volumes or exposed as environment variables.

Usage Characteristics

- **Enhanced Security**: Compared to plaintext configuration maps (Kubernetes ConfigMap), Secrets offer better security by storing sensitive information using Base64 encoding. This mechanism, combined with Kubernetes' ability to control access, significantly reduces the risk of data exposure.
- **Flexibility and Management**: Using Secrets provides a more secure and flexible approach than hardcoding sensitive information directly into Pod definition files or container images. This separation simplifies the management and modification of sensitive data without requiring changes to application code or container images.

Supported Types

Kubernetes supports various types of Secrets, each tailored for specific use cases. The platform typically supports the following types:

- **Opaque**: A general-purpose Secret type used to store arbitrary key-value pairs of sensitive data, such as passwords or API keys.
- **TLS**: Specifically designed to store TLS (Transport Layer Security) protocol certificate and private key information, commonly used for HTTPS communication and secure ingress.
- **SSH Key**: Used to store SSH private keys, often for secure access to Git repositories or other SSH-enabled services.
- **SSH Authentication (kubernetes.io/ssh-auth)**: Stores authentication information for data transmitted over the SSH protocol.

- **Username/Password (kubernetes.io/basic-auth)**: Used to store basic authentication credentials (username and password).
- **Image Pull Secret**: Stores the JSON authentication string required for pulling container images from private image repositories.

Usage Methods

Secrets can be consumed by applications within pods through different methods:

- **As Environment Variables**: Sensitive data from a Secret can be injected directly into a container's environment variables.
- **As Mounted Files (Volume)**: Secrets can be mounted as files within a pod's volume, allowing applications to read sensitive data from a specified file path.

Note: Pod instances in workloads can only reference Secrets within the same namespace. For advanced usage and YAML configurations, refer to the Kubernetes official documentation ↗.

Creating an Opaque type Secret

```
kubectl create secret generic my-secret \
  --from-literal=username=admin \
  --from-literal=password=Pa$$w0rd
```

YAML

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: YWRtaW4= # base64 of "admin"
  password: UGEkJHcwcmQ= # base64 of "Pa$$w0rd"
```

You can decode them like:

```
echo YWRtaW4= | base64 --decode  # output: admin
```

Creating a container registry type Secret

```
kubectl create secret docker-registry my-container-registry-creds \
  --docker-username=myuser \
  --docker-password=mypass \
  --docker-server=https://registry.example.com \
  --docker-email=my@example.com
```

YAML

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-container-registry-creds
type: kubernetes.io/dockerconfigjson
data:
  .dockerconfigjson: eyJhdXRocyI6eyJodHRwczovL2luZGV4LmRvY2tlci5pby92MS8i
OnsidXNlcm5hbWUiOiJteXVzZXIiLCJwYXNzd29yZCI6Im15cGFzcyIsImVtYWlsIjoibXlAZ
XhhbXBsZS5jb20iLCJhdXRoIjoiYlhsMWMyVnlPbTE1Y0dGemN3PT0ifX19
```

K8s automatically converts your username, password, email, and server information into standard login format:

```
{
  "auths": {
    "https://registry.example.com": {
      "username": "myuser",
      "password": "mypass",
      "email": "my@example.com",
      "auth": "bXl1c2VyOm15cGFzcw=="  # base64(username:password)
    }
  }
}
```

This JSON is then base64 encoded and used as the data field value of the Secret.

Use it in a Pod:

```
imagePullSecrets:
  - name: my-container-registry-creds
```

Creating a Basic Auth type Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: basic-auth-secret
type: kubernetes.io/basic-auth
stringData:
  username: myuser
  password: mypass
```

Creating a SSH-Auth type Secret

Use Case: Store SSH private keys (e.g., for Git access).

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: ssh-key-secret
type: kubernetes.io/ssh-auth
stringData:
  ssh-privatekey: |
    -----BEGIN OPENSSH PRIVATE KEY-----
    ...
    -----END OPENSSH PRIVATE KEY-----
```

Creating a TLS type Secret

Use Case: TLS certs (used by Ingress, webhooks, etc.)

```
kubectl create secret tls tls-secret \
--cert=path/to/tls.crt \
--key=path/to/tls.key
```

YAML

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: tls-secret
type: kubernetes.io/tls
data:
  tls.crt: <base64>
  tls.key: <base64>
```

Creating a Secret by using the web console

1. Go to **Container Platform**.
2. In the left navigation bar, click **Configuration** > **Secrets**.

3. Click **Create Secret**.
4. Configure the parameters.
   Note: In the form view, sensitive data such as the input username and password will automatically be encoded in Base64 format before being stored in the Secret. The converted data can be previewed in the YAML view.
5. Click **Create**.

How to Use a Secret in a Pod

As Environment Variables

```
env:
  - name: DB_USERNAME
    valueFrom:
      secretKeyRef:
        name: my-secret
        key: username
```

From the secret named `my-secret`, take the value with the key `username` and assign it to the environment variable `DB_USERNAME`.

As Mounted Files (Volume)

```
volumes:
  - name: secret-volume
    secret:
      secretName: my-secret

volumeMounts:
  - name: secret-volume
    mountPath: "/etc/secret"
```

Follow-up Actions

When creating workloads for native applications in the same namespace, you can reference the Secrets that have already been created.

Operations

You can click the (⋮) on the right side of the list page or click **Actions** in the upper right corner of the details page to update or delete the Secret as needed.

Operation	Description
Update	After adding or updating a Secret, workloads that have referenced this Secret (or its configuration items) via environment variables need to have their Pods rebuilt for the new configuration to take effect.
Delete	• After deleting a Secret, workloads that have referenced this Secret (or its configuration items) via environment variables may be impacted due to the inability to find the reference source when rebuilding Pods. • Please do not delete the Secrets automatically generated by the platform, as this may prevent the platform from functioning properly. For example: Secrets of type service-account-token that contain authentication information for namespace resources and Secrets in system namespaces (such as kube-system).

Creating a Secret by using new web console

This page mainly describes the legacy console workflow, but the same capability can also be reached from different views in the new web console.

Create or update from the Applications view

Use the **Applications** view when the Secret is managed as part of the current project scope.

Before creating or updating the Secret from this view, confirm that:

- the selected cluster is already associated with the current project
- the target namespace is the same namespace used by the consuming workloads
- the selected Secret type and key names match the expected workload configuration
- the Secret appears afterward in the selected project and cluster context

Create or update from the Clusters view

Use the **Clusters** view when you start from a concrete cluster and manage Secrets as namespace resources in that cluster.

Before creating or updating the Secret from this view, confirm that:

- you have opened the target cluster first
- you have selected or confirmed the target namespace in that cluster
- you are reviewing workload startup, mounted-file, or image-pull impact inside that namespace
- the Secret appears afterward in the expected namespace

Verify workload impact

No matter which view you use, also verify that:

- the referenced workloads are still in the same namespace as the Secret
- any required Pod recreation, rollout, restart, or image-pull revalidation has been completed
- deleting the Secret does not break later workload startup, mount, or image pull because of a missing reference

Be especially careful before deleting platform-generated or workload-critical Secrets.

Application Observability

Monitoring Dashboards

- Supports viewing resource monitoring data for workload components on the platform for the past 7 days (with configurable monitoring data retention period). Includes statistics for applications, workloads, pods, and trends/rankings of CPU/memory usage.
- Supports Namespace-Level monitoring.
- Supported Workload-Level Monitoring: **Applications**, **Deployments**, **DaemonSets**, **StatefulSets**, and **Pods**

TOC

Prerequisites

- Installation of Monitoring Plugins

Namespace-Level Monitoring Dashboards

Procedure

1. **Container Platform**, click **Observe** > **Dashboards**.
2. View monitoring data under the namespace. Three dashboards are provided: **Applications Overview**, **Workloads Overview**, and **Pods Overview**.
3. Switch between dashboards to monitor target **Overview**.

Creating Namespace-Level Monitoring Dashboard

1. **Administrator**, create a dedicated monitoring dashboard by referring to Creating Monitoring Dashboard to create a dedicated monitoring dashboard.
2. Configure the following labels to display the Namespace-Level Monitoring dashboard on the **Container Platform**:
   - `cpaas.io/dashboard.folder: container-platform`
   - `cpaas.io/dashboard.tag.overview: "true"`

Workload-Level Monitoring

> This procedure demonstrates how to view pod monitoring through the Deployment interface.

Default Monitoring Dashboard

Procedure

1. **Container Platform**, click **Workloads** > **Deployments**.
2. Click a Deployment name from the list.

3. Navigate to the **Monitoring** tab to view default monitoring metrics.

Metric interpretation

Monitoring Resource	Metric Granularity	Technical Definition
CPU	**Utilization/Usage**	**Utilization** = Usage/Limit (limits) Assess container limit configuration. High utilization indicates insufficient limits. **Usage** represents actual resource consumption.
Memory	**Utilization/Usage**	**Utilization** = Usage/Limit (limits) Evaluation method same as CPU. High rate may cause component instability.
Network Traffic	**Inflow Rate/Outflow Rate**	Network traffic (bytes/sec) flowing into/out of pods.
Network Packet	**Receiving Rate/Transmit Rate**	Network packets (count/sec) received/sent by pods.
Disk Rate	**Read/Write**	Read/write throughput (bytes/sec) of mounted volumes per workload.
Disk IOPS	**Read/Write**	Input/Output Operations Per Second (IOPS) of mounted volumes per workload.

Custom Monitoring Dashboard

4. Click the **Toggle Icon** to switch to custom dashboards. Refer to Add Panel in Custom Dashboard to create dedicated **Workload-Level** monitoring dashboard.

> **INFO**
>
> Hover over chart curves to view per-pod metrics and PromQL expressions at specific timestamps. If exceeding 15 pods, only top 15 entries sorted in descending order are displayed.

Logs

Aggregate container runtime logs with visual query capabilities. When applications, workloads, or other resources exhibit abnormal behavior, log analysis helps diagnose root causes.

TOC

Procedure

> This procedure demonstrates how to view container runtime logs through the Deployment interface.

1. **Container Platform**, click **Workloads** > **Deployments**.
2. Click a Deployment name from the list.
3. Navigate to the **Logs** tab to view detailed records.

Operation	Description
Pod/Container	Switch between Pods and Containers using the dropdown selector to view the corresponding logs.

Operation	Description
Previous Logs	View logs from terminated containers (available when container restartCount > 0).
Lines	Configure display log buffer size: 1k/10k/100k lines.
Wrap Line	Toggle line wrapping for long log entries (enabled by default).
Find	Full-text search with highlight matching and Enter-to-navigate.
Raw	Unprocessed log streams directly captured from container runtime interfaces (CRI) without formatting, filtering, or truncation.
Export	Download raw logs.
Full Screen	Click truncated line to view full content in modal dialog.

WARNING

- **Truncation Handling**: Log entries exceeding 2000 characters will be truncated with an ellipsis `...`
  - Trimmed portions cannot be matched by the page's find function.
  - Click the ellipsis `...` marker in truncated lines to view full content in a modal dialog.
- **Copy Reliability**: Avoid direct copying from rendered log viewer when seeing truncation markers (...) or ANSI color codes. Always use **Export**, **Raw** function for complete logs.
- **Retention Policy**: Live logs follow Kubernetes log rotation configuration. For historical analysis, use Logs ↗ .

Events

Event information generated by Kubernetes resource state changes and operational status updates, with integrated visual query interface.When applications, workloads, or other resources encounter exceptions, real-time event analysis helps troubleshoot root causes.

TOC

Procedure

> This procedure demonstrates how to view container runtime evens through the Deployment interface.

1. **Container Platform**, click **Workloads** > **Deployments**.
2. Click a Deployment name from the list.
3. Navigate to the **Events** tab to view detailed records.

Event records interpretation

Resource event records: Below the event summary panel, all matching events within the specified time range are listed. Click event cards to view complete event details. Each card displays:

- **Resource Type**: Kubernetes resource type represented by icon abbreviations:
  - `P` = Pod
  - `RS` = ReplicaSet
  - `D` = Deployment
  - `SVC` = Service
- **Resource Name**: Target resource named.
- **Event Reason**: Kubernetes-reported reason (e.g., FailedScheduling).
- **Event Level**: Event severity.
  - `Normal` : Informational
  - `Warning` : Requires immediate attention
- **Time**: Last Occurrence time, Occurrence Count.

> **INFO**
>
> Kubernetes allows administrators to configure event retention periods through the Event TTL controller with a default retention period of 1 hour. Expired events are automatically purged by the system. For comprehensive historical records, access the All Events.

How To

Setting Scheduled Task Trigger Rules

The scheduled task trigger rules support the input of Crontab expressions.

TOC

Time Conversion

Time conversion rule: Local time - time zone offset = UTC

Taking **Beijing time to UTC time** as an example:

Beijing is in the East Eight Time Zone, with a time difference of 8 hours between Beijing time and UTC. The time conversion rule is:

`Beijing Time - 8 = UTC`

Example 1: Beijing time 9:42 converts to UTC time: 42 09 - 00 08 = 42 01, which means the UTC time is 1:42 AM.

Example 2: Beijing time 4:32 AM converts to UTC time: 32 04 - 00 08 = -68 03. If the result is negative, it indicates the previous day, requiring another conversion: -68 03 + 00 24 = 32 20, which means the UTC time is 8:32 PM of the previous day.

Writing Crontab Expressions

Basic format and value range of Crontab: `minute hour day month weekday`, with the corresponding value ranges as shown in the table below:

Minute	Hour	Day	Month	Weekday
[0-59]	[0-23]	[1-31]	[1-12] or [JAN-DEC]	[0-6] or [SUN-SAT]

The special characters allowed in the `minute hour day month weekday` fields include:

- `,`: Value list separator, used to specify multiple values. For example: `1,2,5,7,8,9`.
- `-`: User-defined value range. For example: `2-4`, which represents 2, 3, 4.
- `*`: Represents the entire time period. For example, when used for minutes, it means every minute.
- `/`: Used to specify the increment of values. For example: `n/m` indicates starting from n, increasing by m each time.

Conversion tool reference

Common Examples:

- Input `30 18 25 12 *` indicates a task triggers at `18:30:00 on December 25th`.
- Input `30 18 25 * 6` indicates a task triggers at `18:30:00 every Saturday`.
- Input `30 18 * * 6` indicates a task triggers at `18:30:00 on Saturdays`.
- Input `* 18 * * *` indicates a task triggers every minute starting from `18:00:00` (including `18:00:00`).
- Input `0 18 1,10,22 * *` indicates a task triggers at `18:00:00 on the 1st, 10th, and 22nd of every month`.
- Input `0,30 18-23 * * *` indicates a task triggers at `00 minutes and 30 minutes of each hour between 18:00 and 23:00 daily`.
- Input `* */1 * * *` indicates a task triggers every minute.

- Input `* 2-7/1 * * *` indicates a task triggers every minute between 2 AM and 7 AM daily.
- Input `0 11 4 * mon-wed` indicates a task triggers at `11:00 AM on the 4th of every month and on every Monday to Wednesday`.

Add ImagePullSecrets to ServiceAccount

If the image repository requires authentication, you need to add the corresponding `ImagePullSecrets` to the `ServiceAccount` used by the application. This ensures that the application can successfully pull images from the private repository.

TOC

Create an ImagePullSecret

To create an ImagePullSecret, please refer to Creating a Secret for detailed steps on creating an ImagePullSecret.

Add an ImagePullSecret to a ServiceAccount

If the Pod of your application uses the `ServiceAccount` `example`, you can add the `ImagePullSecret` to the `example` `ServiceAccount` in the namespace where your application is located.

Edit the `ServiceAccount` `example` with patch command:

