Deckhouse Platform User's guide

Version latest

Generated: November 21, 2026

Contents

Contents 2
Configuring authentication for applications 11
Overview 11
Interface 11
Changing and resetting a local user’s password 12
Password change required by policy or administrator 12
Changing the password in the web interface 12
Signing out on all devices 12
Enabling Authentication in a Web Application 12
Authentication via proxy (for applications without OIDC support) 13
Authentication for applications with OIDC support 14
Additional resources 15
Configuring Vertical Pod Autoscaler (VPA) for Applications 15
How Vertical Pod Autoscaler (VPA) works 15
How VPA interacts with limits 16
Monitoring VPA in Grafana 18
VPA configuration examples 18
Priority classes 19
Using a priority class in a Deployment 19
Step-by-step preemption demonstration 20
Environment separation by priority 22
Protecting stateful applications 22
Demo of stateful application protection 23
Step 1. Create a protected StatefulSet with a PDB 23
Step 2. Simulate a resource shortage 24
Step 3. Observe the protection mechanisms 24
How protection mechanisms work in a critical situation 25
Operations and diagnostics 25
Pod does not start due to insufficient resources 25
Practical diagnostics when preemption is impossible 26
Practical check when no suitable pods are available 26
Practical example “Pod limit per node” 27
Useful commands for monitoring priorities 29
Incoming traffic processing 30
NLB features and purpose 30
ALB features and purpose 30
Utilizing Network Load Balancer (NLB) 31
Examples of Service configuration 31
Shared IP address for multiple Services 31
Forcing an IP address assignment 31
Assigning an IPAddressPool (BGP mode) 32
Utilizing Application Load Balancer (ALB) 32
Comparison of ALB options 32
Ingress NGINX 32
Kubernetes Gateway API 33
Istio 33
How to tell what is available in the cluster 33
Next steps 33
Additional resources 34
Publishing applications using the Kubernetes Gateway API 34
Publishing applications using the Kubernetes Gateway API 34
Publishing an application with ListenerSet and HTTPRoute 34
Checking publication 36
Working with GRPCRoute, TLSRoute, TCPRoute, and UDPRoute objects 36
GRPCRoute for gRPC traffic 36
TLS passthrough with TLSRoute 37
TLS termination at the gateway with TCPRoute 39
Additional TCP and UDP ports 40
Publishing the app via a different gateway 42
Linking routes in one namespace to a ListenerSet in another 42

Configuring TLS parameters with BackendTLSPolicy 43
Supported HTTPRoute annotations 45
Publishing an application when the Istio sidecar is enabled 46
WAF on HTTPRoute 47
Additional resources 49
Publishing applications using the Ingress NGINX Controller 49
Publishing applications using the Ingress NGINX Controller 49
Verification 51
gRPC load balancing 51
Additional resources 51
Publishing applications using Istio 51
Publishing applications using Istio 51
Publishing applications using Ingress NGINX 51
Publishing applications using Istio Ingress Gateway resource 52
Canary deployment with VirtualService 53
Additional resources 54
Additional networks for use in application pods 55
Creating a project network (user network) 55
Connecting additional networks to pods 56
IPAM for additional networks (IP allocation and assignment) 57
Features of using IPAM in a DP cluster 57
Allocating a pool of IP addresses for the project network and enabling IPAM 57
Assigning IP addresses to network interfaces of pods connected to an additional network 58
Automatic IP address assignment 58
Manual (explicit) creation of IPAddress with type Auto 58
Manually assigning a static IP address to an additional pod interface 59
Verification of IP address assignment to an Interface 60
Additional resources 61
Connecting physical network interfaces to DPDK application pods 61
Connecting physical network interfaces to pods 61
Creating a pod with a device from the Underlay network 62
IP addresses of virtual machines 64
IP addresses of VMs 64
Assigning a specific IP address 65
Keeping an IP address in the project 66
IPAM for the main network 67
Switching a machine to the shared IPAM 68
Additional resources 68
Accessing applications on a virtual machine 68
Headless service 69
Service of the ClusterIP type 69
Service of the NodePort type 70
Service of the LoadBalancer type 71
Publishing VM services with Ingress 72
Additional network interfaces of virtual machines 73
Additional network interfaces 73
Interface naming and ARP flux in the guest OS 75
IPAM for additional network interfaces 76
Configuring the guest OS for interfaces added on the fly 77
Additional resources 78
Intra-cluster communication 78
Standard load balancer 78
Limitations of the standard Service load balancer 78
Example Service for a standard load balancer 78
Example of a standard load balancer for a PostgreSQL cluster 79
Creating a StatefulSet for PostgreSQL 79
Advanced load balancer 80
Load balancer structure 80
Configuring the load balancer 80
Example configuration of advanced ServiceWithHealthchecks load balancers 80
Hosting two independent load balancers on a single virtual machine 81
Creating the virtual machine 81

Manifests for advanced load balancers for the web and SMTP services 82
Additional resources 83
Managing authorization and access to workload with Istio 83
Authorization 83
AuthorizationPolicy resource 83
Additional resources 84
Configuring request routing with Istio 84
VirtualService resource 84
Additional resources 84
Enabling Istio for applications 84
Additional resources 85
Managing request balancing between service endpoints with Istio 85
DestinationRule resource 86
Additional resources 86
Configuring Locality failover with Istio 86
Additional resources 87
Configuring request retries with Istio 87
VirtualService resource 87
Additional resources 88
Configuring resources for istio-proxy sidecars 88
Supported annotations 88
Configuration examples 88
Additional resources 89
Configuring Circuit Breaker 89
Example Circuit Breaker configuration 90
Additional resources 90
Configuring Canary deployment 90
Example Canary deployment configuration with Ingress NGINX 91
Creating a Deployment and Service for the stable version 91
Creating a Deployment and Service for the Canary version 91
Configuring Ingress for Canary deployment 92
Gradually increasing traffic to the Canary version 93
Rolling back or completing the Canary deployment 93
Additional annotations for Canary deployment 93
Example Canary deployment configuration with Istio 94
Distribution based on the presence of a cookie 94
Distribution based on probability 95
Additional resources 95
Configuration of container image vulnerability scanning 95
Accessing scan results 95
Viewing a scan report for your application 95
Viewing CIS compliance check results 96
Viewing scan results 96
Resource types for scan results 97
Resource-level security 97
VulnerabilityReport 97
ConfigAuditReport 99
ExposedSecretReport 100
SbomReport 101
Cluster-level security 104
RbacAssessmentReport 104
ClusterComplianceReport 107
Additional resources 108
Configuration of using TLS certificates 108
Working with certificates 108
Viewing certificate details 108
Automatic certificate requesting 109
Requesting a wildcard certificate using DNS in Cloudflare 109
Requesting a wildcard certificate using DNS in AWS Route53 110
Requesting a wildcard certificate using DNS in Google 111
Requesting a self-signed certificate 112
Generating a self-signed certificate 112

Certificate generation example 113
Protecting credentials 114
Support for tls-acme annotation 115
Additional resources 116
Deckhouse Platform web UI 117
Accessing the web UI 117
Deckhouse 117
“Overview” subsection 117
“Updates” subsection 118
“Modules” subsection 119
“Global settings” subsection 119
Kubernetes 120
Node management 121
“Node groups” subsection 121
“Instance classes” subsection 126
“Nodes” subsection 128
“Static instances” subsection 129
Multitenancy 130
“Projects” subsection 130
“Project templates” subsection 131
“User namespaces” subsection 133
Access 134
“User operations” subsection 134
Network 136
“Ingress controllers” subsection 136
Storage 137
“Storage classes” subsection 137
“Persistent volume” subsection 137
“Local volumes” subsection 138
Security 140
“CVE scanner” subsection 140
Monitoring 140
“Overview” subsection 140
“Metric processing” subsection 141
“Metric delivery” subsection 142
“Grafana data sources” subsection 142
“Grafana dashboards” subsection 143
“Active alerts” subsection 144
Logging 144
“Log delivery” subsection 144
“Log collection” subsection 145
Viewing API information 146
Additional resources 147
Grafana web interface 147
Monitoring system web UI 147
Main page 148
Working with dashboards 148
Data filtering 149
Working with data 150
Dashboard descriptions 153
Viewing security event logs 166
Additional resources 168
Web interface for managing alerts 169
Accessing the web UI 169
Web interface for downloading CLI utilities 170
Accessing the web interface 170
Additional resources 171
Secret management web interface 171
Accessing the stronghold web UI 171
Managing secrets engines 172
Viewing a secrets engine 172
Managing access to data and stronghold features 176

Working with authentication methods 177
User groups 177
Entities and aliases 178
Managing temporary access to secrets and resources (leases) 179
Managing access control policies 180
Using additional tools 181
Monitoring stronghold Raft cluster 183
Monitoring activity and load 183
Sealing and unsealing the secrets storage 184
Using the stronghold CLI 184
Additional resources 185
Web interface for visualizing the cluster network stack 185
Accessing the web interface 185
Filtering data for display 186
Working with the network flow diagram 187
Working with the network flows and events table 188
Configuring the column set 188
Viewing row details 189
Using multitenancy 190
Additional namespaces 190
Creating a project 190
Automatically creating a project from a namespace 191
Viewing available cluster-wide resources 192
Additional resources 192
Overview 192
How it works 192
Managed PostgreSQL 193
Check available resources 193
Check storage 194
Main example of creating Postgres 194
Select PostgresClass 195
Placement restrictions 196
Configure resources 196
Change resources of an existing Postgres 196
Check CPU and memory limits via PostgresClass 197
Select deployment mode 198
Cluster mode 198
Standalone mode 198
Configure topology and replication mode 199
Configure topology 199
Placement without zone selection 199
Placement within a single zone 200
Configure replication 200
Check replication mode 200
Availability mode 201
Consistency mode 201
ConsistencyAndAvailability mode 203
Change replication mode of an existing cluster 203
Create logical database and user 204
PostgreSQL user 204
Declarative user management 205
Logical database 206
Connect to PostgreSQL 206
Connect from within the cluster 207
External connection to PostgreSQL 207
Publish PostgreSQL for external access 207
Connect by IP address 209
Connect with TLS verification 210
Configure PostgreSQL parameters 211
Change an allowed parameter 211
Restrictions on parameter overrides 212
Validate parameter values 212

Configure TLS 212
K8s mode 213
Monitoring and alerts 213
Backup and restore 214
Create a snapshot 214
Restore from a PostgresSnapshot 217
Check status 218
Additional resources 218
Application and Infrastructure Monitoring 218
Available by default 219
Dashboards 219
Alerts 219
Monitoring configuration 219
Additional resources 219
Monitoring dashboards 220
Dashboard types 220
Preinstalled dashboards 220
Ingress Nginx 220
Resource consumption (Main) 220
Security 221
Custom dashboards 221
Using the observability module 221
Access control 222
Converting dashboards from GrafanaDashboardDefinition 223
Using GrafanaDashboardDefinition 224
Additional resources 224
Configuring Application Monitoring 225
Configuring metrics collection via labels and annotations 225
Example of configuring metrics collection from a Service 226
Example of configuring metrics collection from a Deployment 227
Additional annotations for advanced configuration 227
Configuring metrics collection using PodMonitor or ServiceMonitor resources 228
Adding custom labels to metrics 229
Configuring metrics collection via scrape_configs using the ScrapeConfig resource 230
Configuring metrics collection using blackbox-exporter 231
Additional resources 232
Application delivery with Argo CD 232
Prerequisites 232
AppProject projects 232
Preparing a namespace 233
Deploying an application 233
Creating an application through the Argo CD web interface 233
Creating an application with the argocd CLI utility 234
Creating an application with an Application object 234
Additional resources 235
Marketplace 236
Prerequisites 236
Additional resources 236
Installing and managing applications 236
Browsing available package versions 236
Packages available in a project 237
Installing an application 238
Naming constraints 238
Checking application status 238
Conditions 238
Brief state description 239
Installing multiple instances 239
Configuring workload resources 240
Autoscaling 240
Updating an application 242
Manual update 242
Automatic updates 242

Rolling back to an earlier version 243
Version compatibility rules 243
upgrade.from constraint 244
Deleting an application 244
FAQ 244
Can an application be updated automatically? 244
Can an Application depend on another Application? 244
Can I install the same application in different namespaces? 245
Additional resources 245
Troubleshooting 245
Verifying Marketplace custom resources 245
Quick application diagnostics 245
Reading individual conditions 246
Viewing DP logs 246
Viewing application pod logs 246
Common condition reasons 246
Additional resources 247
Configuration of applications logs collection 247
Configuring application log collection 248
Examples 249
Creating a source in a namespace and reading logs from all pods in it with direction to Loki 249
Reading pods in a specified namespace with a specific label 249
Additional resources 250
Virtual machines 250
Images and disks 250
Working with a virtual machine 250
Snapshots, cloning, and pools 251
Networking 251
Device passthrough 251
Additional resources 251
Quick start 251
Additional resources 254
Virtual machine images 255
Sources and storage options 255
Creating an image from an HTTP server 255
Verifying the integrity of a downloaded image 256
Storing an image in a PVC 257
Creating an image from a container image registry 258
Uploading an image from the command line 259
Creating an image from a disk 260
Creating an image from a disk snapshot 260
Additional resources 261
Virtual machine disks 261
How storage affects a disk 262
Creating an empty disk 263
Creating a disk from an image 263
Uploading a disk from the command line 265
Additional resources 266
Resizing and migrating virtual machine disks 266
Changing the disk size 266
Migrating disks to other storage 267
Additional resources 268
Exporting virtual machine disk data 268
Additional resources 268
Creating a virtual machine 269
Creating a virtual machine 269
Virtual machine life cycle 271
Diagnostics by phase 271
Conditions of a running VM 272
Eviction from a node 272
Viewing the state in the web interface 272
Additional resources 273

Virtual machine CPU and memory 273
Configuring CPU and coreFraction 273
Automatic coreFraction (Auto) 274
Sizing policy 275
CPU topologies 276
Additional resources 277
Guest OS and bootloader of a virtual machine 278
Additional resources 280
Initial provisioning and the guest OS agent 281
VM initialization scripts 281
Cloud-Init 281
Sysprep 283
Guest OS agent 283
Additional resources 284
Connecting to a virtual machine 284
Connecting to a virtual machine 285
SPICE 285
Startup policy and VM state management 286
Additional resources 287
Changing the virtual machine configuration 287
Changing the number of cores without a restart 289
Changing the amount of memory without a restart 290
Additional resources 291
Placing virtual machines on nodes 291
Tolerance to node restrictions 292
Simple label binding (nodeSelector) 292
Preferred binding (Affinity) 293
Avoiding co-location (AntiAffinity) 294
Placing VMs across availability zones 295
Additional resources 295
Attaching disks and images to a virtual machine 296
Attaching through the VM specification 296
Attaching through VirtualMachineBlockDeviceAttachment 297
Disk naming in the guest OS 299
Network interface naming in the guest OS 299
Additional resources 300
Live migration of virtual machines 300
How live migration works 300
Requirements and limitations 301
Checking whether a VM is ready for migration 302
Specifics of the Migratable condition 303
Starting a live migration 303
Configuring the migration policy 305
Migrations with insufficient network bandwidth 306
Migrations started by the system 307
Live VM migration on a placement parameter change 308
Additional resources 308
Collecting debug information about a virtual machine 308
Additional resources 309
Snapshots of disks and virtual machines 309
Consistent snapshots 309
Creating disk snapshots 310
Recovering disks from snapshots 311
Creating VM snapshots 311
Recovering a VM 312
Additional resources 313
Cloning virtual machines 314
Creating a clone of an existing VM 315
Creating a clone from a VM snapshot 316
Additional resources 318
Virtual machine pools 318
Attaching a shared CD-ROM (or any shared image) to all replicas 320

Scaling the pool 320
Deleting specific replicas 321
Reusable disks (reclaim) 321
Pool limitations and specifics 322
Additional resources 322
USB devices in a virtual machine 323
Project devices (USBDevice) 323
USBDevice conditions 323
Attaching a USB device to a VM 323
Additional resources 324
GPU devices in a virtual machine 324
Additional resources 325
PCI devices in a virtual machine 325
Attaching a PCI device to a VM 325
Limitations 326
Additional resources 326

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Configuring authentication for applications

Overview

Authentication is the process of verifying a user's identity. In Deckhouse Platform (DP), end-to-end authentication is implemented, allowing user verification when accessing any DP interface or cluster resources. Cluster users can also use DP to enable authentication for their applications.

Depending on the DP configuration, authentication can use either an internal user database or external authentication providers. Connecting an external provider allows the use of existing credentials (e.g., LDAP, GitLab, GitHub, etc.) for access. It also enables using the same credentials to authenticate in multiple DP clusters.

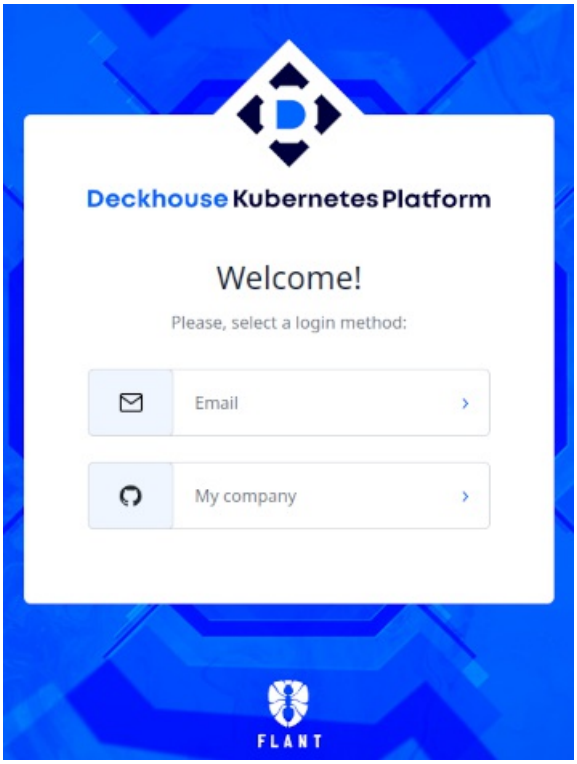
From the perspective of a cluster user or an application developer, it does not matter how the DP administrator configured authentication — the user interface and authentication methods for applications will be the same.

> To use authentication in DP, configuration is required.

Interface

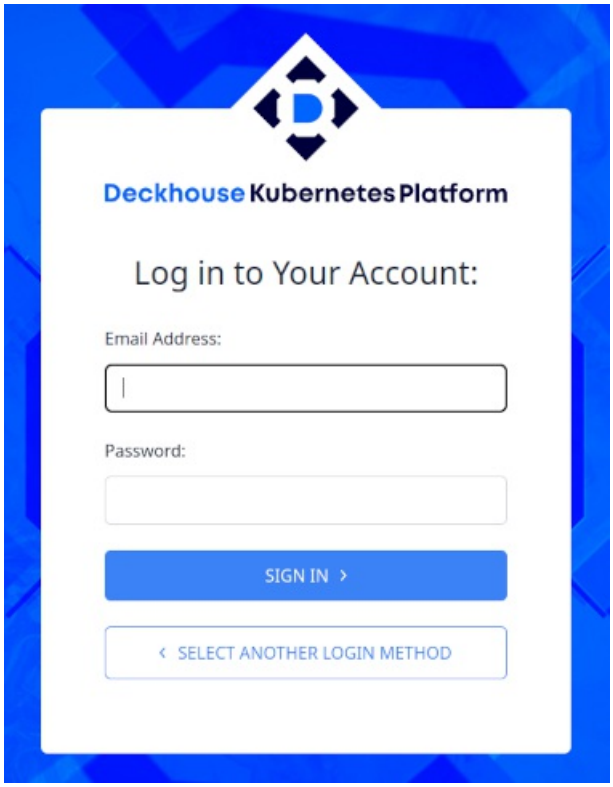
The authentication interface appears upon the first request to a resource for which authentication is enabled — DP redirects the user to the authentication page. If the user is already authenticated (e.g., via an external identity provider), DP will redirect the request back to the original resource, enriched with authentication data. If authentication has not yet occurred, the user will see the authentication interface.

Example of DP authentication interface:



The authentication interface allows users to choose an authentication method if multiple methods are configured. If only one external provider is configured, the user is redirected directly to that provider's login page. If local users are defined in DP, the user will be prompted to enter a username and password.

Example of DP authentication interface with username and password input:



Changing and resetting a local user’s password

If local authentication is configured in the cluster, a user can change their password on their own.

Password change required by policy or administrator

During login, DP may redirect a local user to the password change form if:

- The password complexity does not meet the configured policy.
- The password has expired according to the configured policy.
- An administrator initiated a forced password change.

The form asks for a new password and its confirmation, and states why the password must be changed. The new password must comply with the cluster password policy.

Changing the password in the web interface

A local user can change their password in the DP web interface. At the next sign-in, enter the current password in the sign-in form. Then the password change form opens. Enter a new password and its confirmation. The new password must comply with the cluster password policy. Changing the password does not end your sessions.

The password change is available only for local accounts. If you sign in through LDAP, GitHub, or another external provider, change the password in that system.

Signing out on all devices

In the DP web interface, you can sign out on all devices. This ends every session of your account, including the sessions in other applications and the kubeconfig files issued to you. Sign in again to continue working.

Enabling Authentication in a Web Application

> **i** To enable authentication in an application, authentication must first be configured at the Deckhouse Platform level.

DP supports two ways of enabling authentication for an application, depending on whether the application is capable of handling authentication requests (i.e., acting as an OIDC client) or not. Both approaches are described below.

Authentication via proxy (for applications without OIDC support)

Authentication for applications that cannot independently handle authentication requests is implemented via a special proxy server. This proxy handles both authentication and authorization, hiding the details of these processes from the application.

To enable authentication for an application deployed in DP, follow these steps:

1. Create a DexAuthenticator object in the application's namespace.

   After the DexAuthenticator object is created, DP will automatically deploy a set of components required for authentication:

   - A Deployment with containers running the authentication/authorization proxy and a Redis data store;
   - A Service pointing to the authentication/authorization proxy;
   - If `applicationIngressClassName` is set, an Ingress resource that handles requests at `https://<applicationDomain>/dex-authenticator` and forwards them to the proxy Service;
   - If `gatewayAPI` is set, an HTTPRoute;
   - Secrets required for integration with the DP authentication system.

   The DexAuthenticator name must be a DNS-1035 label of at most 63 characters: it starts with a lowercase letter and contains only lowercase letters, digits and hyphens. The name is a part of the name of its Service, so a DexAuthenticator with another name cannot be created.

   Example DexAuthenticator manifest:

   ```
   apiVersion: deckhouse.io/v1
   kind: DexAuthenticator
   metadata:
     # Prefix for Dex authenticator pod names.
     # For example, if the name prefix is `app-name`, pods will be named like `app-
   name-dex-authenticator-7f698684c8-c5cjg`.
     name: app-name
     # Namespace where Dex authenticator will be deployed.
     namespace: app-ns
   spec:
     # Domain name of your application. Requests to this domain will be redirected
   to Dex for authentication.
     applicationDomain: "app-name.kube.my-domain.com"
     # Whether to send an `Authorization: Bearer` header to the application. Useful
   with NGINX's auth_request.
     # If sendAuthorizationHeader is true, add Authorization to the
   nginx.ingress.kubernetes.io/auth-response-headers annotation of the
   application's Ingress or to the alb.network.deckhouse.io/auth-response-headers
   annotation of its HTTPRoute.
     sendAuthorizationHeader: false
     # Name of the Secret containing the TLS certificate.
     applicationIngressCertificateSecretName: "ingress-tls"
     # Name of the Ingress class to use for the Ingress resource created for the
   Dex authenticator.
     applicationIngressClassName: "nginx"
     # Duration for which the user session remains active.
     keepUsersLoggedInFor: "720h"
     # List of groups whose users are allowed to authenticate.
     allowedGroups:
     - everyone
     - admins
     # List of IP addresses and CIDRs allowed to authenticate.
     whitelistSourceRanges:
     - 1.1.1.1/32
     - 192.168.0.0/24
   ```

   Note the following capabilities when configuring authentication:

   - The `applicationDomain` parameter in the DexAuthenticator defines the main domain of the

application. Additional domains can be specified via the `additionalApplications.domain` parameter.

- The `whitelistSourceRanges` and `additionalApplications.whitelistSourceRanges` parameters allow you to restrict authentication access to a specific list of IP addresses. The limit applies to the `/dex-authenticator` endpoints. To restrict access to the application itself, or to let some networks in without authentication, use the `nginx.ingress.kubernetes.io/whitelist-source-range` and `nginx.ingress.kubernetes.io/satisfy` annotations of the application's Ingress.

  Refer to the Authorization section for information about configuring authorization.

2. Add the following annotations to the application's Ingress resource:

   - `nginx.ingress.kubernetes.io/auth-signin: https://$host/dex-authenticator/sign in`
   - `nginx.ingress.kubernetes.io/auth-response-headers: X-Auth-Request-User,X-Auth-Request-Email`
   - `nginx.ingress.kubernetes.io/auth-url: https://<NAME>-dex-authenticator.<NS>.svc.<CLUSTER_DOMAIN>/dex-authenticator/auth`

     Where:

     - `<NAME>` is the value of `metadata.name` from the DexAuthenticator;
     - `<NS>` is the value of `metadata.namespace` from the DexAuthenticator;
     - `<CLUSTER_DOMAIN>` is the cluster domain (defined in the `network.clusterDomain` parameter of the `control-plane-manager` module, by default it is `cluster.local`).

       Example (for a DexAuthenticator named `app-name` in namespace `app-ns`):

```
 annotations:
   nginx.ingress.kubernetes.io/auth-signin: https://$host/dex-authenticator/sign_in
   nginx.ingress.kubernetes.io/auth-url: https://app-name-dex-authenticator.app-ns.svc.cluster.local/dex-authenticator/auth
   nginx.ingress.kubernetes.io/auth-response-headers: X-Auth-Request-User,X-Auth-Request-Email
```

⚠ When setting `sendAuthorizationHeader: true`, list all necessary headers in the `nginx.ingress.kubernetes.io/auth-response-headers` annotation of the Ingress, since the `Authorization` header is not transmitted by default:

```
nginx.ingress.kubernetes.io/auth-response-headers: X-Auth-Request-User,X-Auth-Request-Email,Authorization
```

Authentication for applications with OIDC support

Applications that can handle authentication requests on their own and act as OIDC clients can directly integrate with the DP authentication system. In this case, the application independently redirects the user to the login page and processes the received OIDC tokens.

To enable authentication for such an application, follow these steps:

1. Create a DexClient object in the application's namespace.

   After the DexClient object is created, Deckhouse will perform the following:

   - An OIDC client will be registered in the DP authentication system with an identifier (`clientID`) in the format:
     `dex-client-<NAME>@<NAMESPACE>`
     where `<NAME>` and `<NAMESPACE>` are `metadata.name` and `metadata.namespace` from the DexClient resource;
   - A `clientSecret` will be automatically generated and saved in a Kubernetes Secret named `dex-client-<NAME>` in the same namespace;
   - You can then use the generated `clientID` and `clientSecret` to configure OIDC support in your application.

2. Specify the allowed redirect URIs.

These URIs define where the provider (Dex) can redirect the user after successful authentication.

3. Restrict access by groups or emails, if needed.
   Use the `allowedGroups` and `allowedEmails` parameters to specify which user groups and users are allowed to authenticate via this client.

4. (Optional) Define trusted peers.
   Use the `trustedPeers` parameter to allow authentication delegation between applications.

   Example of a DexClient object:

```
apiVersion: deckhouse.io/v1
kind: DexClient
metadata:
  name: myname
  namespace: mynamespace
spec:
  redirectURIs:
  - https://app.example.com/callback
  - https://app.example.com/callback-reserve
  allowedGroups:
  - Everyone
  - admins
  trustedPeers:
  - opendistro-sibling
```

5. Retrieve the `clientSecret`.
   The secret will be automatically created:

```
apiVersion: v1
kind: Secret
metadata:
  name: dex-client-myname
  namespace: mynamespace
type: Opaque
data:
  clientSecret: c2VjcmV0
```

6. Configure your application as an OIDC client. Use the `clientID`, `clientSecret`, and `redirectURIs`, along with the Dex provider URL.
   You can get the Dex URL (`https://dex.<publicDomainTemplate>`) using the following command:

```
d8 k -n d8-user-authn get ingress dex -o jsonpath="{.spec.rules[*].host}"
```

Additional resources

- Module `control-plane-manager` parameters
- Module `user-authn` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Configuring Vertical Pod Autoscaler (VPA) for Applications

How Vertical Pod Autoscaler (VPA) works

The Vertical Pod Autoscaler (VPA) automates container resource management and can significantly

improve application performance. VPA is especially useful when it’s difficult to estimate resource needs in advance. When VPA is used in an appropriate operating mode, it sets requested resources based on actual usage data collected from the monitoring system. You can also configure it to only provide recommendations without applying changes automatically.

How VPA interacts with limits

VPA manages the container’s resource **requests**, but it does not manage **limits** unless explicitly configured to do so.

VPA calculates recommended values based on resource usage by the container. This behavior can affect the ratio between requests and limits:

- If requests and limits are equal, VPA will only update the requests, leaving limits unchanged.
- If limits are not specified, VPA will only update requests.
- If limits are set but not controlled by VPA, the ratio between requests and limits may shift.

1. Example 1. In the cluster, we have:
    - A VPA object:

      ```
      apiVersion: autoscaling.k8s.io/v1
      kind: VerticalPodAutoscaler
      metadata:
        name: test2
      spec:
        targetRef:
          apiVersion: "apps/v1"
          kind: Deployment
          name: test2
        updatePolicy:
          updateMode: "Initial"
      ```

    - A pod with specified resources:

      ```
      resources:
      limits:
        cpu: 2
      requests:
        cpu: 1
      ```

      If the container consumes 1 CPU, VPA will recommend 1.168 CPU. In this case, the ratio between requests and limits is 100% (since the request is managed, but the limit remains unchanged). When the pod is recreated, VPA will update the resources as follows:

      ```
      resources:
      limits:
        cpu: 2336m
      requests:
        cpu: 1168m
      ```

2. Example 2. The cluster contains:
    - A VPA object:

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: test2
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: test2
  updatePolicy:
    updateMode: "Initial"
```

- A pod with specified resources:

```
resources:
limits:
  cpu: 1
requests:
  cpu: 750m
```

In this case, the ratio between requests and limits will be 25%. If VPA recommends 1.168 CPU, the container’s resources will be adjusted to:

```
  resources:
    limits:
      cpu: 1557m
    requests:
      cpu: 1168m
```

If resources are not limited, VPA might assign excessively high resource values, which can cause issues.

To prevent this, you can:

- Use the `maxAllowed` parameter in the VPA specification:

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: my-app-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: my-app
  updatePolicy:
    updateMode: "InPlaceOrRecreate"
  resourcePolicy:
    containerPolicies:
    - containerName: hamster
      minAllowed:
        memory: 100Mi
        cpu: 120m
      maxAllowed:
        memory: 300Mi
        cpu: 350m
      mode: Auto
```

- Configure a `LimitRange` in the cluster:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: my-app-limits
spec:
  limits:
  - default:
      cpu: 2
      memory: 4Gi
    defaultRequest:
      cpu: 500m
      memory: 256Mi
    type: Container
```

Monitoring VPA in Grafana

To efficiently manage resources using the Vertical Pod Autoscaler (VPA), it is recommended to use Grafana dashboards. These dashboards allow you to track the current status of VPA, its configuration, and the percentage of pods on which it is active.

Grafana provides several levels of detail for VPA-related information. Key dashboards include:

- Main / Namespace — Displays general VPA usage per namespace.
- Main / Namespace / Controller — Provides VPA metrics for specific controllers.
- Main / Namespace / Controller / Pod — The most granular view, showing data for each individual pod.

Key columns to monitor:

- VPA type — Shows the current value of `updatePolicy.updateMode`, which defines the VPA operating mode. This field appears in the following dashboards:
  - Main / Namespace
  - Main / Namespace / Controller
  - Main / Namespace / Controller / Pod
- VPA % (Percentage of pods with VPA enabled) — Displays the percentage of pods within a namespace that have VPA enabled. This helps quickly assess how much of the cluster is covered by automatic resource scaling via VPA.

VPA configuration examples

1. Example of a minimal `VerticalPodAutoscaler` resource:

   ```
   apiVersion: autoscaling.k8s.io/v1
   kind: VerticalPodAutoscaler
   metadata:
     name: my-app-vpa
   spec:
     targetRef:
       apiVersion: "apps/v1"
       kind: StatefulSet
       name: my-app
   ```

2. Example of a full VerticalPodAutoscaler resource:

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: my-app-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: my-app
  updatePolicy:
    updateMode: "InPlaceOrRecreate"
  resourcePolicy:
    containerPolicies:
    - containerName: hamster
      minAllowed:
        memory: 100Mi
        cpu: 120m
      maxAllowed:
        memory: 300Mi
        cpu: 350m
      mode: Auto
```

Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.

The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Priority classes

A priority class defines which workloads are more important when a node runs out of resources. You can assign a class in a Deployment manifest, verify preemption, and figure out why a pod is stuck in `Pending`.

The priority class of a virtual machine (VM) is passed to its `virt-launcher` pod, so the scheduler uses the same rules to choose which VMs to preempt as it does for pods.

Using a priority class in a Deployment

Deckhouse Platform already provides a set of priority classes. The following example shows how to use a priority class in a Deployment pod template.

Create a file `deployment-with-priority.yaml` to deploy an application with the Deckhouse Platform predefined class `production-high` (value `9000`, see Available priority classes):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      # Specify the priority class name.
      priorityClassName: production-high
      containers:
      - name: app
        image: nginx:latest
        ports:
        - containerPort: 80
```

Apply the manifest to the cluster:

```
d8 k apply -f deployment-with-priority.yaml
```

Check the priorities of the created pods:

```
d8 k get pods -l app=my-app -o custom-
columns=NAME:.metadata.name,CLASS:.spec.priorityClassName,PRIORITY:.spec.priority
```

Example output:

```
NAME                       CLASS              PRIORITY
my-app-7d8f9b5c4f-2xq9t    production-high    9000
my-app-7d8f9b5c4f-4p7k2    production-high    9000
my-app-7d8f9b5c4f-9w5r8    production-high    9000
```

⚠ You cannot change the `priorityClassName` field of a running pod. This field is immutable. To change the priority, update the template in the Deployment and recreate the pods. For a VM, change the priority class in its specification and restart the VM if required.

Step-by-step preemption demonstration

This example shows how the scheduler preempts a lower-priority pod when a node lacks sufficient resources for a higher-priority pod. First, a pod with the `develop` class (`1000`) is started on a worker node and consumes most of the free resources. Then a pod with the `production-high` class (`9000`) is created with the same `requests` — there is no room left on the node, and the scheduler preempts the lower-priority pod in favor of the new one.

In the examples, replace `worker-0` with the name of your worker node.

i The example is designed for a single worker node with 4 CPU and 8 Gi of memory. If there are multiple nodes, pods may be placed on different nodes, and preemption may not occur. Adjust `requests` so that the low-priority pod consumes most of the node's free resources.

Check the list of nodes in the cluster:

```
d8 k get nodes
```

Example output:

```
NAME        STATUS   ROLES                  AGE   VERSION
master-0    Ready    control-plane,master   14d   v1.34.9
worker-0    Ready    worker                 14d   v1.34.9
```

Check resources on the worker node:

```
d8 k describe node worker-0 | grep -E "Capacity|Allocatable|Allocated" -A 5
```

Example output:

```
Capacity:
  cpu:                4
  memory:             8174932Ki
Allocatable:
  cpu:                3800m
  memory:             7174932Ki
Allocated resources:
  cpu:                1200m (31%)
  memory:             2Gi (28%)
```

Create a low-priority pod that consumes a significant portion of the free resources, then a high-priority pod that requests more than remains available.

Create a file `low-priority-pod.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: low-priority-pod
spec:
  priorityClassName: develop
  containers:
  - name: app
    image: nginx
    resources:
      requests:
        cpu: "2"
        memory: "4Gi"
```

Apply the manifest:

```
d8 k apply -f low-priority-pod.yaml
```

Wait until the pod reaches `Running` status:

```
d8 k get pods low-priority-pod
```

Example output:

```
NAME               READY   STATUS    RESTARTS   AGE
low-priority-pod   1/1     Running   0          10s
```

Create a file `high-priority-pod.yaml` to run a high-priority pod that requests more resources than remain available on the node:

```
apiVersion: v1
kind: Pod
metadata:
  name: high-priority-pod
spec:
  priorityClassName: production-high
  containers:
  - name: app
    image: nginx
    resources:
      requests:
        cpu: "2"
        memory: "4Gi"
```

Apply the manifest:

```
d8 k apply -f high-priority-pod.yaml
```

Check pod status:

```
d8 k get pods | grep priority
```

Example output:

```
NAME                READY   STATUS    RESTARTS   AGE
high-priority-pod   1/1     Running   0          5s
```

Pod `high-priority-pod` is in `Running` status. The `low-priority-pod` pod may no longer appear in the output because the scheduler preempted it within seconds. In some cases, you may see `low-priority-pod` in `Pending` status.

