Menu

Developer

Overview

Introduction

Advantages

Use Cases

Cross-Cutting Cloud-Native Principles

Concepts

Features

Building Application

Namespace Management

Application Observability

Source to Image

Registry

Node Isolation Strategy

OAM Application

Quick Start

Creating a simple application via image

Introduction

Important Notes

Prerequisites

Workflow Overview

Procedure

Building Applications

Overview

Namespace Management

Application Lifecycle Management

Kubernetes Workload Management

Concepts

Guides

How To

Registry

Introduction

Principles and namespace isolation

Authentication and authorization

Advantages

Application Scenarios

Install

How To

Source to Image

Introduction

Source to Image Concept

Core Features

Core Benefits

Application scenarios

Usage Limitations

Install

Architecture

Guides

How To

Node Isolation Strategy

Introduction

Advantages

Application Scenarios

Architecture

Concepts

Guides

Permissions

FAQ

FAQ

Why shouldn't multiple ResourceQuotas exist in a namespace when importing it?

Why shouldn't multiple LimitRanges exist or a LimitRange that is not named `default` in a namespace when importing it?

Menu

Overview

Introduction

Introduction

Advantages

Use Cases

Cross-Cutting Cloud-Native Principles

Concepts

Resource Unit Description

Application Types

Workload Types

Features

Features

Building Application

Namespace Management

Application Observability

Source to Image

Registry

Node Isolation Strategy

OAM Application

Introduction

The Developer view module empowers developers with cloud-native application orchestration and operational capabilities. It provides a unified interface for application composition from multiple sources while integrating built-in observability tools for production operations.

TOC

Advantages

> The Developer view module delivers the following key advantages:

1.

Unified Application Orchestration

- Images: Deploy from public/private registries with image
- YAML: Direct Kubernetes resource declarations with schema validation
- Source to Image (S2I): Build containerized applications directly from source code
- Helm Charts: Deploy packaged applications from curated Application Catalog
- Implements GitOps-aligned application composition using multiple approaches

2.

Holistic Lifecycle Management

> Implements declarative management for workloads and namespaces:

- Progressive Delivery: Canary/Blue-Green deployments via ServiceMesh
- Resource Governance:
  - Namespace provisioning with RBAC policies
  - Resource allocation policies via HPA/VPA
  - Dynamic scaling with Cluster Autoscaler integration
- Workflow Automation: CI/CD pipeline integration with Tekton

1. Enterprise-Grade Namespace Controls

> Implements multi-tenant namespace management:

- Complete lifecycle management
- Resource Guarantees:
  - ResourceQuota and LimitRange configurations
  - Configurable overcommit ratios for CPU/Memory

1. Full-Stack Observability

> Integrated monitoring stack with:

- Event Correlation: Kubernetes Event and Audit log integration
- Log Analytics: Log aggregation
- Metrics dashboard: Monitoring and Custom alert rules

Use Cases

> The main use cases of the Developer module include:

- Multi-Cloud Deployment

Organizations distribute workloads across multiple cloud providers (AWS, Azure, GCP) to avoid vendor lock-in, optimize costs, and ensure resilience. Cloud-native application delivery enables consistent deployment pipelines that abstract provider-specific implementations.

- Hybrid Cloud Environments

Enterprises maintain on-premises infrastructure alongside public cloud resources. Cloud-native delivery provides unified application deployment across hybrid environments while managing heterogeneous infrastructure complexities.

- Edge Computing Integration

As edge computing gains prominence, applications must run in centralized clouds, edge devices, and regional edge nodes. Cloud-native delivery extends deployment capabilities to these distributed edge environments.

- Development-to-Production Pipeline

Cloud-native methodologies enable seamless promotion of applications from development through testing/staging to production, preserving configuration consistency while accommodating environment-specific requirements.

- Global Multi-Region Deployments

For globally distributed applications, cloud-native delivery ensures consistent deployments across geographic regions, addressing latency optimization and data locality compliance.

- Disaster Recovery and Workloads Continuity

Cloud-native delivery facilitates disaster recovery environment provisioning that mirrors production systems, enabling rapid failover and ensuring uninterrupted operations.

Cross-Cutting Cloud-Native Principles

> These scenarios leverage core cloud-native principles:

- Containerization

- Infrastructure-as-Code (IaC)
- Declarative configurations
- Immutable infrastructure
- GitOps workflows

These ensure consistency, reliability, and automation across heterogeneous computing environments.

Menu

Concepts

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.

Application Types

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

- **Application**: A complete business application composed of one or more associated computing components (Workloads), internal routes (Services), and other native Kubernetes resources. It supports creation through UI editing, YAML orchestration, and templates, and can run in development, testing, or production environments. Different types of native applications can be created in the following ways:
  - **Create from Image**: Quickly create applications using existing container images.
  - **Create from Catalog**: Create applications using Helm Chart packages.
  - **Create from YAML**: Create applications using YAML configuration files.
  - **Create from Code**: Create applications using source code.
- **Operator Backed App**: Based on application components (Operator backed), you can quickly deploy a component application and leverage the capabilities of Operators to automate the entire lifecycle management of the application.
- **OAM Application**: Used to define the model of cloud-native applications. Compared to container or Kubernetes orchestration logic, OAM focuses more on the "application" itself. Based on OAM, common capabilities of applications are encapsulated into high-level interfaces for use, throughout the entire process of application deployment, development, and operations.

Workload Types

In addition to creating native applications and component applications, workloads can also be directly created in **Container Platform > Computing Components**:

- **Deployment**: The most commonly used workload controller for deploying stateless applications. It can ensure that a specified number of Pod replicas are running in the cluster, supporting rolling updates and rollbacks, suitable for stateless application scenarios such as web services and API services.
- **DaemonSet**: Ensures that each node in the cluster (or specific nodes) runs a Pod replica. When a node joins the cluster, the Pod is automatically created; when a node is removed from the cluster, those Pods are also reclaimed. Suitable for scenarios requiring logging, monitoring, etc., to run on each node.
- **StatefulSet**: A workload controller for managing stateful applications. It maintains a fixed identity for each Pod and provides stable storage and network identity, which remains unchanged even if the Pod is rescheduled. Suitable for stateful applications such as databases and distributed caches.
- **Job**: A workload for running one-time tasks. A Job creates one or more Pods and ensures that the specified number of Pods successfully complete the task before terminating. It is suitable for batch processing, data migration, and other one-time task scenarios.
- **CronJob**: Used to manage Jobs scheduled to run based on time. You can set the time expression for task execution, and the system will automatically create and run the Job at the scheduled time. Suitable for periodic tasks such as data backup, report generation, and periodic cleaning.

In addition to creating the above computing components through the platform's form page, the platform also supports creating Pods and Containers through CLI tools:

- **Pod**: The smallest deployable unit in Kubernetes, a Pod can contain one or more containers that share storage, network, and configuration declarations. Pods are typically managed by controllers (such as Deployments).

- **Container**: A standard software unit that packages the code and all dependencies, allowing applications to run quickly and reliably across different computing environments. Containers run inside Pods and share the Pod's resources.

Features

TOC

Building Application

- Creating Application

  Support multiple ways to create an Application, including image, yaml, codes and catalog.

- Application Operation

  Use Application to orchestrate and operate the workloads and their related resources.

- Workloads Management

  Manage the lifecycle of the workloads.

Namespace Management

- Namespace Lifecycle Management

  Manage the lifecycle of the namespace.

- Resource Quota and Limit Management

  Manage the resource quota and limit of the namespace.

- Namespace Resource Overcommit

  Allow overcommit the reousurce of the namespace.

Application Observability

- Logs

  Query the history logs or real time logs of the applications.

- Events

  Query the events collected from the applications.

- Monitoring

  Monitor the application status and firing alerts when abnormalities occur.

Source to Image

- Build image from source

  Build image from the source code of the git repository and push the image to the image repository.

Registry

- Out-of-the-box Registry Server

  Easily deploy an registry server available for the platform.

Node Isolation Strategy

- Node Isolation

  Support project-level node isolation to avoid resource contention between projects.

OAM Application

- Efficient operation and maintenance

  Through OAM applications, application operation and maintenance personnel can focus on business logic and manage applications from the application perspective rather than the platform perspective, reducing the threshold for application operation and maintenance. Platform operation and maintenance personnel can handle platform plugins, operation and maintenance plugins, and other configurations uniformly, thereby improving operational efficiency.

- Portability

  The OAM application model includes configurations related to application operation and maintenance, service governance, etc. Compared with applications deployed through Operators, Charts, and other methods, OAM applications can be repeatedly deployed through YAML, making cross-environment migration easier. Even without Kubernetes and specific vendors, OAM applications can run normally on various platforms.

- Scalability

  Several types of components pre-installed on the platform can meet most application development needs: network services, stateful applications, and native Kubernetes resources. In addition, the platform also provides the ability to extend components and

traits, making it easy for developers to use custom-designed and encapsulated components and traits.

Menu

Quick Start

Creating a simple application via image

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
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 **Platform Management** > **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 (e.g., Docker Hub)
- Example: `index.docker.io/library/nginx:latest`

Verification Requirement

- Cluster network must have egress access to registry endpoints

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., `index.docker.io/library/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)

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`

Parameter	Description
Workload Name	Select the `Deployment` created previously.
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.