```
kubectl patch serviceaccount example -p '{"imagePullSecrets": [{"name":
"my-registry-creds"}]}' -n <namespace>
```

Replace `<namespace>` with the namespace where your application is located, and `my-registry-creds` with the name of the `ImagePullSecret` you created.

You can verify the addition of the `ImagePullSecret` by describing the `ServiceAccount`:

```
kubectl describe serviceaccount example -n <namespace>
Name:                example
Namespace:           <namespace>
Labels:              <none>
Annotations:         <none>
Image pull secrets:  my-registry-creds
Mountable secrets:   <none>
Tokens:              <none>
Events:              <none>
```

You should see the `Image pull secrets` section showing the added secret.

> **NOTE**
>
> **Note**: If your Pod does not specify a `ServiceAccount`, it will use the `default` `ServiceAccount` in the namespace by default. You can add the `ImagePullSecret` to the `default` `ServiceAccount` in the same way.

Verify that imagePullSecrets are set for new Pods

When you create a new Pod that uses the `ServiceAccount` `example`, the Pod will automatically use the `ImagePullSecrets` specified in the `ServiceAccount`.

You can verify this by run the command:

```
kubectl get pod <pod-name> -n <namespace> -o=jsonpath='{.spec.imagePullSecrets}'
```

Use an ImagePullSecret with an OCI Connector

When a workload image is selected through an OCI Connector, the workload still needs an image pull Secret to authenticate to the Connector proxy. The Connector avoids distributing long-lived credentials for the original registry to workload namespaces, but this does not mean that the workload needs no Kubernetes image pull Secret. The Connector proxy Secret uses a ServiceAccount token that is authorized to access the Connector instead of the original registry password.

You can reference the Secret directly from the Pod template, or attach it to the ServiceAccount used by the workload. Either method is sufficient.

Prerequisites

Before creating the Secret, confirm the following:

- Alauda DevOps Connectors is installed, the Connector is Ready, and `enable-pod-image-pull-via-connector` is enabled.
- The OCI proxy is exposed through NodePort or Ingress, and every target node's container runtime can access the proxy address. If the proxy uses HTTP, the runtime must allow that proxy address.
- The ServiceAccount used to create the token is authorized to access the Connector.
- The Secret is created in the same namespace as the workload.

Create the Connector proxy Secret

Get the Connector proxy address and create a token for the workload ServiceAccount:

```
kubectl get connector <connector-name> -n <connector-namespace> \
  -o jsonpath='{.status.proxy.httpAddress.url}'

kubectl create token <service-account> \
  -n <workload-namespace> \
  --duration=<approved-duration>
```

Create the image pull Secret:

```
kubectl create secret docker-registry <connector-pull-secret> \
  --docker-server='<connector-proxy-host-and-port>' \
  --docker-username='u' \
  --docker-password='<service-account-token>' \
  --docker-email='<email>' \
  -n <workload-namespace>
```

The `--docker-server` value must identify the Connector proxy, not the original registry. If `.status.proxy.httpAddress.url` includes a Connector path, use the proxy host and port as the registry server. The command creates a Secret of type `kubernetes.io/dockerconfigjson`.

The token must come from a ServiceAccount that has permission to access the Connector. A token created without an approved duration is valid for approximately one hour by default. Select `--duration` according to your organization's security policy, and rotate or recreate the Secret before the token expires. Do not treat a manually copied token as a permanent credential or rotation mechanism.

Reference the Secret

To reference the Secret directly from the Pod template, configure:

```
spec:
  imagePullSecrets:
    - name: <connector-pull-secret>
```

Alternatively, attach the Secret to the ServiceAccount used by the workload:

```
kubectl patch serviceaccount <service-account> \
  -n <workload-namespace> \
  -p '{"imagePullSecrets":[{"name":"<connector-pull-secret>"}]}'
```

The Pod template reference and ServiceAccount inheritance are alternative valid sources. You do not need to configure both.

Verify and troubleshoot

After creating or updating the workload, verify the Connector path in this order:

1. Check that the workload Pod template has the expected `connectors.<baseDomain>/connectors` annotation, including mappings for every container and init container that uses an OCI Connector.
2. Check the image pull Secret selected by the Pod or inherited from its ServiceAccount.
3. Check that the actual Pod image was rewritten to the Connector proxy address.
4. Check that the Pod reaches `Running` (or `Completed` for a completed Job), and then review its Events.

Run the following commands:

```
kubectl get serviceaccount <service-account> -n <workload-namespace> -o yaml
kubectl get pod <pod-name> -n <workload-namespace> -o jsonpath='{.spec.imagePullSecrets}'
kubectl get pod <pod-name> -n <workload-namespace> -o jsonpath='{.spec.containers[*].image}'
kubectl get pod <pod-name> -n <workload-namespace>
kubectl describe pod <pod-name> -n <workload-namespace>
```

If the actual Pod image still points to the original registry, check the Connector annotation, Secret namespace and type, ServiceAccount association, and token validity first. An HTTPS error for the original HTTP registry can indicate that the workload did not enter the Connector proxy path; do not work around it by permanently allowing the original registry as an insecure registry.

Service Serving Certificate

TOC

Overview

Service serving certificates provide automated TLS certificate generation and management for Services in the container cloud platform. This capability enables secure HTTPS communication and encrypted traffic between internal components, ensuring data privacy and system security.

Features

This automated certificate provisioning mechanism provides the following key features:

- **Zero-touch Provisioning**: Developers do not need to manually generate Certificate Signing Requests (CSRs) or interact with certificate authorities. Certificate issuance is fully automated via standard Kubernetes annotations.
- **Cross-namespace CA Synchronization**: Eliminates the hassle of manually copying the platform root CA to every business namespace. The root CA certificate is automatically synchronized, enabling services to easily verify each other's identities.
- **Automated Lifecycle Management**: Leveraging `cert-manager` ensures that certificates are not only automatically issued but also seamlessly renewed before expiration, minimizing operational overhead and preventing outages due to expired certificates.
- **Standardized Security**: Ensures that TLS certificates are generated using a unified, centrally managed `ClusterIssuer`, maintaining consistent security standards across all applications on the platform.

Configuration and Deployment (For Administrators)

To implement the above functions, platform system administrators need to provision the related admission policies (`ClusterPolicy`) for `Kyverno` and the necessary RBAC permissions in advance.

Prerequisites

Before proceeding, ensure that the **Alauda Container Platform Compliance with Kyverno** plugin is installed and enabled in your cluster. For installation details, please refer to Install Compliance Plugin.

1. Configure RBAC Permissions

To grant Kyverno the permissions to create corresponding resources and call related interfaces, aggregate the following permissions to the platform's background controller:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    rbac.kyverno.io/aggregate-to-background-controller: "true"
    rbac.kyverno.io/aggregate-to-admission-controller: "true"
    rbac.kyverno.io/aggregate-to-reports-controller: "true"
  name: kyverno:serving-certs
rules:
  - apiGroups:
      - cert-manager.io
    resources:
      - certificates
    verbs:
      - create
      - update
      - delete
      - get
      - list
      - watch
  - apiGroups:
      - ""
    resources:
      - secrets
    verbs:
      - create
      - update
      - delete
      - get
      - list
      - watch
```

2. Create Certificate

Create the following Certificate resource to act as the service root CA:

```yaml
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: service-root-ca
  namespace: cert-manager
spec:
  commonName: ServiceRootCA
  isCA: true
  issuerRef:
    kind: ClusterIssuer
    name: cpaas-ca
  secretName: service-root-ca
```

3. Configure Kyverno Policies

Create and apply the following three core `ClusterPolicy` resources:

- **clone-ca-secret**: Listens to Namespace creation events and generates the `service-root-ca` Secret resource for new namespaces.
- **sync-ca-rotation**: Listens to changes in `service-root-ca`, obtains the real CA certificate content via the API, and synchronizes the `ca.crt` value to the Secret in the target namespace.
- **generate-service-cert**: Listens to Service resources with the `service.alauda.io/serving-cert-secret-name` annotation and generates a `Certificate` issuance request with wildcard domain support based on its name and namespace.

Save the following YAML content as a file and execute the application (`kubectl apply -f`):

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: clone-ca-secret
spec:
  background: true
  rules:
  - name: generate-ca-secret-shell
    match:
      any:
      - resources:
          kinds:
          - Namespace
    exclude:
      any:
      - resources:
          namespaces:
          - kube-system
          - cpaas-system
          - cert-manager
    generate:
      apiVersion: v1
      kind: Secret
      name: service-root-ca
      namespace: "{{request.object.metadata.name}}"
      synchronize: true
      generateExisting: true
      data:
        kind: Secret
        type: Opaque
        metadata:
          labels:
            app.kubernetes.io/managed-by: kyverno
            generate.kyverno.io/policy-name: clone-ca-secret

  - name: initial-sync-on-creation
    match:
      any:
      - resources:
          kinds:
          - Secret
          names:
          - service-root-ca
```

```yaml
    exclude:
      any:
      - resources:
          namespaces:
          - cert-manager
          - kube-system
          - cpaas-system
    context:
    - name: cacrt
      apiCall:
        method: GET
        urlPath: "/api/v1/namespaces/cert-manager/secrets/service-root-c
a"
        jmesPath: 'data."ca.crt"'
    mutate:
      patchStrategicMerge:
        data:
          ca.crt: "{{ cacrt }}"
---
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: sync-ca-rotation
spec:
  mutateExistingOnPolicyUpdate: true
  background: true
  rules:
  - name: rotation-sync
    match:
      any:
      - resources:
          kinds:
          - Secret
          namespaces:
          - cert-manager
          names:
          - service-root-ca
    mutate:
      targets:
      - apiVersion: v1
        kind: Secret
        name: service-root-ca
        selector:
          matchLabels:
```

```yaml
          app.kubernetes.io/managed-by: kyverno
          generate.kyverno.io/policy-name: clone-ca-secret
      patchStrategicMerge:
        data:
          ca.crt: '{{ request.object.data."ca.crt" }}'
---
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: generate-service-cert
spec:
  background: true
  rules:
    - name: generate-cert
      match:
        any:
          - resources:
              kinds:
                - Service
              annotations:
                service.alauda.io/serving-cert-secret-name: "?*"
      generate:
        apiVersion: cert-manager.io/v1
        kind: Certificate
        name: "{{request.object.metadata.name}}-cert"
        namespace: "{{request.object.metadata.namespace}}"
        synchronize: true
        data:
          spec:
            secretName: '{{request.object.metadata.annotations."service.a
lauda.io/serving-cert-secret-name"}}'
            issuerRef:
              name: cpaas-ca
              kind: ClusterIssuer
            dnsNames:
              - "{{request.object.metadata.name}}"
              - "{{request.object.metadata.name}}.{{request.object.metada
ta.namespace}}"
              - "{{request.object.metadata.name}}.{{request.object.metada
ta.namespace}}.svc"
              - "{{request.object.metadata.name}}.{{request.object.metada
ta.namespace}}.svc.cluster.local"
```

User Guide (For Developers)

For application developers, once the underlying cluster policies are in place, there is no need to worry about the underlying certificate issuance logic. You only need to append an exclusive annotation when defining a business Service resource to achieve automatic certificate provisioning:

1. **Add Annotation to the Service**

Under the business namespace (e.g., `my-namespace`), create or update your Service, and add the `service.alauda.io/serving-cert-secret-name` annotation to specify the name of the generated certificate Secret. For example:

```
apiVersion: v1
kind: Service
metadata:
  name: my-secure-service
  namespace: my-namespace
  annotations:
    service.alauda.io/serving-cert-secret-name: "my-secure-service-tls" # Will generate a certificate Secret named my-secure-service-tls for you
spec:
  ports:
    - port: 443
      targetPort: 8443
  selector:
    app: my-app
```

2. **Verify the Generated Certificate Secret**

After applying the above Service:

- Check if the auto-generated TLS certificate Secret exists in the namespace:

```
kubectl get secret my-secure-service-tls -n my-namespace
```

- Check if the root CA certificate used for verification has also been automatically copied to the current namespace:

```
kubectl get secret service-root-ca -n my-namespace
```

3. **Mount and Use the Certificate in Pods**

Since the CA certificate Secret (`service-root-ca`) and the current application's TLS serving certificate Secret (`my-secure-service-tls`) are in place, you can mount them as data volumes in workloads such as your target application's `Deployment` or `StatefulSet` .

Here is a specific `Deployment` YAML example demonstrating how to mount these certificates:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: my-namespace
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app-container
          image: my-app-image:latest
          volumeMounts:
            # Mount the serving certificate
            - name: tls-cert
              mountPath: "/etc/tls/certs"
              readOnly: true
            # Mount the root CA certificate
            - name: root-ca
              mountPath: "/etc/tls/ca"
              readOnly: true
      volumes:
        # Reference the automatically generated Service certificate Secret
        - name: tls-cert
          secret:
            secretName: my-secure-service-tls
        # Reference the automatically synchronized CA certificate Secret
        - name: root-ca
          secret:
            secretName: service-root-ca
```

For more configuration details regarding workloads via the web console, refer to the Configure Containers section.

Images

Overview of images

How To

Creating images

Learning container best practices

Including metadata in images

Managing images

Image pull policy

Using image pull secrets

Overview of images

TOC

Understanding containers and images

Containers and images are important concepts to understand when you set out to create and manage containerized software. An image holds a set of software that is ready to run, while a container is a running instance of a container image. Those different versions are represented by different tags on the same image name.

Images

Containers in Alauda Container Platform are based on OCI-formatted container images. An image is a binary that includes all of the requirements for running a single container, as well as metadata describing its needs and capabilities.

You can think of it as a packaging technology. Containers have access only to resources defined in the image unless granted additional access at creation time. By deploying the same image in multiple containers across multiple hosts and load balancing between them, Alauda Container Platform can provide redundancy and horizontal scaling for a service packaged into an image.

You can use the `nerdctl` or `container` CLI directly to build images, but Alauda Container Platform also supplies builder images that assist with creating new images by adding your code or configuration to existing images.

Because applications develop over time, a single image name can actually refer to many different versions of the same image. Each different image is referred to uniquely by its hash, a long hexadecimal number such as fd44297e2ddb050ec4f…, which is usually shortened to 12 characters, such as fd44297e2ddb.

Image registry

An image registry is a content server that can store and serve container images. For example:

- Quay.io Container Registry ↗
- Alauda Container Platform Registry

A registry contains a collection of one or more image repositories, which contain one or more tagged images. Alauda Container Platform can supply its own image registry for managing custom container images.

Image repository

An image repository is a collection of related container images and tags identifying them. For example, the Alauda Container Platform Jenkins images are in the repository:

```
jenkins-2-centos7
```

Image tags

An image tag is a label applied to a container image in a repository that distinguishes a specific image from other images in an image stream. Typically, the tag represents a version number of some sort. For example, here :v3 .11.59-2 is the tag:

```
jenkins-2-centos7:v3.11.59-2
```

You can add additional tags to an image. For example, an image might be assigned the tags :v3 .11.59-2 and :latest .

Image IDs

An image ID is a SHA (Secure Hash Algorithm) code that can be used to pull an image. A SHA image ID cannot change. A specific SHA identifier always references the exact same container image content. For example:

```
jenkins-2-centos7@sha256:ab312bda324
```

Containers

The basic units of Alauda Container Platform applications are called containers. Linux container technologies are lightweight mechanisms for isolating running processes so that they are limited to interacting with only their designated resources. The word container is defined as a specific running or paused instance of a container image.

Many application instances can be running in containers on a single host without visibility into each others' processes, files, network, and so on. Typically, each container provides a single service, often called a micro-service, such as a web server or a database, though containers can be used for arbitrary workloads.

The Linux kernel has been incorporating capabilities for container technologies for years. Early container runtime projects introduced convenient management interfaces for running isolated applications on a host. More recently, the Open Container Initiative ↗ has developed open standards for container formats and container runtimes. Alauda Container Platform and Kubernetes add the ability to orchestrate OCI-formatted containers across multi-host installations.

Though you do not directly interact with container runtimes when using Alauda Container Platform, understanding their capabilities and terminology is important for understanding their role in Alauda Container Platform and how your applications function inside of containers.

How To

Creating images

Learn how to create your own container images, based on pre-built images that are ready to help you. The process includes learning best practices for writing images, defining metadata for images, testing images, and using a custom builder workflow to create images to use with Alauda Container Platform Registry. After you create an image, you can push it to the **Alauda Container Platform Registry**.

TOC

Learning container best practices

When creating container images to run on Alauda Container Platform there are a number of best practices to consider as an image author to ensure a good experience for consumers of those images. Because images are intended to be immutable and used as-is, the following guidelines help ensure that your images are highly consumable and easy to use on Alauda Container Platform.

General container image guidelines

The following guidelines apply when creating a container image in general, and are independent of whether the images are used on Alauda Container Platform.

Reuse images

Wherever possible, base your image on an appropriate upstream image using the FROM statement. This ensures your image can easily pick up security fixes from an upstream image when it is updated, rather than you having to update your dependencies directly.

In addition, use tags in the FROM instruction, for example, `alpine:3.20`, to make it clear to users exactly which version of an image your image is based on. Using a tag other than latest ensures your image is not subjected to breaking changes that might go into the latest version of an upstream image.

Maintain compatibility within tags

When tagging your own images, try to maintain backwards compatibility within a tag. For example, if you provide an image named image and it currently includes version `1.0`, you might provide a tag of `image:v1`. When you update the image, as long as it continues to be compatible with the original image, you can continue to tag the new image `image:v1`, and downstream consumers of this tag are able to get updates without being broken.

If you later release an incompatible update, then switch to a new tag, for example `image:v2`. This allows downstream consumers to move up to the new version at will, but not be inadvertently broken by the new incompatible image. Any downstream consumer using `image:latest` takes on the risk of any incompatible changes being introduced.

Avoid multiple processes

Do not start multiple services, such as a database and `SSHD`, inside one container. This is not necessary because containers are lightweight and can be easily linked together for orchestrating multiple processes. Alauda Container Platform allows you to easily colocate and co-manage related images by grouping them into a single pod.

This colocation ensures the containers share a network namespace and storage for communication. Updates are also less disruptive as each image can be updated less frequently and independently. Signal handling flows are also clearer with a single process as you do not have to manage routing signals to spawned processes.

Use exec in wrapper scripts

Many images use wrapper scripts to do some setup before starting a process for the software being run. If your image uses such a script, that script uses `exec` so that the script's process is replaced by your software. If you do not use `exec`, then signals sent by your container runtime go to your wrapper script instead of your software's process. This is not what you want.

If you have a wrapper script that starts a process for some server. You start your container, for example, using `podman run -i`, which runs the wrapper script, which in turn starts your process. If you want to close your container with `CTRL+C`. If your wrapper script used `exec` to start the server process, `podman` sends SIGINT to the server process, and everything works as you expect. If you did not use `exec` in your wrapper script, `podman` sends SIGINT to the process for the wrapper script and your process keeps running like nothing happened.

Also note that your process runs as `PID 1` when running in a container. This means that if your main process terminates, the entire container is stopped, canceling any child processes you launched from your `PID 1` process.

Clean temporary files

Remove all temporary files you create during the build process. This also includes any files added with the `ADD` command. For example, run the `yum clean` command after performing `yum install` operations.

You can prevent the `yum` cache from ending up in an image layer by creating your `RUN` statement as follows:

```
RUN yum -y install mypackage && yum -y install myotherpackage && yum clean all -y
```

Note that if you instead write:

```
RUN yum -y install mypackage
RUN yum -y install myotherpackage && yum clean all -y
```

Then the first `yum` invocation leaves extra files in that layer, and these files cannot be removed when the `yum clean` operation is run later. The extra files are not visible in the final image, but they are present in the underlying layers.

The current container build process does not allow a command run in a later layer to shrink the space used by the image when something was removed in an earlier layer. However, this may change in the future. This means that if you perform an `rm` command in a later layer, although the files are hidden it does not reduce the overall size of the image to be downloaded. Therefore, as with the `yum clean` example, it is best to remove files in the same command that created them, where possible, so they do not end up written to a layer.

In addition, performing multiple commands in a single `RUN` statement reduces the number of layers in your image, which improves download and extraction time.

Place instructions in the proper order

The container builder reads the `Dockerfile` and runs the instructions from top to bottom. Every instruction that is successfully executed creates a layer which can be reused the next time this or another image is built. It is very important to place instructions that rarely change at the top of your `Dockerfile`. Doing so ensures the next builds of the same image are very fast because the cache is not invalidated by upper layer changes.

For example, if you are working on a `Dockerfile` that contains an `ADD` command to install a file you are iterating on, and a `RUN` command to `yum install` a package, it is best to put the `ADD` command last:

```
FROM foo
RUN yum -y install mypackage && yum clean all -y
ADD myfile /test/myfile
```

This way each time you edit `myfile` and rerun `podman build`, the system reuses the cached layer for the `yum` command and only generates the new layer for the `ADD` operation.

If instead you wrote the Dockerfile as:

```
FROM foo
ADD myfile /test/myfile
RUN yum -y install mypackage && yum clean all -y
```

Then each time you changed `myfile` and reran `podman build`, the `ADD` operation would invalidate the `RUN` layer cache, so the `yum` operation must be rerun as well.

Mark important ports

The `EXPOSE` instruction makes a port in the container available to the host system and other containers. While it is possible to specify that a port should be exposed with a `podman run` invocation, using the `EXPOSE` instruction in a `Dockerfile` makes it easier for both humans and software to use your image by explicitly declaring the ports your software needs to run:

- Exposed ports show up under `podman ps` associated with containers created from your image.
- Exposed ports are present in the metadata for your image returned by `podman inspect`.
- Exposed ports are linked when you link one container to another.

Set environment variables

It is good practice to set environment variables with the `ENV` instruction. One example is to set the version of your project. This makes it easy for people to find the version without looking at the `Dockerfile`. Another example is advertising a path on the system that could be used by another process, such as `JAVA_HOME`.

Avoid default passwords

Avoid setting default passwords. Many people extend the image and forget to remove or change the default password. This can lead to security issues if a user in production is assigned a well-known password. Passwords are configurable using an environment variable instead.

If you do choose to set a default password, ensure that an appropriate warning message is displayed when the container is started. The message should inform the user of the value of the default password and explain how to change it, such as what environment variable to set.

Avoid sshd

It is best to avoid running `sshd` in your image. You can use `podman exec` to access containers running on your local host. In a cluster, use `kubectl exec` to access containers managed by Alauda Container Platform. Installing and running `sshd` in your image opens additional attack surface and increases patching burden.

Use volumes for persistent data

Images use a volume for persistent data. This way Alauda Container Platform mounts the network storage to the node running the container, and if the container moves to a new node the storage is reattached to that node. By using the volume for all persistent storage needs, the content is preserved even if the container is restarted or moved. If your image writes data to arbitrary locations within the container, that content could not be preserved.

All data that needs to be preserved even after the container is destroyed must be written to a volume. Container engines support a `readonly` flag for containers, which can be used to strictly enforce good practices about not writing data to ephemeral storage in a container. Designing your image around that capability now makes it easier to take advantage of it later.

Explicitly defining volumes in your `Dockerfile` makes it easy for consumers of the image to understand what volumes they must define when running your image.

See the Kubernetes documentation ↗ for more information on how volumes are used in Alauda Container Platform.

> **Note:**
> Even with persistent volumes, each instance of your image has its own volume, and the filesystem is not shared between instances. This means the volume cannot be used to share state in a cluster.

Including metadata in images

Defining image metadata helps Alauda Container Platform better consume your container images, allowing Alauda Container Platform to create a better experience for developers using your image. For example, you can add metadata to provide helpful descriptions of your image, or offer suggestions on other images that may also be needed.

The metadata set covers the fields needed by the current use cases. Additional metadata or use cases may be added in the future.

Defining image metadata

You can use the `LABEL` instruction in a `Dockerfile` to define image metadata. Labels are similar to environment variables in that they are key value pairs attached to an image or a container. Labels are different from environment variable in that they are not visible to the running application and they can also be used for fast look-up of images and containers.