Check preemption events:

```
d8 k get events -A --field-selector reason=Preempted
```

Example output:

```
NAMESPACE   LAST SEEN   TYPE     REASON      OBJECT                  MESSAGE
default     68s         Normal   Preempted   pod/low-priority-pod   Preempted by pod
d9d25b95-4a7d-4214-8a30-8ce1fd616f67 on node worker-0
```

Environment separation by priority

A namespace itself does not affect preemption: the scheduler compares only pod priority classes, regardless of which namespaces the pods are in. To protect production workloads, use a lower priority class in test and develop environments (`develop`, `staging`), and a higher one in production (`production-low`, `production-medium`, `production-high`).

Protecting stateful applications

Stateful applications (for example, databases and message queues) store data in memory or persistent volumes (PVC). Protecting them requires a special approach: abruptly destroying a pod without a graceful shutdown can corrupt data, and mass preemption of replicas can make the service unavailable.

For more on protection from eviction, see Protecting a pod from eviction. The Demo of stateful application protection subsection shows a practical scenario where these mechanisms work together.

This example uses a combination of three mechanisms to protect stateful applications:

- A high `priorityClassName` — in the example, the StatefulSet uses `production-medium` (`6000`). For real critical stateful applications, usually use `production-high` (`9000`) or higher; see Available priority classes. This reduces the likelihood that the pods will be selected for preemption.
- PodDisruptionBudget (PDB) guarantees a minimum number of running replicas (for example, `minAvailable: 5`).

- The `terminationGracePeriodSeconds` parameter gives the application time to write data to disk and close transactions before the pod terminates (30–60 seconds is recommended).

Do not create a PodDisruptionBudget for VMs: it does not affect VM eviction. The time for a graceful guest operating system (OS) shutdown is set by the `.spec.terminationGracePeriodSeconds` parameter in the VM specification.

Demo of stateful application protection

This example deploys a sample stateful application with the `production-medium` class (`6000`), a PodDisruptionBudget, and `terminationGracePeriodSeconds` to limit the scale of preemption and give pods time to shut down gracefully. Then an `emergency-task` pod is created with the higher `production-high` class (`9000`) and a large memory request — to simulate a critical resource shortage on the node and show how the protection mechanisms work.

Step 1. Create a protected StatefulSet with a PDB

Create a file `stateful-protect.yaml`:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mock-stateful
spec:
  serviceName: mock-stateful
  replicas: 7
  selector:
    matchLabels:
      app: mock-stateful
  template:
    metadata:
      labels:
        app: mock-stateful
    spec:
      priorityClassName: production-medium
      terminationGracePeriodSeconds: 30
      containers:
      - name: app
        image: busybox
        command:
        - sh
        - -c
        - |
          trap 'echo ">>> START: Saving data to disk..."; sleep 10; echo ">>> END:
Data saved, exiting."' TERM
          echo "Application started and running..."
          while true; do sleep 1; done
        resources:
          requests:
            cpu: "100m"
            memory: "256Mi"
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: mock-stateful-pdb
spec:
  minAvailable: 5
  selector:
    matchLabels:
      app: mock-stateful
```

Apply the configuration:

```
d8 k apply -f stateful-protect.yaml
```

Wait until all 7 pods are in `Running` status:

```
d8 k get pods -l app=mock-stateful -w
```

Step 2. Simulate a resource shortage

Create a file `emergency-task.yaml` to run a pod with the predefined `production-high` class (value `9000`), which is higher than the stateful application's class (`production-medium`, value `6000`):

```
apiVersion: v1
kind: Pod
metadata:
  name: emergency-task
spec:
  priorityClassName: production-high
  containers:
  - name: task
    image: busybox
    command: ["sleep", "infinity"]
    resources:
      requests:
        cpu: "1"
        memory: "5Gi"
```

Apply the manifest:

```
d8 k apply -f emergency-task.yaml
```

Step 3. Observe the protection mechanisms

Check pod status:

```
d8 k get pods | grep -E 'mock-stateful|emergency-task'
```

Example output:

```
NAME                       READY   STATUS        RESTARTS   AGE
emergency-task             0/1     Pending       0          5s
mock-stateful-0            1/1     Terminating   0          55s
mock-stateful-1            1/1     Terminating   0          53s
mock-stateful-2            1/1     Running       0          51s
mock-stateful-3            1/1     Running       0          49s
mock-stateful-4            1/1     Running       0          47s
mock-stateful-5            1/1     Running       0          45s
mock-stateful-6            1/1     Terminating   0          43s
```

⚠ Logs of preempted pods are available only while the pod is in `Terminating` status. To see graceful shutdown, run `d8 k logs` before the pod is fully deleted.

Check the logs of a terminating pod:

```
d8 k logs mock-stateful-0 --tail=20
```

Example output:

```
Application started and running...
>>> START: Saving data to disk...
>>> END: Data saved, exiting.
```

How protection mechanisms work in a critical situation

Because the priority of the `emergency-task` pod created in Demo of stateful application protection (`9000`, class `production-high`) is higher than that of the stateful application (`6000`, class `production-medium`), and there are no other preemption candidates, the scheduler must select the stateful application for preemption. The protection mechanisms then take effect:

1. In this case, the PodDisruptionBudget tries to limit the impact, but because the request is extremely large, the scheduler violates the PDB while still trying to minimize the number of pods removed.
2. The `terminationGracePeriodSeconds` parameter helps preserve data even if preemption is complete.

Operations and diagnostics

This section provides practical commands to check pod status and scheduler events. Descriptions of problem causes and possible cluster-level actions are in the Operations and diagnostics admin section.

In the Step-by-step preemption demonstration, the `high-priority-pod` reaches `Running`. The following subsections cover two cases where the same pod stays in `Pending`: insufficient CPU or memory, and the scheduler finding no pods to preempt.

Pod does not start due to insufficient resources

If the cluster has no free resources and preemption is not possible, the pod remains in `Pending`.

Check the pod status:

```
d8 k get pod high-priority-pod
```

Example output:

```
NAME                READY   STATUS    RESTARTS   AGE
high-priority-pod   0/1     Pending   0          2m
```

Check the pod events:

```
d8 k describe pod high-priority-pod | grep -A10 "Events:"
```

Look for messages like the following in the `Events` section:

```
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  2m    default-scheduler  0/3 nodes are available: 2
Insufficient cpu, 1 Insufficient memory. preemption: 0/3 nodes are available: 3
Preemption is not helpful for scheduling.
```

What these messages mean:

- `Insufficient cpu` or `Insufficient memory` — nodes lack the requested CPU or memory resources.
- `Preemption is not helpful for scheduling` — preempting existing pods will not free enough resources (for example, all pods have equal or higher priority).

Check available resources on the nodes:

```
d8 k describe nodes | grep -A 5 "Allocated resources"
```

Example output:

```
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests    Limits
  --------           --------    ------
  cpu                3800m (95%)  4200m (105%)
  memory             7Gi (87%)    8Gi (100%)
```

Possible solutions:

- Reduce `requests` in the pod manifest.
- Add resources to the cluster (new nodes or larger existing ones).
- Delete unused pods.

Practical diagnostics when preemption is impossible

If lower-priority pods are not preempted even though resources appear free, check events with reason `FailedPreemption`:

```
d8 k get events -A --field-selector reason=FailedPreemption --sort-
by='.metadata.creationTimestamp'
```

Example output:

```
NAMESPACE   LAST SEEN   TYPE      REASON             OBJECT
MESSAGE
default     30s         Warning   FailedPreemption   pod/high-priority-pod     no
preemption victims found for pod
```

The message `no preemption victims found for pod` usually means there are no lower-priority pods on the node that can be preempted for the new pod. For a walkthrough of this case, see Practical check when no suitable pods are available.

A pod may also fail to start for reasons unrelated to the priority class: for example, there are no suitable nodes because of taints and tolerations. In that case, events show `FailedScheduling` with messages such as `untolerated taint(s)`, not `FailedPreemption`.

For possible cluster-level actions, see Operations and diagnostics.

Practical check when no suitable pods are available

This example uses the cluster state after Demo of stateful application protection. Pod `mock-stateful-0` has priority `6000`. On the target node, the other `mock-stateful` pods have the same priority, `emergency-task` has a higher priority (`9000`), and system pods have an even higher priority. Therefore, the scheduler cannot preempt any of them to schedule `mock-stateful-0`: there are no lower-priority pods on the node.

If events contain the message `No preemption victims found for incoming pod`, check pod priorities on the node:

```
d8 k get pods --all-namespaces -o wide --field-selector spec.nodeName=worker-0 -o custom-
columns=NAME:.metadata.name,NODE:.spec.nodeName,PRIORITY:.spec.priority --sort-by=.spec.priority
```

Example output:

```
NAME                                                              NODE       PRIORITY
mock-stateful-5                                                   worker-0   6000
mock-stateful-2                                                   worker-0   6000
mock-stateful-3                                                   worker-0   6000
mock-stateful-4                                                   worker-0   6000
emergency-task                                                    worker-0   9000
multitenancy-manager-5968799d76-ktjgl                             worker-0
2000000000
csi-node-s82x4                                                    worker-0
2000001000
agent-wqzxq                                                       worker-0
2000001000
early-oom-6tkzg                                                   worker-0
2000001000
safe-agent-updater-ntrzq                                          worker-0
2000001000
kubernetes-api-proxy-worker-0                                     worker-0
2000001000
node-exporter-cddjm                                               worker-0
2000001000
oom-kills-exporter-pplfk                                          worker-0
2000001000
```

Check the exact reason why `mock-stateful-0` could not preempt other pods:

```
d8 k describe pod mock-stateful-0 | grep -A10 "Events:"
```

Example message from the pod events:

```
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  13s   default-scheduler  0/2 nodes are available:
preemption: 0/2 nodes are available: 1 No preemption victims found for incoming pod
```

Message	Meaning
`0/2 nodes are available`	The cluster has 2 nodes, but none can accommodate the pod.
`1 No preemption victims found for incoming pod`	On the node with insufficient memory, there are no lower-priority pods that can be preempted.

For cluster-level actions, see Operations and diagnostics.

Practical example “Pod limit per node”

Even when CPU and memory become available, the maximum number of pods per node can prevent a high-priority pod from starting.

Check the pod limit on the node:

```
d8 k describe node worker-0 | grep pods -A2
```

Example output:

```
Capacity:
  pods:  120
```

Check the current number of pods on the node:

```
d8 k get pods --all-namespaces -o wide | grep worker-0 | wc -l
```

Example output:

```
64
```

The node currently has 64 of 120 pods. There is still room.

Create a file `pod-filler.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-filler
spec:
  replicas: 110
  selector:
    matchLabels:
      app: filler
  template:
    metadata:
      labels:
        app: filler
    spec:
      priorityClassName: develop
      containers:
      - name: filler
        image: busybox
        command: ["sleep", "infinity"]
        resources:
          requests:
            cpu: "1m"
            memory: "5Mi"
```

Apply the manifest:

```
d8 k apply -f pod-filler.yaml
```

Wait until the Deployment fills the node:

```
d8 k get pods -l app=filler -o wide | grep worker-0 | wc -l
```

Create a file `high-priority-limit-pod.yaml` to run a high-priority pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: high-priority-limit-pod
spec:
  priorityClassName: production-high
  containers:
  - name: app
    image: nginx
    resources:
      requests:
        cpu: "100m"
        memory: "256Mi"
```

Apply the manifest:

```
d8 k apply -f high-priority-limit-pod.yaml
```

Check the status of the high-priority pod:

```
d8 k get pod high-priority-limit-pod
```

Example output:

```
NAME                       READY   STATUS    RESTARTS   AGE
high-priority-limit-pod    0/1     Pending   0          11s
```

Check the reason in the events:

```
d8 k describe pod high-priority-limit-pod | grep -A10 "Events:"
```

Example output:

```
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  11s   default-scheduler  0/2 nodes are available: 1 Too
many pods, 1 node(s) had untolerated taint(s).
```

Verify that no preemption occurred:

```
d8 k get events -A --field-selector reason=Preempted
```

Example output:

```
No resources found
```

The issue: the pod limit on the node has already been reached, so even a high-priority pod cannot start. Preempting existing pods does not help because the number of pods stays the same (the preempted pod is replaced by the new one).

Useful commands for monitoring priorities

Pod count by priority class:

```
d8 k get pods -A -o jsonpath='{range .items[*]}{.spec.priorityClassName}{"\n"}{end}' |
sort | uniq -c | sort -rn
```

Example output:

```
     34 cluster-medium
     30 cluster-low
     26 system-cluster-critical
     18 system-node-critical
      6 production-high
```

Preemption events across all namespaces:

```
d8 k get events -A --field-selector reason=Preempted -o custom-
columns=NAMESPACE:.metadata.namespace,POD:.involvedObject.name,MESSAGE:.message
```

Example output:

```
NAMESPACE          POD                                                  MESSAGE
d8-chrony          chrony-master-9wzbl                                  Preempted by
pod ac651aed-... on node master-0
d8-console         backend-58f9989c9d-4svjw                             Preempted by
pod ac651aed-... on node master-0
d8-monitoring      prometheus-main-0                                    Preempted by
pod 91f6e071-... on node worker-0
default            log-collector-dlxpv                                  Preempted by
pod 91f6e071-... on node worker-0
```

> **i** Events are retained for a limited time (usually about an hour). If preemption has not occurred recently, these commands may return nothing — repeat the preemption demonstration and run the commands immediately afterward.

Most frequently preempted pods:

```
d8 k get events -A --field-selector reason=Preempted -o jsonpath='{range .items[*]}
{.involvedObject.name}{"\n"}{end}' | sort | uniq -c | sort -rn | head -10
```

Example output:

```
      2 prometheus-main-0
      2 memcached-0
      1 user-api-77494dc777-jzp7p
      1 upmeter-dex-authenticator-7f54c8dfb4-wwv22
      1 upmeter-dex-authenticator-7f54c8dfb4-wqfn5
      1 upmeter-dex-authenticator-7f54c8dfb4-h6784
      1 upmeter-dex-authenticator-7f54c8dfb4-bxsvk
      1 upmeter-dex-authenticator-7f54c8dfb4-28fgt
      1 upmeter-agent-4chrw
      1 status-dex-authenticator-786c6cc554-mfsdw
```

> **i** **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Incoming traffic processing

Network Load Balancer (NLB) and Application Load Balancer (ALB) are used to provide external access to applications deployed in a cluster managed by Deckhouse Platform.

NLB features and purpose

NLB operates at the transport layer. It balances TCP and UDP traffic at the IP and port level.

Key advantages:

- High performance
- Minimal latency during traffic transmission
- Simple configuration

NLB is suitable for applications that use TCP/UDP protocols, such as databases.

ALB features and purpose

ALB operates at the application layer. It analyzes the contents of incoming requests (for example, HTTP headers, URL paths, cookies) and can route traffic based on them.

Advantages of ALB:

- Support for HTTP(S) protocols and gRPC
- Flexible routing (path-based, host-based)
- Ability to terminate SSL/TLS
- Integration with authentication and authorization mechanisms

ALB is suitable for web applications, APIs, and other services where intelligent routing and working with HTTP requests are important.

> **i** **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Utilizing Network Load Balancer (NLB)

NLB is provided by using Services of LoadBalancer type.

Examples of Service configuration

Shared IP address for multiple Services

To make Services use the same IP address, add the `network.deckhouse.io/load-balancer-shared-ip-key` annotation to them:

```
apiVersion: v1
kind: Service
metadata:
  name: dns-service-tcp
  namespace: default
  annotations:
    network.deckhouse.io/load-balancer-shared-ip-key: "key-to-share-1.2.3.4"
spec:
  type: LoadBalancer
  ports:
    - name: dnstcp
      protocol: TCP
      port: 53
      targetPort: 53
  selector:
    app: dns
---
apiVersion: v1
kind: Service
metadata:
  name: dns-service-udp
  namespace: default
  annotations:
    network.deckhouse.io/load-balancer-shared-ip-key: "key-to-share-1.2.3.4"
spec:
  type: LoadBalancer
  ports:
    - name: dnsudp
      protocol: UDP
      port: 53
      targetPort: 53
  selector:
    app: dns
```

Forcing an IP address assignment

To enforce an IP address for a Service, add the `network.deckhouse.io/load-balancer-ips`

annotation:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  annotations:
    network.deckhouse.io/load-balancer-ips: 192.168.217.217
spec:
  ports:
    - port: 80
      targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

Assigning an IPAddressPool (BGP mode)

In BGP LoadBalancer mode, an IP address can be allocated from a specific address pool using the `network.deckhouse.io/load-balancer-pool` annotation. For L2 LoadBalancer mode, you need to use the MetalLoadBalancerClass configuration.

Example:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  annotations:
    network.deckhouse.io/load-balancer-pool: production-public-ips
spec:
  ports:
    - port: 80
      targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.

The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Utilizing Application Load Balancer (ALB)

Application deployment and application-level traffic balancing in Deckhouse Platform (DP) can be performed using the following tools:

- Ingress NGINX Controller (`ingress-nginx` module).
- Kubernetes Gateway API (`alb` module).
- Istio (`istio` module).

Comparison of ALB options

The following sections describe each ALB option and typical scenarios for using it.

Ingress NGINX

ALB based on the Ingress NGINX Controller uses the nginx web server and is implemented by the `ingress-nginx` module. This option is suitable for:

- Basic traffic routing based on domains or URLs.

- Using SSL/TLS to secure traffic.

Kubernetes Gateway API

ALB is implemented using the Kubernetes Gateway API via the `alb` module. Gateways run on Envoy Proxy. Reception and routing are described using standard API objects (Gateway, ListenerSet, HTTPRoute, and, if necessary, GRPCRoute, TLSRoute, TCPRoute, UDPRoute, BackendTLSPolicy). The controller deploys the ingress infrastructure and validates the configuration to prevent conflicting handlers.

The Gateway API model separates responsibilities between the cluster administrator (ClusterALBInstance), the namespace administrator (ALBInstance and ListenerSet — hostname, TLS, ports), and application developers (HTTPRoute and other route objects).

Use this option for:

- Publishing applications using the Gateway API model instead of the classic Ingress.
- A cluster-wide entry point or a separate gateway for an application or team within your namespace.
- HTTP/HTTPS, gRPC, TCP, UDP, and TLS termination or passthrough.
- Per-route WAF or an Istio sidecar on the gateway proxy.
- GeoIP and OpenTelemetry on the gateway (configured by an administrator — “Using GeoIP and GeoLite2” and “Configuring OpenTelemetry tracing”).
- Route parameters not included in the specification, via HTTPRoute annotations.

For a comparison with `ingress-nginx` and terminology notes, read “Comparison of the ingress-nginx and alb modules”.

Istio

An ALB based on the `istio` module supports traffic management in a service mesh. Use an Istio-based ALB for:

- Routing for canary deployment and similar scenarios.
- Traffic distribution between application versions and microservices.
- Mutual TLS (mTLS) for encrypting traffic between Pods.
- Request tracing.

How to tell what is available in the cluster

Before publishing an application, check which ALB mechanisms are enabled and configured:

1. Verify that the required module is enabled — the `STATE` column should show `Enabled`:

   ```
   d8 k get moduleconfig ingress-nginx alb istio
   ```

2. For Ingress NGINX — list IngressNginxController resources and note the IngressClass name:

   ```
   d8 k get ingressnginxcontrollers
   d8 k get ingressclass
   ```

3. For Gateway API — verify that ClusterALBInstance or ALBInstance exists and is in the `Ready` state, then find the managed Gateway and ListenerSet objects:

   ```
   d8 k get clusteralbinstances,albinstances --all-namespaces
   d8 k get gateway,listenerset --all-namespaces
   ```

4. For Istio — check IngressIstioController and the ingress gateway class label provided by the cluster administrator:

   ```
   d8 k get ingressistiocontrollers
   ```

Ask the cluster administrator for the IngressClass, Gateway name and namespace, or Istio ingress class to use in application manifests.

Next steps

Once the administrator has configured the infrastructure, you can publish the application:

- using the Kubernetes Gateway API (`alb` module);

- using the Ingress NGINX Controller (the `ingress-nginx` module);
- using Istio (the `istio` module).

Additional resources

- Publishing applications using the Kubernetes Gateway API
- Publishing applications using the Ingress NGINX Controller
- Publishing applications using Istio
- Incoming traffic balancing
- Module `alb` documentation
- Module `ingress-nginx` documentation
- Module `istio` documentation

> **i** **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Publishing applications using the Kubernetes Gateway API

Publishing applications using the Kubernetes Gateway API

Applications can be published through a cluster-wide gateway (a ClusterALBInstance created by the cluster administrator) or through a dedicated gateway in the application namespace (an ALBInstance).

Creating the managed Gateway (ClusterALBInstance or ALBInstance, inlets, enabling the module) is an administrator task. Infrastructure setup is described in “Enabling the module and creating a Gateway”.

This scenario assumes that the ClusterALBInstance or ALBInstance object has already been created and has reached the `Ready` state. Ask the administrator for the name and namespace of the managed Gateway, or read the Gateway name from the instance’s `status.gateway` field:

```
d8 k get clusteralbinstance <CLUSTER_ALB_INSTANCE_NAME> \
  -o jsonpath='{.status.gateway}{"\n"}'
d8 k -n <NAMESPACE> get albinstance <ALB_INSTANCE_NAME> \
  -o jsonpath='{.status.gateway}{"\n"}'
```

For ClusterALBInstance, the managed Gateway usually lives in the `d8-alb` namespace. For ALBInstance, it lives in the same namespace as the ALBInstance object. Status field descriptions are in `status`.

The namespace administrator creates a ListenerSet attached to that Gateway (`spec.parentRef`). Application developers create HTTPRoute objects that attach to the ListenerSet.

The `d8-http` and `d8-https` listeners serve system tasks such as gateway availability checks or cert-manager HTTP-01 challenges. Do not attach application routes to them — use a ListenerSet instead.

Publishing an application with ListenerSet and HTTPRoute

Example of a ListenerSet and HTTPRoute for publishing an application through a managed Gateway:

```
apiVersion: gateway.networking.k8s.io/v1
kind: ListenerSet
metadata:
  name: app-listeners
  namespace: prod
spec:
  parentRef:
    name: public-gw  # The name of the Gateway object from the ClusterALBInstance
status, provided by the administrator.
    namespace: d8-alb
  listeners:
```

```
    - name: app-http
      port: 80 # ListenerSet HTTP listeners use port 80. This is not the HostPort
inlet httpPort on the node.
      protocol: HTTP
      hostname: app.example.com
    - name: app-https
      port: 443 # ListenerSet HTTPS listeners use port 443. This is not the HostPort
inlet httpsPort on the node.
      protocol: HTTPS
      hostname: app.example.com
      tls:
        mode: Terminate
        certificateRefs:
          - name: app-tls   # Secret with the TLS certificate (issue with cert-manager
or provide manually).
            namespace: prod
---
# Route for HTTP traffic
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: app-http-route
  namespace: prod
spec:
  parentRefs:
    - name: app-listeners # ListenerSet name.
      namespace: prod
      kind: ListenerSet
      group: gateway.networking.k8s.io
      sectionName: app-http
      port: 80 # ListenerSet HTTP listeners use port 80. This is not the HostPort
inlet httpPort on the node.
  hostnames:
    - app.example.com
  rules:
    - backendRefs:
        - name: app-svc # Reference to the internal load balancer of the application.
          port: 8080
---
# Route for HTTPS traffic
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: app-https-route
  namespace: prod
spec:
  parentRefs:
    - name: app-listeners # ListenerSet name.
      namespace: prod
      kind: ListenerSet
      group: gateway.networking.k8s.io
      sectionName: app-https
      port: 443 # ListenerSet HTTPS listeners use port 443. This is not the HostPort
inlet httpsPort on the node.
  hostnames:
    - app.example.com
  rules:
    - backendRefs:
        - name: app-svc # Reference to the internal load balancer of the application.
          port: 8080
```

Checking publication

After you apply the ListenerSet and HTTPRoute, check the object status:

```
d8 k -n <NAMESPACE> get listenerset
d8 k -n <NAMESPACE> describe listenerset <LISTENERSET_NAME>
d8 k -n <NAMESPACE> get httproute
d8 k -n <NAMESPACE> describe httproute <HTTPROUTE_NAME>
```

Confirm the following:

- ListenerSet listeners are `Programmed` or `Accepted`.
- The HTTPRoute `parentRefs` point to the intended ListenerSet and the `Accepted` conditions are true.
- The entry-point address and DNS for the hostname match what the administrator provided (for the `LoadBalancer` inlet, the address usually comes from a Service in the `d8-alb` namespace).

Check application reachability from a client. Substitute the entry-point address and hostname (a successful response is `200` or another status code expected from the application):

```
curl -vk \
  --resolve app.example.com:443:<ENTRY_POINT_ADDRESS> \
  https://app.example.com/
```

If the route is not accepted, check the Gateway name from the instance `status`, the ListenerSet namespace, and the port and `sectionName` in `parentRefs`. Hostname and port conflicts with other ListenerSet objects show up as a condition on the conflicting listener’s status, visible in the `describe listenerset` output from the previous step.

Working with GRPCRoute, TLSRoute, TCPRoute, and UDPRoute objects

Besides HTTPRoute, you can use GRPCRoute (gRPC traffic), TLSRoute (TLS passthrough), and TCPRoute/UDPRoute (arbitrary TCP and UDP traffic) to publish applications.

GRPCRoute for gRPC traffic

The GRPCRoute object is intended for gRPC traffic. For it, create the ListenerSet object with an HTTPS listener, then add the GRPCRoute object:

```
apiVersion: gateway.networking.k8s.io/v1
kind: ListenerSet
metadata:
  name: grpc-listeners
  namespace: prod
spec:
  parentRef:
    name: app-gw # The name of the Gateway object from the ALBInstance status.
    namespace: prod
  listeners:
    - name: grpc-https
      port: 443
      protocol: HTTPS
      hostname: grpc.example.com
      tls:
        mode: Terminate
        certificateRefs:
          - name: grpc-tls  # Reference to the Secret with the TLS certificate.
            namespace: prod
---
apiVersion: gateway.networking.k8s.io/v1
kind: GRPCRoute
metadata:
  name: grpc-route
  namespace: prod
spec:
  parentRefs:
    - name: grpc-listeners # ListenerSet name.
      namespace: prod
      kind: ListenerSet
      group: gateway.networking.k8s.io
      sectionName: grpc-https
      port: 443
  hostnames:
    - grpc.example.com
  rules:
    - backendRefs:
        - name: grpc-svc # Name of the gRPC backend Service.
          port: 9090
```

TLS passthrough with TLSRoute

For TLS passthrough, when traffic must be decrypted on the application side, either a TLS listener or an HTTPS listener can be used.

Additional ports are set in `spec.inlet.additionalPorts` on the object that owns the gateway:

- For a cluster-wide Gateway — on ClusterALBInstance (configured by the cluster administrator).
- For a namespaced gateway — on ALBInstance (configured by the namespace administrator, or by you if you can edit the object).

For a ClusterALBInstance example, see “Opening an additional TCP/UDP port on the cluster-wide gateway”. The example below uses ALBInstance.

- TLS listener
- HTTPS listener

Because the TLS listener in this example uses an additional port, ask an administrator to add `additionalPorts` to the object that owns the gateway (ClusterALBInstance or ALBInstance), or add it yourself if you have the required permissions:

```
apiVersion: network.deckhouse.io/v1alpha1
kind: ALBInstance
metadata:
  name: app-gw
  namespace: prod
spec:
  gatewayName: app-gw
  inlet:
    type: LoadBalancer
    additionalPorts:
      - port: 8443   # An additional TCP port to accept TLS traffic.
        protocol: TCP
```

Next, configure ListenerSet and TLSRoute objects:

```
apiVersion: gateway.networking.k8s.io/v1
kind: ListenerSet
metadata:
  name: tls-pass-listeners
  namespace: prod
spec:
  parentRef:
    name: app-gw # The name of the Gateway object from the ALBInstance status.
    namespace: prod
  listeners:
    - name: tls-pass
      port: 8443         # In this case 8443 port is used for TLS.
      protocol: TLS
      hostname: pass.example.com
      tls:
        mode: Passthrough  # TLS passthrough mode is set explicitly.
---
apiVersion: gateway.networking.k8s.io/v1
kind: TLSRoute
metadata:
  name: tls-pass-route
  namespace: prod
spec:
  parentRefs:
    - name: tls-pass-listeners # ListenerSet name.
      namespace: prod
      kind: ListenerSet
      group: gateway.networking.k8s.io
      sectionName: tls-pass
      port: 8443         # In this case 8443 port is used for TLS.
  hostnames:
    - pass.example.com
  rules:
    - backendRefs:
        - name: tls-pass-svc  # Reference to the internal load balancer of the
application.
          port: 8443
```

The HTTPS listener variant is useful when the standard handler on port `443` should be used: no extra port needs to be opened for TLS passthrough.

```
apiVersion: gateway.networking.k8s.io/v1
kind: ListenerSet
metadata:
  name: https-pass-listeners
  namespace: prod
spec:
  parentRef:
    name: app-gw # The name of the Gateway object from the ALBInstance status.
    namespace: prod
  listeners:
    - name: https-pass
      port: 443 # In this case 443 (HTTPS) port is reused for TLS.
      protocol: HTTPS
      hostname: pass.example.com
      tls:
        mode: Passthrough  # TLS passthrough mode is set explicitly.
---
apiVersion: gateway.networking.k8s.io/v1
kind: TLSRoute
metadata:
  name: https-pass-route
  namespace: prod
spec:
  parentRefs:
    - name: https-pass-listeners # ListenerSet name.
      namespace: prod
      kind: ListenerSet
      group: gateway.networking.k8s.io
      sectionName: https-pass
      port: 443 # In this case 443 (HTTPS) port is reused for TLS.
  hostnames:
    - pass.example.com
  rules:
    - backendRefs:
        - name: tls-pass-svc # Reference to the internal load balancer of the
application.
          port: 8443
```

TLS termination at the gateway with TCPRoute

If TLS must be terminated on the gateway and then forwarded to the backend as a TCP stream (for example, when the application accepts decrypted TCP rather than HTTPS), create a ListenerSet object with a TLS listener in `Terminate` mode, then attach a TCPRoute object:

```
apiVersion: gateway.networking.k8s.io/v1
kind: ListenerSet
metadata:
  name: tls-term-listeners
  namespace: prod
spec:
  parentRef:
    name: app-gw # The name of the Gateway object from the ALBInstance status.
    namespace: prod
  listeners:
    - name: tls-term
      port: 443 # In this case 443 (HTTPS) port is reused for TLS.
      protocol: TLS
      hostname: term.example.com
      tls:
        mode: Terminate
        certificateRefs:
          - name: term-tls  # Reference to the Secret with the TLS certificate.
            namespace: prod
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TCPRoute
metadata:
  name: tls-term-route
  namespace: prod
spec:
  parentRefs:
    - name: tls-term-listeners # ListenerSet name.
      namespace: prod
      kind: ListenerSet
      group: gateway.networking.k8s.io
      sectionName: tls-term
      port: 443 # In this case 443 (HTTPS) port is reused for TLS.
  rules:
    - backendRefs:
        - name: tcp-svc # Reference to the internal load balancer of the application.
          port: 8080
```

Additional TCP and UDP ports

For TCP and UDP ports from `additionalPorts`, attach the route directly to the managed Gateway listener instead of creating a separate ListenerSet. Otherwise the controller rejects the configuration because of overlapping handlers.

Set the additional ports on the object that owns the gateway: ClusterALBInstance for a cluster-wide Gateway, or ALBInstance for a namespaced gateway. For ClusterALBInstance, use the example in “Opening an additional TCP/UDP port on the cluster-wide gateway”.

- TCP
- UDP

To publish a TCP service, ask an administrator to expose an additional TCP port on the object that owns the gateway (ClusterALBInstance or ALBInstance), or expose it yourself if you have the required permissions:

```
apiVersion: network.deckhouse.io/v1alpha1
kind: ALBInstance
metadata:
  name: app-gw
  namespace: prod
spec:
  gatewayName: app-gw
  inlet:
    type: LoadBalancer
    additionalPorts:
      - port: 9000
        protocol: TCP
```

The `alb` module controller creates the TCP listener on the managed Gateway automatically from `spec.inlet.additionalPorts`. Attach the TCPRoute to that listener:

```
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TCPRoute
metadata:
  name: tcp-route
  namespace: prod
spec:
  parentRefs:
    - name: app-gw # The name of the Gateway object from the ALBInstance status.
      namespace: prod
      kind: Gateway
      group: gateway.networking.k8s.io
      sectionName: tcp-port-9000
      port: 9000
  rules:
    - backendRefs:
        - name: tcp-svc # Reference to the internal load balancer of the application.
          port: 9000
```

To publish a UDP service, ask an administrator to expose an additional UDP port on the object that owns the gateway (ClusterALBInstance or ALBInstance), or expose it yourself if you have the required permissions:

```
apiVersion: network.deckhouse.io/v1alpha1
kind: ALBInstance
metadata:
  name: app-gw
  namespace: prod
spec:
  gatewayName: app-gw
  inlet:
    type: LoadBalancer
    additionalPorts:
      - port: 5353
        protocol: UDP
```

The `alb` module controller creates the UDP listener on the managed Gateway automatically from `spec.inlet.additionalPorts`. Attach the UDPRoute to that listener:

```
apiVersion: gateway.networking.k8s.io/v1
kind: UDPRoute
metadata:
  name: udp-route
  namespace: prod
spec:
  parentRefs:
    - name: app-gw # The name of the Gateway object from the ALBInstance status.
      namespace: prod
      kind: Gateway
      group: gateway.networking.k8s.io
      sectionName: udp-port-5353
      port: 5353
  rules:
    - backendRefs:
        - name: udp-svc # Reference to the internal load balancer of the application.
          port: 5353
```

Publishing the app via a different gateway

If an application needs to move to another managed Gateway object, change the route attachment in stages:

1. Obtain a new managed Gateway from the cluster administrator (a new ClusterALBInstance or ALBInstance), so that the controller creates a new Gateway object. Creating managed Gateways is described in “Enabling the module and creating a Gateway”.
2. Create a ListenerSet object with the same hostnames, ports, and TLS settings. The new Gateway object must be specified in `spec.parentRef`.
3. Add one more `parentRefs` entry to the existing HTTPRoute object, pointing to the new ListenerSet object.
4. Verify traffic through the new gateway path.
5. After verification, remove the reference to the obsolete ListenerSet object from `parentRefs` of the HTTPRoute object.

Linking routes in one namespace to a ListenerSet in another

By default, the Gateway API does not allow routes to reference objects in other namespaces — this must be allowed explicitly. If an HTTPRoute object is created in one namespace and must be attached to a ListenerSet in another namespace, add a ReferenceGrant in the namespace of the target ListenerSet.

The example below shows a shared ListenerSet in namespace `shared-gw`, an application HTTPRoute in namespace `prod`, and a ReferenceGrant that allows this attachment:

```
apiVersion: gateway.networking.k8s.io/v1
kind: ListenerSet
metadata:
  name: shared-listeners
  namespace: shared-gw
spec:
  parentRef:
    name: public-gw
    namespace: d8-alb
  listeners:
    - name: app-https
      port: 443
      protocol: HTTPS
      hostname: app.example.com
      tls:
        mode: Terminate
        certificateRefs:
          - name: app-tls
            namespace: shared-gw
---
apiVersion: gateway.networking.k8s.io/v1
kind: ReferenceGrant
metadata:
  name: allow-prod-httproute-to-shared-listeners
  namespace: shared-gw
spec:
  from:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      namespace: prod
  to:
    - group: gateway.networking.k8s.io
      kind: ListenerSet
      name: shared-listeners
---
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: app-route
  namespace: prod
spec:
  parentRefs:
    - name: shared-listeners
      namespace: shared-gw
      kind: ListenerSet
      group: gateway.networking.k8s.io
      sectionName: app-https
      port: 443
  hostnames:
    - app.example.com
  rules:
    - backendRefs:
        - name: app-svc
          port: 8080
```

Configuring TLS parameters with BackendTLSPolicy

If traffic from the gateway to the backend must use TLS, create a BackendTLSPolicy in the namespace of the backend Service. The example below shows an HTTPRoute, a backend Service with a named port, a ConfigMap with a CA bundle, and a BackendTLSPolicy that configures TLS validation for that backend:

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: app-route
  namespace: prod
spec:
  parentRefs:
    - name: app-listeners
      namespace: prod
      kind: ListenerSet
      group: gateway.networking.k8s.io
      sectionName: app-https
      port: 443
  hostnames:
    - app.example.com
  rules:
    - backendRefs:
        - name: app-svc
          port: 8443
---
apiVersion: v1
kind: Service
metadata:
  name: app-svc
  namespace: prod
spec:
  selector:
    app: app
  ports:
    - name: https
      port: 8443
      targetPort: 8443
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-backend-ca
  namespace: prod
data:
  ca.crt: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
---
apiVersion: gateway.networking.k8s.io/v1
kind: BackendTLSPolicy
metadata:
  name: app-svc-tls
  namespace: prod
spec:
  targetRefs:
    - group: ""
      kind: Service
      name: app-svc
      sectionName: https
  validation:
    hostname: app.internal.example.com
    caCertificateRefs:
      - group: ""
        kind: ConfigMap
        name: app-backend-ca
```

GeoIP and OpenTelemetry tracing are configured by the administrator on ClusterALBInstance or ALBInstance, following “Using GeoIP and GeoLite2” and “Configuring OpenTelemetry tracing”.