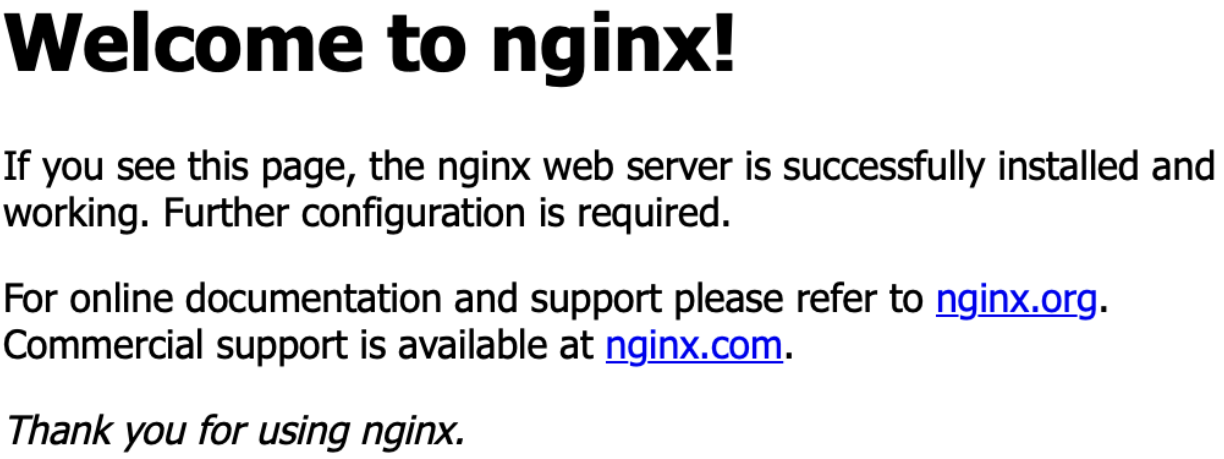
Validate Application Accessibility

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



Menu

Building Applications

Overview

Overview

Namespace Management

Application Lifecycle Management

Kubernetes Workload Management

Concepts

Understanding Parameters

Overview

Core Concepts

Use Cases and Scenarios

CLI Examples and Practical Usage

Best Practices

Troubleshooting Common Issues

Advanced Usage Patterns

Understanding Startup Commands

Overview

Core Concepts

Use Cases and Scenarios

CLI Examples and Practical Usage

Best Practices

Advanced Usage Patterns

Understanding Environment Variables

Overview

Core Concepts

Use Cases and Scenarios

CLI Examples and Practical Usage

Best Practices

Guides

Namespaces

Pre-Application-Creation Preparation

Creating Applications

Post-Application-Creation Configuration

Operation and Maintenance

Application Observability

Workloads

Working with Helm charts

1. Understanding Helm

2 Deploying Helm Charts as Applications via CLI

3. Deploying Helm Charts as Applications via UI

Pod

How To

Setting Scheduled Task Trigger Rules

Time Conversion

Writing Crontab Expressions

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 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).

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 namespace-level monitoring dashboards, including dedicated views for Applications, Workloads, and Pods, and also support customizing monitoring 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.

Menu

Concepts

Understanding Parameters

Overview

Core Concepts

Use Cases and Scenarios

CLI Examples and Practical Usage

Best Practices

Troubleshooting Common Issues

Advanced Usage Patterns

Understanding Startup Commands

Overview

Core Concepts

Use Cases and Scenarios

CLI Examples and Practical Usage

Best Practices

Advanced Usage Patterns

Understanding Environment Variables

Overview

Core Concepts

Use Cases and Scenarios

CLI Examples and Practical Usage

Best Practices

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 Docker 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 Docker

In Docker 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

```
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 off;" -c

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

# 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 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

```
# ❌ 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

```
# ✅ 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/sh

# 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 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 Docker images
- Define the main process (PID 1) inside the container
- Work in conjunction with parameters (args) to form the complete command line

Relationship with Docker and Parameters

Understanding the relationship between Docker instructions and Kubernetes fields:

Docker	Kubernetes	Purpose
ENTRYPOINT	`command`	Defines the executable
CMD	`args`	Provides default arguments

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

```
# 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	Docker 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/

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

# One-time task
kubectl run task --image=busybox --restart=Never --command -- /bin/sh -c "ech
```

Using kubectl create job

```
# Create a job with custom command
kubectl create job backup --image=postgres:13 --dry-run=client -o yaml -- pg_

# 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

```
# ✅ 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 -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%
          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.

Understanding Environment Variables

TOC

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)
Field Reference	Pod/Container metadata	Dynamic runtime information

Source Type	Description	Use Case
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:

```
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
```

```
        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 development
```

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

```
# ✅ 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

```
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
```

Menu

Guides

Namespaces

Creating Namespaces

Understanding namespaces

Creating namespaces by using web console

Creating namespace by using CLI

Importing Namespaces

Overview

Use Cases

Prerequisites

Procedure

Resource Quota

Understanding Resource Requests & Limits

Quotas

Limit Range

Understanding Limit Range

Create Limit Range by using CLI

Pod Security Admission

Security Modes

Security Standards

Configuration

Overcommit Ratio

UnderStanding Namespace Resource Overcommit Ratio

CRD Define

Creating overcommit ratio by using CLI

Creating/Updating Overcommit Ratio by using web console

Managing Namespace Members

Importing Members

Adding Members

Removing Members

Updating Namespaces

Updating Quotas

Updating Container LimitRanges

Updating Pod Security Admission

Deleting/Removing Namespaces

Deleting Namespaces

Removing Namespaces

Pre-Application-Creation Preparation

Configuring ConfigMap

Understanding Config Maps

Config Map Restrictions

ConfigMap vs Secret

Creating a ConfigMap by using the web console

Creating a ConfigMap by using the CLI

Operations

View, Edit and Delete by using the CLI

Ways to Use a ConfigMap in a Pod

ConfigMap vs Secret

Configuring Secrets

Understanding Secrets

Creating an Opaque type Secret

Creating a Docker registry type Secret

Creating a Basic Auth type Secret

Creating a SSH-Auth type Secret

Creating a TLS type Secret

Creating a Secret by using the web console

How to Use a Secret in a Pod

Follow-up Actions

Operations

Creating Applications

Creating applications from Image

Creating applications from Chart

Creating applications from YAML

Creating applications from Code

Creating applications from Operator Backed

Creating applications by using CLI

Prerequisites

Procedure

Example

Reference

Post-Application-Creation Configuration

Configuring HPA

Understanding Horizontal Pod Autoscalers

Prerequisites

Creating a Horizontal Pod Autoscaler

Calculation Rules

Configuring VerticalPodAutoscaler (VPA)

Understanding VerticalPodAutoscalers

Prerequisites

Creating a VerticalPodAutoscaler

Follow-Up Actions

Configuring CronHPA

Understanding Cron Horizontal Pod Autoscalers

Prerequisites

Creating a Cron Horizontal Pod Autoscaler

Schedule Rule Explanation

Operation and Maintenance

Status Description

Applications

Starting and Stopping Applications

Starting the Application

Stopping the Application

Updating Applications

Importing Resources

Removing/Batch Removing Resources

Exporting Applications

Exporting Helm Charts

Exporting YAML to Local

Exporting YAML to Code Repository (Alpha)

Updating and deleting Chart Applications

Important Notes

Prerequisites

Status Analysis Description

Version Management for Applications

Creating a Version Snapshot

Rolling Back to a Historical Version

Deleting Applications

Health Checks

Understanding Health Checks

YAML file example

Health Checks configuration parameters by using web console

Troubleshooting probe failures

Application Observability

Monitoring Dashboards

Prerequisites

Namespace-Level Monitoring Dashboards

Workload-Level Monitoring

Logs

Procedure

Events

Procedure

Event records interpretation

Workloads

Deployments

Understanding Deployments

Creating Deployments

Managing Deployments

Troubleshooting by using CLI

DaemonSets

Understanding DaemonSets

Creating DaemonSets

Managing DaemonSets

StatefulSets

Understanding StatefulSets

Creating StatefulSets

Managing StatefulSets

CronJobs

Understanding CronJobs

Creating CronJobs

Execute Immediately

Deleting CronJobs

Jobs

Understanding Jobs

YAML file example

Execution Overview

Working with Helm charts

Working with Helm charts

1. Understanding Helm

2 Deploying Helm Charts as Applications via CLI

3. Deploying Helm Charts as Applications via UI

Pod

Introduction

Pod Parameters

Deleting Pods

Use Cases

Procedure

Container

Menu

Namespaces

Creating Namespaces

Understanding namespaces

Creating namespaces by using web console

Creating namespace by using CLI

Importing Namespaces

Overview

Use Cases

Prerequisites

Procedure

Resource Quota

Understanding Resource Requests & Limits

Quotas

Limit Range

Understanding Limit Range

Create Limit Range by using CLI

Pod Security Admission

Security Modes

Security Standards

Configuration

Overcommit Ratio

UnderStanding Namespace Resource Overcommit Ratio

CRD Define

Creating overcommit ratio by using CLI

Creating/Updating Overcommit Ratio by using web console

Managing Namespace Members

Importing Members

Adding Members

Removing Members

Updating Namespaces

Updating Quotas

Updating Container LimitRanges

Updating Pod Security Admission

Deleting/Removing Namespaces

Deleting Namespaces

Removing Namespaces

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 namespace by using CLI

YAML file examples

```yaml
# example-namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: example
  labels:
    pod-security.kubernetes.io/audit: baseline # Option, to ensure security,
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/warn: baseline

# 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

# 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-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
kubectl create limitrange example-limitrange --namespace=example --default='c
```

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:

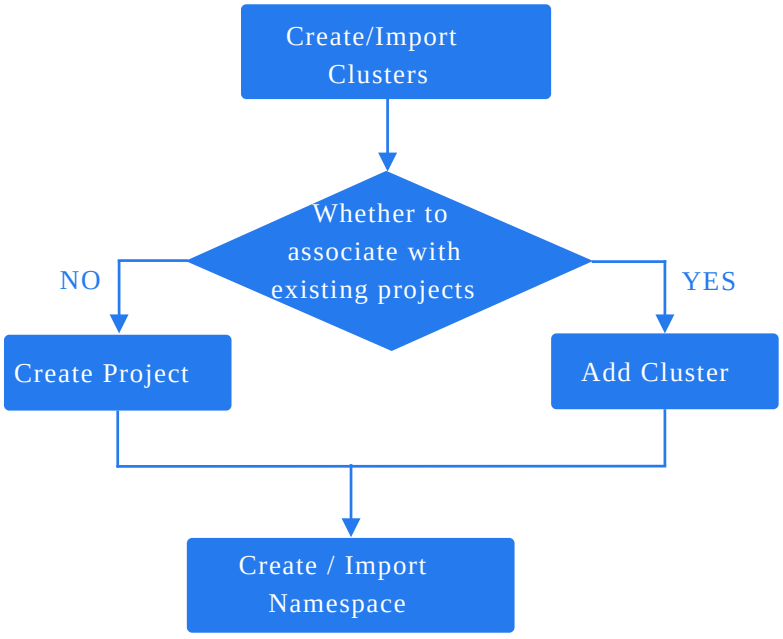
- A Namespace can only be associated with one Project at any given time.

- 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**.

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> --ha
```

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.

Extended Resources Quotas

Extended resource quotas are defined via **ConfigMap**. If the ConfigMap is missing, the resource category will not appear.

ConfigMap Field Descriptions

Field	Description
data.dataType	Data type (e.g., `vGPU`).
data.defaultValue	Default value (empty = no default).
data.descriptionEn	English tooltip text (displayed when hovering over the field).
data.descriptionZh	Chinese tooltip text (displayed when hovering over the field).
data.excludeResources	Mutually exclusive resources (comma-separated).
data.group	Resource group (e.g., `MPS`).
data.groupI18n	Group name in English/Chinese for UI dropdowns.
data.key	Specifies the value of the key. A configuration dictionary can only describe one key.
data.labelEn/data.labelZh	The English/Chinese name of the resource, which can be viewed and selected in the drop-down options corresponding to the quota types. This field serves the same function as the data.groupI18n field but is only applicable when the same resource has a single value, ensuring compatibility with the old version of the configuration dictionary (ConfigMap).
data.limits	Indicates whether to configure limits for the resources. Valid values include: disabled indicates limits cannot be configured for the resource, required indicates it must be input, and optional indicates it is optional input.
data.requests	Indicates whether to configure requests for the resources. Valid values include: disabled indicates requests cannot be configured for the resource, required

Field	Description
	indicates it must be input, optional indicates it is optional input, and fromLimits indicates it will use the same configuration as limits.
data.relatedResources	Associated resources. This field is reserved and currently cannot be used.
data.resourceUnit	Resource unit (e.g., `cores`, `GiB`). Not support input in Chinese.
data.runtimeClassName	Runtime class (default: `nvidia` for GPU).
metadata.labels	Mandatory labels: • `features.cpaas.io/type: CustomResourceLimitation` • `features.cpaas.io/group: <groupName>` • `features.cpaas.io/enabled`: `true` or `false`, the label is mandatory and indicates whether it is enabled, default is true.

Field	Description

metadata.name	The format is `cf-crl-<*groupName*>-<*name*>`, where
	• cf-crl is a fixed field and cannot be changed.
	• groupName is the name of the corresponding resource group, e.g., gpu-manager, galaxy, etc.
	• name is the resource name:
	  • Resource name can be standard resource type names, e.g., cpu, memory, pods, etc. The standard resource names must comply with Kubernetes' qualified name rules and must exist within the defined standard resource types in Kubernetes.
	  • Resource names can also be special resource types starting with specific prefixes, such as: `hugepages-` or `requests.hugepages-.`

metadata.namespace	Must be `kube-public`

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 --default='c
```

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.

TOC

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.

Mode	Behavior	Use Case
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.
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

```bash
# 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"]
```

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 overcom
  memory: 4 # Absence of this field indicates inheritance of the cluster over
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.

Parameter	Description
	• 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.

1. Click **Update**.

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

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.

  5.1.

  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.

  5.2.

  Click **Import All**.

- Import one or more users from the list as namespace members.

  5.1.

  Click the checkbox in front of the username/display name to select one or more users.

  5.2.

  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.

  5.3.

  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**.

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 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
```

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 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
```

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 CLI

> Update Pod Security Admission CLI command

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
```

Menu

Pre-Application-Creation Preparation

Configuring ConfigMap

Understanding Config Maps

Config Map Restrictions

ConfigMap vs Secret

Creating a ConfigMap by using the web console

Creating a ConfigMap by using the CLI

Operations

View, Edit and Delete by using the CLI

Ways to Use a ConfigMap in a Pod

ConfigMap vs Secret

Configuring Secrets

Understanding Secrets

Creating an Opaque type Secret

Creating a Docker registry type Secret

Creating a Basic Auth type Secret

Creating a SSH-Auth type Secret

Creating a TLS type Secret

Creating a Secret by using the web console

How to Use a Secret in a Pod

Follow-up Actions

Operations

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:

```
# my-app-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-app-config
  namespace: default
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.

ConfigMap vs Secret

Feature	ConfigMap	Secret
Data Type	Non-sensitive	Sensitive (e.g., passwords)
Encoding	Plaintext	Base64-encoded
Use Cases	Configs, flags	Passwords, tokens

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 lines, ot
key1=value1
key2=value2
key3=value3
```

5.

Click **Create**.

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 -n k-1
```

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.
Delete	After deleting a ConfigMap, workloads that have referenced this ConfigMap (or its configuration items) through environment variables

Operations	Description
	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 -n k-1 -o yaml
kubectl edit configmap app-config -n k-1
kubectl delete configmap app-config -n k-1
```

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 (kubernetes.io/dockerconfigjson)**: Stores the JSON authentication string required for pulling container images from private image repositories (Docker Registry).

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 Docker registry type Secret

```
kubectl create secret docker-registry my-docker-creds \
  --docker-username=myuser \
  --docker-password=mypass \
  --docker-server=https://index.docker.io/v1/ \
  --docker-email=my@example.com
```

YAML

```
apiVersion: v1
kind: Secret
metadata:
  name: my-docker-creds
type: kubernetes.io/dockerconfigjson
data:
  .dockerconfigjson: eyJhdXRocyI6eyJodHRwczovL2luZGV4LmRvY2tlci5pby92MS8iOnsi
```

K8s automatically converts your username, password, email, and server information into the Docker standard login format:

```
{
  "auths": {
    "https://index.docker.io/v1/": {
      "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-docker-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).

```
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

```
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).

Menu

Creating Applications

Creating applications from Image

Creating applications from Chart

Creating applications from YAML

Creating applications from Code

Prerequisites

Procedure

Creating applications from Operator Backed

Procedure

Troubleshooting

Creating applications by using CLI

Prerequisites

Procedure

Example

Reference

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.

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 (docker-registry-projectname), which includes the project name projectname in this web console and the project name containers in the image repository.

1. 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:

Parameters	Description
	• **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`

Parameters	Description
	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.

Workload 2 - Configure Pod

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

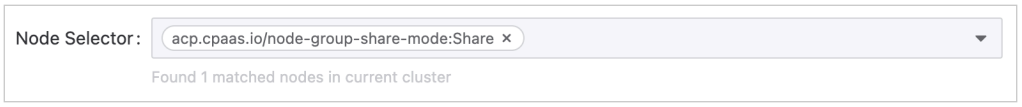
1.

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	Required **only** when pulling images from third-party registries (via manual image URL input). **Note**: Images from the platform's integrated registry automatically inherit associated secrets.

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

1. **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`).

Parameters	Description
	• `labelSelector`: Filters target Pods using label queries.

1. **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. • **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	Description
	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↗.
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:

2.1. Start before app containers (sequential execution).

2.2. Release resources after completion.

2.3. 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,

Parameters	Description
	ensure that the **Service** is associated with the workload under the application; otherwise, service discovery and access for workload will fail.

1. 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.

Operation	Description

Update	• **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**:

Operation	Description
	• Ingress Rule Deletion: External access remains available via `LB_IP:NodePort` 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.
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`.

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.

Type	Purpose
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`).

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`.
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.
	  • `InitializeFailed`: There was an exception in the initialization process before the Chart was downloaded.

Validated	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.
	  • `DependenciesCheckFailed`: Chart dependency check failed.
	  • `PermissionCheckFailed`: The current user lacks permission to perform operations on certain resources.
	  • `ConsistentNamespaceCheckFailed`: When deploying applications through templates in native applications, the Chart contains resources that require cross-namespace deployment.

Synced	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 `ChartSyncFailed`, and you can check the specific failure reason in the message column.

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 choose the one-click YAML creation method. This approach offers more flexible configuration of basic information and resources for your cloud-native application.

TOC

Precautions

When both Linux and Windows nodes exist in the cluster, to prevent scheduling the application on incompatible nodes, you must configure node selection. For example:

```
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 `docker pull` command.

Procedure

1.

Container Platform, and navigate to **Application** > **Applications**.

2.

Click **Create**.

3.

Select the **Create from YAML**.

4.

Complete the configuration and click **Create**.

5.

The corresponding **Deployment** can be viewed on the Details page.