See the Dockerfile reference ↗ for more information on the `LABEL` instruction.

The label names are typically namespaced. The namespace is set accordingly to reflect the project that is going to pick up the labels and use them. For Kubernetes the namespace is io.k8s.

Managing images

With Alauda Container Platform you can interact with images, depending on where the registries of the images are located, any authentication requirements around those registries, and how you want your builds and deployments to behave.

TOC

Image pull policy

Each container in a pod has a container image. After you have created an image and pushed it to a registry, you can then refer to it in the pod.

Image pull policy overview

When Alauda Container Platform creates containers, it uses the container `imagePullPolicy` to determine if the image should be pulled prior to starting the container. There are three possible values for `imagePullPolicy`:

Table `imagePullPolicy` values:

Value	Description
Always	Always pull the image.
IfNotPresent	Only pull the image if it does not already exist on the node.
Never	Never pull the image.

If a container imagePullPolicy parameter is not specified, Alauda Container Platform sets it based on the image tag:

1. If the tag is latest, Alauda Container Platform defaults imagePullPolicy to Always.
2. Otherwise, Alauda Container Platform defaults imagePullPolicy to IfNotPresent.

Using image pull secrets

If you are using the Alauda Container Platform image registry, then your pod service account should already have the correct permissions and no additional action should be required.

However, for other scenarios, such as referencing images across Alauda Container Platform projects or from secured registries, additional configuration steps are required.

Allowing pods to reference images from other secured registries

To pull a secured container from other private or secured registries, you must create a pull secret from your container client credentials, such as `Podman`, and add it to your service account.

Container clients use a configuration file to store authentication details to log in to secured or insecure registry:

These files store your authentication information if you have previously logged in to a secured or insecure registry.

Creating a pull secret

You can obtain the image pull secret to pull an image from a private container image registry or repository. You can refer to Pull an Image from a Private Registry ↗.

Using a pull secret in a workload

You can use a pull secret to allow workloads to pull images from a private registry with one of the following methods:

- By linking the secret to a `ServiceAccount`, which automatically applies the secret to all pods using that service account.
- By defining `imagePullSecrets` in the pod specification, which is useful for environments like GitOps or ArgoCD.

You can use a secret for pulling images for pods by adding the secret to your service account. Note that the name of the service account should match the name of the service account that pod uses.

Example output:

```yaml
apiVersion: v1
imagePullSecrets:
- name: default-cfg-123456
- name: <pull_secret_name>
kind: ServiceAccount
metadata:
  name: default
  namespace: default
secrets:
- name: <pull_secret_name>
```

Instead of linking the secret to a service account, you can alternatively reference it directly in your pod or workload definition. This is useful for GitOps workflows such as ArgoCD. For example:

Example pod specification:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: <secure_pod_name>
spec:
  containers:
  - name: <container_name>
    image: your.registry.io/my-private-image
  imagePullSecrets:
  - name: <pull_secret_name>
```

Example ArgoCD workflow:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: <example_workflow>
spec:
  entrypoint: <main_task>
  imagePullSecrets:
  - name: <pull_secret_name>
```

Registry Image Use

ACP provides two Registry documentation tracks:

Track	Use it for	Continue with
Legacy Registry workflows	Existing environments that still use the earlier Registry service, `image-registry.cpaas-system.svc`, legacy push limit ConfigMaps, and the current `ac get images` / `ac delete images` workflows.	How To
Registry v2 image use	Pull, push, list, tag, and manage images in the Operator-managed ACP Registry after Registry v2 has been installed.	Registry v2

The legacy Registry pages remain available for compatibility. Registry v2 administrator tasks, including installation, configuration, and exposure, are documented in Registry v2 administration.

Registry v2: Image Use

How To

Registry v2: Image Use

Use this section for developer and workload access to **Registry v2**, the Operator-managed ACP image registry.

Registry v2 administrator tasks, including installation, configuration, and exposure, are documented in Registry v2 administration.

The legacy Registry documentation remains available under How To for environments that still use the earlier `cpaas-system` Registry service.

TOC

Documentation Structure

Need	Continue with
Access Registry v2 from workloads or external clients.	Accessing the registry
Use `ac` to list, tag, import, mirror, delete, and inspect images.	Managing images with ac
Review Registry v2 components and administrator settings.	Registry v2 administration

Standard Address

Use the fully qualified internal service address for workloads running in the cluster:

```
image-registry.image-registry-system.svc:5000
```

Use an external Ingress hostname only for developer machines, CI systems, or other clients outside the cluster. See Exposing the registry.

Registry v2: Accessing the Registry

Use this page to access **Registry v2** from inside the cluster, from external clients, and through the ACP Image API.

TOC

Prerequisites

- Registry v2 is installed and available in the target cluster.
- The image namespace exists.
- The current user or service account has the required namespace permissions.

Required Namespace Permissions

Your user account or workload service account must have permissions in the namespace that owns the image repository. Ask a namespace administrator to grant the required image roles when you need cross-namespace access.

Registry v2 uses ImageStream layer authorization:

Operation	Typical role	Image API permission
Pull	`system:image-puller`	`image.alauda.io` `imagestreams/layers` `get`
Push	`system:image-pusher`	`image.alauda.io` `imagestreams/layers` `update`
Delete	`system:image-deleter`	Image API delete permissions for the target image metadata

For role binding examples, see Managing Registry v2 access and cleanup.

Access the Registry from Inside the Cluster

Use the internal service address for workloads inside the cluster:

```
image-registry.image-registry-system.svc:5000
```

Example workload image reference:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: team-a
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      serviceAccountName: default
      containers:
        - name: app
          image: image-registry.image-registry-system.svc:5000/team-a/my-
app:v1
```

For a custom service account, make sure it has pull permission in the image namespace. The Operator injects the managed pull secret when Registry v2 is configured to manage service account pull secrets.

Create the service account in the workload namespace:

```
kubectl create serviceaccount app-puller -n team-b
```

Have a namespace administrator grant the service account pull permission in the image namespace:

```
kubectl create rolebinding image-puller-app-puller \
  --clusterrole=system:image-puller \
  --serviceaccount=team-b:app-puller \
  -n team-a
```

Verify that the managed pull secret is injected:

```
kubectl -n team-b get serviceaccount app-puller \
  -o jsonpath='{.imagePullSecrets[*].name}{"\n"}'
```

Expected result:

- The command returns at least one image pull secret name.

If no pull secret is listed, ask a Registry administrator to verify that managed service account pull secrets are enabled and that the workload namespace is not ignored.

Use the service account in the workload:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: team-b
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      serviceAccountName: app-puller
      containers:
        - name: app
          image: image-registry.image-registry-system.svc:5000/team-a/my-app:v1
```

Verify the rollout:

```
kubectl -n team-b rollout status deployment/my-app --timeout=300s
kubectl -n team-b get pods -l app=my-app
```

Authenticate an External OCI Client

For external access, use `ac registry login` to write credentials for a ACP account that has namespace permissions:

```
ac registry login \
  --registry registry.example.com
```

To write credentials to a specific Docker-compatible auth file, set `DOCKER_CONFIG` or use `--to`:

```
export DOCKER_CONFIG=/tmp/registry-auth

ac registry login \
  --registry registry.example.com \
  --to "$DOCKER_CONFIG/config.json"
```

If the registry uses a private CA, configure client trust before push and pull operations. Use `--insecure` only for an HTTP endpoint or a non-production test certificate that the client does not trust.

Push and Pull Images

Tag and push an image:

```
nerdctl tag my-app:latest registry.example.com/team-a/my-app:v1
nerdctl push registry.example.com/team-a/my-app:v1
```

Pull by tag:

```
nerdctl pull registry.example.com/team-a/my-app:v1
```

Pull by digest:

```
nerdctl pull registry.example.com/team-a/my-app@sha256:<digest>
```

Query Image API Resources

List ImageStreams:

```
kubectl get imagestreams.image.alauda.io -A
ac get imagestreams -A
```

Show one ImageStream:

```
ac get imagestreams my-app -n team-a -o yaml
```

Show the current and historical digest for a tag:

```
ac get imagestreamtags my-app:v1 -n team-a -o wide
```

Show the Image metadata for a digest:

```
ac get imagestreamimages my-app@sha256:<digest> -n team-a -o yaml
```

List Registry HTTP image repositories visible to the current user:

```
ac get images --registry-url=https://registry.example.com
ac get images -n team-a --registry-url=https://registry.example.com
```

Registry v2: Managing Images with ac

Use the latest `ac` image commands to work with **Registry v2** and the ACP Image API. Registry v2 is selected by `modern` registry mode and provides Image and ImageStream workflows for ACP clusters. Compatibility entry points remain available for environments that still use the legacy Registry.

TOC

Registry Mode

`ac` selects the image backend from the current cluster's registry mode:

Mode	Backend	Use it for
`modern`	Registry v2 `image.alauda.io/v1` Image and ImageStream API plus the registry service.	Registry v2 management.
`legacy`	Legacy ACP Registry catalog, tag, and manifest metadata.	Environments that still use the legacy Registry.
`auto`	Automatic detection.	Discovery. If both generations are detected, set the mode explicitly before image management.

Check and set the mode:

```
ac config get-registry-mode
ac config set-registry-mode modern
```

Use `modern` mode before managing ImageStreams, importing images, tagging images, or viewing Image API usage through the Registry v2 Image API graph.

Command Groups

Workflow	Command	Registry mode behavior
Discover and log in to the integrated Registry	`ac registry info` , `ac registry login`	Uses the current registry mode to resolve the Registry host.
List and delete integrated Registry images	`ac get images` , `ac delete images`	Mode-aware compatibility commands.
Manage ImageStreams	`ac create imagestream` , `ac create imagestreamtag` , `ac import-image` , `ac tag` , `ac set image-lookup`	Requires `modern` mode.

Workflow	Command	Registry mode behavior
Query Image API resources	`ac get imagestreams`, `ac get imagestreamtags`, `ac get imagestreamimages`	Requires the Registry v2 Image API.
Work with OCI images directly	`ac image info`, `ac image mirror`, `ac image append`, `ac image extract`	Talks to registries directly. Use `--insecure` only for scoped test or private-CA cases.

Administrative usage reporting, cleanup, and trust verification are documented in Managing Registry v2 access and cleanup.

Discover and Log In to the Registry

Print the Registry host selected by the current registry mode:

```
ac registry info
```

Print the internal host:

```
ac registry info --internal
```

Check that the Registry `/v2/` endpoint is reachable:

```
ac registry info --check
```

Write Registry credentials to the default Docker config file:

```
ac registry login
```

Write credentials to a temporary or CI auth file:

```
export DOCKER_CONFIG=/tmp/registry-auth

ac registry login \
  --registry registry.example.com \
  --to "$DOCKER_CONFIG/config.json"
```

Use `--skip-check` when the registry endpoint is not reachable from the current machine but the credential file still needs to be written. Use `--insecure` only when the endpoint is HTTP or uses a certificate that the client does not trust.

Manage ImageStreams

Create an empty ImageStream:

```
ac create imagestream demo
```

Create an ImageStream with local image lookup enabled:

```
ac create imagestream demo --lookup-local
```

Create an ImageStreamTag from an external image:

```
ac create imagestreamtag demo:latest \
  --from-image=quay.io/example/demo:1.0
```

Import an external image into an ImageStreamTag. Without `--confirm`, the command previews the import. Add `--confirm` to persist the tag:

```
ac import-image demo:latest \
  --from=quay.io/example/demo:1.0 \
  --confirm
```

Common import options:

Option	Purpose
`--scheduled`	Periodically import the external image or repository.
`--insecure`	Import from a registry that is not fully secured by trusted HTTPS.
`--reference-policy=source|local`	Control whether workloads use the source reference or the local Registry reference.
`--import-mode=Legacy|PreserveOriginal`	Select how manifest lists and related image metadata are imported.
`-o json|yaml`	Print import results in a structured format.

Tag an external container image into an ImageStreamTag:

```
ac tag --source=docker quay.io/example/demo:1.0 demo:latest
```

Copy the current image behind one ImageStreamTag to another:

```
ac tag --source=imagestreamtag team-a/demo:stable team-b/demo:latest
```

Delete a tag:

```
ac tag -d demo:old
```

Enable local image lookup for one ImageStream:

```
ac set image-lookup demo
```

Disable local image lookup:

```
ac set image-lookup demo --enabled=false
```

List local lookup state:

```
ac set image-lookup --list
```

Query Images and ImageStreams

List ImageStreams:

```
ac get imagestreams -A
```

Show an ImageStream:

```
ac get imagestreams demo -n team-a -o yaml
```

Show the current digest and tag history for a tag:

```
ac get imagestreamtags demo:latest -n team-a -o wide
```

Show ImageStreamImage metadata by digest:

```
ac get imagestreamimages demo@sha256:<digest> -n team-a -o yaml
```

List images through the integrated Registry compatibility entry point:

```
ac get images
```

In `modern` mode, `ac get images` lists cluster-scoped Image resources with `NAME` and `IMAGE REFERENCE` columns. In `legacy` mode, it keeps the old ACP Registry catalog output and lists repository tags.