Supported HTTPRoute annotations

Because the current Gateway API specification does not yet cover all features required for a Deckhouse Platform (DP) cluster, the module provides HTTPRoute annotations for the missing options. The controller reads these keys from `HTTPRoute.metadata.annotations`.

Annotation	Description
`alb.network.deckhouse.io/tls-disable-protocol`	Disables the listener protocol for the route with the specified hostname (for example, value `http2`). This may be required in rare cases when a shared certificate with several DNS names is used together with request redirection
`alb.network.deckhouse.io/whitelist-source-range`	A comma-separated list of subnets in CIDR format. Sets an IP filter at route level and overrides the global whitelist (for example, `10.1.1.10/32, 10.2.2.2/32`)
`alb.network.deckhouse.io/response-headers-to-add`	JSON object with additional response headers (for example, `{"Strict-Transport-Security": "max-age=31536000; includeSubDomains"}`)
`alb.network.deckhouse.io/session-affinity`	JSON for session affinity with cookie mode (`mode`, `path`, `cookieName`, `ttl`, etc.); not every field is required (for example, `{"mode": "cookie", "path": "/path", "cookieName": "mycookie", "ttl": 0}`)
`alb.network.deckhouse.io/hash-key`	Consistent hashing for Service backends of the HTTPRoute object (for example, `source-ip`)
`alb.network.deckhouse.io/service-upstream`	`"true"`: traffic to the upstream goes through the corresponding Service object instead of directly to pods
`alb.network.deckhouse.io/basic-auth-secret`	`namespace/secret` with htpasswd data for HTTP basic auth on this route
`alb.network.deckhouse.io/satisfy`	`all` or `any`: defines whether both checks must be satisfied (whitelist and basic-auth) or only one of them (default `all`)
`alb.network.deckhouse.io/auth-url`	Defines the URL of the external authentication service
`alb.network.deckhouse.io/auth-signin`	Defines the redirect URL for authentication when `401` is returned by external authentication
`alb.network.deckhouse.io/auth-response-headers`	Comma-separated list: additional headers from the authentication service response to pass upstream (on top of the standard allowlist)
`alb.network.deckhouse.io/mod-security`	JSON configuration for the per-route ModSecurity/Coraza WAF
`alb.network.deckhouse.io/rewrite-target`	Allows rewriting paths for rules with `RegularExpression` type by using regex capture groups (for example, `/my-path/\1`)
`alb.network.deckhouse.io/buffer-max-request-bytes`	Defines the buffer size that may be used when requests are buffered; the value is in bytes (integer). By default Envoy Proxy does not buffer requests
`alb.network.deckhouse.io/limit-rps`	RPS limit for a route

Annotation	Description
`alb.network.deckhouse.io/backend-tls-settings`	For example, `{"mode": "SIMPLE", "insecureSkipVerify": true, "clientCertificate": "", "privateKey": "", "caCertificates": "", "sni": "example.com", "secret": "<NAMESPACE>/<SECRET_NAME>"}`; allows explicit configuration of TLS connection parameters to the upstream. `<NAMESPACE>` — Secret namespace; `<SECRET_NAME>` — Secret name
`alb.network.deckhouse.io/idle-timeout`	Sets the per-route Envoy `idle timeout`, in seconds. Similar to `ingress-nginx proxy-read-timeout`/`proxy-send-timeout`; this is an inactivity timeout, not a total request timeout
`alb.network.deckhouse.io/proxy-buffer-size`	Sets the maximum size of response headers when configured on an upstream cluster; if exceeded, Envoy returns `503`. Similar to `nginx.ingress.kubernetes.io/proxy-buffer-size`

Publishing an application when the Istio sidecar is enabled

When the Istio sidecar is enabled for the gateway proxy through the `istioSidecar` parameter of ALBInstance or ClusterALBInstance, traffic to the backend must reach the sidecar through the Service and carry the Service FQDN in the `Host` header.

Configure the HTTPRoute as follows:

- Add the `alb.network.deckhouse.io/service-upstream: "true"` annotation so that traffic goes through the Service object instead of directly to pods. This is the equivalent of the `ingress-nginx nginx.ingress.kubernetes.io/service-upstream: "true"` annotation.
- Add a `URLRewrite` filter that sets `hostname` to the FQDN of the backend Service object. This replaces the `ingress-nginx nginx.ingress.kubernetes.io/upstream-vhost` annotation.

Example of an HTTPRoute configured for a gateway with an Istio sidecar:

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: myservice
  namespace: myns
  annotations:
    alb.network.deckhouse.io/service-upstream: "true" # Traffic goes through the
Service object so the Istio sidecar can process it.
spec:
  parentRefs:
    - name: app-listeners # ListenerSet name.
      namespace: myns
      kind: ListenerSet
      group: gateway.networking.k8s.io
      sectionName: app-https
      port: 443
  hostnames:
    - myservice.example.com
  rules:
    - filters:
        - type: URLRewrite
          urlRewrite:
            hostname: myservice.myns.svc # FQDN of the backend Service object, so the
sidecar identifies the destination.
      backendRefs:
        - name: myservice
          port: 80
```

WAF on HTTPRoute

The `alb.network.deckhouse.io/mod-security` annotation enables the ModSecurity/Coraza WAF (Web Application Firewall) for a specific HTTPRoute. The WAF is configured per route and does not affect other routes unless the same annotation is added there.

Supported annotation fields:

Field	Description
`mode`	WAF engine mode: `on`, `off`, or any other value for `DetectionOnly`
`preset`	Optional ruleset. Currently only `owasp-crs` is supported. If the field is omitted, no ruleset is loaded
`paranoiaLevel`	Optional CRS paranoia level from `1` to `4`. Applied only when `preset` is `owasp-crs`
`configRef.namespace`	Optional namespace of the ConfigMap with custom rules. Defaults to the namespace of the HTTPRoute
`configRef.name`	Name of the ConfigMap with custom rules
`configRef.key`	Optional key in the ConfigMap. If omitted, all keys are read in sorted order
`directives`	Optional inline list of ModSecurity/Coraza directives appended after the ruleset and ConfigMap rules

Directive order:

1. Base directives shipped with the module (`@coraza.conf`, `SecRuleEngine`, `SecResponseBodyAccess Off`).
2. Rules from the ruleset in `preset`.
3. Rules from `configRef`.
4. Directives from the annotation’s `directives` field.

Directives from the annotation are applied last, so they can override the ruleset or ConfigMap rules.

The examples below show the `alb.network.deckhouse.io/mod-security` annotation on an HTTPRoute.

- mode: on
- OWASP CRS
- ConfigMap

Minimal configuration: WAF enabled without a ruleset or custom directives.

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: app
  namespace: prod
  annotations:
    alb.network.deckhouse.io/mod-security: |
      {
        "mode": "on"
      }
spec:
  hostnames:
    - app.example.com
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: ListenerSet
      name: app-listeners
      namespace: prod
      sectionName: app-https
      port: 443
  rules:
    - matches:
        - path:
            type: PathPrefix
            value: /
      backendRefs:
        - name: app-svc
          port: 8080
```

WAF with the OWASP CRS ruleset and a `paranoiaLevel` value:

```
metadata:
  annotations:
    alb.network.deckhouse.io/mod-security: |
      {
        "mode": "on",
        "preset": "owasp-crs",
        "paranoiaLevel": 1
      }
```

WAF with OWASP CRS, custom rules from a ConfigMap, and extra directives in the annotation:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: waf-rules
  namespace: prod
data:
  rules.conf: |
    SecRule ARGS:test "@streq block" \
      "id:1000001,phase:2,deny,status:403,msg:'test waf block'"
---
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: app
  namespace: prod
  annotations:
    alb.network.deckhouse.io/mod-security: |
      {
        "mode": "on",
        "preset": "owasp-crs",
        "paranoiaLevel": 1,
        "configRef": {
          "name": "waf-rules",
          "key": "rules.conf"
        },
        "directives": [
          "SecResponseBodyAccess Off"
        ]
      }
```

Rule syntax reference:

- Coraza syntax and `SecRule` format
- ModSecurity variables reference
- ModSecurity operators reference
- ModSecurity `SecRuleEngine` and related directives

Current WAF notes and limitations:

- Only the `owasp-crs` ruleset is supported.
- `paranoiaLevel` is ignored when `preset` is omitted or differs from `owasp-crs`.
- Valid `paranoiaLevel` values are `1`–`4`. In practice it is recommended to start with `1`.
- The WAF currently inspects only incoming requests to the application and can block such requests when rules match. Responses sent back to the client are not inspected.
- Rules from ConfigMap values may be multiline: lines ending with `\` are joined automatically.

Additional resources

- Utilizing Application Load Balancer (ALB)
- Module `alb` documentation

> **i** **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Publishing applications using the Ingress NGINX Controller

Publishing applications using the Ingress NGINX Controller

To publish applications, the cluster administrator must create an Ingress controller. The setup procedure is described in “Load balancing configuration examples”. Ask the administrator for the controller `ingressClass` name and specify it in the Ingress resource that routes traffic to your application.

- HTTP
- HTTPS

Example of a basic Ingress for publishing an application over HTTP:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  namespace: prod
spec:
  ingressClassName: nginx # IngressClass name provided by the cluster administrator.
  rules:
  - host: app.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: productpage
            port:
              number: 80
```

For HTTPS, reference a Secret with the certificate in the `tls` section. The Secret must exist in the same namespace as the Ingress. The certificate can be issued by `cert-manager` or created by an administrator.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  namespace: prod
spec:
  ingressClassName: nginx
  tls:
    - hosts:
        - app.example.com
      secretName: application-tls # Secret with tls.crt and tls.key in the prod
namespace.
  rules:
  - host: app.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: productpage
            port:
              number: 80
```

Additional routing options are set with Ingress annotations. The full list of supported annotations is in the `ingress-nginx` module description. Common examples:

- `nginx.ingress.kubernetes.io/rewrite-target` — path rewrite.
- `nginx.ingress.kubernetes.io/whitelist-source-range` — allowed CIDR list.
- `nginx.ingress.kubernetes.io/backend-protocol: "GRPC"` — backend protocol for gRPC when the Service port name does not indicate gRPC.

Verification

After you apply the Ingress, check the object and the controller address:

```
d8 k -n <NAMESPACE> get ingress
d8 k -n <NAMESPACE> describe ingress <INGRESS_NAME>
```

Confirm that the Ingress status includes an address and that DNS for `host` points to the controller entry point provided by the administrator.

Check application reachability, substituting the entry-point address (a successful response is `200` or another status code expected from the application):

```
curl -vk \
  --resolve app.example.com:443:<ENTRY_POINT_ADDRESS> \
  https://app.example.com/
```

gRPC load balancing

This section applies to gRPC publishing through the Ingress NGINX Controller (`ingress-nginx`). For Gateway API, use GRPCRoute.

For automatic gRPC service load balancing behind Ingress NGINX, assign a name with the prefix or value `grpc` to the port in the corresponding Service object. If the port name is different, add the `nginx.ingress.kubernetes.io/backend-protocol: "GRPC"` annotation to the Ingress.

Additional resources

- Module `ingress-nginx` documentation

> **i** **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Publishing applications using Istio

Publishing applications using Istio

Application publishing with Istio is configured in two layers. The cluster administrator deploys the IngressIstioController (and related infrastructure) as described in “Istio Ingress Gateway”. Application developers create Gateway and VirtualService resources as shown below.

When deploying an application using Istio, choose one of the following options: “Publishing applications using Ingress NGINX” or “Publishing applications using Istio Ingress Gateway resource”.

Publishing applications using Ingress NGINX

To publish an application using Ingress NGINX, the cluster administrator must configure the Ingress controller by adding an Istio sidecar to it.

To publish an application, prepare an Ingress resource that references a Service. Set `ingressClassName` to the controller with the Istio sidecar (the administrator provides the value). Required annotations for the Ingress resource:

- `nginx.ingress.kubernetes.io/service-upstream: "true"`: With this annotation, the Ingress controller will send requests to the Service’s ClusterIP (from the Service CIDR range) instead of sending them directly to the application’s Pods. The `istio-proxy` sidecar container intercepts traffic only toward the Service CIDR range. All other requests are sent directly.
- `nginx.ingress.kubernetes.io/upstream-vhost: productpage.bookinfo.svc`: With this annotation, the sidecar can identify the application service the request is intended for.

Examples:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: productpage
  namespace: bookinfo
  annotations:
    # Enables proxying traffic to the Service's ClusterIP via nginx instead of
directly to Pod IPs.
    nginx.ingress.kubernetes.io/service-upstream: "true"
    # In Istio, all routing is based on the `Host:` request header.
    # This avoids the need to inform Istio about the existence of the external domain
`productpage.example.com`;
    # the internal domain known to Istio is used instead.
    nginx.ingress.kubernetes.io/upstream-vhost: productpage.bookinfo.svc
spec:
  ingressClassName: nginx # IngressClass name of the controller with the Istio
sidecar.
  rules:
    - host: productpage.example.com
      http:
        paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: productpage
              port:
                number: 9080
```

```
apiVersion: v1
kind: Service
metadata:
  name: productpage
  namespace: bookinfo
spec:
  ports:
  - name: http
    port: 9080
  selector:
    app: productpage
  type: ClusterIP
```

Publishing applications using Istio Ingress Gateway resource

To publish an application using the Istio Ingress Gateway, the cluster administrator must create an IngressIstioController resource.

To publish an application using the Istio Ingress Gateway resource, create a Gateway. In `spec.selector`, specify the label that references `ingressGatewayClass` and the Secret name provided by the cluster administrator:

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: gateway-app
  namespace: app-ns
spec:
  selector:
    # Value of ingressGatewayClass provided by the cluster administrator.
    istio.deckhouse.io/ingress-gateway-class: istio-hp
  servers:
    - port:
        # Standard template for using the HTTP protocol.
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - app.example.com
    - port:
        # Standard template for using the HTTPS protocol.
        number: 443
        name: https
        protocol: HTTPS
      tls:
        mode: SIMPLE
        # Secret with the certificate and key created by the administrator in the d8-
ingress-istio namespace.
        # Supported Secret formats: https://istio.io/latest/docs/tasks/traffic-
management/ingress/secure-ingress/#key-formats.
        credentialName: app-tls-secret
      hosts:
        - app.example.com
```

Then define routing rules with a VirtualService that links the Gateway to the service it serves:

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: vs-app
  namespace: app-ns
spec:
  gateways:
    - gateway-app # Name of the Gateway resource created above.
  hosts:
    - app.example.com
  http:
    - route:
        - destination:
            host: app-svc # Name of the service that should receive traffic.
```

Canary deployment with VirtualService

For a general overview of canary in Deckhouse Platform (DP), see “Canary deployment”. The example below uses VirtualService and DestinationRule.

To shift traffic gradually between application versions, use a DestinationRule with subsets and weights in the VirtualService. The example below sends 90% of traffic to the stable version and 10% to canary:

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: app-svc
  namespace: app-ns
spec:
  host: app-svc
  subsets:
    - name: stable
      labels:
        version: v1
    - name: canary
      labels:
        version: v2
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: vs-app
  namespace: app-ns
spec:
  gateways:
    - gateway-app
  hosts:
    - app.example.com
  http:
    - route:
        - destination:
            host: app-svc
            subset: stable
          weight: 90
        - destination:
            host: app-svc
            subset: canary
          weight: 10
```

Pods of each version must have the `version: v1` and `version: v2` labels that match the DestinationRule subsets. Before changing weights, confirm that both versions are selected by the `app-svc` Service.

After you apply the manifests, send a few requests to `app.example.com` and check that traffic is split roughly 90/10. To identify which version responded, use whatever the application itself exposes to distinguish `stable` from `canary` — for example, a value in the response body or a dedicated HTTP header:

```
for i in $(seq 1 20); do curl -s https://app.example.com/ | grep version; done \
  | sort | uniq -c
```

Adjust the weights if needed and recheck.

Additional resources

- Utilizing Application Load Balancer (ALB)
- Module `istio` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Additional networks for use in application pods

DP implements the ability to use additional software-defined networks (hereinafter referred to as additional networks) for application workloads (pods, virtual machines). You can use the following types of networks:

- Cluster (public) — a network that is publicly available in each project, configured and managed by the administrator. An example is a public WAN network or a shared network for traffic exchange between projects. To create such a network and use it for application pods, contact the cluster administrator.
- Project network (user network) — a network accessible within a namespace, created and managed by the user using the NetworkClass manifest provided by the administrator.

For more information about additional software-defined networks, see Configuring and connecting additional virtual networks for use in application pods.

Creating a project network (user network)

To create a network for a specific project, use the Network and NetworkClass custom resources provided to you by the administrator:

1. Create and apply the Network manifest by specifying the name of the NetworkClass obtained from the administrator in the `spec.networkClass` field:

   ```
   apiVersion: network.deckhouse.io/v1alpha1
   kind: Network
   metadata:
     name: my-network
     namespace: my-namespace
   spec:
     networkClass: my-network-class # The name of the NetworkClass obtained from
   the administrator.
   ```

   > **i** Static identification of the VLAN ID number from the pool assigned by the cluster or network administrator is supported. If the value of the `spec.vlan.id` field is not specified, the VLAN ID will be assigned dynamically.

2. After creating the Network object you can check its status:

   ```
   d8 k -n my-namespace get network my-network -o yaml
   ```

   Example of the status of a Network object:

```
apiVersion: network.deckhouse.io/v1alpha1
kind: Network
metadata:
...
status:
  bridgeName: d8-br-600
  conditions:
  - lastTransitionTime: "2025-09-29T14:51:26Z"
    message: All node interface attachments are ready
    reason: AllNodeInterfaceAttachmentsAreReady
    status: "True"
    type: AllNodeAttachementsAreReady
  - lastTransitionTime: "2025-09-29T14:51:26Z"
    message: Network is operational
    reason: NetworkReady
    status: "True"
    type: Ready
  nodeAttachementsCount: 1
  observedGeneration: 1
  readyNodeAttachementsCount: 1
  vlanID: 600
```

After creating a network, you can connect it to pods.

Connecting additional networks to pods

You can connect cluster networks and project networks to pods. To do this, use the pod annotation, specifying the parameters of the additional networks to be connected.

Example of a pod manifest with two additional networks added (the cluster network `my-cluster-network` and the project network `my-network`):

> **i** The `ifName` field (optional) specifies the name of the TAP interface within the subnet. The `mac` field (optional) specifies the MAC address to be assigned to the TAP interface.

```
apiVersion: v1
kind: Pod
metadata:
  name: app-with-additional-networks
  namespace: my-namespace
  annotations:
    network.deckhouse.io/networks-spec: |
      [
        {
          "type": "Network",
          "name": "my-network",
          "ifName": "veth_mynet",
          "mac": "aa:bb:cc:dd:ee:ff"
        },
        {
          "type": "ClusterNetwork",
          "name": "my-cluster-network",
          "ifName": "veth_public"
        }
      ]
spec:
  containers:
    - name: app
    # other parameters...
```

IPAM for additional networks (IP allocation and assignment)

The IPAM (IP Address Management) mechanism allows you to allocate IP addresses (IPv4 addresses are supported) from pools and assign them to additional network interfaces of pods connected to cluster networks and project networks.

The cluster administrator is responsible for allocating IP addresses for connection to cluster networks. They enable and configure IPAM for networks and define the IP address pool for them. Users can assign addresses and configure IPAM in project networks.

Features of using IPAM in a DP cluster

IPAM in a DP cluster has the following usage features:

- IPAM is enabled **at the network level** via the `spec.ipam.ipAddressPoolRef` parameter (for ClusterNetwork IPAM, the cluster administrator enables it).
- IP address assignment to the pod interface is described in the annotation `network.deckhouse.io/networks-spec` added to the pod through the following fields:
- `ipAddressNames` — a list of IPAddress objects to be assigned to this interface (if the parameter is not specified, IPAddress can be created automatically).
  - `skipIPAssignment` — control of IPAddress reservation/tracking. If `skipIPAssignment: true`, IPAddress reservation/tracking is enabled, but the IP address is **not assigned** to the interface within the pod (advanced usage option).
- **Only IPv4** is supported.

⚠ If multiple additional networks with IPAM enabled are connected to a single pod, it is recommended to explicitly specify `ipAddressNames` for each interface (creating separate IPAddresses). Automatically created `IPAddress` is bound to the pod and may not be suitable for multiple IPAM networks at the same time.

Allocating a pool of IP addresses for the project network and enabling IPAM

i To allocate a pool of addresses for the project network (Network), create an IPAddressPool resource **in the same namespace** as the project network (pods connected to the network).

To allocate a pool of addresses and assign them to network interfaces of pods connected to the project network, perform the following steps:

1. Create an address pool. To do this, use the IPAddressPool resource.

   Example:

   ```
   apiVersion: network.deckhouse.io/v1alpha1
   kind: IPAddressPool
   metadata:
     name: my-net-pool
     namespace: my-namespace
   spec:
     leaseTTL: 1h
     pools:
       - network: 192.168.10.0/24
         ranges:
           - 192.168.10.50-192.168.10.200
         routes:
           - destination: 10.10.0.0/16
             via: 192.168.10.1
   ```

   i The `spec.pools[].ranges` parameter is optional. If it is not specified, the entire CIDR from `spec.pools[].network` is considered available (except for network/broadcast addresses, see the behavior of `/31` and `/32`).

2. Enable IPAM on the network. To do this, specify the parameters of the IPAddressPool created in the

previous step in the `spec.ipam.ipAddressPoolRef` parameter of the Network resource.

Example:

```
apiVersion: network.deckhouse.io/v1alpha1
kind: Network
metadata:
  name: my-network
  namespace: my-namespace
spec:
  networkClass: my-network-class
  ipam:
    ipAddressPoolRef:
      kind: IPAddressPool
      name: my-net-pool
```

After allocating a pool of IP addresses for the project network, they can be assigned to the network interfaces of the pods connected to this network.

Assigning IP addresses to network interfaces of pods connected to an additional network

DP implements automatic IP address assignment for additional pod interfaces, as well as the ability to manually assign specific static IP addresses to additional pod interfaces.

Automatic IP address assignment

To have the IP address for the additional network interface pod selected automatically from the pool, add the annotation `network.deckhouse.io/networks-spec` to the pod. In this annotation, specify the network parameters with IPAM enabled.

Example (the IP address will be automatically selected from the pool created for the `my-network` network and assigned to the `net1` interface):

```
apiVersion: v1
kind: Pod
metadata:
  name: app-with-ipam
  namespace: my-namespace
  annotations:
    network.deckhouse.io/networks-spec: |
      [
        {
          "type": "Network",
          "name": "my-network",
          "ifName": "net1"
        }
      ]
spec:
  containers:
    - name: app
      image: nginx
```

In this case, an IPAddress object (type `Auto`) will be automatically created, and an IP address will be automatically selected from the pool attached to the additional network (in the example, `my-network`) and assigned to the pod network interface.

Manual (explicit) creation of IPAddress with type `Auto`

You can also manually create an IPAddress object with `spec.type: Auto` (without specifying the `static.ip` parameter). In this case, the controller will allocate a free address from the pool attached to the additional network (in the example, `my-network`), and you can bind it to a specific interface using the `ipAddressNames` parameter in the `network.deckhouse.io/networks-spec annotation`.

Example:

1. Create an IPAddress object:

```
apiVersion: network.deckhouse.io/v1alpha1
kind: IPAddress
metadata:
  name: app-net1-auto
  namespace: my-namespace
spec:
  networkRef:
    kind: Network        # or ClusterNetwork
    name: my-network
  type: Auto
```

2. Assign an IP address from the pool to the pod interface:

```
apiVersion: v1
kind: Pod
metadata:
  name: app-with-manual-auto-ip
  namespace: my-namespace
  annotations:
    network.deckhouse.io/networks-spec: |
      [
        {
          "type": "Network",
          "name": "my-network",
          "ifName": "net1",
          "ipAddressNames": ["app-net1-auto"]
        }
      ]
spec:
  containers:
    - name: app
      image: nginx
```

Manually assigning a static IP address to an additional pod interface

To assign a specific static IP address to an additional pod interface, follow these steps:

1. Create an IPAddress in the pod namespace and specify which network it is intended for and which IP address is required:

```
apiVersion: network.deckhouse.io/v1alpha1
kind: IPAddress
metadata:
  name: app-net1-static
  namespace: my-namespace
spec:
  networkRef:
    kind: Network
    name: my-network
  type: Static
  static:
    ip: 192.168.10.50
```

2. Connect the network to the pod and specify the created IPAddress in `ipAddressNames`:

```
apiVersion: v1
kind: Pod
metadata:
  name: app-with-static-ip
  namespace: my-namespace
  annotations:
    network.deckhouse.io/networks-spec: |
      [
        {
          "type": "Network",
          "name": "my-network",
          "ifName": "net1",
          "ipAddressNames": ["app-net1-static"]
        }
      ]
spec:
  containers:
    - name: app
      image: nginx
```

Verification of IP address assignment to an Interface

To verify that an IP address is assigned to the interface, follow these steps:

1. Check the allocated address and phase at `IPAddress` (the phase should be `Allocated`):

   ```
   d8 k -n my-namespace get ipaddress app-net1-static -o yaml
   ```

   Output example:

   ```
   NAME                TYPE   KIND      NAME    ADDRESS         NETWORK
   PHASE       AGE
   ipaddress-auto-1    Auto   Network   mynet   192.168.12.1    192.168.12.0/24
   Allocated   4d1h
   ipaddress-auto-2    Auto   Network   mynet   192.168.12.2    192.168.12.0/24
   Allocated   4d1h
   ```

2. Check the pod annotation `network.deckhouse.io/networks-status` (including `ipAddressConfigs` and routes):

   ```
   d8 k -n my-namespace get pod app-with-static-ip -o
   jsonpath='{.metadata.annotations.network\.deckhouse\.io/networks-status}   ' |
   jq
   ```

   Output example:

```
[
  {
    "type": "Network",
    "name": "mynet",
    "ifName": "aabbcc",
    "mac": "ae:1c:68:7a:00:8f",
    "vlanID": 0,
    "ipAddressConfigs": [
      {
        "name": "ipaddress-auto-1",
        "address": "192.168.12.1",
        "network": "192.168.12.0/24"
      }
    ],
    "conditions": [
      {
        "type": "Configured",
        "status": "True",
        "lastTransitionTime": "2026-02-26T10:06:49Z",
        "reason": "InterfaceConfiguredSuccessfully",
        "message": ""
      },
      {
        "type": "Negotiated",
        "status": "True",
        "lastTransitionTime": "2026-02-26T10:06:49Z",
        "reason": "Up",
        "message": ""
      }
    ]
  }
]
```


Additional resources

- Module `sdn` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Connecting physical network interfaces to DPDK application pods

If your namespace (project) hosts high-performance workloads that require direct access to hardware (e.g., DPDK applications), you can use direct connection of physical network interfaces (Physical Functions and Virtual Functions) to pods via Kubernetes Dynamic Resource Allocation (DRA).

Physical network interfaces can be connected to pods in one of two modes:

- `Shared`: Virtual Functions (VF) are created from Physical Functions (PF) using SR-IOV, and multiple pods can share the same hardware.
- `Dedicated`: Each pod gets exclusive access to the entire PF.

For more information about the capabilities and features of working with Underlay networks in DP, see the section Configuring and connecting underlay networks for hardware device forwarding.

Connecting physical network interfaces to pods

To use physical network interfaces (PF/VF) directly in pods for DPDK applications, you need to:

1. Ensure that your namespace has been marked by the administrator for use with UnderlayNetwork.
2. Create a pod with an annotation requesting the Underlay network device.

Creating a pod with a device from the Underlay network

Create a pod that requests a device from an UnderlayNetwork. The pod annotation should specify:

- `type: "UnderlayNetwork"`: Indicates this is a physical device request
- `name: "underlay-network-name"`: The name of the UnderlayNetwork resource created by the administrator
- `bindingMode`: The binding mode for the device (VFIO-PCI, DPDK, or NetDev).

Example pod configuration for DPDK mode (universal mode that automatically selects the appropriate driver for the network adapter vendor):

```
apiVersion: v1
kind: Pod
metadata:
  name: dpdk-app
  namespace: mydpdk
  annotations:
    network.deckhouse.io/networks-spec: |
      [
        {
          "type": "UnderlayNetwork",
          "name": "dpdk-shared-network",
          "bindingMode": "DPDK"
        }
      ]
spec:
  containers:
  - name: dpdk-container
    image: dpdk-app:latest
    securityContext:
      privileged: false
      capabilities:
        add:
        - NET_ADMIN
        - NET_RAW
        - IPC_LOCK
    volumeMounts:
    - mountPath: /hugepages
      name: hugepage
    resources:
      limits:
        hugepages-2Mi: 4Gi
        memory: 4Gi
        cpu: 4
      requests:
        cpu: 4
        memory: 4Gi
    command: ["/bin/sh", "-c", "sleep infinity"]
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
```

i For DPDK applications, it is important to:

- Configure `capabilities` (NET_ADMIN, NET_RAW, IPC_LOCK) to run in non-privileged mode instead of using `privileged: true`
- Mount hugepages volumes, as DPDK requires hugepages for efficient memory management.

> **i** For VF devices in Shared mode, you can optionally specify a `vlanID` in the annotation to configure VLAN tagging on the VF:
>
> ```
> network.deckhouse.io/networks-spec: |
>   [
>     {
>       "type": "UnderlayNetwork",
>       "name": "dpdk-shared-network",
>       "bindingMode": "VFIO-PCI",
>       "vlanID": 100
>     }
>   ]
> ```

After creating the pod, verify that the device was allocated by checking the `network.deckhouse.io/networks-status` annotation:

```
d8 k -n mydpdk get pod dpdk-app -o
jsonpath='{.metadata.annotations.network\.deckhouse\.io/networks-status}' | jq
```

You can also check the ResourceClaim that was automatically created:

```
d8 k -n mydpdk get resourceclaim
```

Example pod status with allocated UnderlayNetwork device:

```
apiVersion: v1
kind: Pod
metadata:
  name: dpdk-app
  namespace: mydpdk
  annotations:
    network.deckhouse.io/networks-spec: |
      [
        {
          "type": "UnderlayNetwork",
          "name": "dpdk-shared-network",
          "bindingMode": "DPDK"
        }
      ]
    network.deckhouse.io/networks-status: |
      [
        {
          "type": "UnderlayNetwork",
          "name": "dpdk-shared-network",
          "bindingMode": "DPDK",
          "netDevInterfaces": [
            {
              "name": "ens1f0",
              "mac": "00:1b:21:bb:aa:cc"
            }
          ],
          "conditions": [
            {
              "type": "Configured",
              "status": "True",
              "reason": "InterfaceConfiguredSuccessfully",
              "message": "",
              "lastTransitionTime": "2025-01-15T10:35:00Z"
            },
            {
              "type": "Negotiated",
              "status": "True",
              "reason": "Up",
              "message": "",
              "lastTransitionTime": "2025-01-15T10:35:00Z"
            }
          ]
        }
      ]
status:
  phase: Running
  ...
```

Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.

The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

IP addresses of virtual machines

Every virtual machine (VM) gets an address in the main cluster network. The sections below cover how to view the assigned address, request a specific one, and keep it in the project.

IP addresses of VMs

Two resources describe the address of a machine in the main cluster network, the cluster-wide address lease and the address reserved for the project.

The `.spec.settings.virtualMachineCIDRs` block in the module settings defines the subnets that machines get IP addresses from. All addresses of a subnet are available except the first and the last one.

If an address pool is configured for the main cluster network in the `sdn` module, the address of a machine in that network is managed by the shared IPAM of that module, that is, by the same IPAddress resource as for additional networks. The address is requested automatically, and existing machines switch to the shared IPAM without changing their addresses and without a restart.

> ⚠ In a cluster with the shared IPAM, the VirtualMachineIPAddress and VirtualMachineIPAddressLease resources, as well as the `.spec.virtualMachineIPAddressName` parameter, are deprecated. They keep working and are described in the sections below, but the main network address of new machines is managed by the shared IPAM, covered in IPAM for the main network.

The cluster-wide VirtualMachineIPAddressLease (`vmipl`) resource manages leases of IP addresses from the shared pool specified in `virtualMachineCIDRs`, while the project VirtualMachineIPAddress (`vmip`) resource reserves leased addresses and binds them to machines. An address is assigned to a machine when the resource moves to the `Attached` phase. The other phases are described in the `.status.phase` field.

By default, Deckhouse Platform (DP) assigns an address to the machine itself and keeps it assigned until the machine is deleted. This works as follows:

- You create a virtual machine named `<VM_NAME>`.
- DP creates a VirtualMachineIPAddress resource named `<VM_NAME>-<HASH>` to request an IP address and bind it to the virtual machine.
- For that resource, a VirtualMachineIPAddressLease lease is created, which picks a random IP address from the shared pool.
- As soon as the VirtualMachineIPAddress resource is created, the virtual machine gets the assigned IP address.

After the machine is deleted, the VirtualMachineIPAddress resource is deleted too, but the address itself stays assigned to the project for a while, and you can request it again.

- Using the CLI
- Using the web interface

To view the list of IP address leases (`vmipl`), run the following command:

```
d8 k get vmipl
```

Example output:

```
NAME             VIRTUALMACHINEIPADDRESS                                  STATUS   AGE
ip-10-66-10-14   {"name":"linux-vm-7prpx","namespace":"default"}     Bound    12h
```

To view the address assigned to a machine, run the following command:

```
d8 k get vmip
```

Example output:

```
NAME             ADDRESS       STATUS     VM          AGE
linux-vm-7prpx   10.66.10.14   Attached   linux-vm    12h
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization → IP addresses**.
3. The list shows the resource name, status, address, type (`Auto` or `Static`), the virtual machine that uses the address, and the resource age.