```yaml
# 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"
            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
```

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

Operator backed applications are collections of resources provided by the Operator. Based on these Operator backed applications, you can quickly deploy a component application and leverage the capabilities of the Operator to automate the entire lifecycle management of the application.

TOC

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 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-Resour
```

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}
```

1. 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"
```

```
            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 ↗

Menu

Post-Application-Creation Configuration

Configuring HPA

Understanding Horizontal Pod Autoscalers

Prerequisites

Creating a Horizontal Pod Autoscaler

Calculation Rules

Configuring VerticalPodAutoscaler (VPA)

Understanding VerticalPodAutoscalers

Prerequisites

Creating a VerticalPodAutoscaler

Follow-Up Actions

Configuring CronHPA

Understanding Cron Horizontal Pod Autoscalers

Prerequisites

Creating a Cron Horizontal Pod Autoscaler

Schedule Rule Explanation

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:

```yaml
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)
  targetCPUUtilizationPercentage: 75 (9)
```

(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.

1. Apply the YAML file to create the HPA:

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

Example output:

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

1. 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
```

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

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

Example output:

```
NAME         REFERENCE                     TARGETS    MINPODS   MAXPODS   REPLIC
hpa-demo   Deployment/deployment-demo   0%/75%     3         7         3
```

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	For businesses with specific scaling rate requirements, you can gradually adapt to changes in business volume by specifying **Scale-Up Step** or **Scale-Down Step**.

Parameter	Description
Step (Alpha)	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
  targetCPUUtilizationPercentage: 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/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: demo
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: demo
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Pods
      pods:
        metricName: metric-demo
        targetAverageValue: 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/v2beta2

autoscaling/v2beta2 supports memory utilization:

```yaml
apiVersion: autoscaling/v2beta2
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:
          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.

Configuring VerticalPodAutoscaler (VPA)

For both stateless and stateful applications, VerticalPodAutoscaler (VPA) automatically recommends and optionally applies more appropriate CPU and memory resource limits based on your business needs, ensuring that pods have sufficient resources while improving cluster resource utilization.

TOC

Understanding VerticalPodAutoscalers

You can create a VerticalPodAutoscaler to recommend or automatically update the CPU and memory resource requests and limits for your pods based on their historical usage patterns.

After you create a VerticalPodAutoscaler, the platform begins to monitor the CPU and memory resource usage of the pods. When sufficient data is available, the VerticalPodAutoscaler calculates recommended resource values based on the observed usage patterns. Depending on the configured update mode, VPA can either automatically apply these recommendations or simply make them available for manual application.

The VPA works by analyzing the resource usage of your pods over time and making recommendations based on this analysis. It can help ensure that your pods have the resources they need without over-provisioning, which can lead to more efficient resource utilization across your cluster.

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 can achieve horizontal or vertical scaling of Pods. When sufficient resources are available, elastic scaling can bring good results, but when cluster resources are insufficient, it may cause Pods to be in a Pending state. Therefore, please ensure that the cluster has sufficient resources or reasonable quotas, or you can configure alerts to monitor scaling conditions.

Supported Features

The VerticalPodAutoscaler 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.

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**..
- The Alauda Container Platform Vertical Pod Autoscaler cluster plugin must be installed in your cluster.

Installing the Vertical Pod Autoscaler Plugin

Before using VPA, you need to install the Vertical Pod Autoscaler cluster 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.

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:

```
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:

- Auto: Automatically sets resource requests when creating pods and updates current pods to recommended resource requests. Currently equivalent to "Recreate". This mode may cause application downtime. Once in-place pod resource updates are supported, "Auto" mode will adopt this update mechanism.
- Recreate: Automatically sets resource requests when creating pods and evicts current pods to update to recommended resource requests. Will not use in-place updates.

- Initial: Only sets resource requests when creating pods, no modifications afterward.
- Off: Does not automatically modify pod resource requests, only provides recommendations in the VPA object.

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.

1. Apply the YAML file to create the VPA:

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

Example output:

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

1. 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.

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 **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

Parameter	Description
	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.
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: "Auto"` - Automatically sets resource requests when creating pods and updates current pods to recommended values. Currently equivalent to "Recreate".
- `updateMode: "Recreate"` - Automatically sets resource requests when creating pods and evicts current pods to update to recommended values.
- `updateMode: "Initial"` - Only sets resource requests when creating pods, no modifications afterward.
- `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 mode, which can cause temporary disruption to your application.

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 (1)
kind: CronHPA (2)
metadata:
  name: my-deployment-cronhpa (3)
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1 (4)
    kind: Deployment (5)
    name: my-deployment (6)
  crons:
  - schedule: "0 0 * * *" (7)
    targetReplicas: 0 (8)
  - schedule: "0 8 * * 1-5" (9)
    targetReplicas: 3 (10)
  - schedule: "0 18 * * 1-5" (11)
    targetReplicas: 1 (12)
```

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
  AM on weekdays (Monday-Friday)
- Scale down to 1 replica at 6
  PM on weekdays

1. 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.

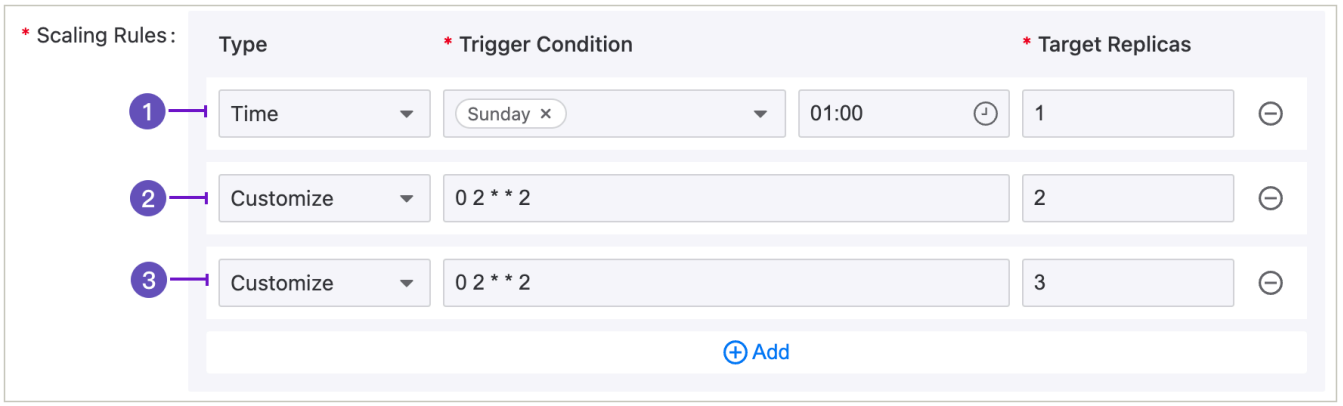
5.

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
   AM every Monday, only 1 pod will be retained.

2. Indicates that starting from 02
   AM every Tuesday, only 2 pods will be retained.

3. Indicates that starting from 02
   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.

Menu

Operation and Maintenance

Status Description

Applications

Starting and Stopping Applications

Starting the Application

Stopping the Application

Updating Applications

Importing Resources

Removing/Batch Removing Resources

Exporting Applications

Exporting Helm Charts

Exporting YAML to Local

Exporting YAML to Code Repository (Alpha)

Updating and deleting Chart Applications

Important Notes

Prerequisites

Status Analysis Description

Version Management for Applications

Creating a Version Snapshot

Rolling Back to a Historical Version

Deleting Applications

Health Checks

Understanding Health Checks

YAML file example

Health Checks configuration parameters by using web console

Troubleshooting probe failures

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.
Failed	Deployment has failed.

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

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.

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**.

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.

Feature	Description
	• 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.

Parameter	Description
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.
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.
Repository Address	The repository address assigned for your use under the integrated tool project.

Export Method	• **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.

File Path	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.
Commit Message	Fill in commit information to identify the content of this submission.
Preview	Preview the YAML file to be submitted and compare differences with the existing YAML in the code repository, displayed with color differentiation.

- 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.

Type	Reason

Initialized	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.

Validated	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.

Type	Reason
Synced	Indicates the state of the Chart template deployment. • When the status is True, it indicates that the Chart template deployment was successful. • 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.

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.

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 smal
          periodSeconds: 5
          failureThreshold: 60 # Allows 60 * 5 = 300 seconds (5 minutes) for
        livenessProbe:
          httpGet:
            path: /healthz
            port: 8080
          initialDelaySeconds: 5  # Delay 5 seconds after Pod starts before c
          periodSeconds: 10       # Check every 10 seconds
          timeoutSeconds: 5       # Timeout after 5 seconds
          failureThreshold: 3     # Consider unhealthy after 3 consecutive fa
        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).

Parameter	Applicable Protocols	Description
Command	EXEC	Container-executable check command (e.g., `sh -c "curl -I localhost:8080 | grep OK"`). **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.

Menu

Application Observability

Monitoring Dashboards

Prerequisites

Namespace-Level Monitoring Dashboards

Workload-Level Monitoring

Logs

Procedure

Events

Procedure

Event records interpretation

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. **Platform Management**, 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

1. Click the **Toggle Icon** to switch to custom dashboards. Refer to Add Pannel 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 under Observe.

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.

Menu

Workloads

Deployments

Understanding Deployments

Creating Deployments

Managing Deployments

Troubleshooting by using CLI

DaemonSets

Understanding DaemonSets

Creating DaemonSets

Managing DaemonSets

StatefulSets

Understanding StatefulSets

Creating StatefulSets

Managing StatefulSets

CronJobs

Understanding CronJobs

Creating CronJobs

Execute Immediately

Deleting CronJobs

Jobs

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

```shell
# 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.