Use structured output:

```
ac get images -o yaml
ac get images -o json
```

Legacy-only options remain available for compatibility:

```
ac get images -n team-a --registry-url=http://old-registry.example.com:50
00
ac get images --fast --registry-url=http://old-registry.example.com:5000
```

Delete Image References

The preferred Registry v2 workflow is to delete ImageStream tags:

```
ac tag -d demo:old
```

The compatibility command `ac delete images` also follows the selected registry mode. It is a dry run unless `--confirm` is set:

```
ac delete images --repo team-a/demo:old
ac delete images --repo team-a/demo:old --confirm
```

Use repeated `--repo` flags to delete multiple tag references:

```
ac delete images \
  --repo team-a/demo:old \
  --repo team-a/api:canary \
  --confirm
```

Deleting image references does not necessarily release blob storage immediately. Run image pruning and registry garbage collection separately when storage reclamation is required.

Work with OCI Images Directly

Show image metadata from a registry:

```
ac image info registry.example.com/team-a/demo:latest
```

Select one platform from a multi-architecture image:

```
ac image info docker.io/library/busybox:latest \
  --filter-by-os=linux/arm64
```

Show manifest list entries:

```
ac image info registry.example.com/team-a/demo:latest \
  --show-multiarch
```

Copy one image to another tag:

```
ac image mirror \
  registry.example.com/team-a/demo:latest \
  registry.example.com/team-a/demo:stable
```

Copy multiple images from a mapping file:

```
cat > mirror-map.txt <<EOF
old-registry.example.com/team-a/demo:latest=registry.example.com/team-a/d
emo:latest
old-registry.example.com/team-a/api:v1=registry.example.com/team-a/api:v1
EOF

ac image mirror \
  -f mirror-map.txt \
  --dry-run \
  --keep-manifest-list

ac image mirror \
  -f mirror-map.txt \
  --continue-on-error \
  --keep-manifest-list \
  --max-per-registry=4
```

Add layers or metadata to an image and push the result:

```
ac image append \
  --from docker.io/library/busybox:latest \
  --to registry.example.com/team-a/busybox:patched \
  layer.tar.gz
```

Preview the append operation:

```
ac image append \
  --from docker.io/library/busybox:latest \
  --to registry.example.com/team-a/busybox:patched \
  layer.tar.gz \
  --dry-run
```

Extract files from an image without a local container runtime:

```
ac image extract docker.io/library/busybox:latest \
  --path /:/tmp/busybox \
  --confirm
```

Use `--insecure` on `ac image` commands only for test registries, HTTP endpoints, or private certificates that are not trusted by the client.

How To

Common CLI Command Operat

Using Alauda Container Platfor

Configure R

Common CLI Command Operations

Use the `ac` CLI for platform login, namespace permission management, and Registry metadata operations. Use a standard OCI client such as `nerdctl` for image transfer operations such as push and pull.

The examples below assume:

- Registry client address: `<REGISTRY_CLIENT_HOSTPORT>`
- Registry API URL: `<REGISTRY_API_URL>`
- Current namespace: `my-ns`

TOC

Before You Begin

Before running the commands in this document, make sure:

- `ac` is installed.
- `nerdctl` is installed if you need to push or pull images.
- You can reach the ACP API endpoint and the Registry address from your current environment.
- You have logged in to ACP and selected the target cluster.
- Your current ACP account, or the ServiceAccount used in a Pod or Job, has the required namespace permissions.

Typical permission requirements:

- Pull images: `system:image-puller`
- Push images: `system:image-pusher`
- List images with `ac get images`: access to the target namespaces
- Delete images with `ac delete images`: permission to delete images in the target namespaces

If you run `ac` inside a Pod, Job, or CronJob:

- The Pod must use a valid `serviceAccountName`.
- The mounted ServiceAccount token must be available.
- The ServiceAccount must have permissions to access the target cluster and Registry-related APIs.

How Authentication Works

`ac` and `nerdctl` use different authentication paths:

- `ac` uses the current ACP login session for Registry-related API access.
- If `ac` runs inside a Pod and no ACP session is available, it can fall back to the mounted ServiceAccount token.

- `nerdctl` authenticates directly to the Registry, typically by using the same ACP account credentials that already have access to the target namespace.

Address usage in this document:

- `<REGISTRY_CLIENT_HOSTPORT>` is the Registry address used by OCI clients such as `nerdctl`.
- `<REGISTRY_API_URL>` is the Registry API endpoint used by `ac get images` and `ac delete images`.
- In external access scenarios, `ac` should typically use an explicitly specified `--registry-url` instead of relying on the default in-cluster Registry address.

Authenticate to ACP

Before using Registry-related commands in `ac`, log in and select the target cluster:

```
ac login <acp-url>
ac config get-clusters
ac config use-cluster <cluster-name>
```

After login, `ac` can use the current session to access Registry-related APIs such as `ac get images` and `ac delete images`.

Grant namespace permissions to a user

Add namespace pull permission for a user.

```
ac create rolebinding <binding-name> --clusterrole=system:image-puller --user=<username> -n <namespace>
```

Add namespace push permissions to a user.

```
ac create rolebinding <binding-name> --clusterrole=system:image-pusher --
user=<username> -n <namespace>
```

Grant namespace permissions to a ServiceAccount

Add namespace pull permission for a service account.

```
ac create rolebinding <binding-name> --clusterrole=system:image-puller --
serviceaccount=<namespace>:<serviceaccount-name> -n <namespace>
```

Add namespace push permission for a service account.

```
ac create rolebinding <binding-name> --clusterrole=system:image-pusher --
serviceaccount=<namespace>:<serviceaccount-name> -n <namespace>
```

Authenticate an OCI Client

For image push and pull, use the same ACP account credentials that have access to the target namespace:

```
nerdctl login <REGISTRY_CLIENT_HOSTPORT> -u <ACP-USERNAME> -p <ACP-PASSWO
RD>
```

If the Registry uses a self-signed certificate or plain HTTP, add the global flag `--insecure-registry`.

Example:

```
nerdctl --insecure-registry login <REGISTRY_CLIENT_HOSTPORT> -u <ACP-USER
NAME> -p <ACP-PASSWORD>
```

List Images

List images from namespaces that the current user is allowed to access:

```
ac get images --registry-url=<REGISTRY_API_URL>

# List images from a specific namespace
ac get images -n my-ns --registry-url=<REGISTRY_API_URL>

# Use structured output
ac get images -o yaml --registry-url=<REGISTRY_API_URL>
```

When `ac` runs outside the cluster, explicitly specifying `--registry-url` is recommended. Otherwise, the CLI may fall back to the default in-cluster Registry address, which is often unreachable from a local workstation.

Example:

```
ac get images --registry-url=<REGISTRY_API_URL>
```

Pull Images

Pull an image from the Registry with `nerdctl`:

```
# Pull an image from the current namespace
nerdctl pull <REGISTRY_CLIENT_HOSTPORT>/my-ns/my-app:latest

# Pull an image from another namespace when you already have permission
nerdctl pull <REGISTRY_CLIENT_HOSTPORT>/shared-ns/base-image:latest
```

Push Images

Push a local image to the current namespace:

```
# Tag the local image with the target repository
nerdctl tag my-app:latest <REGISTRY_CLIENT_HOSTPORT>/my-ns/my-app:v1

# Push it to the Registry
nerdctl push <REGISTRY_CLIENT_HOSTPORT>/my-ns/my-app:v1
```

Copy an image from another registry into ACP Registry:

```
# Pull the source image
nerdctl pull remote.registry.io/demo/my-app:latest

# Retag it for ACP Registry
nerdctl tag remote.registry.io/demo/my-app:latest <REGISTRY_CLIENT_HOSTPORT>/my-ns/my-app:latest

# Push it to ACP Registry
nerdctl push <REGISTRY_CLIENT_HOSTPORT>/my-ns/my-app:latest
```

Delete Images

Use `ac delete images` to remove image tags from the Registry:

```
# Preview the deletion result without removing anything
ac delete images --repo=my-ns/my-app:v1 --registry-url=<REGISTRY_API_URL>

# Confirm the deletion
ac delete images --repo=my-ns/my-app:v1 --registry-url=<REGISTRY_API_URL> --confirm
```

Notes:

- This operation removes the image manifest reference from the Registry, but it does not guarantee that the underlying image data is reclaimed immediately.
- In the current implementation, if multiple tags point to the same manifest digest, deleting one tag may also remove the sibling tags that reference the same manifest.

Example:

```
ac delete images \
  --repo=my-ns/my-app:v1 \
  --registry-url=<REGISTRY_API_URL> \
  --confirm
```

Using Alauda Container Platform Registry in Kubernetes Clusters

The Alauda Container Platform (ACP) Registry provides secure container image management for Kubernetes workloads.

TOC

Registry Access Guidelines

- **Internal Address Recommended**: For images stored in the cluster's registry, always prioritize using the internal service address `image-registry.cpaas-system.svc` when deploying within the cluster. This ensures optimal network performance and avoids unnecessary external routing.

- **External Address Usage**: The external ingress domain (e.g. `registry.cluster.local`) is primarily intended for:
  - Image pushes/pulls from outside the cluster (e.g., developer machines, CI/CD systems)
  - Cluster-external operations requiring registry access

Deploy Sample Application

1. Create an application named `my-app` in the `my-ns` namespace.
2. Store the application image in the registry at `image-registry.cpaas-system.svc/my-ns/my-app:v1`.
3. The **default** ServiceAccount in each namespace is automatically configured with an imagePullSecret for accessing images from `image-registry.cpaas-system.svc`.

Example Deployment:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: my-ns
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: main-container
        image: image-registry.cpaas-system.svc/my-ns/my-app:v1
        ports:
        - containerPort: 8080
```

Cross-Namespace Access

To allow users from `my-ns` to pull images from `shared-ns`, the administrator of `shared-ns` can create a role binding to grant the necessary permissions.

Example Role Binding

```
# Access images from shared namespace (requires permissions)
kubectl create rolebinding cross-ns-pull \
  --clusterrole=system:image-puller \
  --serviceaccount=my-ns:default \
  -n shared-ns
```

Best Practices

- **Registry Usage**: Always use `image-registry.cpaas-system.svc` for deployments to ensure security and performance.
- **Namespace Isolation**: Leverage namespace isolation for better security and management of images.
  - Use namespace-based image paths: `image-registry.cpaas-system.svc/<namespace>/<image>:<tag>`.
- **Access Control**: Use role bindings to manage cross-namespace access for users and service accounts.

Verification Checklist

1. Validate image accessibility for the default ServiceAccount in `my-ns`:

   ```
   kubectl auth can-i get images.registry.alauda.io --namespace my-ns --as
   =system:serviceaccount:my-ns:default
   ```

2. Validate image accessibility for a user in `my-ns`:

```
kubectl auth can-i get images.registry.alauda.io --namespace my-ns --as
=<username>
```

Troubleshooting

- **Image Pull Errors**: Check the imagePullSecrets in the pod spec and ensure they are correctly configured.
- **Permission Denied**: Ensure the user or ServiceAccount has the necessary role bindings in the target namespace.
- **Network Issues**: Verify network policies and service configurations to ensure connectivity to the internal registry.
- **DNS Failures**: Check the content of `/etc/hosts` file on the node, ensure DNS resolution for the `image-registry.cpaas-system.svc` is correctly configured.
  - Verify node's /etc/hosts configuration to ensure correct DNS resolution of `image-registry.cpaas-system.svc`
  - Example showing registry service mapping (ClusterIP of image-registry service):

    ```
    # /etc/hosts
    127.0.0.1   localhost localhost.localdomain
    10.4.216.11 image-registry.cpaas-system image-registry.cpaas-system.s
    vc image-registry.cpaas-system.svc.cluster.local # cpaas-generated-no
    de-resolver
    ```

  - **How to get `image-registry` current ClusterIP**:

    ```
    kubectl get svc -n cpaas-system image-registry -o jsonpath='{.spec.cl
    usterIP}'
    ```

Configure Registry Push Limits

The Alauda Container Platform Registry includes a built-in proxy that can enforce image push limits for OCI registry API requests.

You can use this capability to:

- Limit the maximum size of pushed images
- Limit the number of tags allowed in a repository
- Apply global limits or path-based override rules

TOC

Before You Begin

- The **Alauda Container Platform Registry** cluster plugin must already be installed.
- You must have permission to update the Registry plugin configuration.
- You must be able to create a ConfigMap in the Registry namespace.
- If you have not installed the Registry yet, see Install Via YAML.

How It Works

To use this feature, enable `registryLimitConfig` in the Registry plugin configuration and create a ConfigMap that defines the limits.

Enable Registry Push Limits

When installing or updating the Registry plugin, enable `registryLimitConfig` and point it to a ConfigMap name.

Example:

```yaml
spec:
  config:
    registryLimitConfig:
      enabled: true
      configMapName: image-registry-limit-config
```

Notes:

- `enabled: true` mounts the limit configuration into the Registry proxy container.
- `configMapName` must reference a ConfigMap that you create manually.
- For new deployments, the recommended ConfigMap name is `image-registry-limit-config`.
- The runtime also accepts the legacy ConfigMap name `registry-gateway-config` for backward compatibility.

Create the Limit ConfigMap

Create a ConfigMap in the same namespace as the Registry deployment.

Example:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: image-registry-limit-config
  namespace: cpaas-system
data:
  max_image_size: "1GB"
  tag_count_limit: "1000"
  rules: |
    - path: ^team-a/.+
      limit:
        max_image_size: "100MB"
        tag_count_limit: "3"
    - path: ^team-b/.+
      limit:
        max_image_size: "5GB"
        tag_count_limit: "20"
```

Apply it:

```
kubectl apply -f image-registry-limit-config.yaml
```

Configuration Keys

Global Keys

Key	Description	Example
`max_image_size`	Maximum size allowed for a pushed image. Supported units: `GB`, `MB`, `KB`, `B`.	`1GB`
`tag_count_limit`	Maximum number of tags allowed in a repository	`1000`

Optional Rule Key

Key	Description
`rules`	Optional path-based override rules

Each rule contains:

- `path`: A regular expression used to match a repository path
- `limit.max_image_size`: The size limit applied when the rule matches
- `limit.tag_count_limit`: The tag count limit applied when the rule matches

Rules are evaluated in order, and the first matching rule takes effect.