Assigning a specific IP address

Instead of a random address from the pool, you can give a machine an address you choose in advance.

- Using the CLI
- Using the web interface

1. Create a VirtualMachineIPAddress resource:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineIPAddress
metadata:
  name: linux-vm-custom-ip
spec:
  staticIP: 10.66.20.77
  type: Static
EOF
```

2. Create a new virtual machine or modify an existing one, and specify the VirtualMachineIPAddress resource you need explicitly in the specification:

```
spec:
  virtualMachineIPAddressName: linux-vm-custom-ip
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization → IP addresses**.
3. Click **Create**.
4. In the **Create resource** window that opens, enter the resource name in the **Name** field.
5. On the **Configuration** tab, select `Static` in the **Type** field, and specify the address you need in the **Static IP address** field.
6. Click **Apply**.
7. Specify the name of the created resource in the `.spec.virtualMachineIPAddressName` parameter of the virtual machine.

Keeping an IP address in the project

To keep the automatically allocated IP address of a virtual machine from being deleted along with the virtual machine itself, do the following.

Get the name of the VirtualMachineIPAddress resource for the given virtual machine:

```
d8 k get vm linux-vm -o jsonpath="{.status.virtualMachineIPAddressName}"
```

Example output:

```
linux-vm-7prpx
```

Remove the `.metadata.ownerReferences` block from the resource you found:

```
d8 k patch vmip linux-vm-7prpx --type=merge --patch '{"metadata":{"ownerReferences":null}}'

# Or make the same changes by editing the resource.

d8 k edit vmip linux-vm-7prpx
```

After the virtual machine is deleted, the VirtualMachineIPAddress resource is preserved and you can reuse it in a newly created virtual machine:

```
spec:
  virtualMachineIPAddressName: linux-vm-7prpx
```

Even if the VirtualMachineIPAddress resource is deleted, the IP address stays leased to the current project or namespace for another 10 minutes. So you can claim it again on request:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineIPAddress
metadata:
  name: linux-vm-custom-ip
spec:
  staticIP: 10.66.20.77
  type: Static
EOF
```

IPAM for the main network

> **i** The shared IPAM works only if an address pool is configured for the main cluster network in the `sdn` module. Without it, machine addresses are managed by the deprecated VirtualMachineIPAddress resources described above.

The address of a machine in the main network is held by the IPAddress resource of the `sdn` module, the same one that addresses additional networks. There are two ways to get an address:

- **Automatic**: If the `ipAddressName` field isn't set for the `Main` network, the controller creates an IPAddress resource bound to the machine and waits for the address before starting it. A machine that failed to get an address stays in the `Pending` phase, and the reason is shown by the `VirtualMachineIPAddressReady` condition.
- **Static**: Create an IPAddress resource of the main network yourself and specify it in the `.spec.networks[]` parameter. Such a resource belongs to you, so it isn't deleted along with the machine and can be reassigned to another one.

An example of a static IPAddress resource for the main network:

```
d8 k apply -f - <<EOF
apiVersion: network.deckhouse.io/v1alpha1
kind: IPAddress
metadata:
  name: linux-vm-main-ip
spec:
  networkRef:
    kind: ClusterNetwork
    name: main
  type: Static
  static:
    ip: 10.66.20.77
EOF
```

An example of a reference to this resource in the machine specification:

```
spec:
  networks:
    - type: Main
      ipAddressName: linux-vm-main-ip
```

Changing the `ipAddressName` field of the main network changes the address of the primary interface, so it requires a restart of the machine.

The current address and the name of the IPAddress resource it is assigned to are shown in the machine status:

```
d8 k get vm linux-vm -o jsonpath='{.status.networks[?(@.type=="Main")]}'
```

Example output:

```
{"id":1,"ipAddress":"10.66.10.14","ipAddressName":"linux-vm-4bkqr","type":"Main"}
```

Switching a machine to the shared IPAM

Machines that don’t reference a VirtualMachineIPAddress resource explicitly switch to the shared IPAM automatically, and the address doesn’t change and the machine isn’t restarted.

A machine with the `.spec.virtualMachineIPAddressName` parameter set keeps working through the deprecated mechanism, because the reference to the resource is set in its specification explicitly.

For the VirtualMachineIPAddress resource listed in the machine specification, DP maintains an IPAddress resource with the same name and the same address. To switch the machine and keep its address, replace one parameter with the other:

```
spec:
  # Remove the virtualMachineIPAddressName parameter.
  networks:
    - type: Main
      # An IPAddress resource with the same name and address.
      ipAddressName: linux-vm-custom-ip
```

Restart the machine to apply the change. The address stays the same, and the VirtualMachineIPAddress resource can be deleted after that.

⚠ Don’t remove the `.spec.virtualMachineIPAddressName` parameter without such a replacement, otherwise the machine gets a new address from the pool.

Additional resources

- Module `sdn` documentation
- Module `virtualization` parameters
- Module `virtualization` custom resources

i **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**

The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Accessing applications on a virtual machine

You can reach a virtual machine (VM) directly by its IP address, but this approach has limitations. You have to know the address in advance, it can change when the machine is recreated, and you can’t reach a group of machines at once. Kubernetes services solve all of these tasks.

A service gives a machine or a group of machines a permanent name that hides their addresses, and distributes requests evenly among them. The name is formed as `<SERVICE NAME>.<NAMESPACE>.svc.<CLUSTER_NAME>`, and within the same namespace the short form `<SERVICE_NAME>` is enough.

Which service type to choose depends on the task:

- `Headless`: Direct access to specific machines inside the cluster without a single entry point.
- `ClusterIP`: A single internal address with balancing between machines.
- `NodePort`: External access through a port on the cluster nodes.
- `LoadBalancer`: External access through an external load balancer.

i If a connection to the VM from a cluster node doesn’t go through, check the `NetworkPolicy` in the project. The policy may deny traffic to the machine.

A machine gets into a service by labels. Assign the machine the label that the service looks for:

- Using the CLI

- Using the web interface

Assign the label with the `d8 k label` command:

```
d8 k label vm linux-vm app=nginx
```

Example output:

```
virtualmachine.virtualization.deckhouse.io/linux-vm labeled
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Virtual machines**.
3. Select the VM you need from the list and click its name.
4. Go to the **Meta** tab.
5. Click **Add** in the **Labels** or **Annotations** section.
6. In the window that opens, set the key and the value, then press **Enter**.
7. Click the **Save** button that appears.

Headless service

A headless service doesn’t allocate an IP address of its own, but returns the addresses of the machines themselves. This way you reach a specific machine by its DNS name without setting up a separate entry point for it. Even for a single machine, this is more convenient than a fixed address, because the name doesn’t change when the machine is recreated.

- Using the CLI
- Using the web interface

Create a service with `clusterIP: None`:

```
d8 k apply -f - <<EOF
apiVersion: v1
kind: Service
metadata:
  name: http
  namespace: default
spec:
  clusterIP: None
  selector:
    # The label the service uses to select virtual machines.
    app: nginx
EOF
```

After creation, you can reach the machine by the `http.default.svc` name.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Network** → **Services**.
3. Click **Create**.
4. In the form that opens, enter the service name in the **Name** field.
5. In the **Type** field, select `Headless`.
6. In the **Workload selector** block, mark the virtual machines you need, and their labels go into the service selector.
7. In the **Ports** block, set the **Port** and **Target port** values.
8. Click **Create**.

Service of the ClusterIP type

`ClusterIP` is the standard service type that provides an internal IP address for accessing the service inside the cluster. This IP address is used to route traffic between different system components and lets virtual machines interact with each other through a predictable and stable address.

- Using the CLI
- Using the web interface

Here is an example of a `ClusterIP` configuration:

```
d8 k apply -f - <<EOF
apiVersion: v1
kind: Service
metadata:
  name: http
spec:
  selector:
    # The label the service uses to decide which virtual machine to route traffic
to.
    app: nginx
EOF
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Network → Services**.
3. In the window that opens, configure the service.
4. Click **Create**.

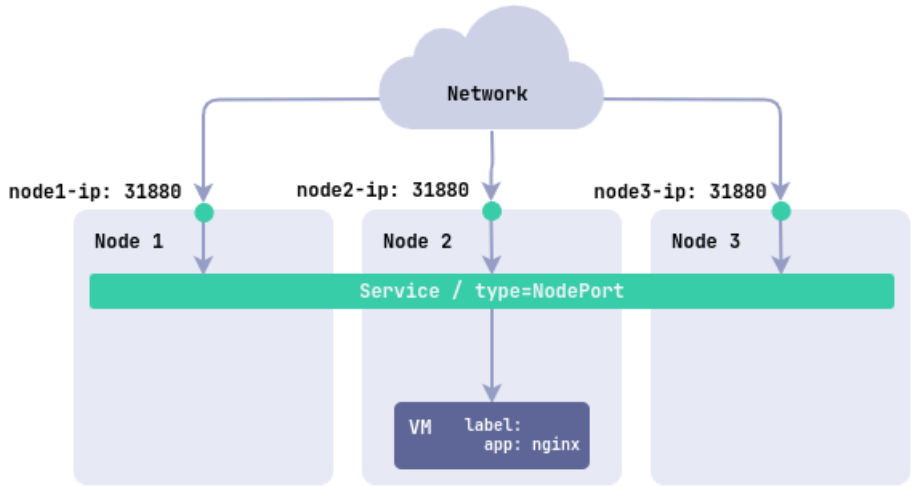
Service of the NodePort type

`NodePort` is an extension of the `ClusterIP` service that provides access to the service through a specified port on all cluster nodes. This makes the service reachable from outside the cluster through the combination of a node IP address and a port, and it suits the cases where an external load balancer isn’t needed.

- Using the CLI
- Using the web interface

Create the following service:

```
d8 k apply -f - <<EOF
apiVersion: v1
kind: Service
metadata:
  name: linux-vm-nginx-nodeport
spec:
  type: NodePort
  selector:
    # The label the service uses to decide which virtual machine to route traffic
to.
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 31880
EOF
```



In this example, a service of the `NodePort` type is created, which opens external port 31880 on all nodes of your cluster. This port routes incoming traffic to internal port 80 of the virtual machine where the Nginx application runs.

If you don’t specify the `nodePort` value explicitly, an arbitrary port is assigned to the service, and you can see it in the service status right after creation.

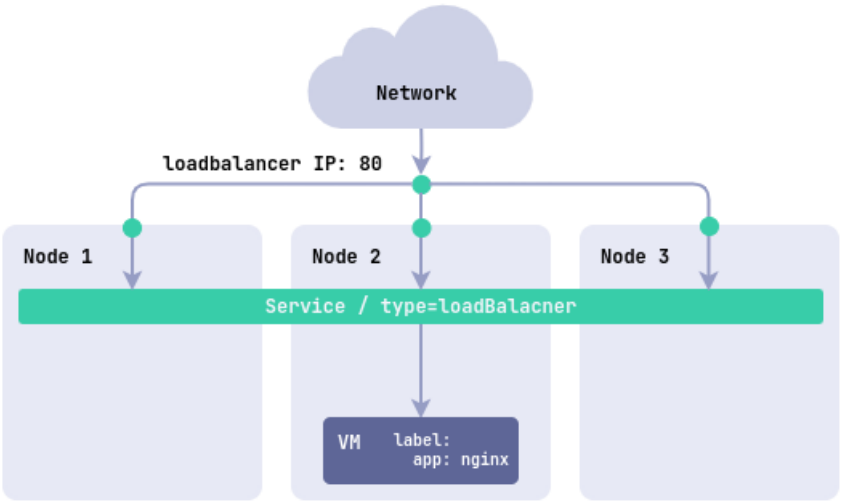
1. Go to the **Projects** tab and select the project you need.
2. Go to **Network** → **Services**.
3. Click **Create**.
4. In the form that opens, enter the service name in the **Name** field.
5. In the **Type** field, select `NodePort`.
6. In the **Workload selector** block, mark the virtual machines you need.
7. In the **Ports** block, set the **Port**, **Target port**, and, if required, **Node port** values.
8. Click **Create**.

Service of the LoadBalancer type

`LoadBalancer` is a service type that automatically creates an external load balancer with a permanent IP address. This balancer distributes incoming traffic among virtual machines, making the service available from the internet.

- Using the CLI
- Using the web interface

```
d8 k apply -f - <<EOF
apiVersion: v1
kind: Service
metadata:
  name: linux-vm-nginx-lb
spec:
  type: LoadBalancer
  selector:
    # The label the service uses to decide which virtual machine to route traffic to
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
EOF
```



1. Go to the **Projects** tab and select the project you need.
2. Go to **Network** → **Services**.
3. Click **Create**.
4. In the form that opens, enter the service name in the **Name** field.
5. In the **Type** field, select `LoadBalancer`.
6. In the **Workload selector** block, mark the virtual machines you need.
7. In the **Ports** block, set the **Port** and **Target port** values.
8. Click **Create**.

9. The external address of the service is shown in the service list, in the **External IP** column.

Publishing VM services with Ingress

`Ingress` lets you manage incoming HTTP/HTTPS requests and route them to different servers within your cluster. This is the most suitable method if you want to use domain names and SSL termination to access your virtual machines.

- Using the CLI
- Using the web interface

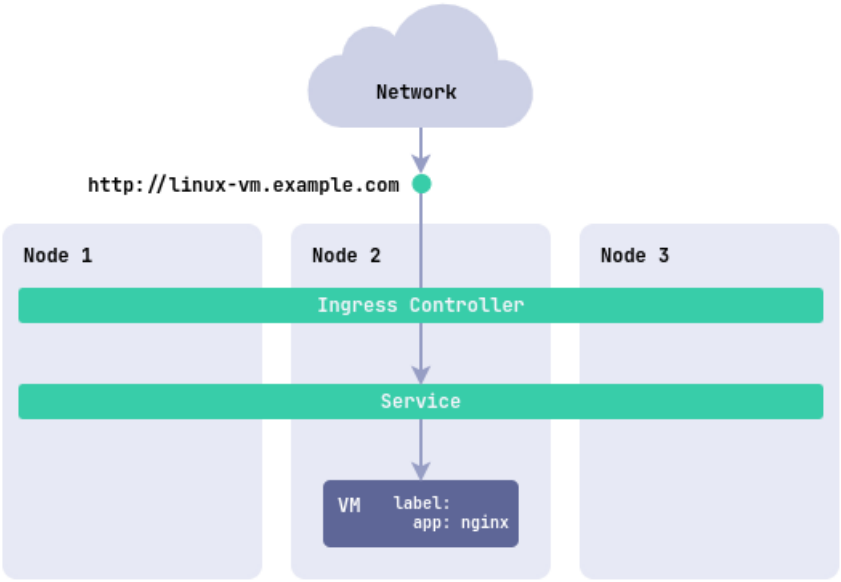
To publish a virtual machine service through `Ingress`, create the following resources:

An internal service to bind with `Ingress`. Example:

```
d8 k apply -f - <<EOF
apiVersion: v1
kind: Service
metadata:
  name: linux-vm-nginx
spec:
  selector:
    # the label the service uses to decide which virtual machine to route traffic to
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
EOF
```

And an `Ingress` resource for publishing. Example:

```
d8 k apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: linux-vm
spec:
  rules:
    - host: linux-vm.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: linux-vm-nginx
                port:
                  number: 80
EOF
```



1. Go to the **Projects** tab and select the project you need.
2. Go to **Network** → **Ingresses**.
3. Click **Create**.
4. In the **Create Ingress** form that opens, enter the resource name in the **Name** field, and select the controller class (`spec.ingressClassName`) in the **Ingress Class** field.
5. In the **Rules** block, click **Add rule (host)** and describe the host and the routing paths to the service you need.
6. If HTTPS is required, click **Add certificate** in the **TLS certificates** block and specify the secret with the certificate; set the **Default backend** if required.
7. Click **Create**.

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Additional network interfaces of virtual machines

Besides the main cluster network, a virtual machine (VM) can be connected to additional networks of the `sdn` module, and its interfaces can be assigned addresses.

Additional network interfaces

Besides the main cluster network, a machine can connect to additional networks, both project ones (Network) and cluster ones (ClusterNetwork). You list the networks you need in the `.spec.networks` block, and if that block isn't set, the machine works only in the main cluster network.

> To work with additional networks, the `sdn` module has to be enabled.

> ⚠ You don't have to specify the main cluster network (`type: Main`), and a machine can work only in additional networks. But if the main network is specified, it has to be first in the list.

Specifics of additional network interfaces:

- the order of networks in `.spec.networks` determines the order in which interfaces are attached inside the virtual machine;
- adding or removing an additional network (`Network` or `ClusterNetwork`) on a running VM applies without a reboot. The ACPI indexes of existing interfaces are preserved when adding or removing, so interface names in the guest OS stay stable;
- adding or removing the main network (`type: Main`) still requires a VM reboot, because it's bound

to the main network interface of the pod and can’t be changed on a running pod;
- to preserve the order of network interfaces inside the guest operating system, add new networks to the end of the `.spec.networks` list and don’t change the order of existing ones;
- network security policies (NetworkPolicy) don’t apply to additional network interfaces;
- the network parameters (IP addresses, gateways, DNS, and so on) for additional networks are configured manually from inside the guest OS (for example, with Cloud-Init), unless IPAM is configured for the network, as covered in IPAM for additional network interfaces.

For each additional network interface, a unique MAC address is created and reserved automatically, which prevents MAC address collisions. The VirtualMachineMACAddress (`vmmac`) and VirtualMachineMACAddressLease (`vmmacl`) resources are used for this.

A MAC address is generated at random from a pool of allowed ranges.

- Ranges: `x2-xx-xx-xx-xx-xx`, `x6-xx-xx-xx-xx-xx`, `xA-xx-xx-xx-xx-xx`, `xE-xx-xx-xx-xx-xx`.
- The first three octets (OUI) are formed from the cluster UUID, and the last three (NIC) are picked at random from 16 million possible combinations.

The cluster-wide VirtualMachineMACAddressLease (`vmmacl`) resource manages leases of addresses from the shared pool, while the project VirtualMachineMACAddress (`vmmac`) resource reserves leased addresses and binds them to machines. An address stays assigned to the machine until it’s deleted.

When a network is removed from the VM configuration:

- The MAC address of the interface is released.
- The related VirtualMachineMACAddress and VirtualMachineMACAddressLease resources are deleted automatically.
- The allocated `IPAddress` resource is deleted automatically (if IPAM was used).

The following example shows how to connect a virtual machine to an additional network:

- Using the CLI
- Using the web interface

List the networks you need in the `.spec.networks` block of the machine specification.

Here is an example of connecting a VM to the main cluster network and the `user-net` project network:

```
spec:
  networks:
    - type: Main # If specified, it has to be first
    - type: Network # Network type (Network \ ClusterNetwork)
      name: user-net # Network name
```

Here is an example of connecting to several networks, including the `corp-net` cluster network:

```
spec:
  networks:
    - type: Main # If specified, it has to be first
    - type: Network
      name: user-net
    - type: ClusterNetwork
      name: corp-net # Network name
```

Here is an example of connecting a VM only to additional networks (without the main cluster network):

```
spec:
  networks:
    - type: Network
      name: isolated-net
    - type: ClusterNetwork
      name: corp-net
```

You can see the information about the connected networks and their MAC addresses in the VM status:

```
status:
  networks:
    - type: Main
    - type: Network
      name: user-net
      macAddress: aa:bb:cc:dd:ee:01
    - type: ClusterNetwork
      name: corp-net
      macAddress: aa:bb:cc:dd:ee:02
```

To view the list of MAC address leases (`vmmacl`), run the following command:

```
d8 k get vmmacl
```

Example output:

```
NAME                    VIRTUALMACHINEMACADDRESS                          STATUS   AGE
mac-5e-e6-19-22-0f-d8   {"name":"vm-01-fz9cr","namespace":"pr-sdn"}   Bound    45s
mac-5e-e6-19-29-89-cf   {"name":"vm-01-99qj6","namespace":"pr-sdn"}   Bound    45s
mac-5e-e6-19-54-f9-be   {"name":"vm-01-5jqxg","namespace":"pr-sdn"}   Bound    45s
```

To check the assigned MAC addresses, run the following command:

```
d8 k get vmmac
```

Example output:

```
NAME          ADDRESS             STATUS     VM      AGE
vm-01-5jqxg   5e:e6:19:54:f9:be   Attached   vm-01   5m42s
vm-01-99qj6   5e:e6:19:29:89:cf   Attached   vm-01   5m42s
vm-01-fz9cr   5e:e6:19:22:0f:d8   Attached   vm-01   5m42s
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization → Virtual machines**.
3. Select the VM you need from the list and click its name.
4. On the **Configuration** tab, scroll down to the **Networks** section and click **Add**.
5. In the **Add network** window that opens, specify the network you need in the **Select network** field.
6. Click **Add**, then click the **Save** button that appears.

To create a project network:

1. Go to the **Projects** tab and select the project you need.
2. Go to **Network → SDN → Networks**.
3. Click **Create**.
4. In the **Create resource** window that opens, enter the network name in the **Name** field.
5. On the **Configuration** tab, select the Network class in the **Network class** field, the network type in the **Type** field, and the VLAN ID in the **VLAN** field. Set **Mtu** and the parameters of the **IPAM** block if required.
6. Click **Apply**.
7. The created networks are shown in the list with the **Status**, **Type**, **VLAN**, and **Network class** columns.

Interface naming and ARP flux in the guest OS

When configuring network interfaces in the guest OS, use stable identifiers (predictable `enpXsY` names or binding by MAC address) instead of `ethX` names, as described in Network interface naming in the guest OS.

On a Linux guest system with several interfaces in the same subnet, the ARP flux problem can occur, where the kernel answers ARP requests through an arbitrary interface rather than the one the request arrived on, which leads to an unstable connection and packet loss because of an incorrect MAC address in the router caches.

To fix this, set the parameters that make the system answer requests strictly through the interface with the target IP and use the correct source address:

```
sysctl -W net.ipv4.conf.all.arp_ignore=1
sysctl -W net.ipv4.conf.all.arp_announce=2
```

Example for cloud-init:

```
write_files:
- path: /etc/sysctl.d/90-arp-strict.conf
  content: |
    net.ipv4.conf.all.arp_ignore=1
    net.ipv4.conf.all.arp_announce=2
```

The parameter values are described in the IP sysctl documentation.

IPAM for additional network interfaces

Deckhouse Platform (DP) can hand out addresses in an additional network itself, if an administrator has configured an address pool for that network (`spec.ipam.ipAddressPoolRef` in the `sdn` module). DP then allocates addresses for the additional VM interfaces and delivers them to the guest OS over DHCP.

Two modes are supported:

- **Automatic (DHCP)**: If the `ipAddressName` field isn't specified in `.spec.networks[]`, the controller automatically creates an IPAddress resource (of the `Auto` type) bound to the VM through `ownerReferences`, and passes it to the `sdn` module. The `sdn` module allocates an address from the pool and delivers it to the guest OS over DHCP. The address is preserved across VM reboots and migrations, because it's bound to the VM rather than to the pod. For this mode to work, the DHCP client has to be enabled on the corresponding interface in the guest OS.
- **Static**: If the `ipAddressName` field is specified in `.spec.networks[]`, the controller uses the IPAddress resource provided by the user (of the `Static` type, `network.deckhouse.io/v1alpha1`). The address is defined by the user and doesn't change automatically.

If an additional network has no IPAM pool configured, the IPAM feature isn't enabled. The interface works in L2-only mode, and IP addressing has to be configured manually in the guest OS.

⚠ If an IPAM pool is configured for an additional network, don't configure a static IP address on the additional interface in the guest OS manually (through Cloud-Init). Use the automatic (DHCP) or static (`ipAddressName`) mode to avoid address conflicts.

i If an additional network has an IPAM pool but the IPAddress resource isn't allocated yet or is in the `Pending` state (for example, because the address pool is exhausted), the interface is temporarily skipped. The VM starts without it, and the `NetworkReady` condition reports the error. Once an IP address becomes available, the interface is attached automatically.

- Using the CLI
- Using the web interface

Here is an example VM configuration with automatic IP address allocation for an additional network:

```
spec:
  networks:
    - type: Main
    - type: ClusterNetwork
      name: corp-net
      # ipAddressName isn't specified → automatic mode (DHCP) is used
```

Here is an example VM configuration with a static IP address for an additional network:

```
spec:
  networks:
    - type: Main
    - type: ClusterNetwork
      name: corp-net
      ipAddressName: my-static-ip # Name of the IPAddress resource (SDN)
```

Here is an example of a static IPAddress resource configuration:

```
apiVersion: network.deckhouse.io/v1alpha1
kind: IPAddress
metadata:
  name: my-static-ip
  namespace: my-namespace
spec:
  networkRef:
    kind: ClusterNetwork
    name: corp-net
  type: Static
  static:
    ip: 192.168.200.42
```

The allocated IP address is shown in the VM status:

```
status:
  ipAddress: 10.66.10.2                 # IP address of the main network.
  virtualMachineIPAddressName: vm-01-main-ip # IPAddress name of the main network.
  networks:
    - type: Main
    - type: ClusterNetwork
      name: corp-net
      macAddress: 32:a6:a1:0a:92:48
      virtualMachineMACAddressName: vm-01-rxzd6
      ipAddress: 192.168.200.4            # IP address of the additional network (from
IPAM).
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Network** → **SDN** → **IP pools**.
3. Click **Create**.
4. In the **Create resource** window that opens, enter the pool name in the **Name** field.
5. On the **Configuration** tab, set the lease lifetime in the **Lease TTL** field, and in the **Pools** block, set the network (**Network**), the address ranges (**Ranges**), and the routes (**Routes**).
6. Click **Apply**.

Configuring the guest OS for interfaces added on the fly

When an additional network interface is attached to an already running VM, the guest OS has to be configured to bring new network interfaces up automatically and request a DHCP lease. By default, Linux doesn’t start a DHCP client on interfaces added on the fly.

To make such interfaces configure themselves, use one of the following approaches in the guest OS:

- **NetworkManager** (Ubuntu, RHEL, CentOS): Configures new interfaces with DHCP automatically, if the `network-manager` service is running.
- **A udev rule** (Alpine and other systems without `network-manager`): Add a udev rule to bring new interfaces up:

  ```
  write_files:
    - path: /etc/udev/rules.d/90-hotplug-network.rules
      content: |
        SUBSYSTEM=="net", ACTION=="add", RUN+="/sbin/ifup %k"
  ```

Interfaces present at VM boot (included in the initial network configuration) don’t need any extra setup,

because the guest OS configures them at startup through Cloud-Init.

Additional resources

- Module `sdn` documentation
- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Intra-cluster communication

To organize intra-cluster communication in Deckhouse Platform, it is recommended to use Services instead of accessing Pods directly. Services provide load balancing between Pods, stable network connectivity, and DNS integration for convenient service discovery. They also support various access scenarios and ensure isolation and security of network traffic.

If necessary, you can use either the standard Service-based load balancer or the advanced load balancer based on the `service-with-healthchecks` module.

Standard load balancer

In Kubernetes, the Service resource is responsible for both internal and external request load balancing. This resource:

- Distributes requests between the application’s working Pods.
- Excludes unhealthy Pods from load balancing.

Readiness probes specified in the specification of the containers belonging to a Pod are used to check whether the Pod is able to handle incoming requests.

Limitations of the standard Service load balancer

The standard Service load balancing mechanism is suitable for most cloud application scenarios but has two limitations:

- If at least one container in a Pod fails the readiness probe, the entire Pod is considered `NotReady` and is excluded from load balancing for all Services it is associated with.
- Each container can only have one probe configured, so you cannot set up separate probes, for example, for read and write availability checks.

Example scenarios where the standard load balancer is insufficient:

- Database:
  - Runs in three Pods: `db-0`, `db-1`, and `db-2`, each containing a single container running the database process.
  - You need to create two Services: `db-write` for writing and `db-read` for reading.
  - Read requests must be balanced across all Pods.
  - Write requests must be balanced only to the Pod designated as the master by the database.
- Virtual machine:
  - A Pod contains a single container running the `qemu` process, acting as a hypervisor for a guest virtual machine.
  - Independent processes, such as a web server and an SMTP server, run on the guest VM.
  - You need to create two Services: `web` and `smtp`, each with its own readiness probe.

Example Service for a standard load balancer

```
apiVersion: v1
kind: Service
metadata:
  name: productpage
  namespace: bookinfo
spec:
  ports:
  - name: http
    port: 9080
  selector:
    app: productpage
  type: ClusterIP
```

Example of a standard load balancer for a PostgreSQL cluster

Creating a StatefulSet for PostgreSQL

For a StatefulSet to work correctly, you need to create a standard Service to generate DNS names for individual Pods. This Service will not be used for direct database access.

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  selector:
    app: postgres
  ports:
    - protocol: TCP
      port: 5432
      targetPort: 5432
```

Example StatefulSet manifest:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
  name: my-ns
spec:
  serviceName: postgres
  replicas: 3
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
        - name: postgres
          image: postgres:13
          ports:
            - containerPort: 5432
          env:
            - name: POSTGRES_USER
              value: postgres
            - name: POSTGRES_PASSWORD
              value: example
```

Advanced load balancer

The advanced load balancer is implemented using the ServiceWithHealthcheck resources from the `service-with-healthchecks` module.

Unlike the standard load balancer where readiness probes are tied to container states, the ServiceWithHealthcheck–based load balancer allows you to configure active probes for individual TCP ports. This way, each load balancer serving the same Pod can operate independently from the others.

Load balancer structure

The load balancer consists of two components:

- Controller — runs on cluster master nodes and manages ServiceWithHealthcheck resources.
- Agents — run on every cluster node and perform probes for Pods running on that node.

The ServiceWithHealthcheck load balancer is designed to be CNI-independent while using standard Service and EndpointSlice resources:

- When a ServiceWithHealthcheck resource is created, the controller automatically creates a Service with the same name in the same namespace, but with an empty `selector` field. This prevents the standard controller from creating EndpointSlice objects, which are normally used for load balancing.
- When an agent detects Pods on its node that fall under a ServiceWithHealthcheck, it runs the configured probes and creates an EndpointSlice with the list of verified IP addresses and ports. This EndpointSlice is linked to the Service created earlier.
- The CNI matches all EndpointSlice objects with the standard Services created earlier and performs load balancing across the verified IP addresses and ports on all cluster nodes.

Migrating from a Service to a ServiceWithHealthchecks resource, for example in a CI/CD pipeline, should not cause difficulties. The ServiceWithHealthchecks specification mostly repeats the standard Service specification, but includes an additional `healthcheck` section. During the resource lifecycle, a Service with the same name is created in the same namespace to route traffic to workloads in the cluster in the usual way (via `kube-proxy` or CNI).

Configuring the load balancer

⚠ To use advanced load balancers:

- Configure the network policies for the project where ServiceWithHealthchecks load balancers will be used.
- If necessary, manually switch to using ServiceWithHealthchecks instead of standard load balancers (replace the existing Service objects with ServiceWithHealthchecks).

For more details, see the `service-with-healthchecks` module documentation.

You can configure this type of load balancing using the ServiceWithHealthchecks resource:

- Its specification is identical to a standard Service with the addition of a `healthcheck` section that contains a set of checks.
- Currently, three types of probes are supported:
  - `TCP`: A basic check using TCP connection establishment.
  - `HTTP`: Sends an HTTP request and expects a specific response code.
  - `PostgreSQL`: Sends an SQL query and expects it to complete successfully.

Example configuration of advanced ServiceWithHealthchecks load balancers

Create a Secret to store the credentials required for database probe access:

```
d8 k -n my-ns create secret generic cred-secret --from-literal=user=postgres --from-literal=password=example cred-secret
```

Example load balancer manifest for reading:

```
apiVersion: network.deckhouse.io/v1alpha1
kind: ServiceWithHealthchecks
metadata:
  name: postgres-read
spec:
  ports:
  - port: 5432
    protocol: TCP
    targetPort: 5432
  selector:
    app: postgres
  healthcheck:
    probes:
    - mode: PostgreSQL
      postgreSQL:
        targetPort: 5432
        dbName: postgres
        authSecretName: cred-secret
        query: "SELECT 1"
```

Example load balancer manifest for writing:

```
apiVersion: network.deckhouse.io/v1alpha1
kind: ServiceWithHealthchecks
metadata:
  name: postgres-write
spec:
  ports:
  - port: 5432
    protocol: TCP
    targetPort: 5432
  selector:
    app: postgres
  healthcheck:
    probes:
    - mode: PostgreSQL
      postgreSQL:
        targetPort: 5432
        dbName: postgres
        authSecretName: cred-secret
        query: "SELECT NOT pg_is_in_recovery()"
```

Hosting two independent load balancers on a single virtual machine

A Linux-based virtual machine runs two applications: an HTTP server (TCP 8080) and an SMTP server (TCP 2525). You need to configure two separate load balancers for these services — a web load balancer and an SMTP load balancer.

Creating the virtual machine

Create a `my-vm` virtual machine based on the examples in the DVP documentation.

In the following manifest example, the label `vm: my-vm` is added for further identification in the load balancers.