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 (docker-registry-projectname), which includes the project name projectname in this web console and the project name containers in the image repository.

1.

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)` . • 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.

Parameters	Description
	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

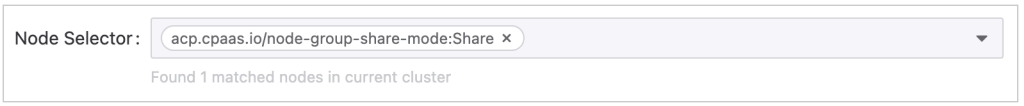
Note: In mixed-architecture clusters deploying single-architecture images, ensure proper Node Affinity Rules are configured for Pod scheduling.

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	Required **only** when pulling images from third-party registries (via manual image URL input). **Note**: Secret for authentication when pulling image from a secured registry.
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.

1. **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**: • `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.

1.

Network Configuration

- Kube-OVN

Parameters	Description
Bandwidth Limits	Enforce QoS for Pod network traffic: • **Egress rate limit**: Maximum outbound traffic rate (e.g., `10Mbps`).

Parameters	Description
	• **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**:

Parameters	Description
	• **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 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↗.

Parameters	Description
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`)
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.
More > **Log Files**	Configure log paths: - Default: Collect `stdout`

Parameters	Description
	- 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:

2.1. Start before app containers (sequential execution).

2.2. Release resources after completion.

2.3. 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

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
.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
```

1. 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
```

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.

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.

Troubleshooting by using CLI

When a Deployment encounters issues, here are some common troubleshooting methods.

Check Deployment status

```
kubectl get deployment nginx-deployment
kubectl describe deployment nginx-deployment # View detailed events and statu
```

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 contai
kubectl logs <pod-name> --previous          # View logs for the previously ter
```

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.

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
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector: # defines how the DaemonSet identifies its managed Pods. Must mat
    matchLabels:
      name: fluentd-elasticsearch
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template: # defines the Pod Template for the DaemonSet. Each Pod created by
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations: # these tolerations are to have the daemonset runnable on
      - 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
      # it may be desirable to set a high priority class to ensure that a Dae
      # 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*, execut
kubectl apply -f example-daemonSet.yaml

# Step 2: To verify the creation and status of your DaemonSet and its associa
kubectl get daemonset fluentd-elasticsearch # View DaemonSet
kubectl get pods -l name=fluentd-elasticsearch -o wide # Check Pods managed b
```

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.

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 (docker-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. • **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.

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).

- 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.

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
  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
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
---
# example-service.yaml
apiVersion: v1
kind: Service
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*, ex
kubectl apply -f example-statefulset.yaml

# Step 2: To verify the creation and status of your StatefulSet and its assoc
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.

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 (docker-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. **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.

Parameters	Description

	• 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**.

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-count>
```

- 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).

- 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> # Example: ku
```

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.

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.

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

```shell
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.

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 (docker-registry-projectname) indicates the platform's project name projectname and the image registry's project name containers.

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**

  5.1. Click the vertical ellipsis (⋮) on the right side of the cronjob list.

  5.2. 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>
```

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).

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>
```

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

Term	Definition	Notes
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.tgz
```