For each rule, set both `limit.max_image_size` and `limit.tag_count_limit`.

Configuration Examples

Example 1: Global Limits Only

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: image-registry-limit-config
  namespace: cpaas-system
data:
  max_image_size: "500MB"
  tag_count_limit: "50"
```

Effect:

- All repositories use the same default size limit and tag count limit.

Example 2: Per-Repository Overrides

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: image-registry-limit-config
  namespace: cpaas-system
data:
  max_image_size: "1GB"
  tag_count_limit: "100"
  rules: |
    - path: ^project-a/.+
      limit:
        max_image_size: "100MB"
        tag_count_limit: "3"
    - path: ^project-b/release/.+
      limit:
        max_image_size: "5GB"
        tag_count_limit: "20"
```

Effect:

- `project-a/*` is limited to `100MB` and `3` tags.
- `project-b/release/*` is limited to `5GB` and `20` tags.
- Other repositories use the global defaults.

Apply Changes

After you create or update the ConfigMap, the Registry proxy loads the new rules automatically.

Allow a short propagation delay before the new limits take effect.

Verify the Configuration

1. Check that the ConfigMap exists:

```
kubectl get configmap image-registry-limit-config -n cpaas-system
```

2. Check that the Registry deployment is running:

```
kubectl get pods -n cpaas-system -l app=image-registry
```

3. Push an image to a repository covered by the rule and verify the result.

Example:

```
nerdctl login <REGISTRY_CLIENT_HOSTPORT> -u <ACP-USERNAME> -p <ACP-PASSWORD>
nerdctl push <REGISTRY_CLIENT_HOSTPORT>/team-a/demo:v1
```

If the Registry uses a self-signed certificate or plain HTTP, add the global flag `--insecure-registry`.

Expected behavior:

- When the image size exceeds the configured limit, the push is rejected.
- When the repository already has the maximum allowed number of tags, the push is rejected.

Recommended Practices

- Start with global defaults, then add path-based rules only where needed.
- Use repository path conventions that make rule matching predictable.
- Keep the ConfigMap name stable across updates.
- Verify changes in a non-production environment before applying them to production.

Troubleshooting

The limits do not take effect

Check the following:

- `spec.config.registryLimitConfig.enabled` is set to `true`
- The ConfigMap name matches `spec.config.registryLimitConfig.configMapName`
- The ConfigMap exists in the Registry namespace
- Repository paths in `rules` match the actual push target

A rule does not match the expected repository

Check the regular expression in `path`.

Rules are matched in order, so an earlier rule may already have taken effect.

Existing environments still use `registry-gateway-config`

The runtime still supports the legacy ConfigMap name.

For new environments, use `image-registry-limit-config`.

Source to Image

Overview

Install

Upgrade

Guides

How To

Overview

Introduction

Alauda Container Platform Builds is a cloud-native container tool provided by Alauda Container Platform that integrates Source to Image (S2I) capabilities with automated pipelines. It accelerates enterprise cloud-native journeys by enabling fully automated CI/CD pipelines that support multiple programming languages, including Java, Go, Python, and Node.js. Additionally, Alauda Container Platform Builds offers visual release management and seamless integration with Kubernetes-native tools like Helm and GitOps, ensuring efficient application lifecycle management from development to production.

TOC

Source to Image Concept

Source to Image (S2I) is a tool and workflow for building reproducible container images from source code. It injects the application's source code into a predefined builder image and automatically completes steps such as compilation and packaging, ultimately generating a runnable container image. This allows developers to focus more on business code development without worrying about the details of containerization.

Core Features

Alauda Container Platform Builds facilitates a full-stack, cloud-native workflow from code to application, enabling multi-language builds and visual release management. It leverages Kubernetes-native capabilities to convert source code into runnable container images, ensuring seamless integration into a comprehensive cloud-native platform.

- **Multi-language Builds**: Supports building applications in various programming languages such as Java, Go, Python, and Node.js, accommodating diverse development needs.
- **Visual Interface**: Provides an intuitive interface that allows you to easily create, configure, and manage build tasks without deep technical knowledge.
- **Full Lifecycle Management**: Covers the entire lifecycle from code commit to application deployment, automating build, deployment, and operational management.
- **Deep Integration**: Seamlessly integrates with your Container Platform product, providing a seamless development experience.
- **High Extensibility**: Supports custom plugins and extensions to meet your specific needs.

Core Benefits

- **Accelerated Development**: Streamlines the build process, speeding up application delivery.
- **Enhanced Flexibility**: Supports building in multiple programming languages.
- **Improved Efficiency**: Automates build and deployment processes, reducing manual intervention.
- **Increased Reliability**: Provides detailed build logs and visual monitoring for easy troubleshooting.

Application scenarios

The main application scenarios for S2I are as follows:

- Web applications
  S2I supports various programming languages, such as Java, Go, Python, and Node.js. By leveraging the Alauda Container Platform application management capabilities, it allows for rapid building and deployment of web applications simply by entering the code repository URL.
- CI/CD
  S2I integrates seamlessly with DevOps pipelines, leveraging Kubernetes-native tools like Helm and GitOps to automate the image building and deployment processes. This enables continuous integration and continuous deployment of applications.

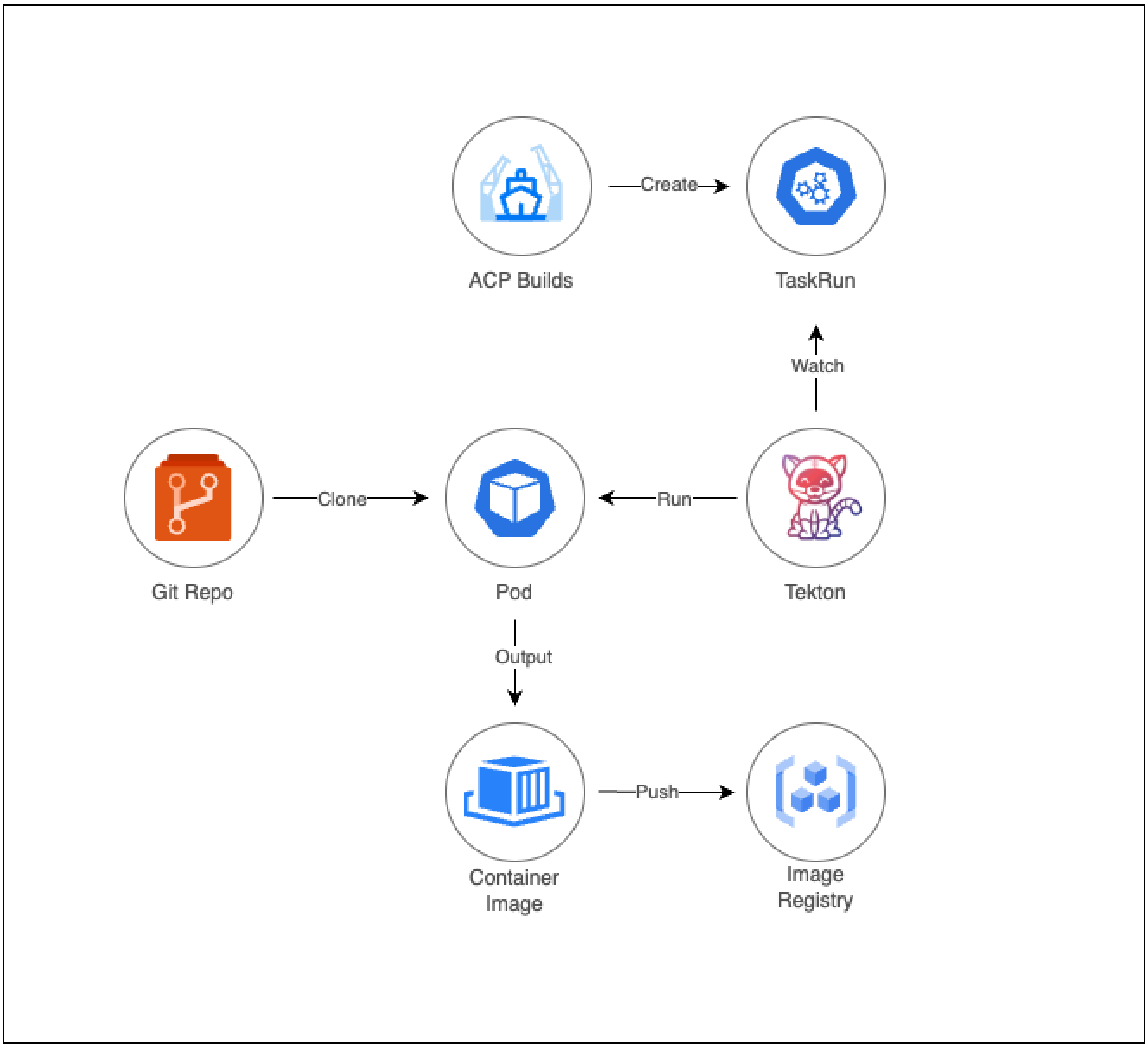
Usage Limitations

> The current version only supports Java, Go, Python, and Node.js languages.

WARNING

Prerequisites: Alauda DevOps Pipelines operator ↗ is now available in the cluster OperatorHub.

Architecture



Source to Image (S2I) capability is implemented through the **Alauda Container Platform Builds** operator, enabling automated container image builds from Git repository source code and subsequent pushes to a designated image registry. The core components include:

- **Alauda Container Platform Builds** operator: Manages the end-to-end build lifecycle and orchestrates Tekton pipelines.

- **Tekton** pipelines: Executes S2I workflows via Kubernetes-native `TaskRun` resources.

Release Notes

TOC

Alauda Container Platform Builds Release Notes

The release notes for the **Alauda Container Platform Builds** operator describe new features and enhancements, deprecated features, and known issues.

> **INFO**
>
> The **Alauda Container Platform Builds** operator is provided as an installable component, with a distinct release cycle from the core Alauda Container Platform. The Alauda Container Platform Builds operator Lifecycle Policy outlines release compatibility.

Supported Versions

Version	Alauda Container Platform Version	Alauda DevOps Pipelines Version
v1.1.0	v4.1	v4.1

v1.1 Release Notes

v1.1.0

1. Security Vulnerability Remediation.
2. Independently Releasable.

Lifecycle Policy

TOC

Version Lifecycle Timeline

Below is the lifecycle schedule for released versions of the Alauda Container Platform Builds Operator:

Version	Release Date	End of Life
v1.1.0	2025-08-15	2027-08-15

Install

Installing Alauda Container Platform Builds

TOC

Prerequisites

> Alauda Container Platform Builds is a container tool offered by Alauda Container Platform that integrates building (capable of Source to Image) and create application.

1. Download the latest version package of **Alauda Container Platform Builds** that matches your platform. If the **Alauda DevOps Pipelines** operator has not been installed on the Kubernetes cluster, it is recommended to download it together.
2. Utilize the `violet` CLI tool to upload **Alauda Container Platform Builds** and **Alauda DevOps Pipelines** packages to your target cluster. For detailed instructions on using `violet`, please refer to the CLI.

Procedure

Install the Alauda Container Platform Builds Operator

1. Log in, and navigate to the **Administrator** page.
2. Click **Marketplace** > **OperatorHub**.
3. Find the **Alauda Container Platform Builds** operator, click **Install**, and enter the **Install** page.

Configuration Parameters:

Parameter	Recommended Configuration
Channel	`Alpha` : The default Channel is set to **alpha**.
Version	Please select the latest version.
Installation Mode	`Cluster` : A single Operator is shared across all namespaces in the cluster for instance creation and management, resulting in lower resource usage.
Namespace	`Recommended` : It is recommended to use the **shipyard-operator** namespace; it will be created automatically if it does not exist.
Upgrade Strategy	Please select the `Manual` . • `Manual` : When a new version is available in the OperatorHub • the **Upgrade** action will not be executed automatically.

4. On the **Install** page, select default configuration, click **Install**, and complete the installation of the **Alauda Container Platform Builds** Operator.

Install the Shipyard instance

1. Click on **Marketplace** > **OperatorHub**.
2. Find the installed **Alauda Container Platform Builds** operator, navigate to **All Instances**.
3. Click **Create Instance** button, and click **Shipyard** card in the resource area.

4. On the parameter configuration page for the instance, you may use the default configuration unless there are specific requirements.