```
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachine
metadata:
  name: my-vm
  namespace: my-ns
  labels:
    vm: my-vm
spec:
  virtualMachineClassName: host
  cpu:
    cores: 1
  memory:
    size: 1Gi
  provisioning:
    type: UserData
    userData: |
      #cloud-config
      ssh_pwauth: True
      users:
      - name: cloud
        passwd:
'$6$rounds=4096$saltsalt$fPmUsbjAuA7mnQNTajQM6ClhesyG0.yyQhvahas02ejfMAq1ykBo1RquzS0
R6GgdIDlvS.kbUwDablGZKZcTP/'
        shell: /bin/bash
        sudo: ALL=(ALL) NOPASSWD:ALL
        lock_passwd: False
  blockDeviceRefs:
    - kind: VirtualDisk
      name: linux-disk
```

Manifests for advanced load balancers for the web and SMTP services

Example web load balancer manifest:

```
apiVersion: network.deckhouse.io/v1alpha1
kind: ServiceWithHealthchecks
metadata:
  name: web
  namespace: my-ns
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    vm: my-vm
  healthcheck:
    probes:
    - mode: HTTP
      http:
        targetPort: 8080
        method: GET
        path: /healthz
```

Example SMTP load balancer manifest:

```
apiVersion: network.deckhouse.io/v1alpha1
kind: ServiceWithHealthchecks
metadata:
  name: smtp
  namespace: my-ns
spec:
  ports:
  - port: 25
    protocol: TCP
    targetPort: 2525
  selector:
    vm: my-vm
  healthcheck:
    probes:
    - mode: TCP
      tcp:
        targetPort: 2525
```

Additional resources

- Module `service-with-healthchecks` documentation

> **i** **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Managing authorization and access to workload with Istio

You can use the istio module to manage authorization and control access to workloads. Before configuring authorization, make sure the module is enabled in the cluster.

Authorization

Authorization is managed using the AuthorizationPolicy resource from Istio. When this resource is created for a Service, the following request decision rules apply:

- If a request matches a `DENY` policy — deny the request.
- If there are no `ALLOW` policies for the Service — allow the request.
- If a request matches an `ALLOW` policy — allow the request.
- All other requests — to be denied.

In other words, if something is explicitly denied, the deny rule takes precedence. If something is explicitly allowed, only explicitly permitted requests are allowed (denies still have priority).

You can use the following arguments when writing authorization rules:

- Service identifiers and wildcards based on them (`mycluster.local/ns/myns/sa/myapp` or `mycluster.local/*`)
- Namespace
- IP ranges
- HTTP headers
- JWT tokens from application requests

AuthorizationPolicy resource

For more details on AuthorizationPolicy, refer to the Istio documentation.

The AuthorizationPolicy resource enables and defines access control to workloads. It supports both ALLOW and DENY rules described above.

Arguments for making authorization decisions:

- `source`:

    - `namespace`
    - `principal` (user identifier obtained after authentication)
    - IP
- `destination`:
    - `method` (`GET`, `POST`, etc.)
    - `host`
    - `port`
    - URI
- `conditions`:
    - HTTP headers
    - `source` arguments
    - `destination` arguments
    - JWT tokens

Additional resources

- Module `istio` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Configuring request routing with Istio

You can use the istio module in Deckhouse Platform to route HTTP and TCP requests.

The main resource for managing routing is VirtualService from Istio, which allows you to configure routing for HTTP or TCP requests.

VirtualService resource

For more details on VirtualService, refer to the Istio documentation.

Using VirtualService is optional. Standard Services will continue to work if their functionality is sufficient. With this resource, you can configure request routing:

- Arguments for making routing decisions:
    - `host`
    - `uri`
    - `weight`
- Parameters of the final destinations:
    - New `host`
    - New `uri`
    - If the `host` is defined via a DestinationRule, requests can be routed to subsets
    - Timeout and retry settings

> ⚠ For `destination` in Istio to work correctly, it must be specified explicitly. If you are using an external API, specify it with a ServiceEntry.

Additional resources

- Module `istio` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Enabling Istio for applications

Enabling Istio for applications is possible if the istio module is enabled and configured in the cluster. This is handled by the cluster administrator.

The essence of enabling Istio is adding a sidecar container to the application Pods, after which Istio can manage the traffic.

The recommended way to add sidecars is to use the sidecar injector. Istio can “inject” a sidecar container into application Pods using the Admission Webhook mechanism. Labels and annotations are used to add sidecars:

- A label on a `namespace` marks your namespace for the sidecar injector component. After the label is applied, new Pods will have sidecar containers added:
  - `istio-injection=enabled`: Uses the global Istio version (`spec.settings.globalVersion` in the ModuleConfig resource);
  - `istio.io/rev=v1x16`: Uses a specific Istio version for this namespace.
- A Pod annotation `sidecar.istio.io/inject` (`"true"` or `"false"`) allows you to locally override the `sidecarInjectorPolicy`. These annotations only work in namespaces marked with one of the labels listed above.

It is also possible to add a sidecar to a specific Pod in a namespace without the `istio-injection=enabled` or `istio.io/rev=vXxYZ` labels by setting the label `sidecar.istio.io/inject=true`.

The Istio proxy, which runs as a sidecar container, also consumes resources and adds overhead:

- Each request is DNATed into Envoy, which processes the request and creates another one. On the receiving side, the same happens.
- Each Envoy stores information about all Services in the cluster, which requires memory. The larger the cluster, the more memory Envoy consumes. The solution is the Sidecar custom resource.

It is also important to prepare the Ingress controller and the application’s Ingress resources:

- Enable `enableIstioSidecar` in the IngressNginxController resource.
- Add annotations to the application’s Ingress resources:
  - `nginx.ingress.kubernetes.io/service-upstream: "true"`: The Ingress controller uses the Service’s ClusterIP as the upstream instead of the Pod IPs. Traffic balancing between Pods is now handled by the sidecar proxy. Use this option only if your Service has a ClusterIP.
  - `nginx.ingress.kubernetes.io/upstream-vhost: "myservice.myns.svc"`: The Ingress controller’s sidecar proxy makes routing decisions based on the `Host` header. Without this annotation, the controller will keep the site address header, for example, `Host: example.com`.

Additional resources

- Module `ingress-nginx` custom resources
- Module `istio` documentation

> **i** **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Managing request balancing between service endpoints with Istio

You can use the istio module to manage request load balancing between service endpoints. Before configuring load balancing, make sure the module is enabled in the cluster.

The main resource for managing request load balancing is DestinationRule from Istio. This resource allows you to configure the following parameters:

- TCP limits and timeouts.
- Algorithms for load balancing between endpoints.
- Rules for detecting endpoint issues and removing them from the load balancer.
- Encryption settings.

⚠ All configured limits apply separately to each client Pod. For example, if you set a limit of one TCP connection for a Service and there are three client Pods, the Service will receive three incoming connections.

DestinationRule resource

For more details on DestinationRule, refer to the Istio documentation. Use this resource to:

- Define a traffic load balancing strategy between service endpoints:
  - Load balancing algorithm (`LEAST_CONN`, `ROUND_ROBIN`, etc.).
  - Detection and exclusion of unhealthy endpoints.
  - TCP connection and request limits for endpoints.
  - Sticky Sessions support.
  - Circuit Breaker configuration.
- Define alternative endpoint groups for handling traffic (applicable to Canary Deployments). Each group can have its own load balancing strategy.
- Configure TLS for outgoing requests.

Additional resources

- Module `istio` documentation

i **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**

The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Configuring Locality failover with Istio

In Deckhouse Platform, you can implement the Locality failover mechanism using the istio module. Before configuring the mechanism, make sure the module is enabled in the cluster.

The Locality failover mechanism manages traffic routing and directs it to a priority failover in case certain service instances become unavailable.

i If necessary, refer to the Locality failover documentation.

With Istio, you can configure priority-based geographic failover between endpoints. Zones are determined based on node labels in the following hierarchy:

- `topology.istio.io/subzone`
- `topology.kubernetes.io/zone`
- `topology.kubernetes.io/region`

This is useful for inter-cluster failover when used together with multicluster configurations.

⚠ To enable Locality Failover, use the DestinationRule resource, which must also include the `outlierDetection` parameter.

Example:

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: helloworld
spec:
  host: helloworld
  trafficPolicy:
    loadBalancer:
      localityLbSetting:
        enabled: true # Enable LF.
    outlierDetection: # Required.
      consecutive5xxErrors: 1
      interval: 1s
      baseEjectionTime: 1m
```

Additional resources

- Module `istio` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Configuring request retries with Istio

You can use the istio module to configure request retries. Before configuring retries, make sure the module is enabled in the cluster.

To configure request retries, use the VirtualService resource from Istio.

> ⚠ By default, when errors occur, all requests (including POST requests) are retried up to three times.

Example:

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: ratings-route
spec:
  hosts:
  - ratings.prod.svc.cluster.local
  http:
  - route:
    - destination:
        host: ratings.prod.svc.cluster.local
    retries:
      attempts: 3
      perTryTimeout: 2s
      retryOn: gateway-error,connect-failure,refused-stream
```

VirtualService resource

If necessary, refer to the VirtualService documentation.

Using VirtualService is optional. Standard Services will continue to work if their functionality is sufficient. With this resource, you can configure request routing:

- Arguments for making routing decisions:
  - `host`
  - `uri`
  - `weight`
- Parameters of the final destinations:
  - New `host`
  - New `uri`
  - If the `host` is defined via a DestinationRule, requests can be routed to subsets
  - Timeout and retry settings

⚠ For `destination` in Istio to work correctly, it must be specified explicitly. If you are using an external API, specify it with a ServiceEntry.

Additional resources

- Module `istio` documentation

i **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**

The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Configuring resources for istio-proxy sidecars

When using the istio module in a cluster, you can manage the resources allocated for istio-proxy sidecars in specific workloads. Annotations are used for this purpose.

Supported annotations

To override global resource limits for istio-proxy sidecars, the following annotations are supported in individual workloads:

Annotation	Description	Example value
`sidecar.istio.io/proxyCPU`	CPU request for a sidecar	`200m`
`sidecar.istio.io/proxyCPULimit`	CPU limit for a sidecar	`"1"`
`sidecar.istio.io/proxyMemory`	Memory request for a sidecar	`128Mi`
`sidecar.istio.io/proxyMemoryLimit`	Memory limit for a sidecar	`512Mi`

⚠ All annotations from the table must be specified in the workload manifest at the same time. Partial configuration is not supported.

Configuration examples

For Deployments:

```
apiVersion: apps/v1
kind: Deployment
metadata:
# ...
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/proxyCPU: 200m
        sidecar.istio.io/proxyCPULimit: "1"
        sidecar.istio.io/proxyMemory: 128Mi
        sidecar.istio.io/proxyMemoryLimit: 512Mi
# ... Rest of the manifest.
```

For ReplicaSets:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
# ...
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/proxyCPU: 200m
        sidecar.istio.io/proxyCPULimit: "1"
        sidecar.istio.io/proxyMemory: 128Mi
        sidecar.istio.io/proxyMemoryLimit: 512Mi
# ... Rest of the manifest.
```

For Pods:

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    sidecar.istio.io/proxyCPU: 200m
    sidecar.istio.io/proxyCPULimit: "1"
    sidecar.istio.io/proxyMemory: 128Mi
    sidecar.istio.io/proxyMemoryLimit: 512Mi
# ... Rest of the manifest.
```

Additional resources

- Module `istio` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Configuring Circuit Breaker

In Deckhouse Platform, the Circuit Breaker mechanism is implemented using Istio (the istio module) and provides the following capabilities:

- Temporarily exclude an endpoint from load balancing if the error limit is exceeded.
- Configure limits on the number of TCP connections and the number of requests to a single endpoint.

- Detect stuck requests and terminate them with an error code (HTTP request timeout).

Example Circuit Breaker configuration

To detect problematic endpoints, use the `outlierDetection` settings in the DestinationRule custom resource. The Outlier Detection algorithm is described in more detail in the Envoy documentation.

Example:

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: reviews-cb-policy
spec:
  host: reviews.prod.svc.cluster.local
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 100 # Maximum number of connections to the host, total across
all endpoints.
      http:
        maxRequestsPerConnection: 10 # The connection will be recreated after every 10
requests.
    outlierDetection:
      consecutive5xxErrors: 7 # Allows up to 7 errors (including `5xx`, TCP timeouts,
and HTTP timeouts)
      interval: 5m          # within 5 minutes,
      baseEjectionTime: 15m   # after which the endpoint will be removed from load
balancing for 15 minutes.
```

You can also use the VirtualService resource to configure HTTP timeouts. These timeouts are also taken into account when calculating endpoint error statistics.

Example:

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: my-productpage-rule
  namespace: myns
spec:
  hosts:
  - productpage
  http:
  - timeout: 5s
    route:
    - destination:
        host: productpage
```

Additional resources

- Module `istio` documentation

> **i** **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Configuring Canary deployment

Canary deployment is an application deployment strategy that allows gradually rolling out a new application version to production. This approach makes it possible to test new versions on a small portion of traffic, minimizing risks and ensuring a smooth transition. With Canary deployment, traffic can be shifted to the new version as confidence in its stability grows, with the ability to quickly roll back to the old version if issues arise. In Deckhouse Platform, Canary deployment can be implemented using the ingress-nginx module or the istio module (recommended).

Example Canary deployment configuration with Ingress NGINX

To implement Canary deployment with Ingress NGINX, annotations and rules are used to route a portion of traffic to the new application version.

Creating a Deployment and Service for the stable version

Example manifest for the stable version:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-v1
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
      version: v1
  template:
    metadata:
      labels:
        app: my-app
        version: v1
    spec:
      containers:
      - name: app
        image: app:v1
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: app-service
spec:
  selector:
    app: my-app
  ports:
  - port: 80
    targetPort: 80
```

Creating a Deployment and Service for the Canary version

Example manifest for the Canary version:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
      version: v2
  template:
    metadata:
      labels:
        app: my-app
        version: v2
    spec:
      containers:
      - name: app
        image: app:v2
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: app-canary-service
spec:
  selector:
    app: my-app
    version: v2
  ports:
  - port: 80
    targetPort: 80
```

Configuring Ingress for Canary deployment

For Canary deployment with Ingress NGINX, special annotations are used:

- `nginx.ingress.kubernetes.io/canary`: Enables Canary mode for the Ingress.
- `nginx.ingress.kubernetes.io/canary-weight`: Specifies the percentage of traffic to be routed to the Canary version.

Example Ingress manifest (10% of traffic will go to the Canary version (`app-canary-service`), 90% to the stable version (`app-service`)):

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-ingress
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-weight: "10" # 10% traffic to Canary.
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: app-canary-service
            port:
              number: 80
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-ingress-main
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: app-service
            port:
              number: 80
```

Gradually increasing traffic to the Canary version

You can gradually increase the percentage of traffic to the Canary version by changing the value of the `nginx.ingress.kubernetes.io/canary-weight` annotation. For example, to send 50% of traffic to the Canary version, update the annotation as follows:

```
nginx.ingress.kubernetes.io/canary-weight: "50"
```

Rolling back or completing the Canary deployment

If the Canary version is stable, you can fully switch traffic to the new version by removing the Canary annotations and updating the main Ingress. If problems occur, you can reduce the percentage of traffic going to the Canary version or disable it completely by setting `nginx.ingress.kubernetes.io/canary-weight: "0"`.

Additional annotations for Canary deployment

- `nginx.ingress.kubernetes.io/canary-by-header`: Routes traffic to the Canary version based on the value of an HTTP header.
- `nginx.ingress.kubernetes.io/canary-by-cookie`: Routes traffic to the Canary version based on the value of a cookie.

Example using a header:

```
nginx.ingress.kubernetes.io/canary-by-header: "canary"
nginx.ingress.kubernetes.io/canary-by-header-value: "true"
```

In this case, traffic will be routed to the Canary version if the request contains the header `canary: true`.

Example Canary deployment configuration with Istio

> **i** Istio is responsible only for flexible request routing based on special request headers (for example, cookies) or simple randomness. The actual routing configuration and switching between Canary versions is managed by the CI/CD system.

It is assumed that the same namespace contains two Deployments with different application versions. Pods of different versions have different labels (`version: v1` and `version: v2`).

Two custom resources need to be configured:

- DestinationRule describing how to identify different versions of your application (subsets).
- VirtualService describing how to distribute traffic between different application versions.

Example:

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: productpage-canary
spec:
  host: productpage
  # Subsets are available only when accessing the host through a VirtualService from
a Pod managed by Istio.
  # These subsets must be specified in the routes.
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

Distribution based on the presence of a cookie

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: productpage-canary
spec:
  hosts:
  - productpage
  http:
  - match:
    - headers:
       cookie:
         regex: "^(.*;?)?(canary=yes)(;.*)?"
    route:
    - destination:
        host: productpage
        subset: v2 # Reference to a subset from DestinationRule.
  - route:
    - destination:
        host: productpage
        subset: v1
```

Distribution based on probability

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: productpage-canary
spec:
  hosts:
  - productpage
  http:
  - route:
    - destination:
        host: productpage
        subset: v1 # Reference to a subset from DestinationRule.
      weight: 90 # Percentage of traffic that goes to Pods with the label `version:
v1`.
  - route:
    - destination:
        host: productpage
        subset: v2
      weight: 10
```

Additional resources

- Module `ingress-nginx` documentation
- Module `istio` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Configuration of container image vulnerability scanning

Deckhouse Platform (DP) follows the CIS Kubernetes Benchmark recommendations, ensuring security both at the component level and across the platform as a whole.

To continuously monitor CIS compliance, DP automatically runs checks in every cluster. The results of these checks are presented in reports and available in Grafana on the `Security / CIS Kubernetes Benchmark` dashboard.

DP provides a built-in tool for automated container image vulnerability scanning based on Trivy.

Below are commands for viewing and filtering reports from vulnerability scans and CIS compliance checks performed in the cluster.

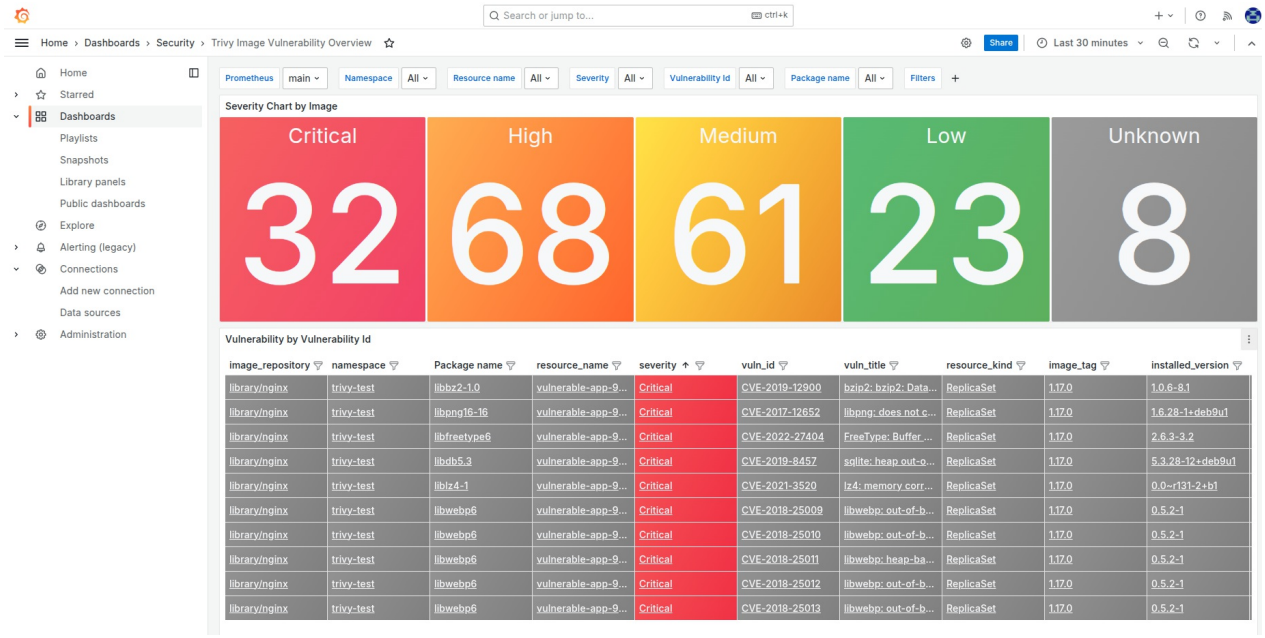
Accessing scan results

The `d8:subsystem:security:viewer` access role or higher grants access to the cluster-wide scan results (for example, ClusterComplianceReport) and to the `operator-trivy` module configuration.

This role and the namespace roles (`d8:namespace:*`) do not grant access to the scan reports in application namespaces (VulnerabilityReport and others): the `operator-trivy` module ships these permissions separately. To get access to such reports, contact the cluster administrator.

Viewing a scan report for your application

To view scan results for your application, use the Grafana dashboard `Security / Trivy Image Vulnerability Overview`. You can filter the results by namespace and resource.



Viewing CIS compliance check results

- To list all resources that failed the check, run:

```
d8 k get clustercompliancereports.aquasecurity.github.io cis -ojson |
  jq '.status.detailReport.results | map(select(.checks | map(.success) | all |
not))'
```

- To search by a specific check ID:

```
check_id="5.7.3"
d8 k get clustercompliancereports.aquasecurity.github.io cis -ojson |
  jq --arg check_id "$check_id" '.status.detailReport.results | map(select(.id ==
$check_id))'
```

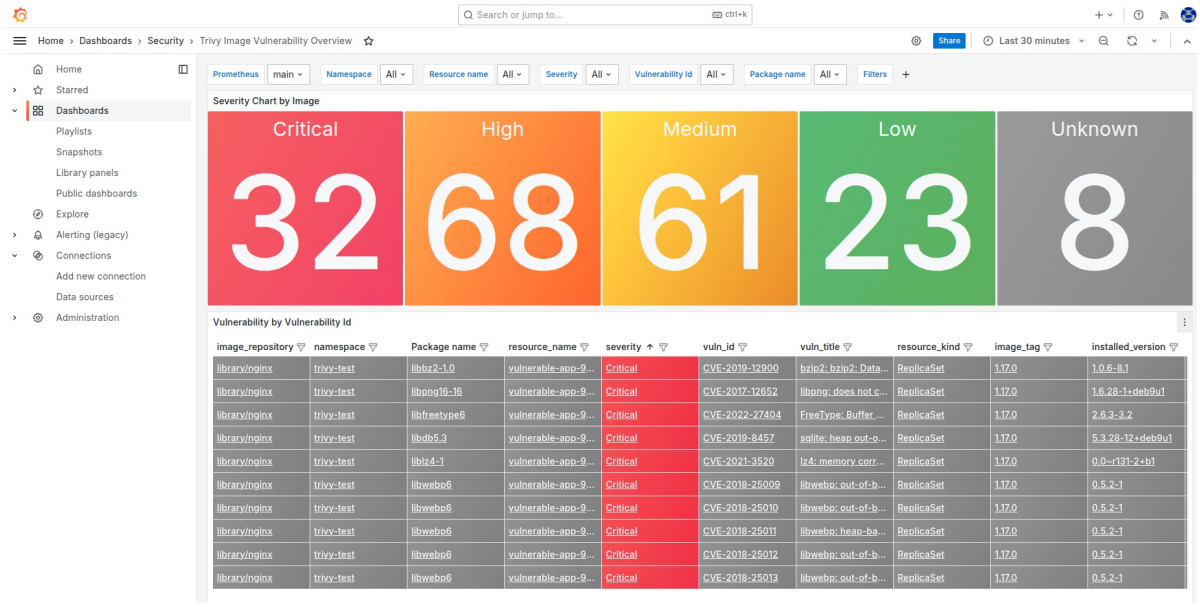
- To search by check description:

```
check_desc="Apply Security Context to Your Pods and Containers"
d8 k get clustercompliancereports.aquasecurity.github.io cis -ojson |
  jq --arg check_desc "$check_desc" '.status.detailReport.results |
map(select(.description == $check_desc))'
```

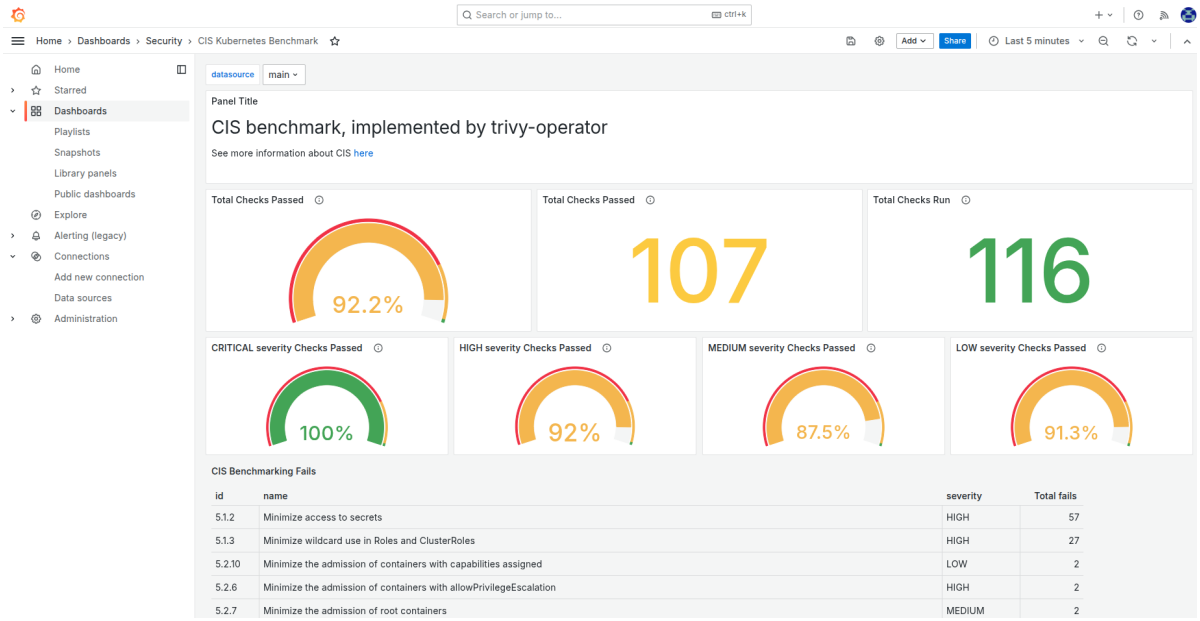
Viewing scan results

Grafana dashboards:

- `Security / Trivy Image Vulnerability Overview`: A summary of vulnerabilities in container images running in the cluster.



- `Security / CIS Kubernetes Benchmark`: Results of cluster compliance checks against the CIS Kubernetes Benchmark.



id	name	severity	Total fails
5.1.2	Minimize access to secrets	HIGH	57
5.1.3	Minimize wildcard use in Roles and ClusterRoles	HIGH	27
5.2.10	Minimize the admission of containers with capabilities assigned	LOW	2
5.2.6	Minimize the admission of containers with allowPrivilegeEscalation	HIGH	2
5.2.7	Minimize the admission of root containers	MEDIUM	2

In the cluster:

- Cluster security reports:
  - ClusterComplianceReport
  - RbacAssessmentReport
- Workload security reports in the cluster:
  - VulnerabilityReport: Vulnerabilities in container images.
  - SbomReport: Software composition in images (SBOM).
  - ConfigAuditReport: Misconfigurations in Kubernetes objects.
  - ExposedSecretReport: Exposed secrets in containers.

DP uses a set of custom resources developed by the Aqua Security Trivy Operator project to represent vulnerability scan results, configuration analysis, and cluster compliance checks.

Below is a description of the key CRDs created by operator-trivy, with examples and links to official documentation.

Resource types for scan results

Resource-level security

VulnerabilityReport

VulnerabilityReport is a resource containing a report of vulnerabilities detected in a container image used by a Kubernetes workload.

The report includes a list of known vulnerabilities in OS packages and applications, grouped by severity levels (Critical, High, Medium, etc.).

For each container in a multi-container workload, operator-trivy creates a separate VulnerabilityReport in the workload's namespace. The association with the Kubernetes resource is defined via the `ownerReference` field.

Resource names are formed based on the following pattern: `<workload-type>-<workload-name>-<container-name>`.

Example:

```
apiVersion: aquasecurity.github.io/v1alpha1
kind: VulnerabilityReport
metadata:
  name: replicaset-nginx-6d4cf56db6-nginx
  namespace: default
  labels:
    trivy-operator.container.name: nginx
    trivy-operator.resource.kind: ReplicaSet
    trivy-operator.resource.name: nginx-6d4cf56db6
    trivy-operator.resource.namespace: default
    resource-spec-hash: 7cb64cb677
  ownerReferences:
    - apiVersion: apps/v1
      kind: ReplicaSet
      name: nginx-6d4cf56db6
      uid: aa345200-cf24-443a-8f11-ddb438ff8659
      controller: true
      blockOwnerDeletion: false
report:
  artifact:
    repository: library/nginx
    tag: '1.16'
  os:
    family: debian
    name: '10.3'
  registry:
    server: index.docker.io
  scanner:
    name: Trivy
    vendor: Aqua Security
    version: 0.35.0
  summary:
    criticalCount: 2
    highCount: 0
    mediumCount: 0
    lowCount: 0
    unknownCount: 0
  vulnerabilities:
    - vulnerabilityID: CVE-2019-20367
      resource: libbsd0
      installedVersion: 0.9.1-2
      fixedVersion: 0.9.1-2+deb10u1
      severity: CRITICAL
      score: 9.1
      target: library/nginx:1.21.6
      primaryLink: https://avd.aquasec.com/nvd/cve-2019-20367
    - vulnerabilityID: CVE-2018-25009
      resource: libwebp6
      installedVersion: 0.6.1-2
      fixedVersion: ''
      severity: CRITICAL
      score: 9.1
      target: library/nginx:1.16
      title: 'libwebp: out-of-bounds read in WebPMuxCreateInternal'
      primaryLink: https://avd.aquasec.com/nvd/cve-2018-25009
```

ConfigAuditReport

ConfigAuditReport is a resource containing results of Kubernetes object configuration checks performed with auditing tools such as Trivy.

The report includes a list of configuration findings, grouped by categories (for example, Security) and severity levels (Critical, High, etc.).

Examples of checks:

- Container runs as a non-root user.
- Container has resource limits.
- Network access configuration (hostNetwork, hostPID, etc.).
- Presence of security flags preventing privilege escalation.

A ConfigAuditReport is generated for any namespaced resource, including:

- Workloads (Pod, Deployment, StatefulSet, etc.)
- Auxiliary resources (Service, ConfigMap, Role, RoleBinding, etc.)

Each report is linked to the checked object via `ownerReference` and stored in the same namespace. Resource names are formed based on the following pattern: `<workload-type>-<workload-name>`.

Example:

```
apiVersion: aquasecurity.github.io/v1alpha1
kind: ConfigAuditReport
metadata:
  name: replicaset-nginx-6d4cf56db6
  namespace: default
  labels:
    trivy-operator.resource.kind: ReplicaSet
    trivy-operator.resource.name: nginx-6d4cf56db6
    trivy-operator.resource.namespace: default
  ownerReferences:
    - apiVersion: apps/v1
      kind: ReplicaSet
      name: nginx-6d4cf56db6
report:
  scanner:
    name: Trivy
    vendor: Aqua Security
    version: '0.22.0'
  summary:
    criticalCount: 2
    highCount: 0
    mediumCount: 0
    lowCount: 9
  checks:
    - checkID: hostPIDSet
      severity: CRITICAL
      messages: ["Host PID is not configured"]
      success: true
    - checkID: notReadOnlyRootFilesystem
      severity: LOW
      messages: ["Filesystem should be read only"]
      success: false
      scope:
        type: Container
        value: nginx
```

ExposedSecretReport

ExposedSecretReport is a resource containing results of scanning for potential secrets detected in a container image used by a Kubernetes workload.

The report lists strings containing sensitive data (for example, tokens, keys, passwords) found in files within the image. Each finding includes a category, rule, severity level, and file path.

For each container in a multi-container workload, operator-trivy creates a separate ExposedSecretReport in the workload's namespace. The association with the Kubernetes resource is

established via `ownerReference`.

Resource names are formed based on the following pattern: `<workload-type>-<workload-name>-<container-name>`.

Example:

```
apiVersion: aquasecurity.github.io/v1alpha1
kind: ExposedSecretReport
metadata:
  name: replicaset-app-67b77f5965-app
  namespace: default
  labels:
    trivy-operator.container.name: app
    trivy-operator.resource.kind: ReplicaSet
    trivy-operator.resource.name: app-67b77f5965
    trivy-operator.resource.namespace: default
  ownerReferences:
    - apiVersion: apps/v1
      kind: ReplicaSet
      name: app-67b77f5965
report:
  artifact:
    repository: myimagewithsecret
    tag: v0.22.0
  registry:
    server: index.docker.io
  scanner:
    name: Trivy
    vendor: Aqua Security
    version: 0.35.0
  secrets:
    - category: Stripe
      ruleID: stripe-access-token
      severity: HIGH
      target: "/app/config/secret.yaml"
      match: "publishable_key: *****"
      title: Stripe
    - category: Stripe
      ruleID: stripe-access-token
      severity: HIGH
      target: "/app/config/secret.yaml"
      match: "secret_key: *****"
      title: Stripe
  summary:
    criticalCount: 0
    highCount: 2
    mediumCount: 0
    lowCount: 0
  updateTimestamp: "2022-06-29T14:29:37Z"
```

SbomReport

SbomReport is a resource containing a Software Bill of Materials (SBOM) report for a container image used by a Kubernetes workload.

It provides a complete list of software components, including system packages and application dependencies found in the container. This information is useful for image composition analysis, security auditing, and vendor compliance requirements.

For multi-container workloads, operator-trivy creates a separate SbomReport for each container. Reports are generated in the same namespace as the workload and linked via `ownerReference`.

Resource names are formed based on the following pattern: `<workload-type>-<workload-name>-`

`<container-name>`.

Example:

```
apiVersion: aquasecurity.github.io/v1alpha1
kind: SbomReport
metadata:
  creationTimestamp: "2023-07-10T09:37:21Z"
  generation: 1
  labels:
    resource-spec-hash: 796669cd5d
    trivy-operator.container.name: kube-apiserver
    trivy-operator.resource.kind: Pod
    trivy-operator.resource.name: kube-apiserver-kind-control-plane
    trivy-operator.resource.namespace: kube-system
  name: pod-kube-apiserver-kind-control-plane-kube-apiserver
  namespace: kube-system
  ownerReferences:
  - apiVersion: v1
    blockOwnerDeletion: false
    controller: true
    kind: Pod
    name: kube-apiserver-kind-control-plane
    uid: 732b4aa7-91f8-40a3-8b21-9627a98a910b
  resourceVersion: "6148"
  uid: 2a5000fe-b97e-46d0-9de7-62fb5fbc6555
report:
  artifact:
    repository: kube-apiserver
    tag: v1.21.1
  components:
    bomFormat: CycloneDX
    components:
    - bom-ref: 9464f5f9-750d-4ea0-8705-c8d067b25b29
      name: debian
      properties:
      - name: aquasecurity:trivy:Class
        value: os-pkgs
      - name: aquasecurity:trivy:Type
        value: debian
      supplier: {}
      type: operating-system
      version: "10.9"
    - bom-ref: pkg:deb/debian/base-files@10.3+deb10u9?arch=amd64&distro=debian-10.9
      licenses:
      - expression: GPL-3.0
        license: {}
      name: base-files
      properties:
      - name: aquasecurity:trivy:LayerDiffID
        value: sha256:417cb9b79adeec55f58b890dc9831e252e3523d8de5fd28b4ee2abb151b7dc8b
      - name: aquasecurity:trivy:LayerDigest
        value: sha256:5dea5ec2316d4a067b946b15c3c4f140b4f2ad607e73e9bc41b673ee5ebb99a3
      - name: aquasecurity:trivy:PkgID
        value: base-files@10.3+deb10u9
      - name: aquasecurity:trivy:PkgType
        value: debian
      - name: aquasecurity:trivy:SrcName
        value: base-files
      - name: aquasecurity:trivy:SrcVersion
        value: 10.3+deb10u9
      purl: pkg:deb/debian/base-files@10.3+deb10u9?arch=amd64&distro=debian-10.9
```

```
      supplier:
        name: Santiago Vila <sanvila@debian.org>
      type: library
      version: 10.3+deb10u9
    - bom-ref: pkg:deb/debian/netbase@5.6?arch=all&distro=debian-10.9
      licenses:
      - expression: GPL-2.0
        license: {}
      name: netbase
      properties:
      - name: aquasecurity:trivy:LayerDiffID
        value: sha256:417cb9b79adeec55f58b890dc9831e252e3523d8de5fd28b4ee2abb151b7dc8b
      - name: aquasecurity:trivy:LayerDigest
        value: sha256:5dea5ec2316d4a067b946b15c3c4f140b4f2ad607e73e9bc41b673ee5ebb99a3
      - name: aquasecurity:trivy:PkgID
        value: netbase@5.6
      - name: aquasecurity:trivy:PkgType
        value: debian
      - name: aquasecurity:trivy:SrcName
        value: netbase
      - name: aquasecurity:trivy:SrcVersion
        value: "5.6"
      purl: pkg:deb/debian/netbase@5.6?arch=all&distro=debian-10.9
      supplier:
        name: Marco d'Itri <md@linux.it>
      type: library
      version: "5.6"
    - bom-ref: pkg:deb/debian/tzdata@2021a-0+deb10u1?arch=all&distro=debian-10.9
      name: tzdata
      properties:
      - name: aquasecurity:trivy:LayerDiffID
        value: sha256:417cb9b79adeec55f58b890dc9831e252e3523d8de5fd28b4ee2abb151b7dc8b
      - name: aquasecurity:trivy:LayerDigest
        value: sha256:5dea5ec2316d4a067b946b15c3c4f140b4f2ad607e73e9bc41b673ee5ebb99a3
      - name: aquasecurity:trivy:PkgID
        value: tzdata@2021a-0+deb10u1
      - name: aquasecurity:trivy:PkgType
        value: debian
      - name: aquasecurity:trivy:SrcName
        value: tzdata
      - name: aquasecurity:trivy:SrcRelease
        value: 0+deb10u1
      - name: aquasecurity:trivy:SrcVersion
        value: 2021a
      purl: pkg:deb/debian/tzdata@2021a-0+deb10u1?arch=all&distro=debian-10.9
      supplier:
        name: GNU Libc Maintainers <debian-glibc@lists.debian.org>
      type: library
      version: 2021a-0+deb10u1
    dependencies:
    - dependsOn:
      - pkg:deb/debian/base-files@10.3+deb10u9?arch=amd64&distro=debian-10.9
      - pkg:deb/debian/netbase@5.6?arch=all&distro=debian-10.9
      - pkg:deb/debian/tzdata@2021a-0+deb10u1?arch=all&distro=debian-10.9
      ref: 9464f5f9-750d-4ea0-8705-c8d067b25b29
    - dependsOn: []
      ref: pkg:deb/debian/base-files@10.3+deb10u9?arch=amd64&distro=debian-10.9
    - dependsOn: []
      ref: pkg:deb/debian/netbase@5.6?arch=all&distro=debian-10.9
    - dependsOn: []
      ref: pkg:deb/debian/tzdata@2021a-0+deb10u1?arch=all&distro=debian-10.9
    - dependsOn:
```

```
      - 9464f5f9-750d-4ea0-8705-c8d067b25b29
      ref: pkg:oci/kube-
apiserver@sha256:53a13cd1588391888c5a8ac4cef13d3ee6d229cd904038936731af7131d193a9?
repository_url=k8s.gcr.io%2Fkube-apiserver&arch=amd64
    metadata:
      component:
        bom-ref: pkg:oci/kube-
apiserver@sha256:53a13cd1588391888c5a8ac4cef13d3ee6d229cd904038936731af7131d193a9?
repository_url=k8s.gcr.io%2Fkube-apiserver&arch=amd64
        name: k8s.gcr.io/kube-apiserver:v1.21.1
        properties:
        - name: aquasecurity:trivy:DiffID
          value:
sha256:417cb9b79adeec55f58b890dc9831e252e3523d8de5fd28b4ee2abb151b7dc8b,sha256:b5013
1762317bbe47def2d426d5c78a353a08b966d36bed4a04aee99dde4e12b,sha256:1e6ed7621dee7e03d
d779486ed469a65af6fb13071d13bd3a89c079683e3b1f0
        - name: aquasecurity:trivy:ImageID
          value:
sha256:771ffcf9ca634e37cbd3202fd86bd7e2df48ecba4067d1992541bfa00e88a9bb
        - name: aquasecurity:trivy:RepoDigest
          value: k8s.gcr.io/kube-
apiserver@sha256:53a13cd1588391888c5a8ac4cef13d3ee6d229cd904038936731af7131d193a9
        - name: aquasecurity:trivy:RepoTag
          value: k8s.gcr.io/kube-apiserver:v1.21.1
        - name: aquasecurity:trivy:SchemaVersion
          value: "2"
        purl: pkg:oci/kube-
apiserver@sha256:53a13cd1588391888c5a8ac4cef13d3ee6d229cd904038936731af7131d193a9?
repository_url=k8s.gcr.io%2Fkube-apiserver&arch=amd64
        supplier: {}
        type: container
      timestamp: "2023-07-10T09:37:21+00:00"
      tools:
      - name: trivy
        vendor: aquasecurity
    serialNumber: urn:uuid:50dbce86-28c5-4caf-9d08-a4aadf23233e
    specVersion: 1.4
    version: 1
  registry:
    server: k8s.gcr.io
  scanner:
    name: Trivy
    vendor: Aqua Security
    version: 0.52.2
  summary:
    componentsCount: 5
    dependenciesCount: 5
  updateTimestamp: "2023-07-10T09:37:21Z
```


Cluster-level security

RbacAssessmentReport

RbacAssessmentReport is a resource containing a report generated from analyzing Kubernetes RBAC (Role-Based Access Control) settings.