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: `docker pull nginx`
2. Save as tar package: `docker save nginx > nginx.latest.tar`
3. Load and push to private registry:

```
docker load -i nginx.latest.tar
docker tag nginx:latest 192.168.80.8:30050/nginx:latest
docker 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/$NAMESPA
--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\":\"IfNotPresent
    },
    "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/$NAMESPA
--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/$NAMESPA
--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-system/
--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.

Menu

Pod

Introduction

Introduction

Pod Parameters

Pod Parameters

Deleting Pods

Deleting Pods

Use Cases

Procedure

Container

Introduction

Debug Container (Alpha)

Implementation Principle

Notes

Use Cases

Procedure

Entering the Container via EXEC

Entering the Container through Applications

Entering the Container through the Pod

Introduction

Refer to the official Kubernetes website documentation: Pod ↗

Menu

Pod Parameters

The platform interface provides various information about the pods for quick reference. Below are some parameter explanations.

Parameter	Description
Resource Requests/Limits	The effective resource (CPU, memory) requests and limits values for the pods. The calculation method for requests and limits values is the same; this document introduces using the limit values as an example, and the specific rules and algorithms are as follows: • When the pods only contains **business containers** (containers), the **CPU/memory** limit value is the sum of the **CPU/memory** limit values of all containers within the pods. **For example**: If the pods includes two business containers with **CPU/memory** limit values of 100m/100Mi and 50m/200Mi, the pods's **CPU/memory** limit value will be 150m/300Mi. • When the pods contains both **init containers** (initContainers) and **business containers**, the calculation steps for the pods's **CPU/memory** limit values are as follows: 1. Take the maximum value of the **CPU/memory** limit values of all init containers. 2. Take the sum of **CPU/memory** limit values of all business containers. 3. Compare the results and take the maximum values of **CPU** and **memory** from both init containers and business containers as the pods's **CPU/memory** limit values. **Calculation Example**: If the pods contains two init containers with **CPU/memory** limit values of 100m/200Mi and

Parameter	Description
	200m/100Mi, the maximum **CPU/memory** limit value for the init containers would be 200m/200Mi. At the same time, if the pods also contains two business containers with **CPU/memory** limit values of 100m/100Mi and 50m/200Mi, the total limit value for the business containers will be 150m/300Mi. Therefore, the comprehensive **CPU/memory** limit value for the pods would be 200m/300Mi.
Source	The computing component to which the pods belongs.
Restart Count	The number of restarts when the pods's status is abnormal.
Node	The name of the node where the pods is located.
Service Account	The Service Account is an account that allows processes and services in the Pod to access the Kubernetes APIServer, providing an identity for the processes and services. The **Service Account** field is visible only when the currently logged-in user has either the platform administrator role or the platform auditor role, and the YAML file of the Service Account can be viewed.

Deleting Pods

Deleting pods may affect the operation of computing components; please proceed with caution.

TOC

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.

Menu

Container

Introduction

Debug Container (Alpha)

Implementation Principle

Notes

Use Cases

Procedure

Entering the Container via EXEC

Entering the Container through Applications

Entering the Container through the Pod

Introduction

Refer to the official Kubernetes website documentation: Containers↗.

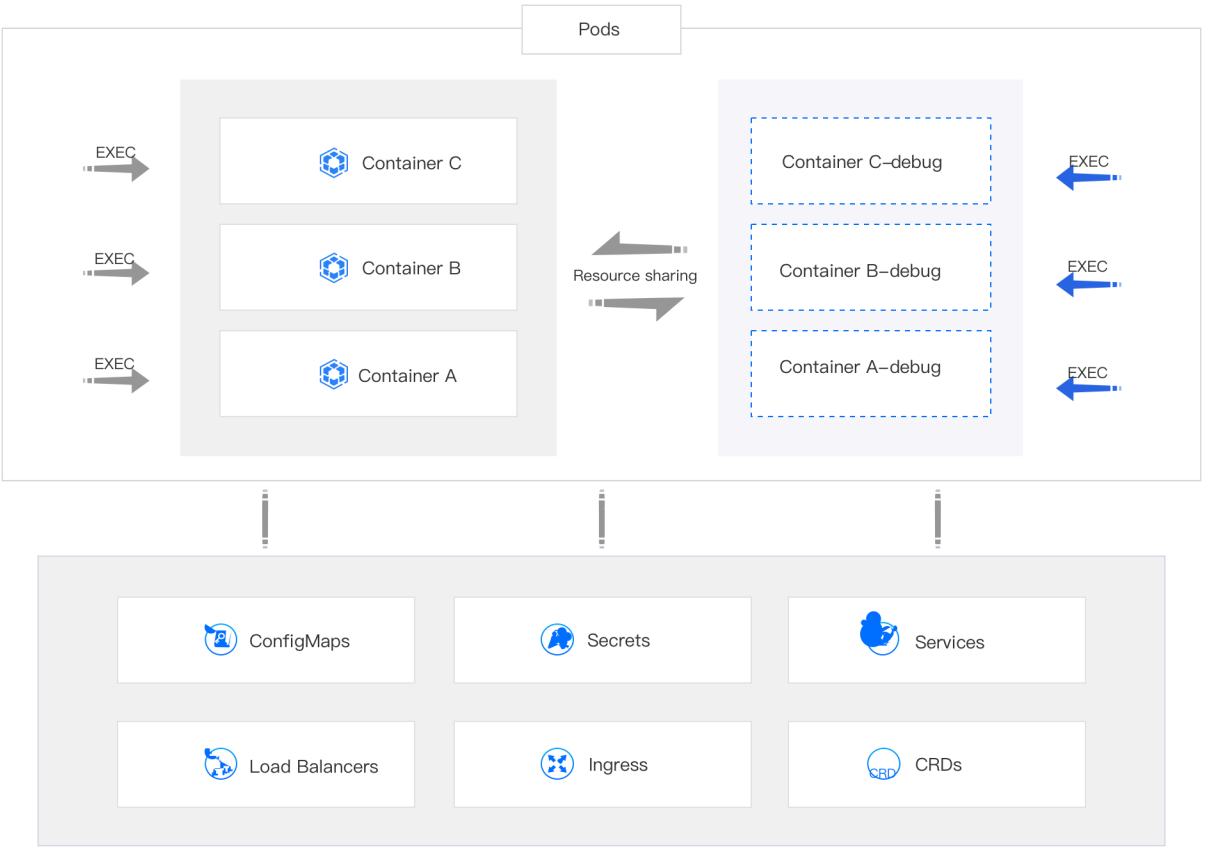
Debug Container (Alpha)

The Debug feature provides relevant tools for debugging running containers, including system, network, and disk utilities.

TOC

Implementation Principle

The Debug feature is implemented through Ephemeral Containers. An Ephemeral Container is a type of container that shares resources with business containers. You can add an Ephemeral Container (for example, *Container A-debug*) to a pod and use debugging tools within that container. The debugging results will be directly applied to the business container (such as *Container A*).



Notes

- You cannot add an Ephemeral Container by directly updating the pod configuration; make sure to enable the Ephemeral Container through the Debug feature.
- The Ephemeral Containers enabled by the Debug feature do not have resource or scheduling guarantees and will not restart automatically. Please avoid running business applications in them, except for debugging purposes.
- Please use the Debug feature cautiously if the resources on the node where the pod is located are about to be exhausted, as it may lead to the eviction of the pod.

Use Cases

Although you can also log into containers and debug using the EXEC feature, many container images do not include the required debugging tools (such as bash, net-tools, etc.) for the sake

of image size reduction. In contrast, the Debug feature, which comes pre-installed with debugging tools, is more suitable for the following scenarios.

- Fault Diagnosis: If a business container encounters an issue, in addition to checking events and logs, you may need to conduct more detailed troubleshooting and resolution within the container.
- Configuration Tuning: If there are flaws in the current business solution, you might want to perform configuration tuning on the business components within the container to devise a new configuration scheme that helps the business run more effectively.

Procedure

1.

Enter the **Container Platform**.

2.

In the left navigation bar, click **Workloads** > **Pods**.

3.

Locate the pod and click ⋮ > **Debug**.

4.

Select the container you wish to debug.

5.

(Optional) If prompted by the interface that **initialization is required**, click **Initialize**.

Note: 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.

6.

Wait for the debugging window to be ready, and then begin debugging.

Tip: Click the command query in the upper right corner to view common tools and their usage.

7.

Once finished, close the debugging window.

Entering the Container via EXEC

TOC

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.

Prerequisites

- The container must be running properly.
- When using the file transfer feature, the `tar` tool must be available in the container, and the container's operating system cannot be Windows.

Procedure

1.

Enter **Container Platform**.

2.

In the left navigation bar, click **Application** > **Applications**.

3.

Click on ***Application Name***.

4.

Locate the workload and click **EXEC** > ***Pod Name***.

5.

Enter the command you wish to execute.

6.

Click **OK** to enter the Web console window and execute command-line operations.

7.

Click **File Transfer**. Enter **Upload Path** to upload files for testing into the container; or enter **Download Path** to download logs and other files from the container to your local machine for analysis.

Entering the Container through the Pod

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.

Prerequisites

- The container must be running properly.
- When using the file transfer feature, the `tar` tool must be available in the container, and the container's operating system cannot be Windows.

Procedure

1.

In the left navigation bar, click **Workloads** > **Pods**.

2.

Click ⋮ > **EXEC** > ***Container Name***.

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 **Upload Path** to upload files for testing into the container; or enter **Download Path** to download logs and other files from the container to your local machine for analysis.

Menu

How To

Setting Scheduled Task Trigger Rules

Time Conversion

Writing Crontab Expressions

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
converts to UTC time: 42 09 - 00 08 = 42 01, which means the UTC time is 1
AM.

Example 2: Beijing time 4
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
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`.

Menu

Registry

Introduction

Introduction

Principles and namespace isolation

Authentication and authorization

Advantages

Application Scenarios

Install

Install Via YAML

When to Use This Method?

Prerequisites

Installing Alauda Container Platform Registry via YAML

Updating/Uninstalling Alauda Container Platform Registry

Install Via Web UI

When to Use This Method?

Prerequisites

Installing Alauda Container Platform Registry cluster plugin using the web console

Updating/Uninstalling Alauda Container Platform Registry

How To

Common CLI Command Operations

Logging in Registry

Add namespace permissions for users

Add namespace permissions for a service account

Pulling Images

Pushing Images

Using Alauda Container Platform Registry in Kubernetes Clusters

Registry Access Guidelines

Deploy Sample Application

Cross-Namespace Access

Best Practices

Verification Checklist

Troubleshooting

Introduction

Building, storing and managing container images is a core part of the cloud-native application development process. Alauda Container Platform(ACP) provides a high-performance, highly-available, built-in container image repository service designed to provide users with a secure and convenient image storage and management experience, greatly simplifying application development, continuous integration/continuous deployment (CI/CD) and application deployment processes within the platform. CD) and application deployment processes within the platform.

Deeply integrated into the platform architecture, Alauda Container Platform Registry provides tighter platform collaboration, simplified configuration, and greater internal access efficiency than an external, independently deployed image repository.

TOC

Principles and namespace isolation

Alauda Container Platform's built-in image repository, as one of the core components of the platform, runs inside the cluster in a highly-available manner and utilizes the persistent

storage capabilities provided by the platform to ensure that the image data is secure and reliable.

One of its core design concepts is logical isolation and management based on Namespace. Within the Registry, image repositories are organized by namespace. This means that each namespace can be considered as a separate “zone” for images belonging to that namespace, and images between different namespaces are isolated by default, unless explicitly authorized.

Authentication and authorization

The authentication and authorization mechanism of Alauda Container Platform Registry is deeply integrated with ACP's platform-level authentication and authorization system, enabling access control as granular as the namespace:

Authentication

Users or automated processes (e.g., CI/CD pipelines on the platform, automated build tasks, etc.) do not need to maintain a separate set of account passwords for the Registry. They are authenticated through the platform's standard authentication mechanisms (e.g., using platform-provided API tokens, integrated enterprise identity systems, etc.). When accessing Alauda Container Platform Registry through the CLI or other tools, it is common to utilize existing platform login sessions or ServiceAccount tokens for transparent authentication.

Authorization

Authorization control is implemented at the namespace level. Pull or Push permissions for an image repository in Alauda Container Platform Registry depend on the platform role and permissions that the user or ServiceAccount has in the corresponding namespace.

- **Typically**, the owner or developer role of a namespace is automatically granted Push and Pull permissions to image repositories under that namespace.
- **Users in other namespaces or users who wish to pull images across namespaces** need to be explicitly granted the appropriate permissions by the administrator of the target

namespace (e.g., bind a role that allows pulling of images via RBAC) before they can access images within that namespace.

- **This namespace-based authorization** mechanism ensures isolation of images between namespaces, improving security and avoiding unauthorized access and modification.

Advantages

Core advantages of Alauda Container Platform Registry:

- **Ready-to-Use:** Rapidly deploy a private image registry without complex configurations.
- **Flexible Access:** Supports both intra-cluster and external access modes.
- **Security Assurance:** Provides RBAC authorization and image scanning capabilities.
- **High Availability:** Ensures service continuity through replication mechanisms.
- **Production-Grade:** Validated in enterprise environments with SLA guarantees.

Application Scenarios

- **Lightweight Deployment:** Implement streamlined registry solutions in low-traffic environments to accelerate application delivery.
- **Edge Computing:** Enable autonomous management for edge clusters with dedicated registries.
- **Resource Optimization:** Demonstrate full workflow capabilities through integrated Source to Image (S2I) solutions when underutilizing infrastructure.

Menu

Install

Install Via YAML

When to Use This Method?

Prerequisites

Installing Alauda Container Platform Registry via YAML

Updating/Uninstalling Alauda Container Platform Registry

Install Via Web UI

When to Use This Method?

Prerequisites

Installing Alauda Container Platform Registry cluster plugin using the web console

Updating/Uninstalling Alauda Container Platform Registry

Install Via YAML

TOC

When to Use This Method?

Recommended for:

- **Advanced users** with Kubernetes expertise who prefer a manual approach.
- **Production-grade deployments** requiring enterprise storage (NAS, AWS S3, Ceph, etc.).
- Environments needing **fine-grained control** over TLS, ingress.
- **Full YAML customization** for advanced configurations.

Prerequisites

- **Install** the **Alauda Container Platform Registry** cluster plugin to a target cluster.
- **Access** to the target Kubernetes cluster with kubectl configured.
- **Cluster admin permissions** to create cluster-scoped resources.
- Obtain a registered **domain** (e.g., registry.yourcompany.com) Create a Domain
- Provide valid **NAS storage** (e.g., NFS, GlusterFS, etc.).
- (Optional) Provide valid **S3 storage** (e.g., AWS S3, Ceph, etc.). If no existing S3 storage is available, deploy a MinIO (Built-in S3) instance in the cluster Deploy MinIO.

Installing Alauda Container Platform Registry via YAML

Procedure

1. **Create a YAML configuration file** named registry-plugin.yaml with the following template:

```yaml
apiVersion: cluster.alauda.io/v1alpha1
kind: ClusterPluginInstance
metadata:
  annotations:
    cpaas.io/display-name: internal-docker-registry
  labels:
    create-by: cluster-transformer
    manage-delete-by: cluster-transformer
    manage-update-by: cluster-transformer
  name: internal-docker-registry
spec:
  config:
    access:
      address: ""
      enabled: false
    fake:
      replicas: 2
    global:
      expose: false
      isIPv6: false
      replicas: 2
      resources:
        limits:
          cpu: 500m
          memory: 512Mi
        requests:
          cpu: 250m
          memory: 256Mi
    ingress:
      enabled: true
      hosts:
        - name: <YOUR-DOMAIN>   # [REQUIRED] Customize domain
          tlsCert: <NAMESPACE>/<TLS-SECRET>  # [REQUIRED] Namespace/SecretNam
      ingressClassName: "<INGRESS-CLASS-NAME>"  # [REQUIRED] IngressClassName
      insecure: false
    persistence:
      accessMode: ReadWriteMany
      nodes: ""
      path: <YOUR-HOSTPATH>  # [REQUIRED] Local path for LocalVolume
      size: <STORAGE-SIZE>  # [REQUIRED] Storage size (e.g., 10Gi)
      storageClass: <STORAGE-CLASS-NAME>  # [REQUIRED] StorageClass name
      type: StorageClass
    s3storage:
```

```
      bucket: <S3-BUCKET-NAME>              # [REQUIRED] S3 bucket name
      enabled: false                         # Set false for local storage
      env:
        REGISTRY_STORAGE_S3_SKIPVERIFY: false  # Set true for self-signed cer
      region: <S3-REGION>                        # S3 region
      regionEndpoint: <S3-ENDPOINT>  # S3 endpoint
      secretName: <S3-CREDENTIALS-SECRET>             # S3 credentials Secret
    service:
      nodePort: ""
      type: ClusterIP
  pluginName: internal-docker-registry
```

1. **Customize the following fields** according to your environment:

```
spec:
  config:
    ingress:
      hosts:
        - name: "<YOUR-DOMAIN>"                # e.g., registry.your-company.
          tlsCert: "<NAMESPACE>/<TLS-SECRET>"   # e.g., cpaas-system/tls-secr
      ingressClassName: "<INGRESS-CLASS-NAME>"  # e.g., cluster-alb-1
    persistence:
      size: "<STORAGE-SIZE>"                     # e.g., 10Gi
      storageClass: "<STORAGE-CLASS-NAME>"      # e.g., cpaas-system-storage
    s3storage:
      bucket: "<S3-BUCKET-NAME>"                 # e.g., prod-registry
      region: "<S3-REGION>"                     # e.g., us-west-1
      regionEndpoint: "<S3-ENDPOINT>"           # e.g., https://s3.amazonaws.c
      secretName: "<S3-CREDENTIALS-SECRET>"     # Secret containing AWS_ACCES
      env:
        REGISTRY_STORAGE_S3_SKIPVERIFY: "true"  # Set "true" for self-signed
```

1. **How to create a secret** for S3 credentials:

```
kubectl create secret generic <S3-CREDENTIALS-SECRET> \
  --from-literal=access-key-id=<YOUR-S3-ACCESS-KEY-ID> \
  --from-literal=secret-access-key=<YOUR-S3-SECRET-ACCESS-KEY> \
  -n cpaas-system
```

Replace `<S3-CREDENTIALS-SECRET>` with the name of your S3 credentials secret.

1. Apply the configuration to your cluster:

```
kubectl apply -f registry-plugin.yaml
```

Configuration Reference

Mandatory Fields

Parameter	Description	Example V
`spec.config.ingress.hosts[0].name`	Custom domain for registry access	`registry.yourcc`
`spec.config.ingress.hosts[0].tlsCert`	TLS certificate secret reference (namespace/secret-name)	`cpaas-system/re tls`
`spec.config.ingress.ingressClassName`	Ingress class name for the registry	`cluster-alb-1`
`spec.config.persistence.size`	Storage size for the registry	`10Gi`
`spec.config.persistence.storageClass`	StorageClass name for the registry	`nfs-storage-sc`
`spec.config.s3storage.bucket`	S3 bucket name for image storage	`prod-image-stor`
`spec.config.s3storage.region`	AWS region for S3 storage	`us-west-1`
`spec.config.s3storage.regionEndpoint`	S3 service endpoint URL	`https://s3.amaz`
`spec.config.s3storage.secretName`	Secret containing S3 credentials	`s3-access-keys`

Verification

1. Check plugin:

```
kubectl get clusterplugininstances internal-docker-registry -o yaml
```

1. Verify registry pods:

```
kubectl get pods -n cpaas-system -l app=internal-docker-registry
```

Updating/Uninstalling Alauda Container Platform Registry

Update

Execute the following command on the global cluster::

```
# <CLUSTER-NAME> is the cluster where the plugin is installed
kubectl edit -n cpaas-system \
  $(kubectl get moduleinfo -n cpaas-system -l cpaas.io/cluster-name=<CLUSTER-
```

Uninstall

Execute the following command on the global cluster:

```
# <CLUSTER-NAME> is the cluster where the plugin is installed
kubectl get moduleinfo -n cpaas-system -l cpaas.io/cluster-name=<CLUSTER-NAME
```

Install Via Web UI

TOC

When to Use This Method?

Recommended for:

- **First-time users** who prefer a guided, visual interface.
- **Quick proof-of-concept setups** in non-production environments.
- Teams with **limited Kubernetes expertise** seeking a simplified deployment process.
- Scenarios requiring **minimal customization** (e.g., default storage configurations).
- **Basic networking setups** without specific ingress rules.
- **StorageClass** configurations for high availability.

Not Recommended for:

- Production environments requiring advanced storage(S3 storage) configurations.
- Networking setups needing specific ingress rules.

Prerequisites

- **Install** the **Alauda Container Platform Registry** cluster plugin to a target cluster using the Cluster Plugin mechanism.

Installing Alauda Container Platform Registry cluster plugin using the web console

Procedure

1. Log in and navigate to the **Administrator** page.
2. Click **Marketplace** > **Cluster Plugins** to access the **Cluster Plugins** list page.
3. Locate the **Alauda Container Platform Registry** cluster plugin, click **Install**, then proceed to the installation page.
4. Configure parameters according to the following specifications and click **Install** to complete the deployment.

The parameter descriptions are as follows:

Parameter	Description
Expose Service	Once enabled, administrators can manage the image repository externally using the access address. This poses significant security risks and should be enabled with extreme caution.
Enable IPv6	Enable this option when the cluster uses IPv6 single-stack networking.
NodePort	When Expose Service is enabled, configure NodePort to allow external access to the Registry via this port.
Storage Type	Select a storage type. Supported types: LocalVolume and StorageClass.

Parameter	Description
Nodes	Select a node to run the Registry service for image storage and distribution. (Available only when Storage Type is LocalVolume)
StorageClass	Select a StorageClass. When replicas exceed 1, select storage with RWX (ReadWriteMany) capability (e.g., File Storage) to ensure high availability. (Available only when Storage Type is StorageClass)
Storage Size	Storage capacity allocated to the Registry (Unit: Gi).
Replicas	Configure the number of replicas for the Registry Pod: • **LocalVolume**: Default is 1 (fixed) • **StorageClass**: Default is 3 (adjustable)
Resource Requirements	Define CPU and Memory resource requests and limits for the Registry Pod.

Verification

1. Navigate to **Marketplace** > **Cluster Plugins** and confirm the plugin status shows **Installed**.
2. Click the plugin name to view its details.
3. Copy the **Registry Address** and use the Docker client to push/pull images.

Updating/Uninstalling Alauda Container Platform Registry

You can update or uninstall the **Alauda Container Platform Registry** plugin from either the list page or details page.

Menu

How To

Common CLI Command Operations

Logging in Registry

Add namespace permissions for users

Add namespace permissions for a service account

Pulling Images

Pushing Images

Using Alauda Container Platform Registry in Kubernetes Clusters

Registry Access Guidelines

Deploy Sample Application

Cross-Namespace Access

Best Practices

Verification Checklist

Troubleshooting

Common CLI Command Operations

The Alauda Container Platform provides command line tools for users to interact with the Alauda Container Platform Registry. The following are some examples of common operations and commands:

Let's assume that Alauda Container Platform Registry for the cluster has a service address of registry.cluster.local and the namespace you are currently working on is my-ns.

> Contact technical services to acquire the kubectl-acp plugin and ensure it is properly installed in your environment.

TOC

Logging in Registry

Log in to the cluster's Registry by logging in to the ACP.

```
kubectl acp login <ACP-endpoint>
```

Add namespace permissions for users

Add namespace pull permission for a user.

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-puller -
```

Add namespace push permissions to a user.

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-pusher -
```

Add namespace permissions for a service account

Add namespace pull permission for a service account.

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-puller -
```

Add namespace push permission for a service account.

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-pusher -
```

Pulling Images

Pulls an image from the registry to inside the cluster (e.g., for Pod deployment).

```
# Pull the image named my-app, labeled latest, from the Registry of the curre
kubectl acp pull registry.cluster.local/my-ns/my-app:latest

# Pull images from other namespaces (e.g. shared-ns) (requires permission to
kubectl acp pull registry.cluster.local/shared-ns/base-image:latest
```

This command verifies your identity and pull permissions in the target namespace, and then pulls the image from the Registry.

Pushing Images

Pushes locally built images or images pulled from elsewhere to a specific namespace in the registry.

You need to first tag (tag) the local image with the address and namespace format of the target Registry using a standard container command line tool such as docker.