5. Click **Create**.

Verification

- After the instance is successfully created, wait approximately 20 minutes, then navigate to **Container Platform** > **Applications** > **Applications** and click **Create**.
- You should see the entry for **Create from Code**. At this time, the installation of Alauda Container Platform Builds is successful, and you can start your S2I journey with the Creating an application from Code.

Upgrade

Upgrading Alauda Container Platform Builds

TOC

Prerequisites

> Alauda Container Platform Builds is a container tool offered by Alauda Container Platform that integrates building (capable of Source to Image) and create application.

1. Download the new version package of **Alauda Container Platform Builds** that matches your platform.
2. Utilize the `violet` CLI tool to upload **Alauda Container Platform Builds** and **Alauda DevOps Pipelines** packages to your target cluster. For detailed instructions on using `violet`, please refer to the CLI.

Procedure

Upgrading the Alauda Container Platform Builds Operator

INFO

If you are upgrading from version v4.0 and earlier, first migrate the **Alauda DevOps Tekton v3** to **Alauda DevOps Pipelines**. For details, see the migration guide.

1. Log in, and navigate to the **Administrator** page.
2. Click **Marketplace** > **OperatorHub**.
3. In the navigation bar, select the cluster where the operator is installed.
4. Find the **Alauda Container Platform Builds** operator and open its **Details** page.
5. Click **Confirm** to start the upgrade, and wait until the operator finishes upgrading.

Guides

Managing applications created from Code

Managing applications created from Code

TOC

Key Features

- Input the code repository URL to trigger the S2I process, which converts the source code into a image and publishes it as an application.
- When the source code is updated, initiate the **Rebuild** action via the visual interface to update the application version with a single click.

Advantages

- Simplifies the process of creating and upgrading applications from code.

- Lowers the barrier for developers, eliminating the need to understand the details of containerization.
- Provides a visual construction process and operational management, facilitating problem localization, analysis, and troubleshooting.

Prerequisites

- Installing Alauda Container Platform Builds is completed.
- Access to a image repository is required; if unavailable, contact the Administrator to Installing Alauda Container Platform Registry

Procedure

1. **Container Platform**, navigate to **Application** > **Application**.
2. Click **Create**.
3. Select the **Create from Code**.
4. Refer to the parameter descriptions below to complete the configuration.

Region	Parameter	Description
Code Repository	**Type**	• **Platform Integrated**: Choose a code repository that is integrated with the platform and already allocated for the current project; the platform supports GitLab, GitHub, and Bitbucket. • **Input**: Use a code repository URL that is not integrated with the platform.
	Integrated Project Name	The name of the integration tool project assigned or associated with the current project by the Administrator.

	Repository Address	Select or input the address of the code repository that stores the source code.
	Version Identifier	Supports creating applications based on branches, tags, or commits in the code repository. Among them: • When the version identifier is a branch, only the latest commit under the selected branch is supported for creating applications. • When the version identifier is a tag or commit, the latest tag or commit in the code repository is selected by default. However, you can also choose other versions as needed.
	Context dir	Optional directory for the source code, used as a context directory for build.
	Secret	When using an input code repository, you can add an authentication secret as needed.
	Builder Image	• An image that includes specific programming language runtime environments, dependency libraries, and S2I scripts. Its main purpose is to convert source code into runnable application images. • The supported builder images, include: Golang, Java, Node.js, and Python.
	Version	Select the runtime environment version that is compatible with your source code to ensure smooth application execution.
Build	**Build Type**	Currently, only the **Build** method is supported for constructing application images. This method simplifies and automates the complex image building

	process, allowing developers to focus solely on code development. The general process is as follows: 1. After installed Alauda Container Platform Builds and creating the Shipyard instance, the system automatically generates cluster-level resources, such as ClusterBuildStrategy, and defines a standardized build process. This process includes detailed build steps and necessary build parameters, thereby enabling Source-to-Image (S2I) builds. For detailed information, refer to: Installing Alauda Container Platform Builds 2. Create Build type resources based on the above strategies and the information provided in the form. These resources specify build strategies, build parameters, source code repositories, output image repositories, and other relevant information. 3. Create BuildRun type resources to initiate specific build instances, which coordinate the entire build process. 4. After completing the BuildRun creation, the system will automatically generate the corresponding TaskRun resource instance. This TaskRun instance triggers the Tekton pipeline build and creates a Pod to execute the build process. The Pod is responsible for the actual build work, which includes: Pulling the source code from the code repository. Calling the specified builder image. Executing the build process.
Image URL	After the build is complete, specify the target image repository address for the application.

Application	-	Fill in the application configuration as needed. For specific details, refer to the parameter descriptions in the Creating applications from Image documentation.
Network	-	• **Target Port**: The actual port that the application inside the container listens on. When external access is enabled, all matching traffic will be forwarded to this port to provide external services. • **Other Parameters**: Please refer to the parameter descriptions in the CreatingIngress documentation.
Label Annotations	-	Fill in the relevant labels and annotations as needed.

5. After filling in the parameters, click on **Create**.
6. You can view the corresponding deployment on the **Details** page.

Related operations

Build

After the application has been created, the corresponding information can be viewed on the details page.

Parameter	Description
Build	Click the link to view the specific build (Build) and build task (BuildRun) resource information and YAML.
Start Build	When the build fails or the source code changes, you can click this button to re-execute the build task.

How To

Creating an application from Code

> Using the powerful capabilities of Alauda Container Platform Builds installation to achieve the entire process from Java source code to create an application, and ultimately enable the application to run efficiently in a containerized manner on Kubernetes.

TOC

Prerequisites

Before using this functionality, ensure that:

- Installing Alauda Container Platform Builds
- There is an accessible image repository on the platform. If not, please contact the Administrator to Installing ACP Registry

Procedure

1. **Container Platform**, click **Applications** > **Applications**.
2. Click **Create**.

3. Select the **Create from Code**.

4. Complete the configuration according to the parameters below:

Parameter	Recommended Configuration
Code Repository	**Type**: `Input` **Repository URL**: `https://github.com/alauda/spring-boot-hello-world`
Build Method	`Build`
Image Repository	Contact the Administrator.
Application	**Application**: `spring-boot-hello-world` **Name**: `spring-boot-hello-world` **Resource Limits**: Use the default value.
Network	**Target Port**: `8080`

5. After filling in the parameters, click **Create**.

6. You can check the corresponding application status on the **Details** page.

Node Isolation Strategy

Node Isolation Strategy provides a project-level node isolation strategy that allows projects to exclusively use cluster nodes.

Introduction

Architecture

Concepts

Guides

Create Node Isolation Strategy

Manage Node Isolation Policies with the CLI

Permissions

Permissions

Introduction

Node Isolation Strategy provides a project-level node isolation strategy that allows projects to exclusively use cluster nodes.

TOC

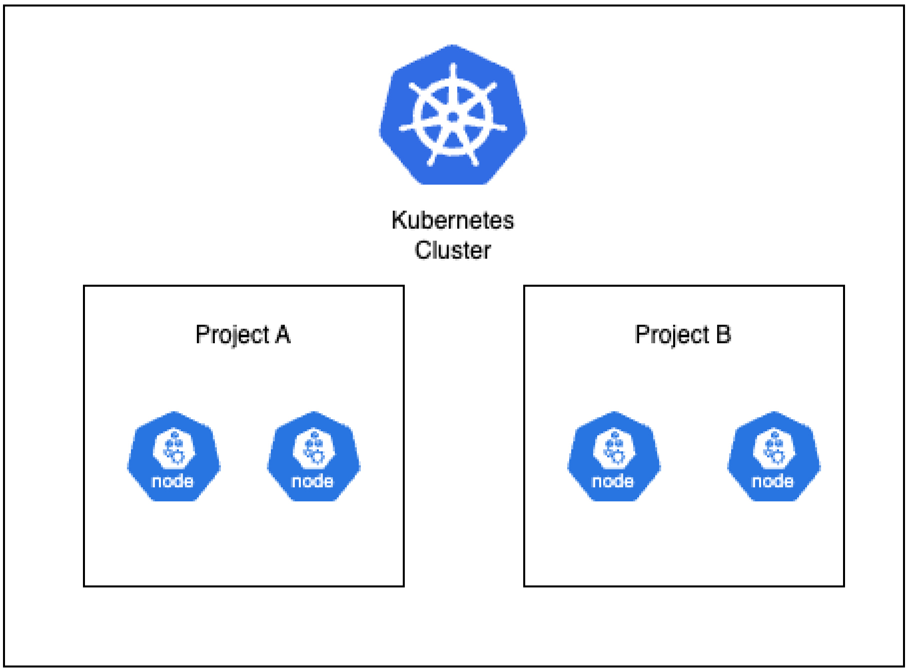
Advantages

Conveniently allocate nodes to projects in an exclusive or shared manner, preventing resource contention between projects.

Application Scenarios

Node Isolation Strategy is suitable for scenarios where enhanced resource isolation between projects is required, and where there is a desire to prevent other projects' components from occupying nodes, which could lead to resource constraints or inability to meet performance requirements.

Architecture



Node Isolation Strategy is implemented based on the Container Platform Cluster Core component, providing the capability of node isolation between projects by allocating nodes on each workload cluster. When containers are created in a project, they are forcibly scheduled to the nodes allocated to that specific project.

Concepts

Core Concepts

Core Concepts

TOC

Node Isolation

Node Isolation refers to isolating nodes in a cluster to prevent containers from different projects from simultaneously using the same node, thereby avoiding resource contention and performance degradation.

Guides

Create Node Isolation Strategy

Create a node isolation policy for the current cluster, allowing specified projects to have exclusive access to the nodes of grouped resources within the cluster, thereby restricting the runnable nodes for Pods under the project, achieving physical resource isolation between projects.

TOC

Create Node Isolation Strategy

1. In the left navigation bar, click on **Security** > **Node Isolation Strategy**.
2. Click on **Create Node Isolation Strategy**.
3. Refer to the instructions below to configure the relevant parameters.

Parameter	Description
Project Exclusivity	Whether to enable or disable the switch for the nodes contained in the project isolation policy configured in the strategy; click to toggle on or off, default is on. When the switch is on, only Pods under the specified project in the policy

Parameter	Description
	can run on the nodes included in the policy; when off, Pods under other projects in the current cluster can also run on the nodes included in the policy apart from the specified project.
Project	The project that is configured to use the nodes in the policy. Click the **Project** dropdown selection box, and check the checkbox before the project name to select multiple projects. **Note**: A project can only have one node isolation policy set; if a project has already been assigned a node isolation policy, it cannot be selected; Supports entering keywords in the dropdown selection box to filter and select projects.
Node	The IP addresses of the compute nodes allocated for use by the project in the policy. Click the **Node** dropdown selection box, and check the checkbox before the node name to select multiple nodes. **Note**: A node can belong to only one isolation policy; if a node already belongs to another isolation policy, it cannot be selected; Supports entering keywords in the dropdown selection box to filter and select nodes.

4. Click **Create**.
   Note:

   - After the policy is created, existing Pods in the project that do not comply with the current policy will be scheduled to the nodes included in the current policy after they are rebuilt;
   - When **Project Exclusivity** is on, currently existing Pods on the nodes will not be automatically evicted; manual scheduling is required if eviction is needed.

Delete Node Isolation Strategy

Note: After the node isolation policy is deleted, the project will no longer be restricted to run on specific nodes, and the nodes will no longer be exclusively used by the project.

1. In the left navigation bar, click on **Security** > **Node Isolation Strategy**.
2. Locate the node isolation policy, click ⋮ > **Delete**.

Manage Node Isolation Policies with the CLI

Node isolation policies provide project-level node isolation. Platform administrators can bind one or more projects to selected cluster nodes and choose whether those nodes are exclusive to the bound projects or shared with other projects.

In versions where the web console no longer provides **Administrator** > **Security Settings** > **Node Isolation Policies**, you can continue to manage node isolation policies with `ac` or `kubectl`.

TOC

How Node Isolation Policies Work

A node isolation policy is stored as a Kubernetes custom resource named `NodeGroup`.

Item	Value
API group	`security.alauda.io`
API version	`v1beta1`
Kind	`NodeGroup`
Resource	`nodegroups.security.alauda.io`
Scope	Cluster

The key fields are:

Field	Description
`metadata.name`	Policy name. It must start and end with a lowercase letter or number, can contain lowercase letters, numbers, and `-`, and can be up to 63 characters long.
`metadata.annotations["alauda.io/display-name"]`	Optional display name.
`spec.shareMode`	Node sharing mode. Valid values are `Exclusive` and `Share`.

Field	Description
`spec.projects[].name`	Names of the projects that use the policy.
`spec.nodes[].name`	Names of the nodes assigned to the policy.
`status.phase`	Policy reconciliation status, for example `Active`.

The sharing modes are:

- `Exclusive`: Only projects bound to the policy can schedule new Pods on the policy nodes. Other projects cannot schedule new Pods on those nodes.
- `Share`: Projects bound to the policy are constrained to the policy nodes, but other projects can still use those nodes.

After a policy is created or updated, the platform labels related `Node` and `Namespace` resources, labels `ProjectQuota` objects when they exist, and injects scheduling constraints when Pods are created or updated. Common labels and taints include:

Type	Name	Description
Label	`acp.cpaas.io/node-group-name`	Node isolation policy name.
Label	`acp.cpaas.io/node-group-share-mode`	Node isolation mode, usually `Exclusive` or `Share`.
Taint	`nodeGroup=<policy-name>:NoSchedule`	Added to policy nodes in `Exclusive` mode to block Pods that do not tolerate the taint.