It includes results of configuration audits performed with tools such as Trivy. Examples of findings include detection of roles that:

- Grant excessive privileges (for example, full access to secrets across all API groups).
- Violate the principle of least privilege.

Each report is associated with a specific Role or ClusterRole and stored in the same namespace as the

evaluated object.

Resource names follow the pattern: `<Role|ClusterRole>-<role-name>`.

Example:

```
apiVersion: aquasecurity.github.io/v1alpha1
kind: RbacAssessmentReport
metadata:
  name: role-868458b9d6
  namespace: kube-system
report:
  checks:
    - category: Kubernetes Security Check
      checkID: KSV051
      description: Check whether role permits creating role bindings and associating
        to privileged role/clusterrole
      messages:
        - ""
      severity: HIGH
      success: true
      title: Do not allow role binding creation and association with privileged
role/clusterrole
    - category: Kubernetes Security Check
      checkID: KSV056
      description: The ability to control which pods get service traffic directed to
        them allows for interception attacks. Controlling network policy allows for
        bypassing lateral movement restrictions.
      messages:
        - ""
      severity: HIGH
      success: true
      title: Do not allow management of networking resources
    - category: Kubernetes Security Check
      checkID: KSV041
      description: Check whether role permits managing secrets
      messages:
        - Role permits management of secret(s)
      severity: CRITICAL
      success: false
      title: Do not allow management of secrets
    - category: Kubernetes Security Check
      checkID: KSV047
      description: Check whether role permits privilege escalation from node proxy
      messages:
        - ""
      severity: HIGH
      success: true
      title: Do not allow privilege escalation from node proxy
    - category: Kubernetes Security Check
      checkID: KSV045
      description: Check whether role permits wildcard verb on specific resources
      messages:
        - ""
      severity: CRITICAL
      success: true
      title: No wildcard verb roles
    - category: Kubernetes Security Check
      checkID: KSV054
      description: Check whether role permits attaching to shell on pods
      messages:
        - ""
      severity: HIGH
```

```
      success: true
      title: Do not allow attaching to shell on pods
    - category: Kubernetes Security Check
      checkID: KSV044
      description: Check whether role permits wildcard verb on wildcard resource
      messages:
        - ""
      severity: CRITICAL
      success: true
      title: No wildcard verb and resource roles
    - category: Kubernetes Security Check
      checkID: KSV050
      description: An effective level of access equivalent to cluster-admin should
not
        be provided.
      messages:
        - ""
      severity: CRITICAL
      success: true
      title: Do not allow management of RBAC resources
    - category: Kubernetes Security Check
      checkID: KSV046
      description: Check whether role permits specific verb on wildcard resources
      messages:
        - ""
      severity: CRITICAL
      success: true
      title: No wildcard resource roles
    - category: Kubernetes Security Check
      checkID: KSV055
      description: Check whether role permits allowing users in a rolebinding to add
        other users to their rolebindings
      messages:
        - ""
      severity: LOW
      success: true
      title: Do not allow users in a rolebinding to add other users to their
rolebindings
    - category: Kubernetes Security Check
      checkID: KSV052
      description: Check whether role permits creating role ClusterRoleBindings and
        association with privileged cluster role
      messages:
        - ""
      severity: HIGH
      success: true
      title: Do not allow role to create ClusterRoleBindings and association with
privileged
        role
    - category: Kubernetes Security Check
      checkID: KSV053
      description: Check whether role permits getting shell on pods
      messages:
        - ""
      severity: HIGH
      success: true
      title: Do not allow getting shell on pods
    - category: Kubernetes Security Check
      checkID: KSV042
      description: Used to cover attacker’s tracks, but most clusters ship logs
quickly
        off-cluster.
```

```
      messages:
        - ""
      severity: MEDIUM
      success: true
      title: Do not allow deletion of pod logs
    - category: Kubernetes Security Check
      checkID: KSV049
      description: Some workloads leverage configmaps to store sensitive data or
configuration
        parameters that affect runtime behavior that can be modified by an attacker
        or combined with another issue to potentially lead to compromise.
      messages:
        - ""
      severity: MEDIUM
      success: true
      title: Do not allow management of configmaps
    - category: Kubernetes Security Check
      checkID: KSV043
      description: Check whether role permits impersonating privileged groups
      messages:
        - ""
      severity: CRITICAL
      success: true
      title: Do not allow impersonation of privileged groups
    - category: Kubernetes Security Check
      checkID: KSV048
      description: Check whether role permits update/create of a malicious pod
      messages:
        - ""
      severity: HIGH
      success: true
      title: Do not allow update/create of a malicious pod
  scanner:
    name: Trivy
    vendor: Aqua Security
    version: '0.22.0'
  summary:
    criticalCount: 1
    highCount: 0
    lowCount: 0
    mediumCount: 0
  updateTimestamp: null
```

ClusterComplianceReport

ClusterComplianceReport is a resource containing a summary of cluster compliance with security requirements.

Currently, compliance checks include the CIS Kubernetes Benchmark, a set of best practices for securely configuring Kubernetes components.

Report structure:

- `spec.compliance.controls`: Description of the evaluated criteria.
- `status`: Results of the checks matching the description under `spec`. These results are based on aggregated reports from various security scanners.

> **i** You can also review the compliance report in the Grafana dashboard `Security / CIS Kubernetes Benchmark`.

Example:

```
apiVersion: aquasecurity.github.io/v1alpha1
kind: ClusterComplianceReport
metadata:
  name: cis
spec:
  compliance:
    controls:
    - checks:
      - id: AVD-KCV-0048
      commands:
      - id: CMD-0001
      description: Ensure that the API server pod specification file has permissions
        of 600 or more restrictive
      id: 1.1.1
      name: Ensure that the API server pod specification file permissions are set
        to 600 or more restrictive
      severity: HIGH
    - checks:
      - id: AVD-KCV-0049
      commands:
      - id: CMD-0002
      description: Ensure that the API server pod specification file ownership is
        set to root:root
      id: 1.1.2
      name: Ensure that the API server pod specification file ownership is set to
        root:root
      severity: HIGH
...
  summary:
    failCount: 9
    passCount: 107
  updateTimestamp: "2025-07-29T06:00:00Z"
```

Additional resources

- Module `operator-trivy` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Configuration of using TLS certificates

Deckhouse Platform (DP) provides built-in tools for managing TLS certificates, simplifying setup and management of traffic encryption for applications running in the cluster.

This page covers the following aspects of certificate usage in DP:

- Manually requesting TLS certificates using the Certificate and ClusterIssuer resources.
- Securely storing and using credentials for accessing certificate authorities (CAs).
- Automatically obtaining certificates using the `tls-acme` annotation in Ingress resources.

> For a general overview of certificate management in DP, the list of supported issuers, and setup recommendations, refer to Certificate management.

Working with certificates

Viewing certificate details

- To list all certificates in the cluster, use the following command:

```
d8 k get certificate --all-namespaces
```

- To check the status of a specific certificate, run:

```
d8 k -n <NAMESPACE> describe certificate <CERTIFICATE-NAME>
```

Automatic certificate requesting

To request a `letsencrypt` certificate, follow these steps:

1. Create a Certificate resource as described in the `cert-manager` documentation. Use the following example as a reference:

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: example-com          # Certificate name.
  namespace: default
spec:
  secretName: example-com-tls  # Name of the Secret that will store the private
key and the certificate.
  issuerRef:
    kind: ClusterIssuer      # Certificate issuer details.
    name: letsencrypt
  commonName: example.com     # Certificate main domain.
  dnsNames:                 # Optional additional certificate domains (at least
one DNS name or IP address).
  - www.example.com
  - admin.example.com
```

2. The `cert-manager` module will automatically run a domain ownership challenge using the method defined in the ClusterIssuer resource (for example, `HTTP-01` or `DNS-01`).
3. The `cert-manager` module will automatically create a temporary Ingress resource for the challenge. This temporary Ingress will not affect your existing Ingress configuration.
4. Once validated, the issued certificate will be stored in the Secret specified in the `secretName` field.

> **i**
>
> If you see a `CAA record does not match issuer` error during certificate issuance, check the DNS records for the domain. To use `letsencrypt`, your domain must have the following CAA record: `issue "letsencrypt.org"`.
>
> For mode details on CAA records, refer to Let's Encrypt documentation.

Requesting a wildcard certificate using DNS in Cloudflare

1. Obtain your `GlobalAPIKey` and `Email`:
   - Go to `dash.cloudflare.com/profile`.
   - Your email is shown at the top under **Email Address**.
   - To view the key, click **View** next to **Global API Key** at the bottom.
2. Edit the `cert-manager` module settings, adding the following section:

```
settings:
  cloudflareGlobalAPIKey: APIkey
  cloudflareEmail: some@mail.somedomain
```

   Or, use an API token instead (recommended):

```
settings:
  cloudflareAPIToken: some-token
  cloudflareEmail: some@mail.somedomain
```

DP will automatically create a ClusterIssuer and Secret for Cloudflare in the `d8-cert-manager` namespace.

3. Create a Certificate resource using Cloudflare for DNS validation. This option becomes available only after the `cloudflareGlobalAPIKey` and `cloudflareEmail` parameters are configured:

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: domain-wildcard
  namespace: app-namespace
spec:
  secretName: tls-wildcard
  issuerRef:
    name: cloudflare
    kind: ClusterIssuer
  commonName: "*.domain.com"
  dnsNames:
  - "*.domain.com"
```

4. Create the Ingress resource:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: domain-wildcard
  namespace: app-namespace
spec:
  ingressClassName: nginx
  rules:
  - host: "*.domain.com"
    http:
      paths:
      - backend:
          service:
            name: svc-web
            port:
              number: 80
        path: /
  tls:
  - hosts:
    - "*.domain.com"
    secretName: tls-wildcard
```

Requesting a wildcard certificate using DNS in AWS Route53

1. Create a user with the required permissions:
    - On the policy management page, create a policy with the following permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "route53:GetChange",
            "Resource": "arn:aws:route53:::change/*"
        },
        {
            "Effect": "Allow",
            "Action": "route53:ChangeResourceRecordSets",
            "Resource": "arn:aws:route53:::hostedzone/*"
        },
        {
            "Effect": "Allow",
            "Action": "route53:ListHostedZonesByName",
            "Resource": "*"
        }
    ]
}
```

- Open the user management page and create a user with the above policy.

2. Edit the `cert-manager` module settings to add the following section:

   ```
   settings:
     route53AccessKeyID: <ACCESS_KEY_ID>
     route53SecretAccessKey: <SECRET_ACCESS_KEY>
   ```

   DP will automatically create a ClusterIssuer and Secret for Route53 in the `d8-cert-manager` namespace.

3. Create a Certificate resource for validation with Route53. This option becomes available only after the `route53AccessKeyID` and `route53SecretAccessKey` parameters are configured:

   ```
   apiVersion: cert-manager.io/v1
   kind: Certificate
   metadata:
     name: domain-wildcard
     namespace: app-namespace
   spec:
     secretName: tls-wildcard
     issuerRef:
       name: route53
       kind: ClusterIssuer
     commonName: "*.domain.com"
     dnsNames:
     - "*.domain.com"
   ```

Requesting a wildcard certificate using DNS in Google

1. Create a ServiceAccount with the required role:

   - Open the policy management page.
   - Choose the target project and create a ServiceAccount under any name (for example, `dns01-solver`).
   - In the created account, create a key by clicking **Add key**. A JSON file with the key data will be downloaded.

   - Encode the JSON file as a Base64 string:

     ```
     base64 project-209317-556c656b81c4.json
     ```

2. Save the Base64 string in the `cloudDNSServiceAccount` parameter.

   DP will automatically create a ClusterIssuer and Secret for CloudDNS in the `d8-cert-manager` namespace.

3. Create a Certificate resource using CloudDNS for validation:

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: domain-wildcard
  namespace: app-namespace
spec:
  secretName: tls-wildcard
  issuerRef:
    name: clouddns
    kind: ClusterIssuer
  dnsNames:
  - "*.domain.com"
```

Requesting a self-signed certificate

To issue a self-signed certificate, specify `selfsigned` as the value for `issuerRef.name`:

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: example-com          # Certificate name.
  namespace: default
spec:
  secretName: example-com-tls  # Name of the Secret that will store the private key
and the certificate.
  issuerRef:
    kind: ClusterIssuer      # Certificate issuer details.
    name: selfsigned
  commonName: example.com     # Certificate main domain.
  dnsNames:                # Optional additional certificate domains (at least one
DNS name or IP address).
  - www.example.com
  - admin.example.com
```

Generating a self-signed certificate

When generating certificates manually, it is important to fill out all fields of the certificate request correctly to ensure that the final certificate is issued properly and can be validated across various services.

It is important to follow these guidelines:

1. Specify domain names in the `SAN` (Subject Alternative Name) field.

   The `SAN` field is a more modern and commonly used method for specifying the domain names covered by the certificate. Some services no longer consider the `CN` (Common Name) field as the source for domain names.

2. Correctly fill out the `keyUsage`, `basicConstraints`, `extendedKeyUsage` fields, specifically:

   - `basicConstraints = CA:FALSE`

     This field determines whether the certificate is an end-entity certificate or a certification authority (CA) certificate. CA certificates cannot be used as service certificates.

   - `keyUsage = digitalSignature, keyEncipherment`

     The `keyUsage` field limits the permissible usage scenarios of this key:

     - `digitalSignature`: Allows the key to be used for signing digital messages and

ensuring data integrity.
    - `keyEncipherment`: Allows the key to be used for encrypting other keys, which is necessary for secure data exchange using TLS (Transport Layer Security).
  - `extendedKeyUsage = serverAuth`

    The `extendedKeyUsage` field specifies additional key usage scenarios required by specific protocols or applications:

    - `serverAuth`: Indicates that the certificate is intended for server use, authenticating the server to the client during the establishment of a secure connection.

It is also recommended to:

1. Issue the certificate for no more than 1 year (365 days).

   The validity period of the certificate affects its security. A one-year validity ensures the cryptographic methods remain current and allows for timely certificate updates in case of threats. Furthermore, some modern browsers now reject certificates with a validity period longer than 1 year.

2. Use robust cryptographic algorithms, such as elliptic curve algorithms (including `prime256v1`).

   Elliptic curve algorithms (ECC) provide a high level of security with a smaller key size compared to traditional methods like RSA. This makes the certificates more efficient in terms of performance and secure in the long term.

3. Do not specify domains in the `CN` (Common Name) field.

   Historically, the `CN` field was used to specify the primary domain name for which the certificate was issued. However, modern standards, such as RFC 2818, recommend using the `SAN` (Subject Alternative Name) field for this purpose. If the certificate is intended for multiple domain names listed in the `SAN` field, specifying one of the domains additionally in `CN` can cause a validation error in some services when accessing domains not listed in `CN`. If non-domain-related information is specified in `CN` (for example, an identifier or service name), the certificate will also extend to these names, which could be exploited for malicious purposes.

Certificate generation example

To generate a certificate, we'll use the `openssl` utility.

1. Fill in the `cert.cnf` configuration file:

```
[ req ]
default_bits       = 2048
default_md         = sha256
prompt             = no
distinguished_name = dn
req_extensions     = req_ext
[ dn ]
C = GB
ST = London
L = London
O = Example Company
OU = IT Department
# CN = Do not specify the CN field.
[ req_ext ]
subjectAltName = @alt_names
[ alt_names ]
# Specify all domain names.
DNS.1 = example.co.uk
DNS.2 = www.example.co.uk
DNS.3 = api.example.co.uk
# Specify IP addresses (if required).
IP.1 = 192.0.2.1
IP.2 = 192.0.4.1
[ v3_ca ]
basicConstraints = CA:FALSE
keyUsage = digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth
[ v3_req ]
basicConstraints = CA:FALSE
keyUsage = digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth
subjectAltName = @alt_names
# Elliptic curve parameters.
[ ec_params ]
name = prime256v1
```

2. Generate an elliptic curve key:

```
openssl ecparam -genkey -name prime256v1 -noout -out ec_private_key.pem
```

3. Create a certificate signing request:

```
openssl req -new -key ec_private_key.pem -out example.csr -config cert.cnf
```

4. Generate a self-signed certificate:

```
openssl x509 -req -in example.csr -signkey ec_private_key.pem -out example.crt -days 365 -
extensions v3_req -extfile cert.cnf
```

Protecting credentials

If you prefer not to store credentials in the DP configuration, you can create a separate Secret and reference it in the ClusterIssuer resource.

To do this, follow these steps:

1. Create a Secret with the access key:

```
d8 k apply -f - <<EOF
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: route53
  namespace: default
data:
  secret-access-key: MY-AWS-ACCESS-KEY-TOKEN
EOF
```

2. Create a ClusterIssuer that references the Secret:

```
d8 k apply -f - <<EOF
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: route53
  namespace: default
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: route53-tls-key
    solvers:
    - dns01:
        route53:
          region: us-east-1
          accessKeyID: MY-AWS-ACCESS-KEY-ID
          secretAccessKeySecretRef:
            name: route53
            key: secret-access-key
EOF
```

3. Request certificates as usual, using the created ClusterIssuer:

```
d8 k apply -f - <<EOF
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: example-com
  namespace: default
spec:
  secretName: example-com-tls
  issuerRef:
    name: route53
  commonName: www.example.com
  dnsNames:
  - www.example.com
EOF
```

Support for tls-acme annotation

DP supports the annotation `kubernetes.io/tls-acme: "true"` in Ingress resources. The `cert-manager-ingress-shim` component watches for this annotation and automatically creates Certificate resources in the same namespace as the Ingress.

> ⚠ When using the annotation, the Certificate resource is linked to the existing Ingress. No separate Ingress is created for domain validation; instead, additional records are added to the existing Ingress. If the Ingress has authentication or a whitelist configured, the validation will fail. Therefore, it’s recommended that you use a Certificate resource directly instead.
>
> If switching from annotation to a manual Certificate, make sure to delete the automatically generated Certificate first. Otherwise, both will try to update the same Secret, potentially exceeding Let’s Encrypt rate limits.