```
# Tag it with the target address:
docker tag my-app:latest registry.cluster.local/my-ns/my-app:v1

# Use the kubectl command to push it to the Registry for the current namespac
kubectl acp push registry.cluster.local/my-ns/my-app:v1
```

Pushes an image from a remote image repository to a specific namespace in the Alauda Container Platform Registry.

```
# If your remote image repository has an image remote.registry.io/demo/my-app
# Use the kubectl command to push it to the namespace(my-ns) of the registry
kubectl acp push remote.registry.io/demo/my-app:latest registry.cluster.local
```

This command verifies your identity and push permissions within the my-ns namespace, and then uploads the locally tagged image to Registry.

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 `internal-docker-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 `internal-docker-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 `internal-docker-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: internal-docker-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 `internal-docker-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: `internal-docker-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=syste
```

1. Validate image accessibility for a user in `my-ns`:

```
kubectl auth can-i get images.registry.alauda.io --namespace my-ns --as=<user
```

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 `internal-docker-registry.cpaas-system.svc` is correctly configured.
  - Verify node's /etc/hosts configuration to ensure correct DNS resolution of `internal-docker-registry.cpaas-system.svc`
  - Example showing registry service mapping (ClusterIP of internal-docker-registry service):

    ```
    # /etc/hosts
    127.0.0.1   localhost localhost.localdomain
    10.4.216.11 internal-docker-registry.cpaas-system internal-docker-regist
    ```

  - **How to get `internal-docker-registry` current ClusterIP**:

    ```
    kubectl get svc -n cpaas-system internal-docker-registry -o jsonpath='{.
    ```

Menu

Source to Image

Introduction

Introduction

Source to Image Concept

Core Features

Core Benefits

Application scenarios

Usage Limitations

Install

Installing Alauda Container Platform Builds

Prerequisites

Procedure

Architecture

Architecture

Guides

Managing applications created from Code

Key Features

Advantages

Prerequisites

Procedure

Related operations

How To

Creating an application from Code

Prerequisites

Procedure

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: Tekton Operator is now available in the cluster OperatorHub.

Menu

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 **Tekton** 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 **Tekton** 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 **Platform Management** 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.

1. 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.

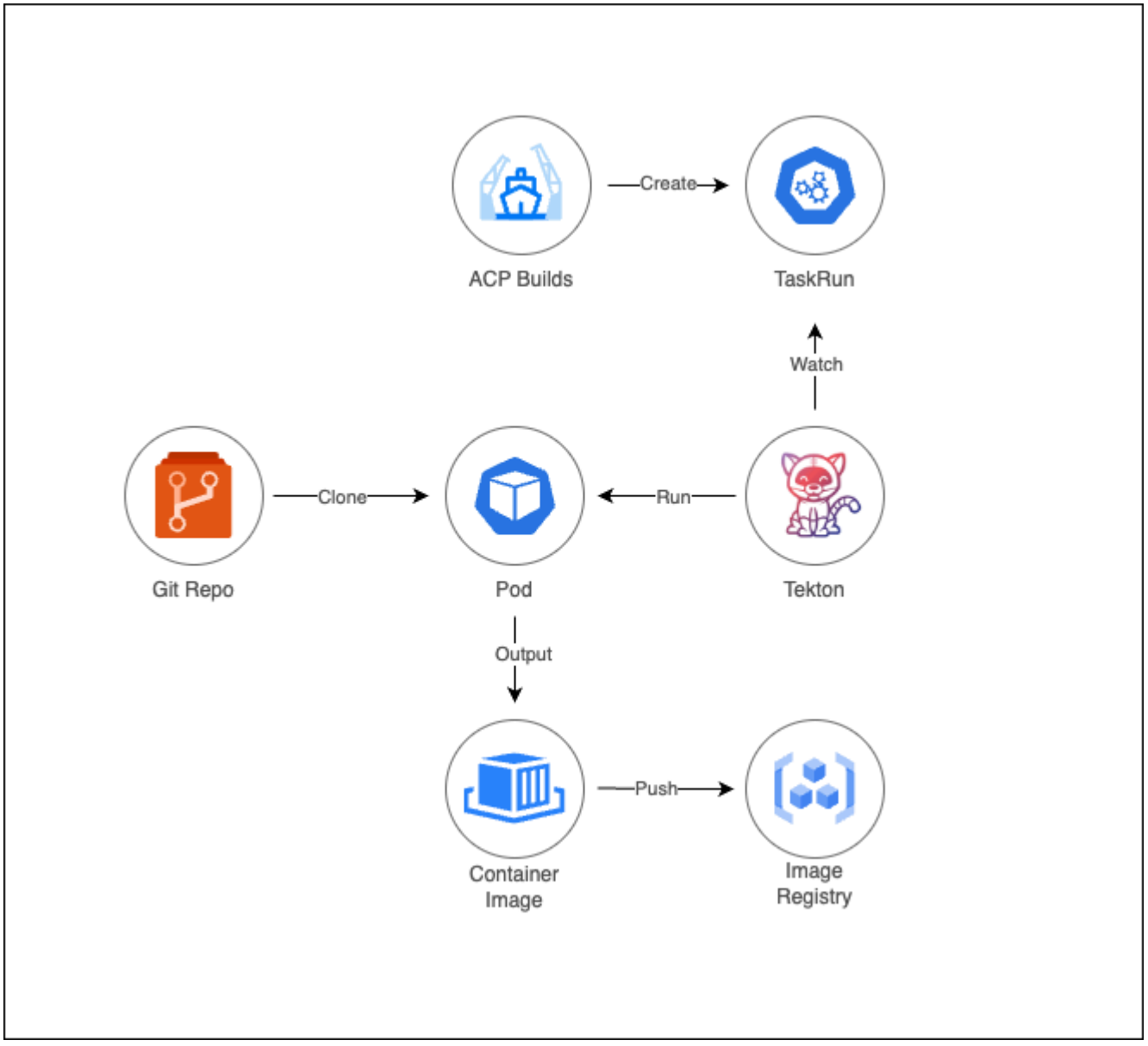
Click **Create**.

Verification

- After the instance is successfully creted, wait approximately "20 mins" before switching to **Container Platform** > **Applications** > **Applications** and click on **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.

Menu

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.

Menu

Guides

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: 4.1. 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 4.2. 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. 4.3.

		Create BuildRun type resources to initiate specific build instances, which coordinate the entire build process. 4.4. 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.

Menu

How To

Creating an application from Code

Prerequisites

Procedure

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 Alauda Container Platform 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.

Menu

Node Isolation Strategy

Node Isolation Strategy provides a project-level node isolation strategy that allows projects to exclusively use cluster nodes.

Introduction

Introduction

Advantages

Application Scenarios

Architecture

Architecture

Concepts

Core Concepts

Node Isolation

Guides

Create Node Isolation Strategy

Create Node Isolation Strategy

Delete Node Isolation Strategy

Permissions

Permissions

Introduction

Node Isolation Strategy provides a project-level node isolation strategy that allows projects to exclusively use cluster nodes.

TOC

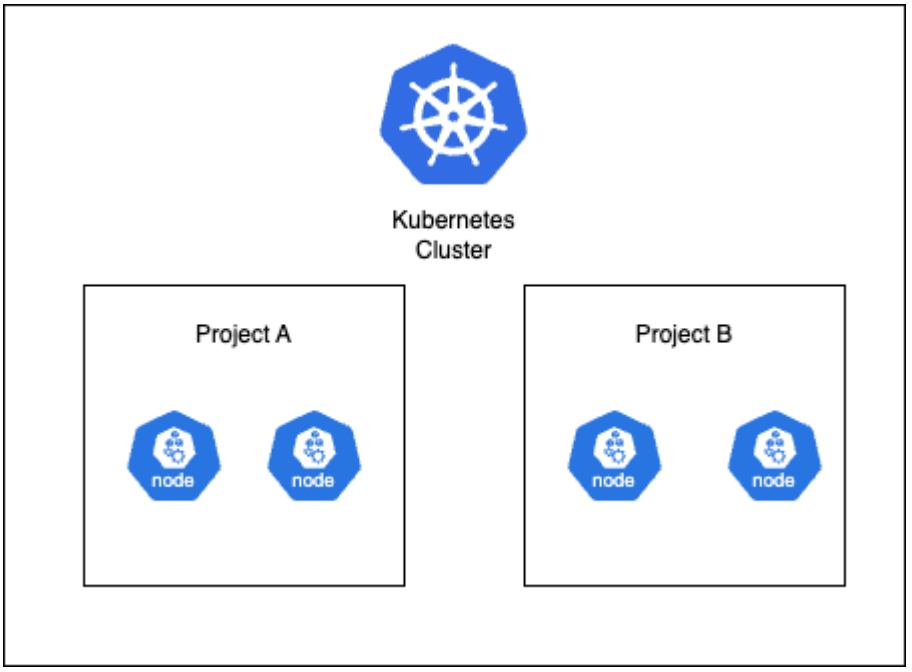
Advantages

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.

Menu

Concepts

Core Concepts

Node Isolation

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.

Menu

Guides

Create Node Isolation Strategy

Create Node Isolation Strategy

Delete Node Isolation Strategy

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.

Parameter	Description
	When the switch is on, only Pods under the specified project in the policy 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**.

Menu

Permissions

Function	Action	Platform Administrator	Platform auditors	Project Manager	Namespace Administrator
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:

```
---
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:

```
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 or a LimitRange that is not named `default` 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.

The platform will create a LimitRange named `default` when creating a namespace. Therefore, before importing a namespace, only a single LimitRange named `default` should exist in the namespace.