Before You Begin

Before you run the commands in this guide, confirm the following:

- The `ac` CLI is installed. If you use `kubectl`, prepare a kubeconfig for the target cluster first.
- Your account can manage `nodegroups.security.alauda.io`.
- You know the target workload cluster name, for example `business-1`.
- You know the project names and node names to bind.
- You have evaluated the impact on existing workloads.

Important limits:

- `NodeGroup` is a cluster-scoped resource. Do not specify `-n` when managing `NodeGroup` resources.
- In one workload cluster, each project can belong to only one node isolation policy.
- In one workload cluster, each node can belong to only one node isolation policy.
- Creating a policy does not move existing Pods. Existing Pods that do not match the policy are constrained only after they are recreated.
- In `Exclusive` mode, existing Pods on the policy nodes are not evicted automatically. Migrate them manually if required.
- The web console creation page filters out control plane nodes and disables projects and nodes that are already used by another policy. When you use the CLI, check conflicts yourself before creating or updating a policy.

Authenticate to ACP

Use `ac` to manage node isolation policies. The `ac` CLI manages ACP multi-cluster sessions and can switch to the target workload cluster.

```
ac login <acp-url> --name <session-name>
ac config get-clusters
ac config use-cluster <cluster-name>
ac config current-context
```

Example:

```
ac login https://acp.example.com --name prod
ac config use-cluster business-1
ac config current-context
```

Confirm that the `NodeGroup` CRD exists:

```
ac api-resources --api-group=security.alauda.io
ac get crd nodegroups.security.alauda.io
```

If you prefer `kubectl`, export kubeconfig from `ac` and then select the target context. Use the value from the `NAME` column in `kubectl config get-contexts` as `<target-context>`.

```
ac config view --raw > kubeconfig
export KUBECONFIG=./kubeconfig
kubectl config get-contexts
kubectl config use-context <target-context>
kubectl get nodegroups.security.alauda.io
```

The following examples use `ac`. If your `kubectl` context is configured correctly, you can replace `ac` with `kubectl`.

View Node Isolation Policies

View all node isolation policies in the current cluster:

```
ac get nodegroups.security.alauda.io
```

View the policy list with selected fields:

```
ac get nodegroups.security.alauda.io \
  -o 'custom-columns=NAME:.metadata.name,MODE:.spec.shareMode,PROJECTS:.s
pec.projects[*].name,NODES:.spec.nodes[*].name,PHASE:.status.phase'
```

View details for one policy:

```
ac get nodegroups.security.alauda.io <policy-name> -o yaml
ac describe nodegroups.security.alauda.io <policy-name>
```

Check whether the policy is reconciled:

```
ac get nodegroups.security.alauda.io <policy-name> \
  -o jsonpath='{.status.phase}{"\n"}'
```

When `status.phase` is `Active`, the controller has reconciled the policy.

Check Available Projects and Nodes

Before you create or update a policy, check whether the projects and nodes are already used by another policy.

If `ProjectQuota` objects exist in the environment, view their node isolation labels:

```
ac get projectquotas.auth.alauda.io \
  -L acp.cpaas.io/node-group-name,acp.cpaas.io/node-group-share-mode
```

View `ProjectQuota` objects that are not bound to a policy:

```
ac get projectquotas.auth.alauda.io \
  --selector='!acp.cpaas.io/node-group-name'
```

View nodes and their node isolation labels:

```
ac get nodes \
  -L acp.cpaas.io/node-group-name,acp.cpaas.io/node-group-share-mode
```

View compute nodes that are not bound to a policy:

```
ac get nodes \
  --selector='!node-role.kubernetes.io/master,!node-role.kubernetes.io/co
ntrol-plane,!acp.cpaas.io/node-group-name'
```

Create a Node Isolation Policy

Create `nodegroup-pro.yaml`:

```yaml
apiVersion: security.alauda.io/v1beta1
kind: NodeGroup
metadata:
  name: pro-nodegroup
  annotations:
    alauda.io/display-name: "pro nodes"
spec:
  shareMode: Exclusive
  projects:
    - name: pro
  nodes:
    - name: node-1
    - name: node-2
```

Set `spec.shareMode` to `Exclusive` when the bound projects must have exclusive access to the nodes. Set it to `Share` when the bound projects must run on the selected nodes but other projects can still use those nodes.

Apply the policy:

```
ac apply -f nodegroup-pro.yaml
```

Confirm the policy status:

```
ac get nodegroups.security.alauda.io pro-nodegroup
ac get nodegroups.security.alauda.io pro-nodegroup -o yaml
```

Confirm that the nodes are labeled:

```
ac get nodes \
  --selector='acp.cpaas.io/node-group-name=pro-nodegroup' \
  -L acp.cpaas.io/node-group-name,acp.cpaas.io/node-group-share-mode
```

For `Exclusive` policies, confirm that a policy node has the `nodeGroup` taint:

```
ac describe node node-1
```

In the output, check that `Taints` contains:

```
nodeGroup=pro-nodegroup:NoSchedule
```

Shared nodes usually do not keep a `nodeGroup=<policy-name>:NoSchedule` taint.

Update a Node Isolation Policy

Before you update a policy, check its current configuration:

```
ac get nodegroups.security.alauda.io <policy-name> -o yaml
```

Update Display Name

```
ac annotate nodegroups.security.alauda.io <policy-name> \
  alauda.io/display-name="<display-name>" \
  --overwrite
```

Change Share Mode

Switch the policy to shared mode:

```
ac patch nodegroups.security.alauda.io <policy-name> \
  --type=merge \
  -p '{"spec":{"shareMode":"Share"}}'
```

Switch the policy to exclusive mode:

```
ac patch nodegroups.security.alauda.io <policy-name> \
  --type=merge \
  -p '{"spec":{"shareMode":"Exclusive"}}'
```

After the policy switches to `Exclusive`, the controller adds `nodeGroup=<policy-name>:NoSchedule` to the policy nodes. Existing Pods are not evicted by this taint, but new Pods are constrained.

Replace Bound Projects

The following command replaces the entire project list with `pro` and `dev`:

```
ac patch nodegroups.security.alauda.io <policy-name> \
  --type=merge \
  -p '{"spec":{"projects":[{"name":"pro"},{"name":"dev"}]}}'
```

Replace Bound Nodes

The following command replaces the entire node list with `node-1` and `node-2`:

```
ac patch nodegroups.security.alauda.io <policy-name> \
  --type=merge \
  -p '{"spec":{"nodes":[{"name":"node-1"},{"name":"node-2"}]}}'
```

> Important:
> `spec.projects` and `spec.nodes` are arrays. A merge patch replaces the entire array. Include the complete final list in the patch to avoid removing projects or nodes unintentionally.

Delete a Node Isolation Policy

After you delete a policy, the bound projects are no longer restricted to the policy nodes, and the nodes are no longer reserved for the policy.

```
ac delete nodegroups.security.alauda.io <policy-name>
```

Confirm that the policy is deleted:

```
ac get nodegroups.security.alauda.io
```

Confirm that related labels are cleaned up:

```
ac get nodes \
  --selector='acp.cpaas.io/node-group-name=<policy-name>'

ac get projectquotas.auth.alauda.io \
  --selector='acp.cpaas.io/node-group-name=<policy-name>'

ac get namespaces \
  --selector='acp.cpaas.io/node-group-name=<policy-name>'
```

If the policy stays in deletion for a long time, do not remove its finalizer directly. Check the node isolation controller and webhook first.

Verify Scheduling Behavior

After you create or update a policy, verify that it is effective.

Verify Labels

View policy nodes:

```
ac get nodes \
  --selector='acp.cpaas.io/node-group-name=<policy-name>' \
  -L acp.cpaas.io/node-group-name,acp.cpaas.io/node-group-share-mode
```

View policy project quotas, if `ProjectQuota` objects exist in the environment:

```
ac get projectquotas.auth.alauda.io \
  --selector='acp.cpaas.io/node-group-name=<policy-name>' \
  -L acp.cpaas.io/node-group-name,acp.cpaas.io/node-group-share-mode
```

View namespaces that belong to the policy projects:

```
ac get namespaces \
  --selector='acp.cpaas.io/node-group-name=<policy-name>' \
  -L acp.cpaas.io/node-group-name,acp.cpaas.io/node-group-share-mode
```

Verify New Pod Scheduling

Create a test Pod in a namespace that belongs to a bound project:

```
ac -n <namespace> run nodegroup-check \
  --image=busybox:1.36 \
  --restart=Never \
  --command -- sleep 3600
```

View the node where the Pod is scheduled:

```
ac -n <namespace> get pod nodegroup-check -o wide
```

View the injected node selector and toleration:

```
ac -n <namespace> get pod nodegroup-check -o yaml
```

In the output, check for the following fields:

```yaml
spec:
  nodeSelector:
    acp.cpaas.io/node-group-name: <policy-name>
  tolerations:
    - key: nodeGroup
      operator: Equal
      value: <policy-name>
      effect: NoSchedule
```

Delete the test Pod after verification:

```
ac -n <namespace> delete pod nodegroup-check
```

> Note:
> Existing Pods go through the scheduling flow again only after they are recreated.

Troubleshooting

Issue	Possible cause	Action
`nodegroups.security.alauda.io` does not exist	The target cluster does not have the node isolation components installed or enabled.	Confirm that you selected the correct cluster, then check the node isolation feature switch and related components.
Create or update is rejected	The project or node already belongs to another policy, or your account does not have permission.	Check the `acp.cpaas.io/node-group-name` labels and confirm your permissions.
The policy does not become `Active`	The controller has not finished reconciliation, or a referenced project or node does not exist.	Check the project and node names, then check the node isolation controller logs.

Issue	Possible cause	Action
New Pods are not scheduled to policy nodes	The namespace is not labeled, the Pod was not recreated, or the webhook did not inject scheduling constraints.	Check the Namespace labels, delete and recreate the Pod, and check the webhook and controller.
Existing Pods from other projects remain on exclusive nodes	This is expected. `NoSchedule` does not evict existing Pods.	Manually recreate, evict, or adjust the related workloads if migration is required.
Node labels do not change after you update the node list	The controller is still syncing or the policy is unhealthy.	Wait briefly and query again. If labels still do not update, check controller logs and `NodeGroup` events.

Recommended Operation Flow

For production operations, use the following flow:

1. Confirm the target workload cluster context.
2. Query the existing `NodeGroup` list.
3. Check whether the projects and nodes are already bound to another policy.
4. Write and save the `NodeGroup` YAML.
5. Create or update the policy with `ac apply -f`.
6. Wait until `status.phase` is `Active`.
7. Check the `Node`, `Namespace`, and `ProjectQuota` labels when the related resources exist.
8. Create a new test Pod in a test namespace to verify scheduling.
9. Recreate business workloads or migrate existing Pods only after verification.

Permissions

Function	Action	Platform Administrator	Platform auditors	Project Manager	N A
nodegroups `acp-nodegroups`	View	✓	✓	✓	
	Create	✓	×	×	
	Update	✓	×	×	
	Delete	✓	×	×	

FAQ

TOC

Why shouldn't multiple ResourceQuotas exist in a namespace when importing it?

When importing a namespace, if the namespace contains multiple ResourceQuota resources, the platform will select the smallest value for each quota item among all ResourceQuotas and merge them, ultimately creating a single ResourceQuota named `default`.

Example:

The namespace `to-import` to be imported contains the following `resourcequota` resources:

```yaml
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: a
  namespace: to-import
spec:
  hard:
    requests.cpu: "1"
    requests.memory: "500Mi"
    limits.cpu: "3"
    limits.memory: "1Gi"
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: b
  namespace: to-import
spec:
  hard:
    requests.cpu: "2"
    requests.memory: "300Mi"
    limits.cpu: "2"
    limits.memory: "2Gi"
```

After importing the `to-import` namespace, the following `default` ResourceQuota will be created in that namespace:

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: default
  namespace: to-import
spec:
  hard:
    requests.cpu: "1"
    requests.memory: "300Mi"
    limits.cpu: "2"
    limits.memory: "1Gi"
```

For each ResourceQuota, the quotas of resources is the minimum value between `a` and `b`.

When multiple ResourceQuotas exist in a namespace, Kubernetes validates each ResourceQuota independently. Therefore, after importing a namespace, it is recommended to delete all ResourceQuotas except for the `default` one. This helps avoid complications in quota calculations due to multiple ResourceQuotas, which can easily lead to errors.

Why shouldn't multiple LimitRanges exist in a namespace when importing it?

When importing a namespace, if the namespace contains multiple LimitRange resources, the platform cannot merge them into a single LimitRange. Since Kubernetes independently validates each LimitRange when multiple exist, and the behavior of which LimitRange's default values Kubernetes selects is unpredictable.

If the namespace only contains a single LimitRange, the platform will created a LimitRange named `default` with the values from that LimitRange.

Therefore, before importing a namespace, only a single LimitRange should exist in the namespace. And after the namespace is imported it is recommended to delete the LimitRanges except for the one named `default` to avoid unpredictable behavior caused by multiple LimitRanges.