Example Ingress configuration with the annotation:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/tls-acme: "true"         # Annotation.
  name: example-com
  namespace: default
spec:
  ingressClassName: nginx
  rules:
  - host: example.com
    http:
      paths:
      - backend:
          service:
            name: site
            port:
              number: 80
        path: /
        pathType: ImplementationSpecific
  - host: www.example.com                # Additional domain.
    http:
      paths:
      - backend:
          service:
            name: site
            port:
              number: 80
        path: /
        pathType: ImplementationSpecific
  - host: admin.example.com               # Another additional domain.
    http:
      paths:
      - backend:
          service:
            name: site
            port:
              number: 80
        path: /
        pathType: ImplementationSpecific
  tls:
  - hosts:
    - example.com
    - www.example.com                   # Additional domain.
    - admin.example.com                  # Another additional domain.
    secretName: example-com-tls           # Name of the Certificate and Secret.
```

Additional resources

- Module `cert-manager` parameters

Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.

The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Deckhouse Platform web UI

The Deckhouse Platform (DP) web UI provides access to cluster management, including monitoring, parameter configuration, node and module management, as well as security and network configuration. Most operations available from the command line through the `d8` utility (Deckhouse CLI) or `kubectl` can also be performed through the web UI.

Accessing the web UI

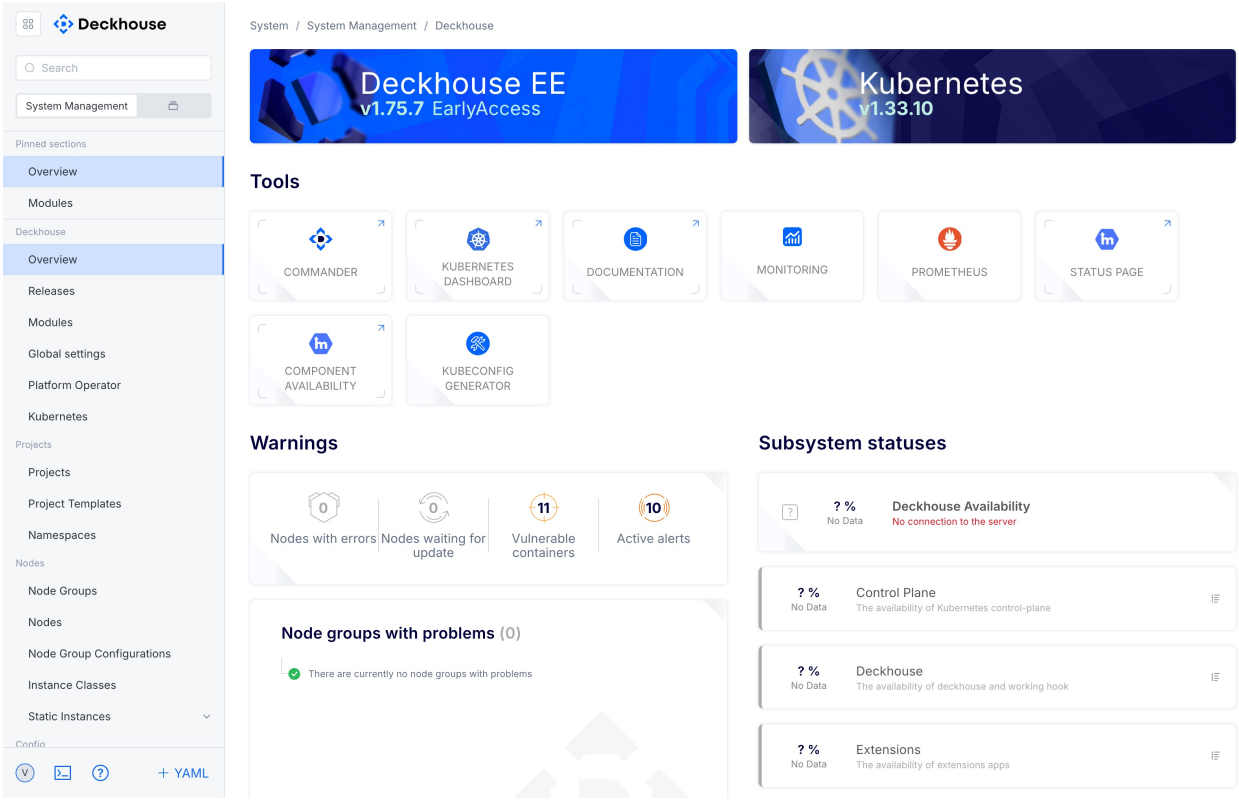
To open the web UI, do as follows:

1. Enter `console.<CLUSTER_NAME_TEMPLATE>` in the browser address bar, where `<CLUSTER_NAME_TEMPLATE>` is a string corresponding to the cluster DNS name template specified in the global parameter `modules.publicDomainTemplate`. The address format may differ depending on the system configuration. Contact your administrator to find out the URL.
2. During the first login, enter the user credentials provided by the security administrator. After successful authentication, the main page of the web UI opens. The page contains the following sections:
   - “Deckhouse”
   - “Node management”
   - “Multitenancy”
   - “Access”
   - “Network”
   - “Storage”
   - “Security”
   - “Monitoring”
   - “Logging”

Deckhouse

“Overview” subsection

The “Overview” subsection contains key information about the Deckhouse Platform (DP) cluster and its components.

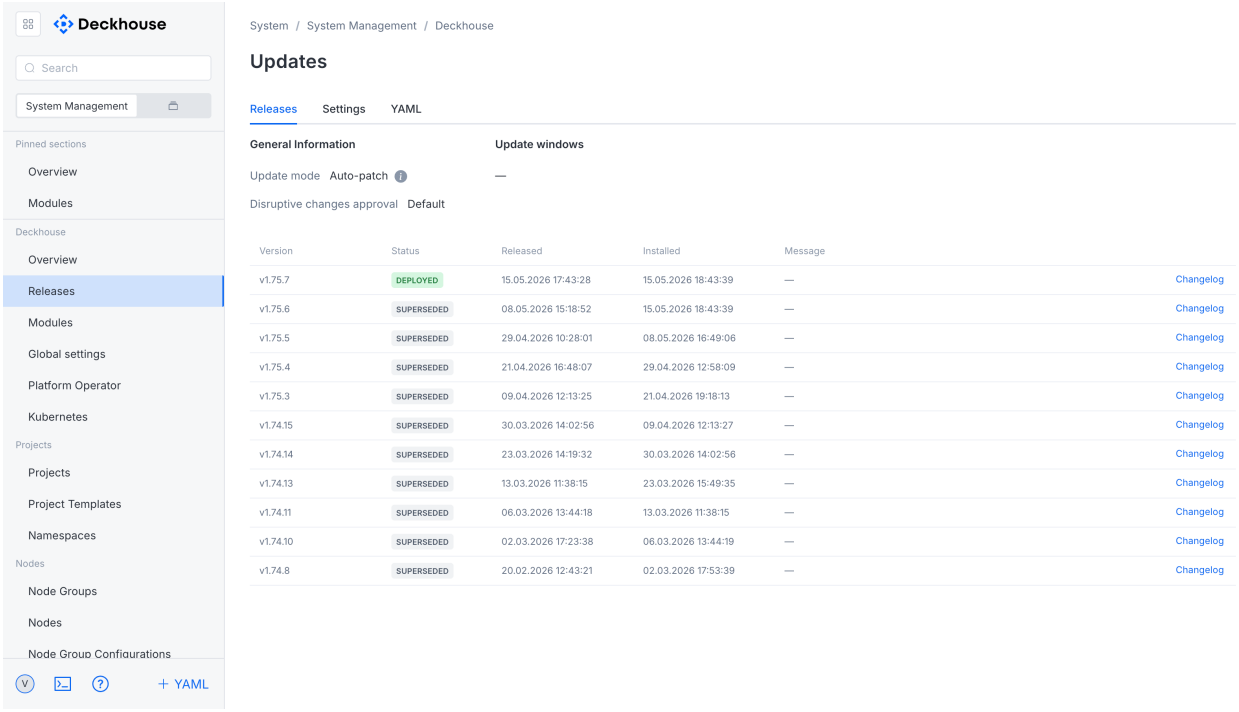


Main elements of the web UI:

- **“Deckhouse” and “Kubernetes” panels**: Display the current Kubernetes version and general platform information.
- **“Tools” panel**: Contains buttons for quick access to the following sections:
  - “Kubernetes dashboard”
  - “Documentation”
  - “Grafana” (metrics monitoring)
  - “Prometheus” (metrics collection and storage)
  - “Status page”
  - “Component availability”
  - “Kubeconfig generator”
- **“Warnings” panel**: Displays errors, pending updates, and active alerts.
- **“Node groups with issues” panel**: Analyzes node health and displays problematic groups.
- **“Subsystem statuses” panel**: Displays the state of cluster services.
- **Resource monitoring**: Includes charts and indicators showing load changes.
- **Sidebar menu**: Navigation through the main sections (the visibility of sections depends on the user access level).
- **User menu (bottom left)**:
  - User information
  - Settings (changing system parameters and viewing the web UI version)
  - YAML editor for configuration management

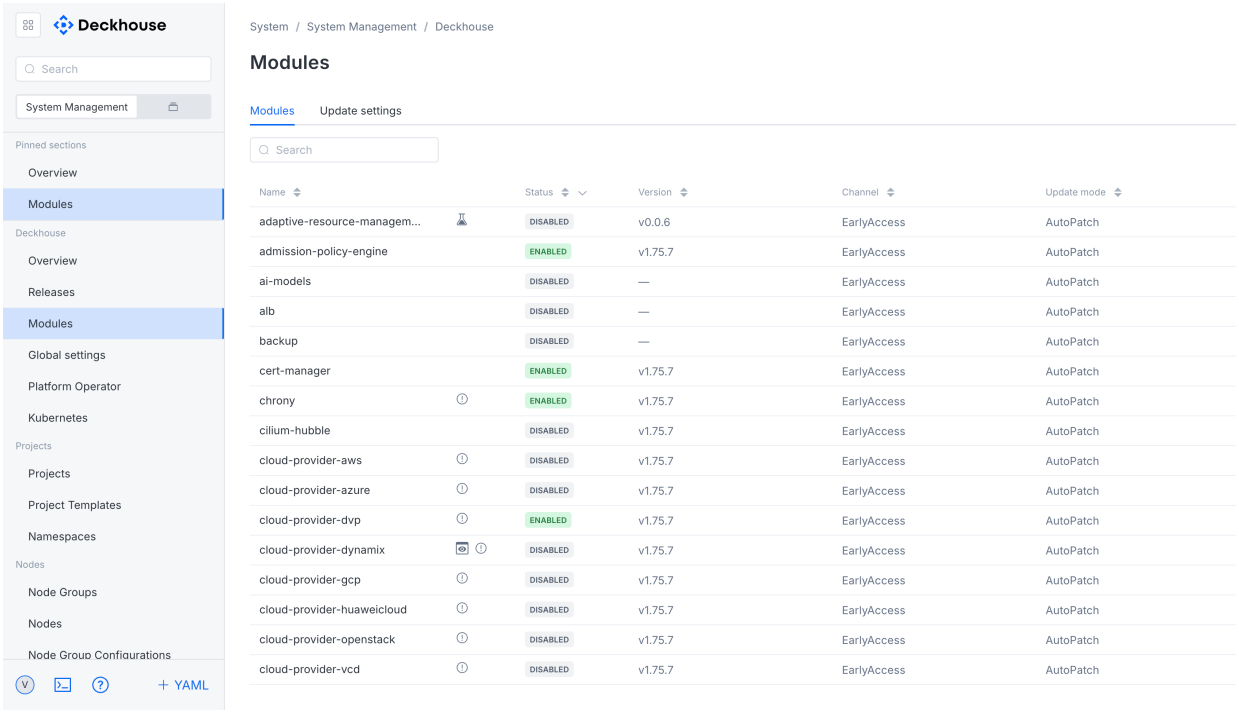
“Updates” subsection

The “Updates” subsection contains information about DP releases.



“Modules” subsection

The “Modules” subsection lists enabled and disabled modules. A filter is available to search for a required module.

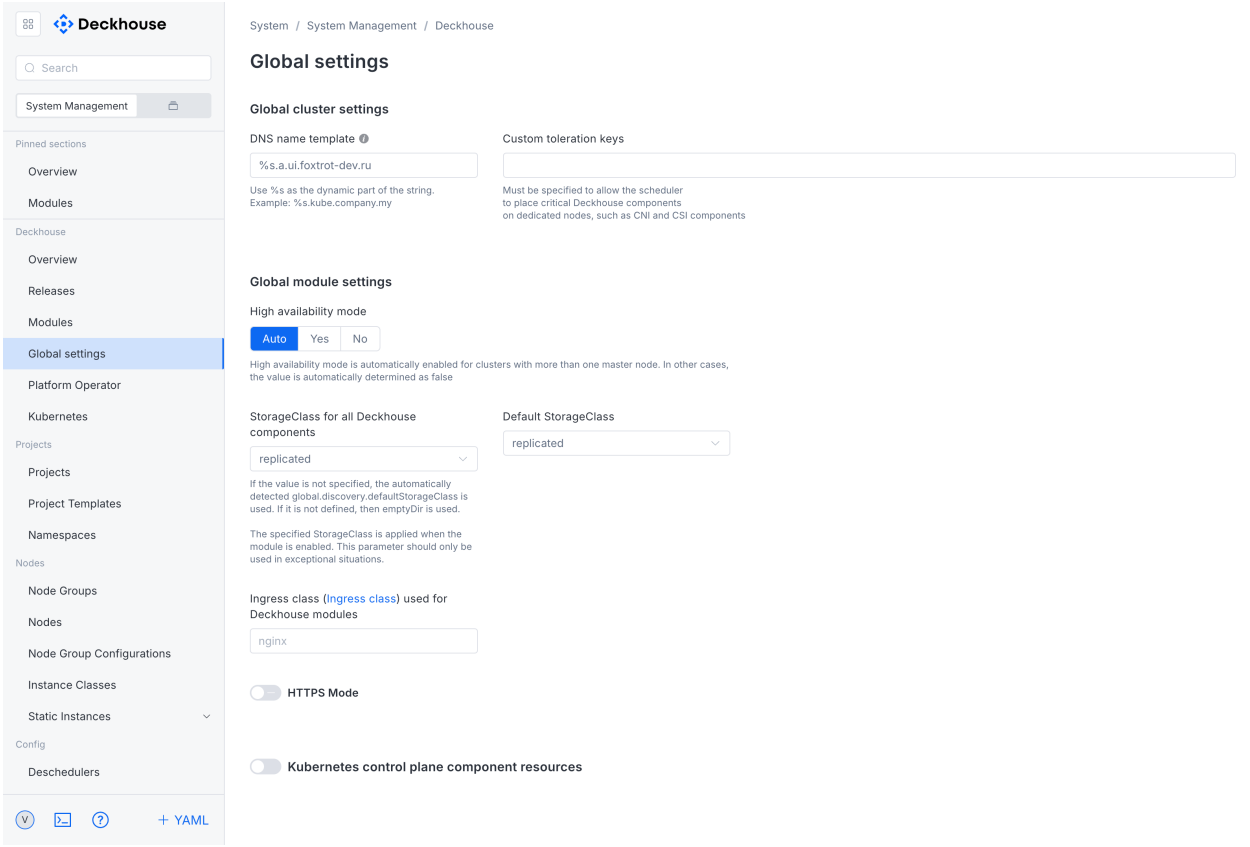


“Global settings” subsection

This subsection provides access to critical DP cluster settings:

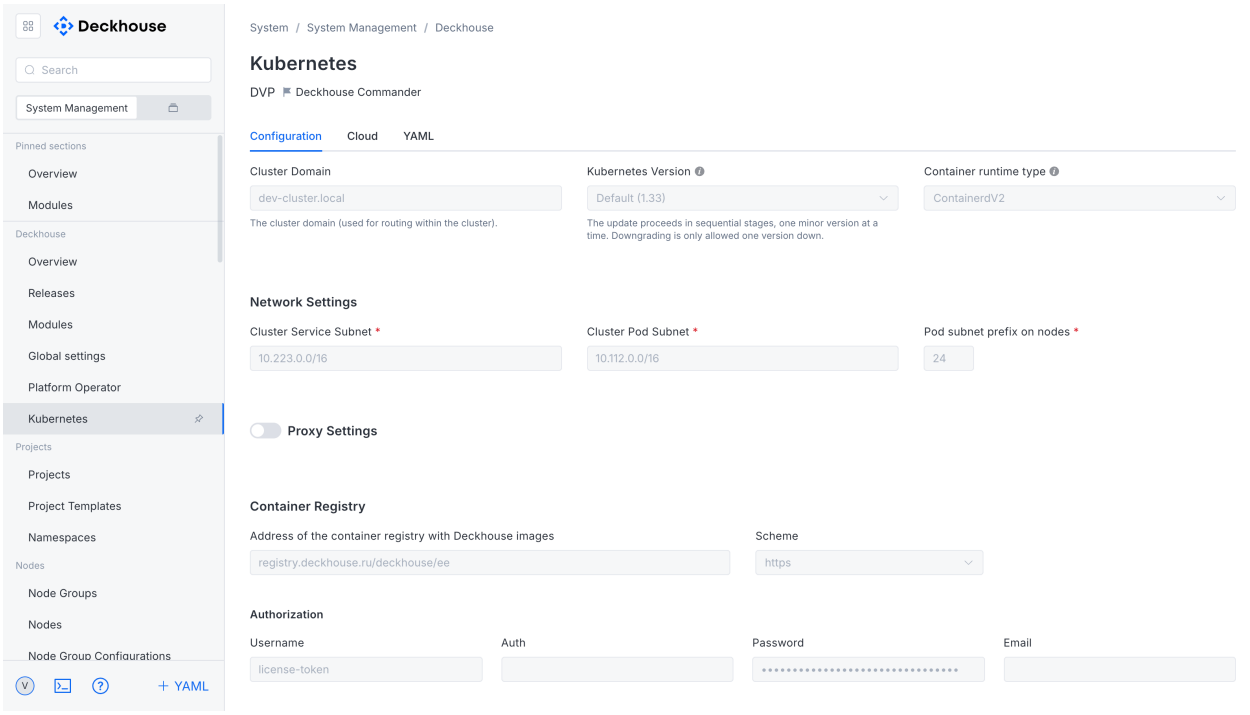
- **“Global cluster settings”**: Configuration of the DNS name template and the list of tolerations.
- **“Global module settings”**: Configuration of the high availability mode, and fields for specifying IngressClass and StorageClass.
- **“HTTPS operating mode”**: Configuration of security certificates.
- **“Kubernetes control plane component resources”**: Configuration of the number of allocated CPU cores and memory size.

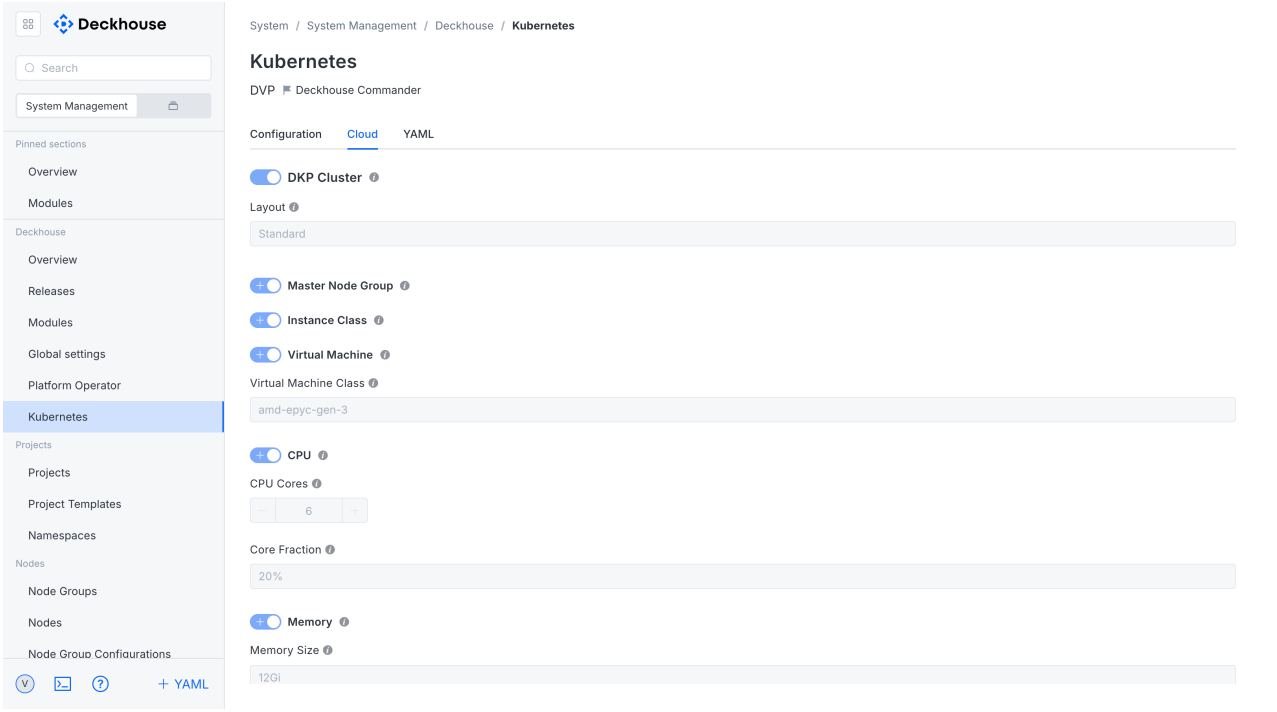
⚠ These settings affect cluster stability, security, and fault tolerance, so modify them with caution.



Kubernetes

The tabs in the “Kubernetes” subsection contain basic information about the Kubernetes cluster.





Node management

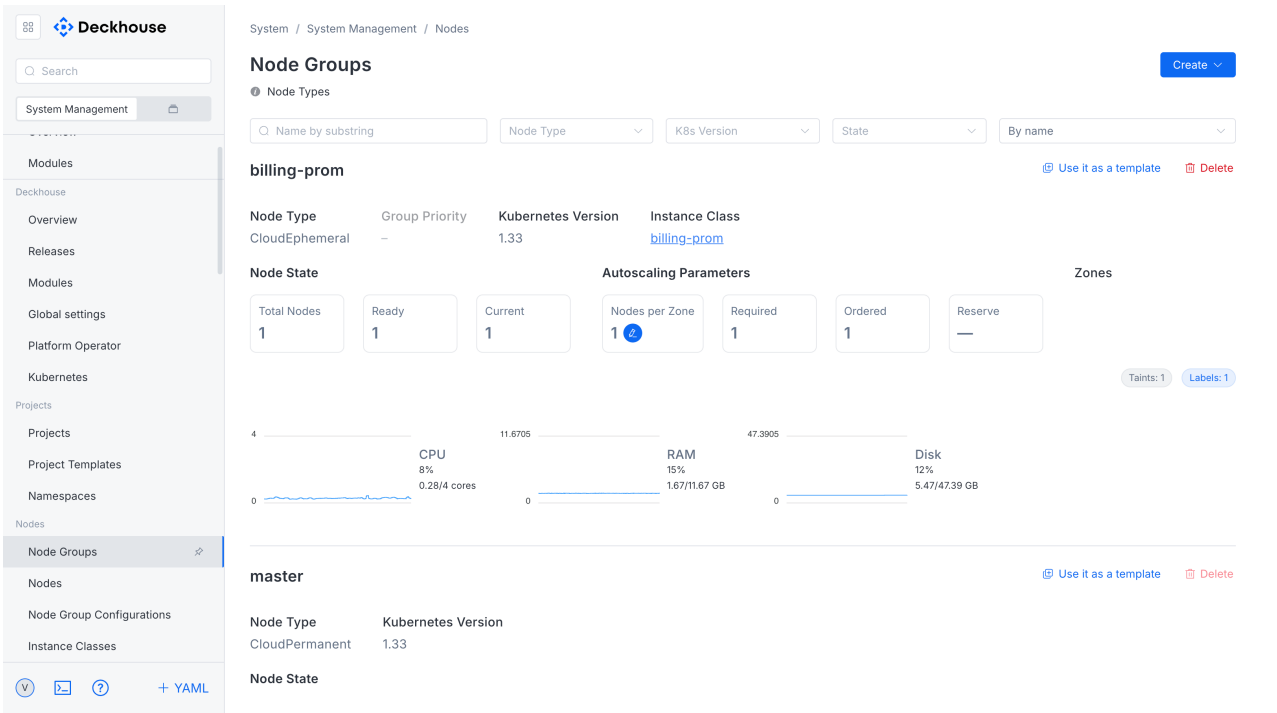
“Node groups” subsection

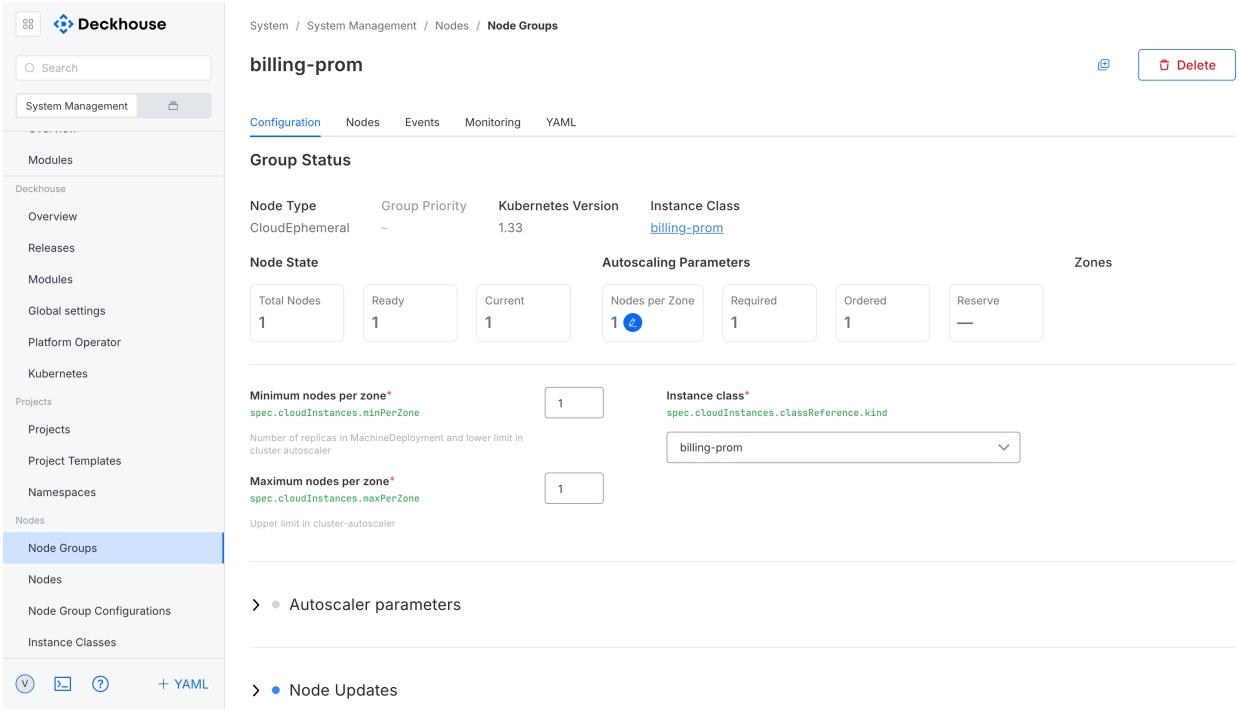
The settings in this subsection let you manage Kubernetes node groups, monitor their state, and add new groups.

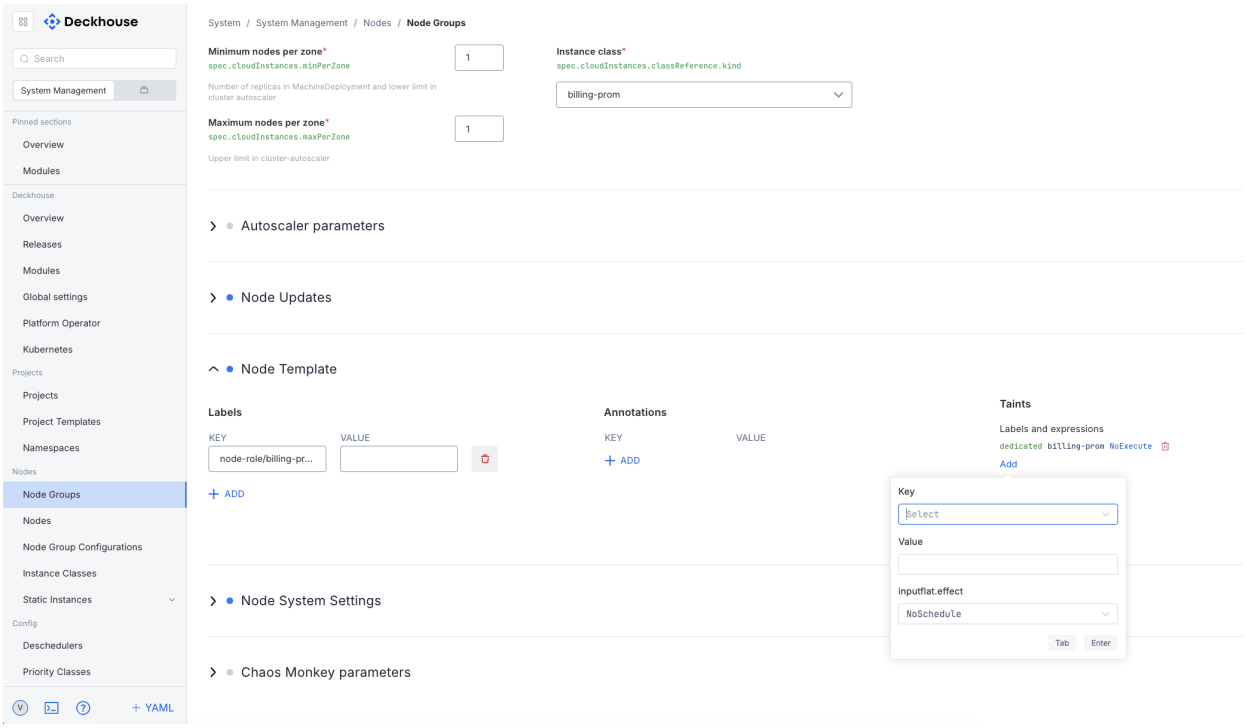
The node group creation form contains the following fields:

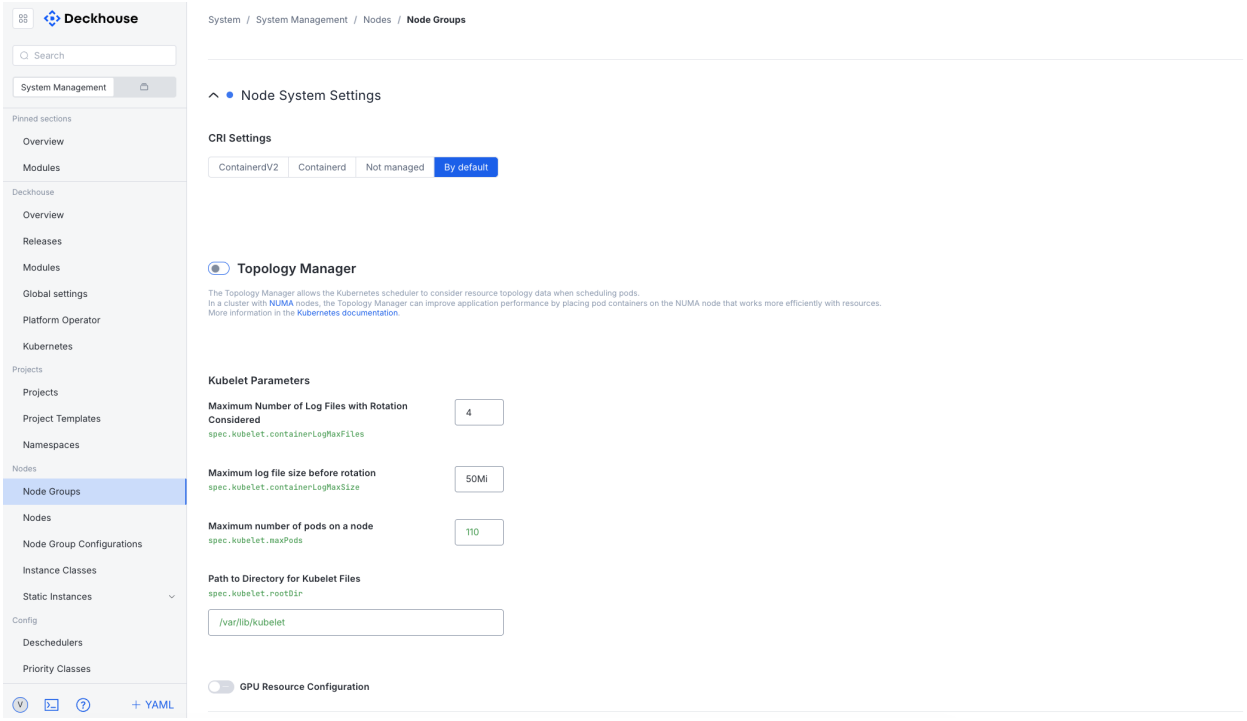
- “Name”
- “Node count”
- “Static instance selector” (cannot be changed after creation)
- “Label expressions”

The lower part of the form contains additional settings, including node update parameters, the node template, system parameters, and Chaos Monkey parameters, which can also be expanded for detailed configuration.



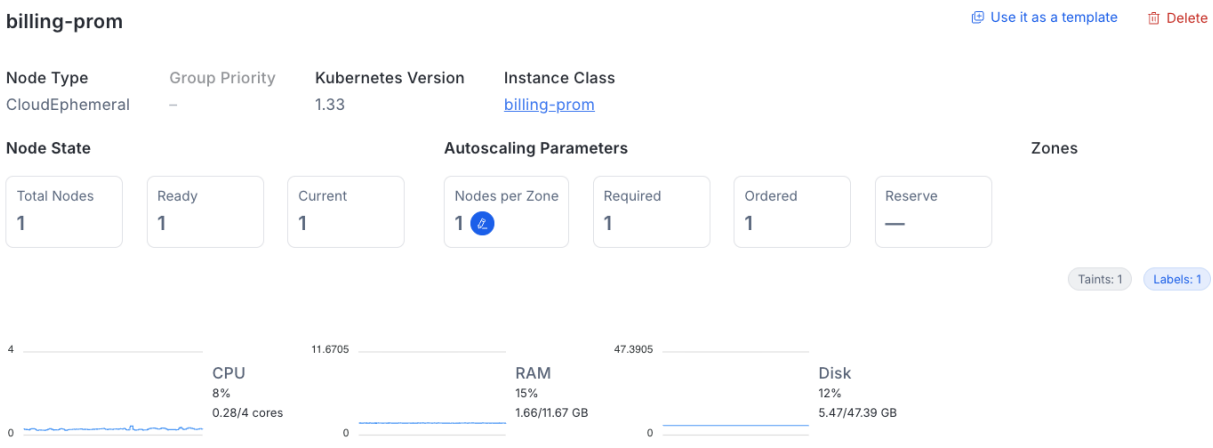

Deckhouse
Search
System Management
Modules
Deckhouse
Overview
Releases
Modules
Global settings
Platform Operator
Kubernetes
Projects
Projects
Project Templates
Namespaces
Nodes
Node Groups
Nodes
Node Group Configurations
Instance Classes
+ YAML
System / System Management / Nodes / Node Groups
billing-prom
Delete
Configuration
Nodes
Events
Monitoring
YAML
Group Status
Node Type
CloudEphemeral
Group Priority
–
Kubernetes Version
1.33
Instance Class
billing-prom
Node State
Total Nodes
1
Ready
1
Current
1
Autoscaling Parameters
Nodes per Zone
1
Required
1
Ordered
1
Reserve
—
Zones
Minimum nodes per zone*
spec.cloudInstances.minPerZone
Number of replicas in MachineDeployment and lower limit in cluster autoscaler
1
Maximum nodes per zone*
spec.cloudInstances.maxPerZone
Upper limit in cluster-autoscaler
1
Instance class*
spec.cloudInstances.classReference.kind
billing-prom
Autoscaler parameters
Node Updates


Deckhouse
Search
System Management
Pinned sections
Overview
Modules
Deckhouse
Overview
Releases
Modules
Global settings
Platform Operator
Kubernetes
Projects
Projects
Project Templates
Namespaces
Nodes
Node Groups
Nodes
Node Group Configurations
Instance Classes
Static Instances
Config
Deschedulers
Priority Classes
+ YAML
System / System Management / Nodes / Node Groups
Minimum nodes per zone*
spec.cloudInstances.minPerZone
Number of replicas in MachineDeployment and lower limit in cluster autoscaler
1
Maximum nodes per zone*
spec.cloudInstances.maxPerZone
Upper limit in cluster-autoscaler
1
Instance class*
spec.cloudInstances.classReference.kind
billing-prom
Autoscaler parameters
Node Updates
Node Template
Labels
KEY
VALUE
node-role/billing-pr...
+ ADD
Annotations
KEY
VALUE
+ ADD
Taints
Labels and expressions
dedicated billing-prom NoExecute
Add
Key
Select
Value
inputflat.effect
NoSchedule
Tab
Enter
Node System Settings
Chaos Monkey parameters

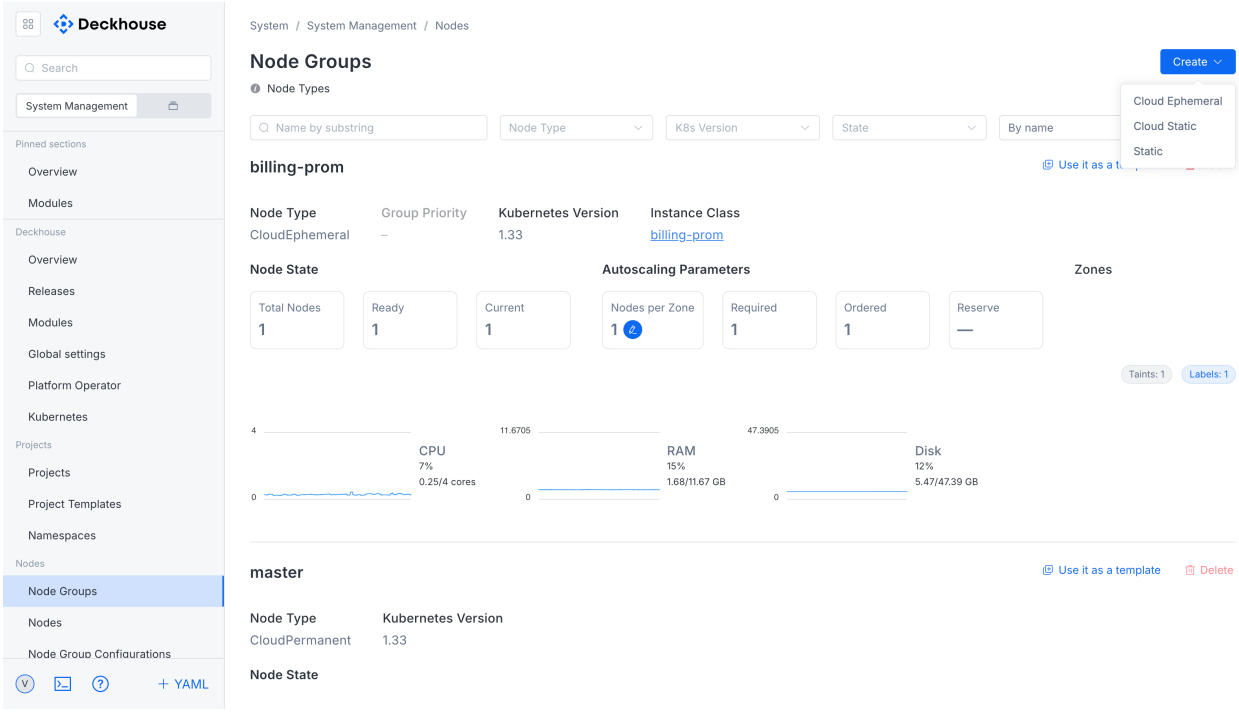


The node group card displays:

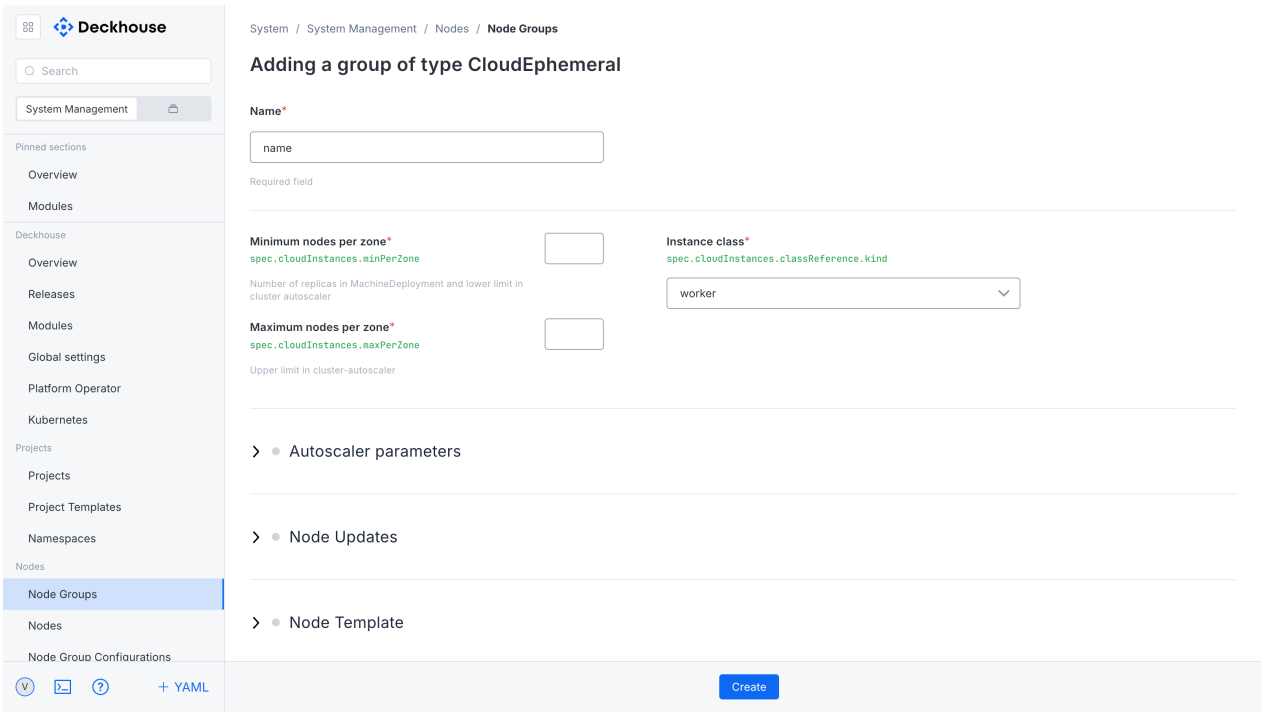
- Node type and Kubernetes version
- Total number of nodes, their readiness and update status
- Load metrics (CPU, memory, disk)
- Taints and labels

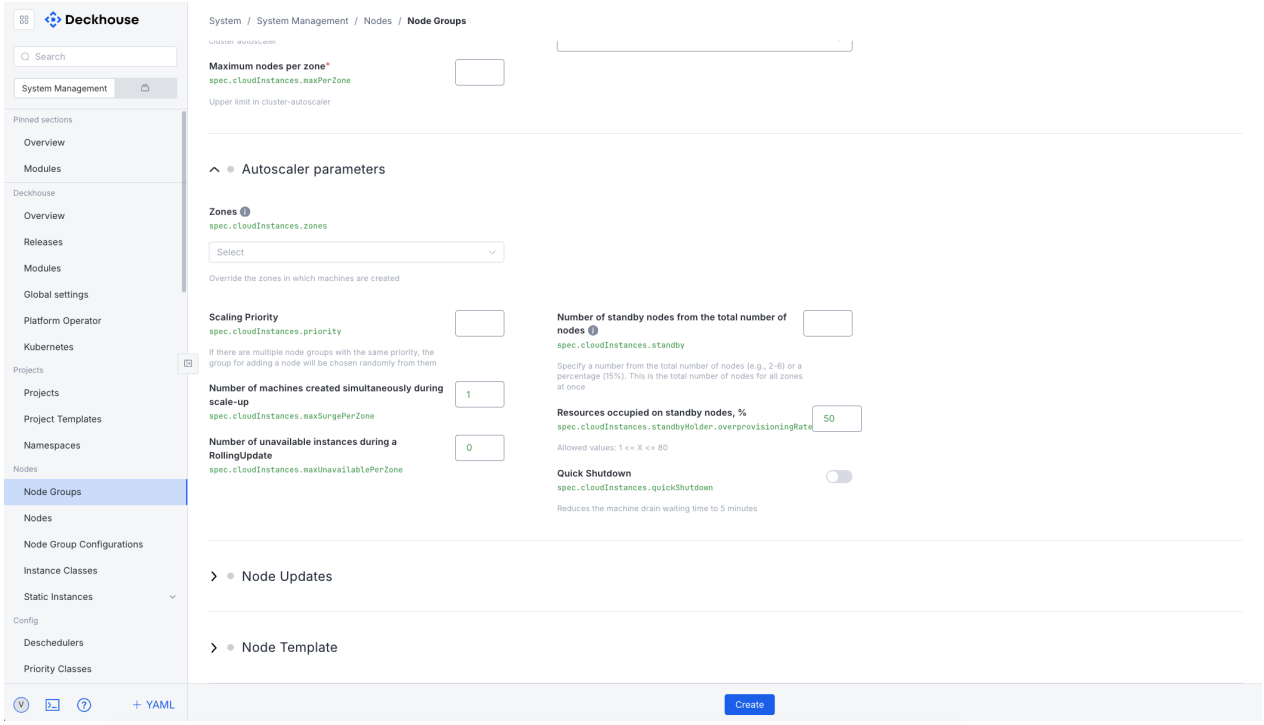


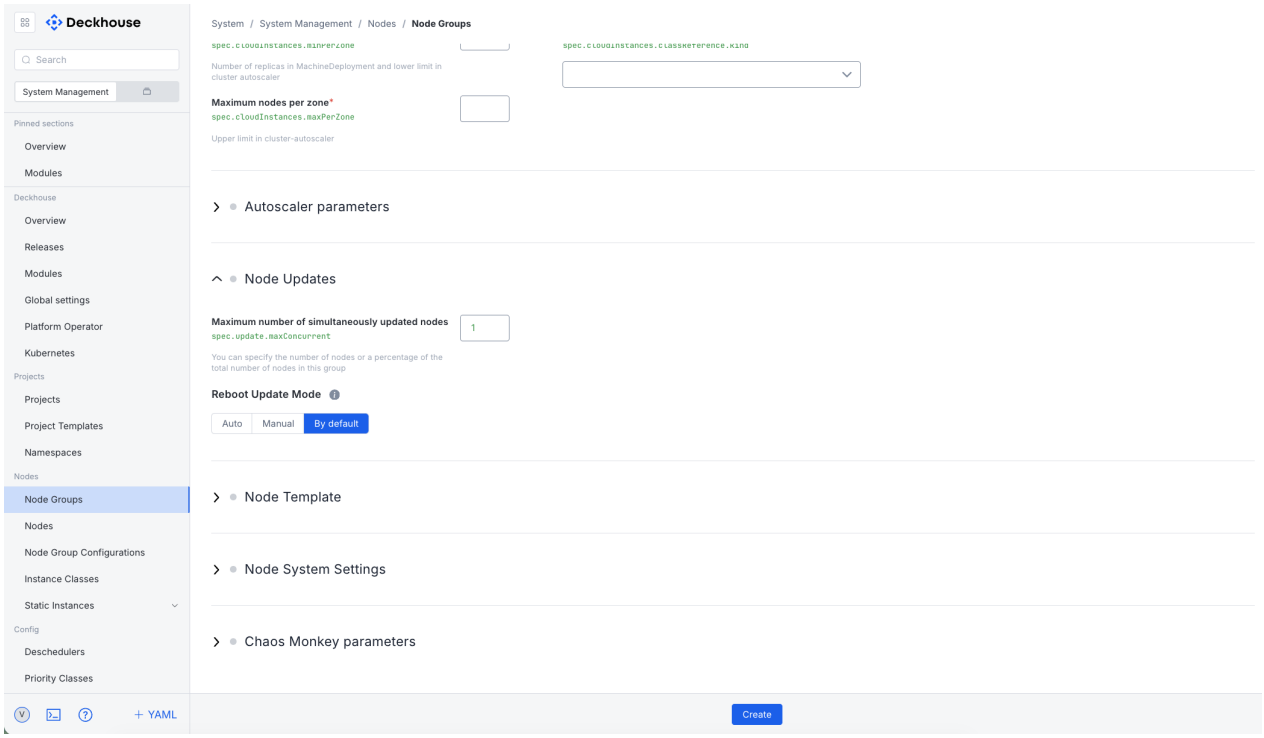
The “Create” button in the upper-right corner opens the form for creating a new node group.

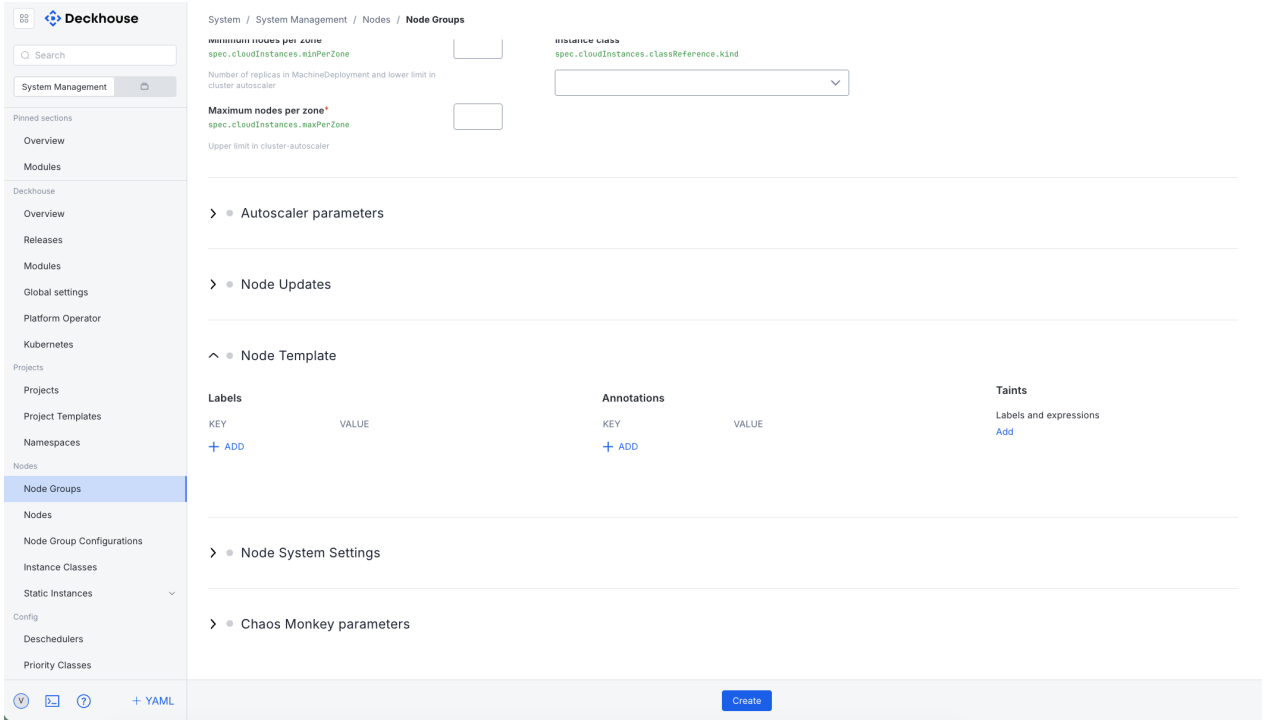


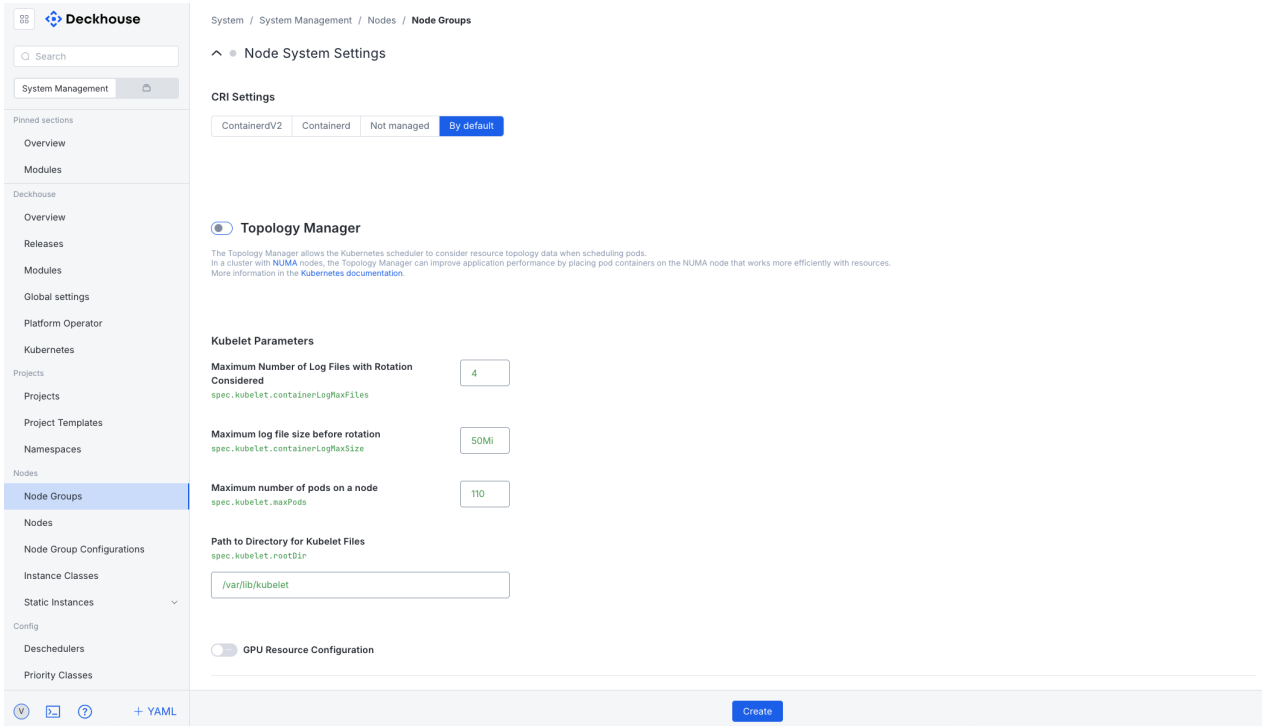
The form lets you specify the required parameters for the new node group.




Deckhouse
Search
System Management
Pinned sections
Overview
Modules
Deckhouse
Overview
Releases
Modules
Global settings
Platform Operator
Kubernetes
Projects
Projects
Project Templates
Namespaces
Nodes
Node Groups
Nodes
Node Group Configurations
Instance Classes
Static Instances
Config
Deschedulers
Priority Classes
YAML
System / System Management / Nodes / Node Groups
Maximum nodes per zone*
spec.cloudInstances.maxPerZone
Upper limit in cluster-autoscaler
Autoscaler parameters
Zones
spec.cloudInstances.zones
Select
Override the zones in which machines are created
Scaling Priority
spec.cloudInstances.priority
If there are multiple node groups with the same priority, the group for adding a node will be chosen randomly from them
Number of machines created simultaneously during scale-up
spec.cloudInstances.maxSurgePerZone
1
Number of unavailable instances during a RollingUpdate
spec.cloudInstances.maxUnavailablePerZone
0
Number of standby nodes from the total number of nodes
spec.cloudInstances.standby
Specify a number from the total number of nodes (e.g., 2-6) or a percentage (15%). This is the total number of nodes for all zones at once
Resources occupied on standby nodes, %
spec.cloudInstances.standbyHolder.overprovisioningRate
50
Allowed values: 1 <= X <= 80
Quick Shutdown
spec.cloudInstances.quickShutdown
Reduces the machine drain waiting time to 5 minutes
Node Updates
Node Template
Create

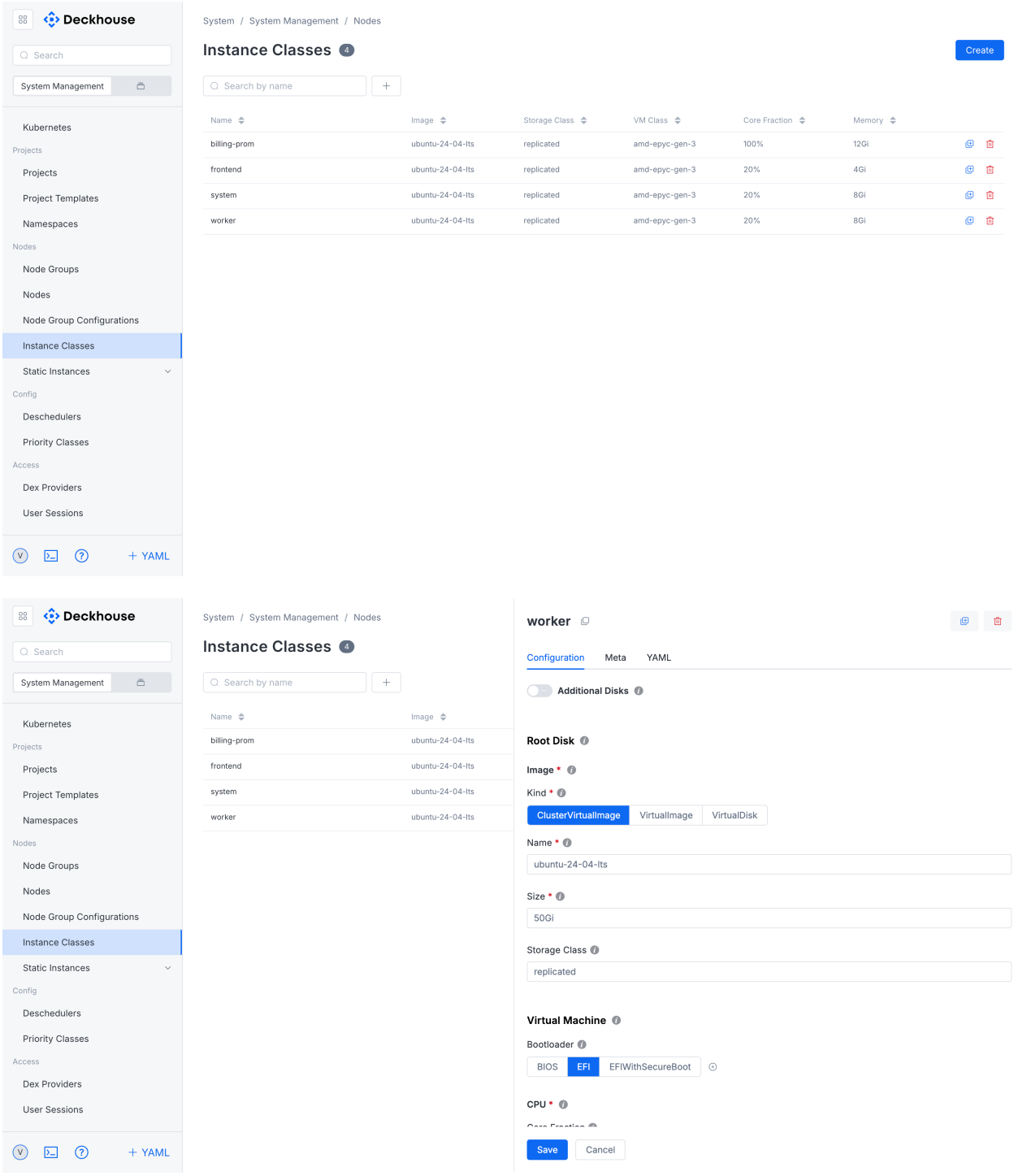

Deckhouse
Search
System Management
Pinned sections
Overview
Modules
Deckhouse
Overview
Releases
Modules
Global settings
Platform Operator
Kubernetes
Projects
Projects
Project Templates
Namespaces
Nodes
Node Groups
Nodes
Node Group Configurations
Instance Classes
Static Instances
Config
Deschedulers
Priority Classes
YAML
System / System Management / Nodes / Node Groups
spec.cloudInstances.minPerZone
Number of replicas in MachineDeployment and lower limit in cluster autoscaler
spec.cloudInstances.classReference.kind
Maximum nodes per zone*
spec.cloudInstances.maxPerZone
Upper limit in cluster-autoscaler
Autoscaler parameters
Node Updates
Maximum number of simultaneously updated nodes
spec.update.maxConcurrent
1
You can specify the number of nodes or a percentage of the total number of nodes in this group
Reboot Update Mode
Auto
Manual
By default
Node Template
Node System Settings
Chaos Monkey parameters
Create



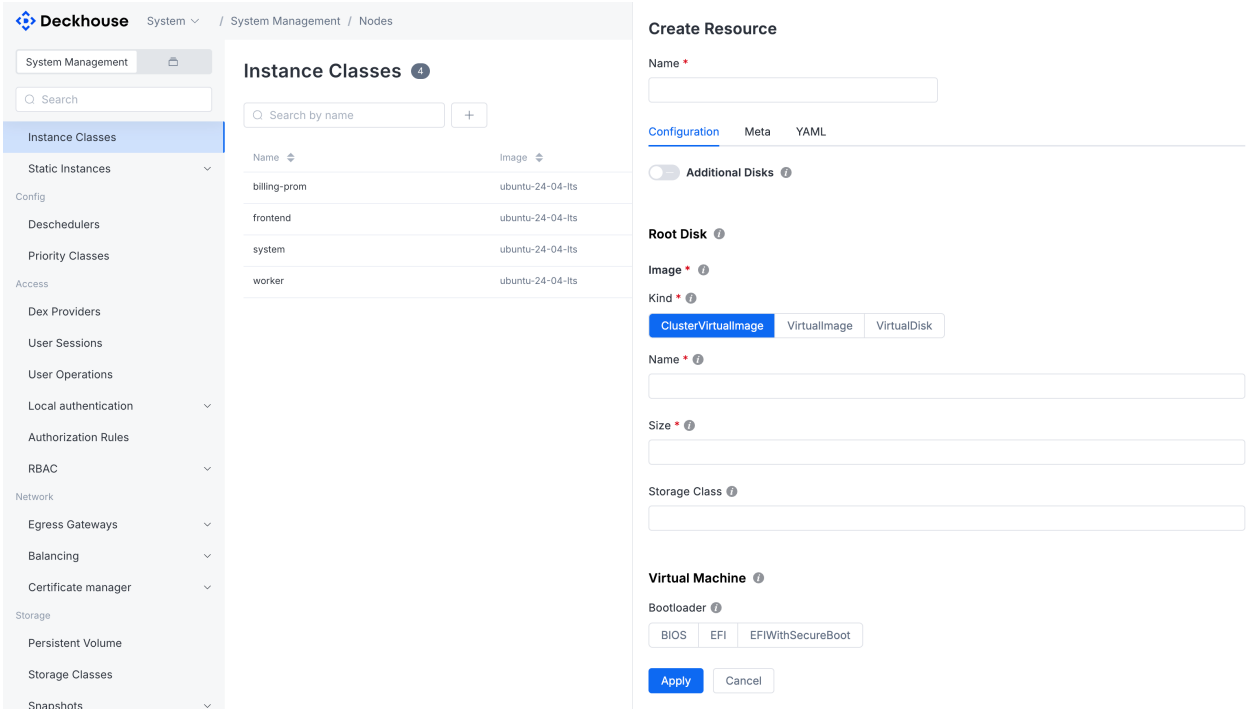


“Instance classes” subsection

This subsection is intended for managing instance classes used in the cluster. The class list can be sorted.



The “Create resource” form lets you specify parameters for a new instance configuration for the cluster.



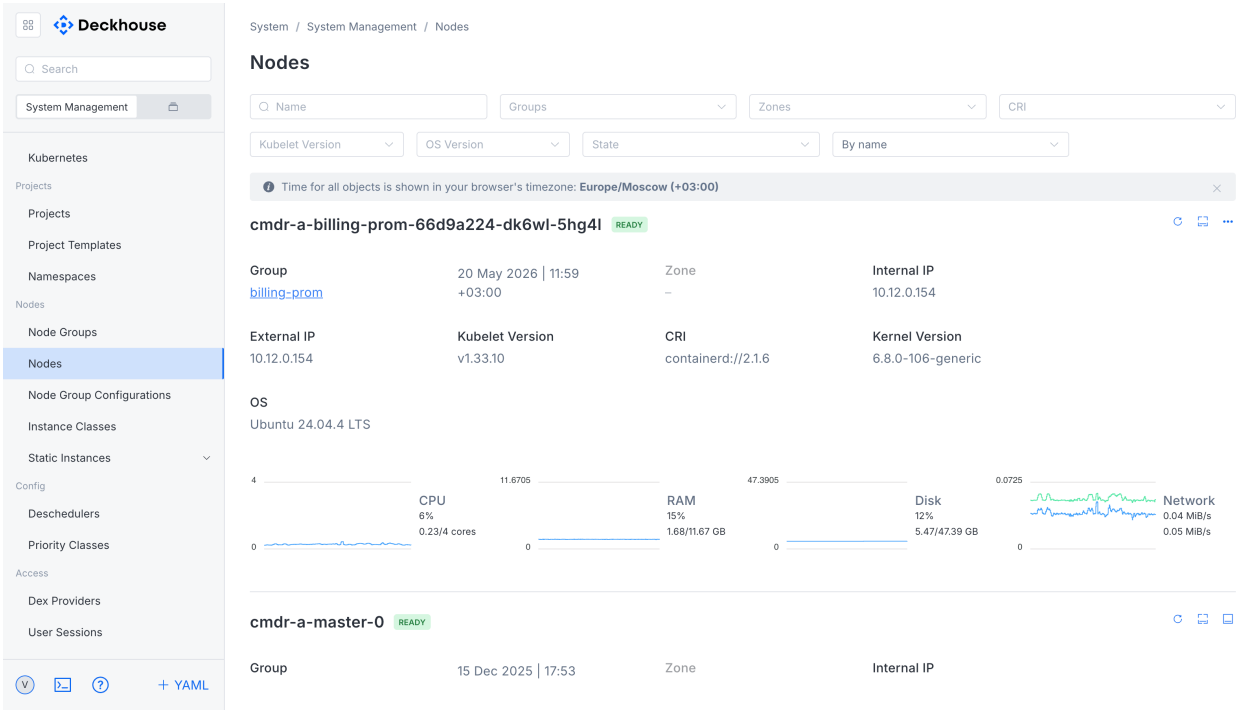
“Nodes” subsection

This subsection provides summary information about all DP cluster nodes with filtering and sorting capabilities.

The node card displays:

- Current node state
- Group
- Date and time
- Availability zone
- Internal and external IP addresses
- Container runtime in use
- Kernel version
- Kubelet version
- Operating system

The subsection also contains charts showing CPU, memory, disk, and network traffic usage, allowing you to monitor node performance. To manage node availability in the cluster, the “Cordon” and “Cordon+Drain” buttons are available.



“Static instances” subsection

This subsection is intended for managing static instances in the cluster and includes two tabs:

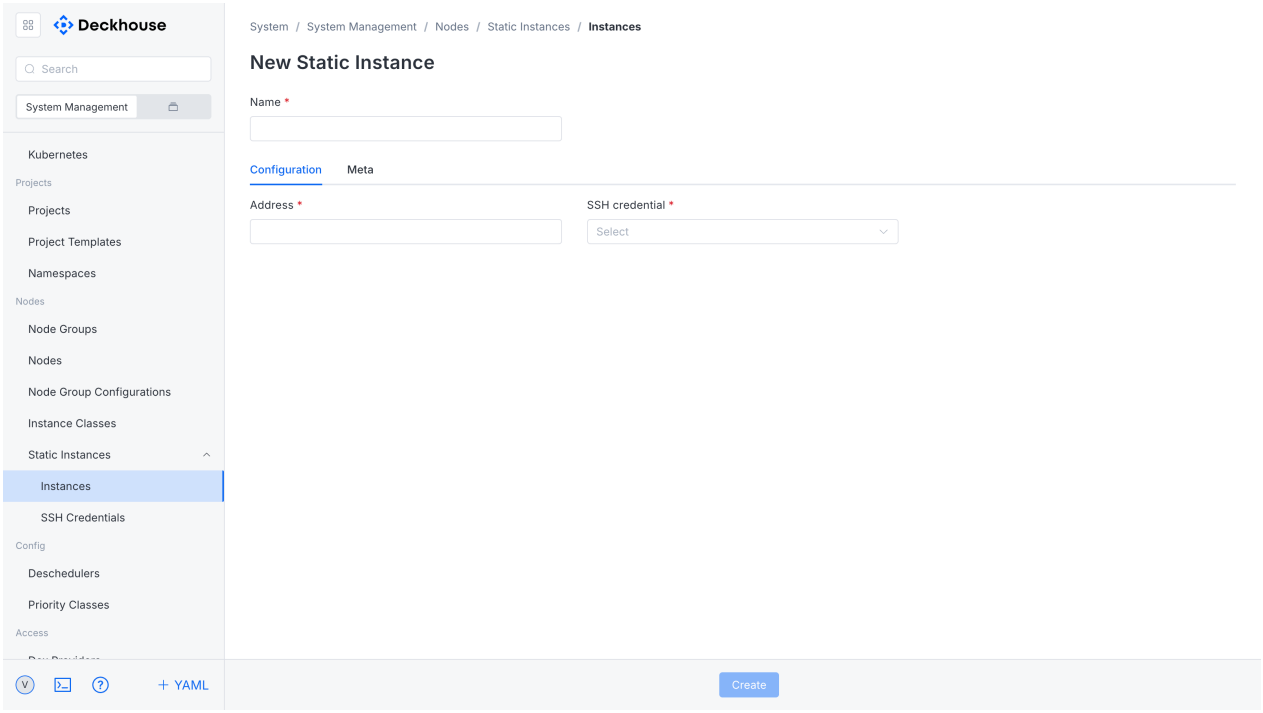
- “Instances”: For working with static instances.
- “SSH access”: For configuring SSH access.

The interface allows you to quickly find and manage static instances.

The “Add instance” button on the “Instances” tab opens the form for adding a new instance to the cluster. The form requires the following information:

- Instance name
- Instance address
- SSH access method from the drop-down list

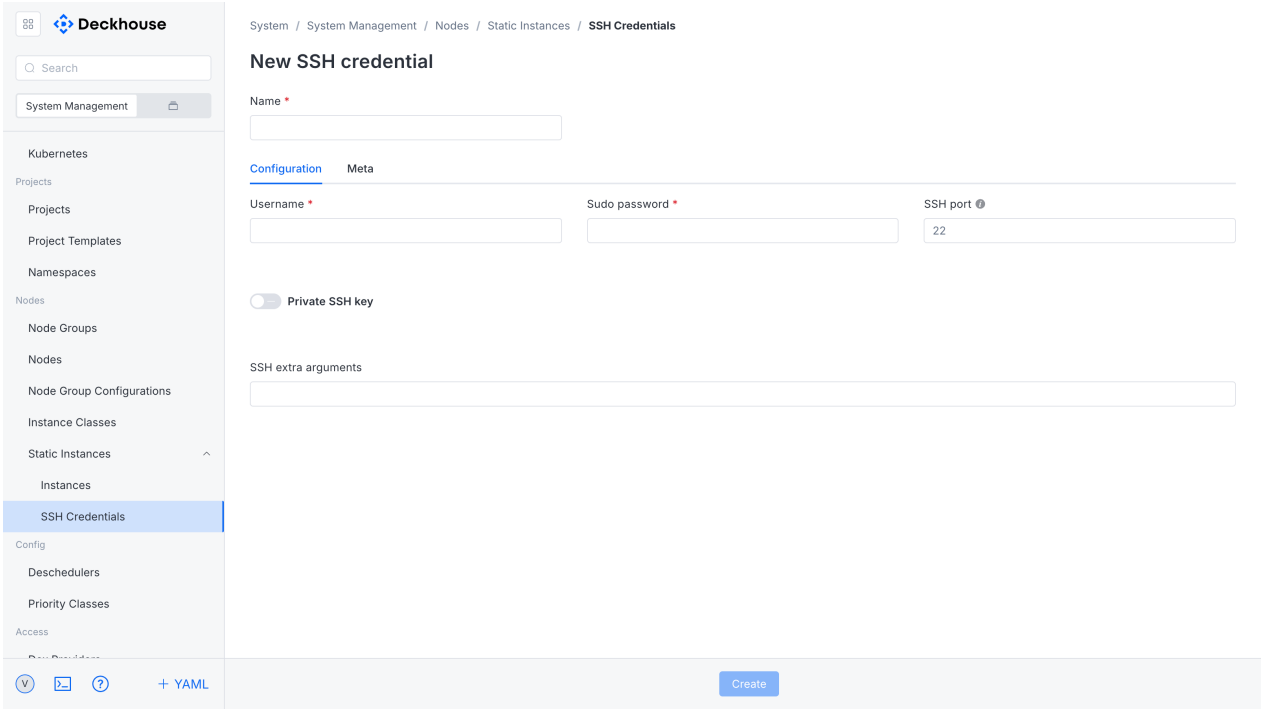
Additionally, you can specify labels by defining keys and values for further identification and management of the instance.



The “Add SSH access” button on the “SSH access” tab opens the form for configuring SSH connections to nodes. The form requires the following information:

- SSH access name
- Username
- Private SSH key
- (Optional) `sudo` password for executing privileged commands

Additional fields are available for changing the SSH port and adding extra SSH arguments.



Multitenancy

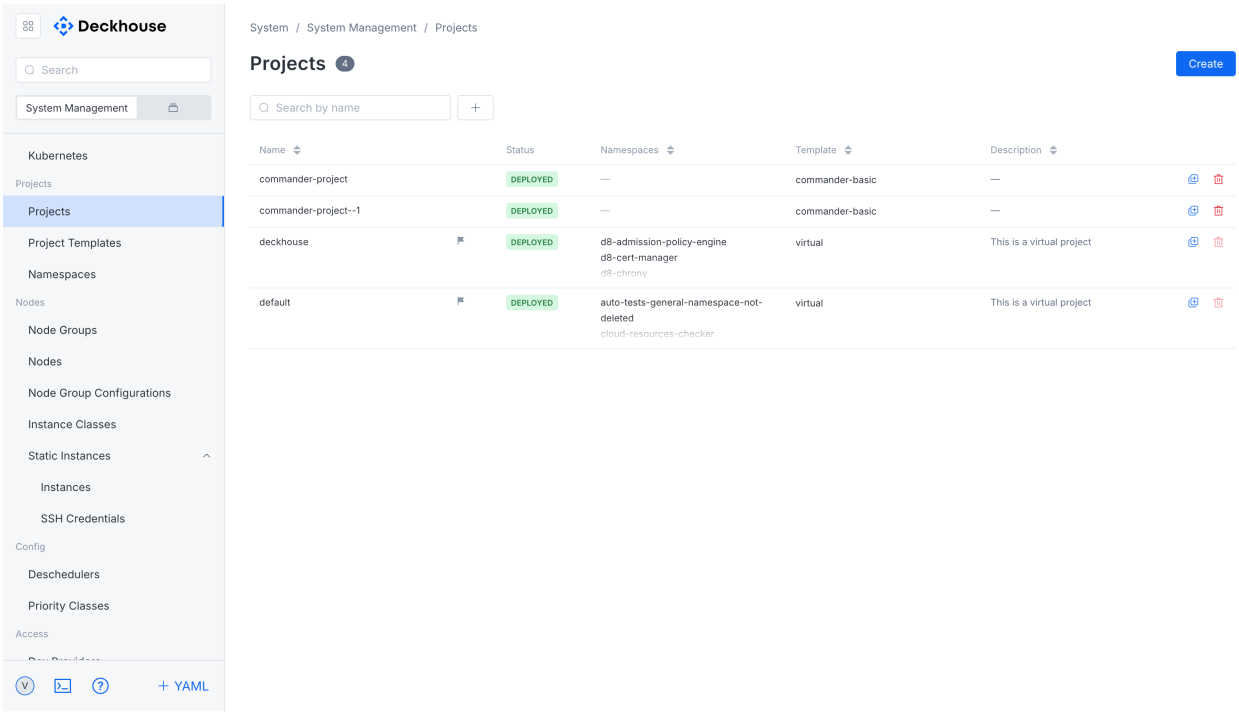
“Projects” subsection

This subsection is intended for creating projects based on prepared templates that define the set of resources to be created and their parameters.

During project creation, the following operations are performed:

- Parameter validation against the OpenAPI schema.
- Template rendering through Helm.
- Deployment of all described resources in an automatically created namespace.

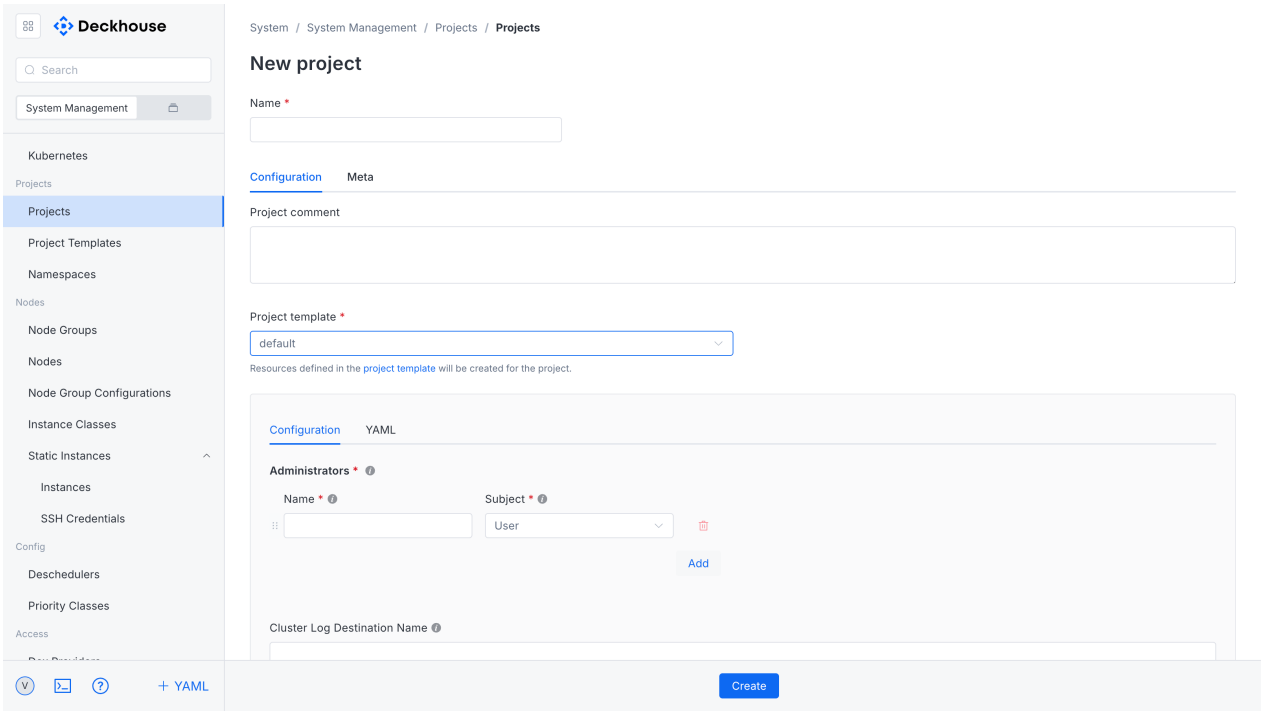
Standard Kubernetes mechanisms are used for access control, resource limits, and network isolation. This allows you to manage security and workload inside the namespace.



The “Create project” button opens the form for adding a new project based on the selected template. In the form, specify the project name and optionally add labels and annotations.

In the central part of the form, select the project template based on which the required resources will be created. If necessary, leave a comment.

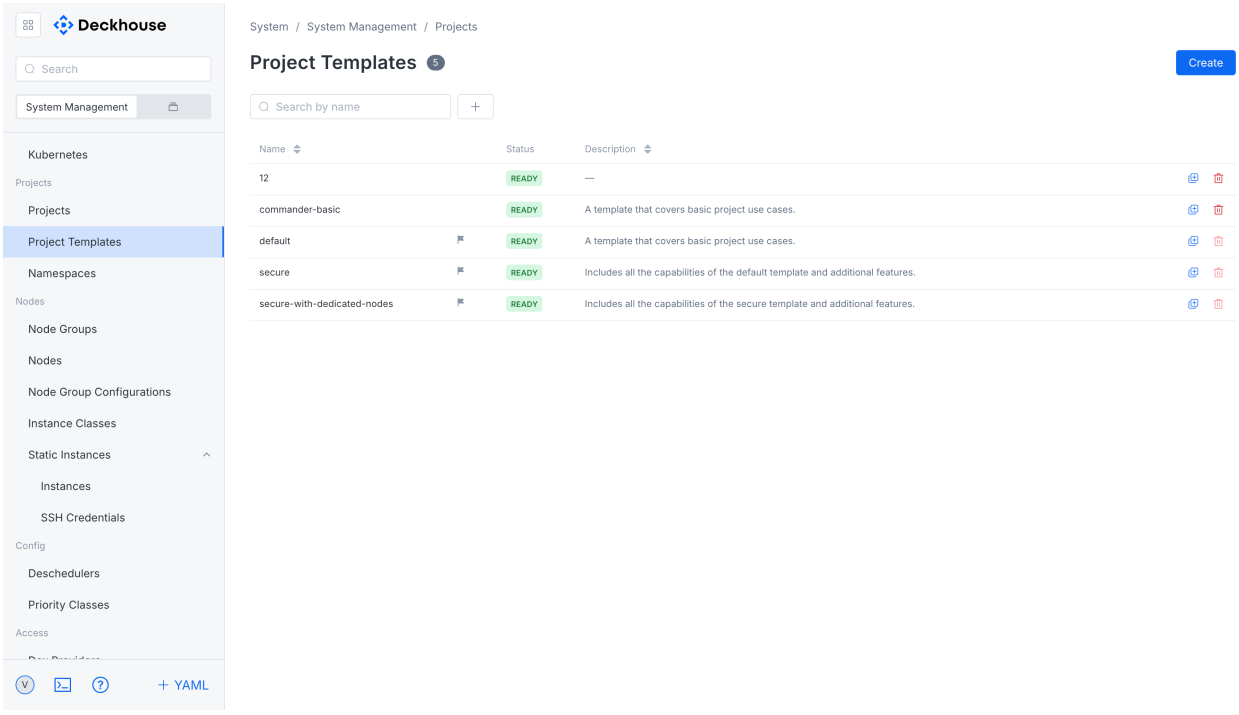
The lower part of the form contains fields for entering template parameters and displaying its structure.



“Project templates” subsection

This subsection is intended for creating project templates.

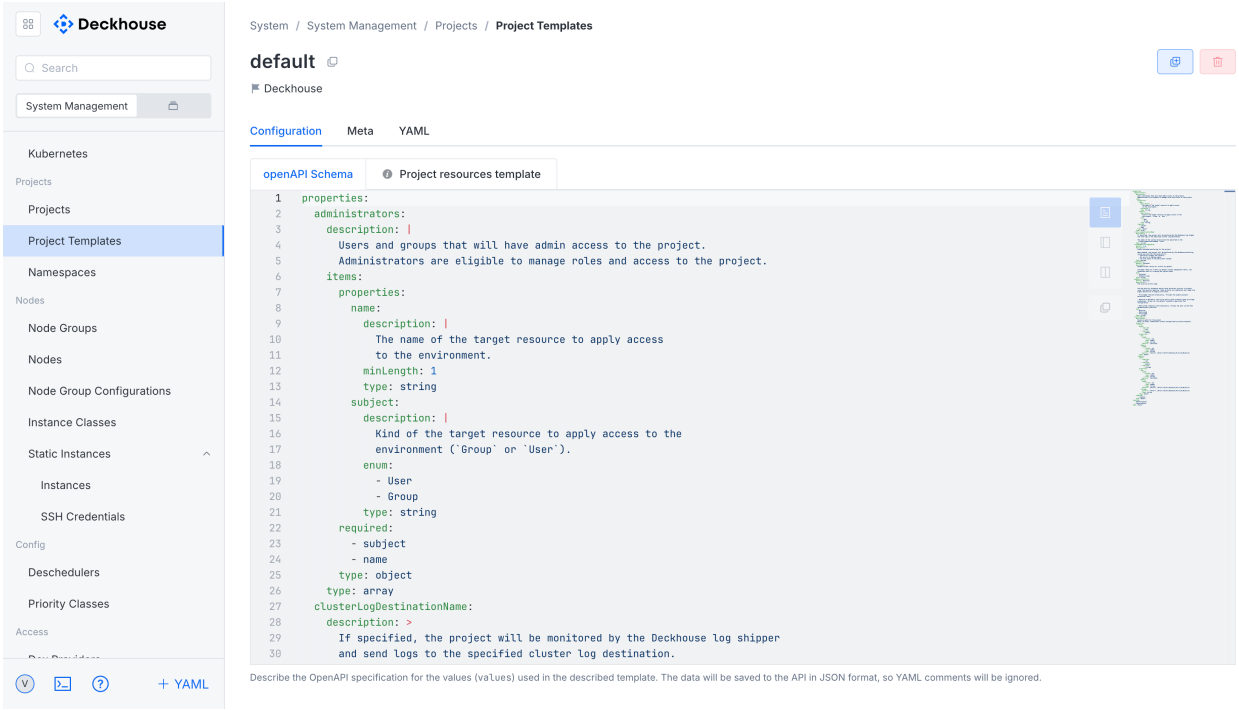
Project templates include basic usage scenarios by default and demonstrate DP capabilities. To add a new template, use the “Create” button in the upper-right corner.

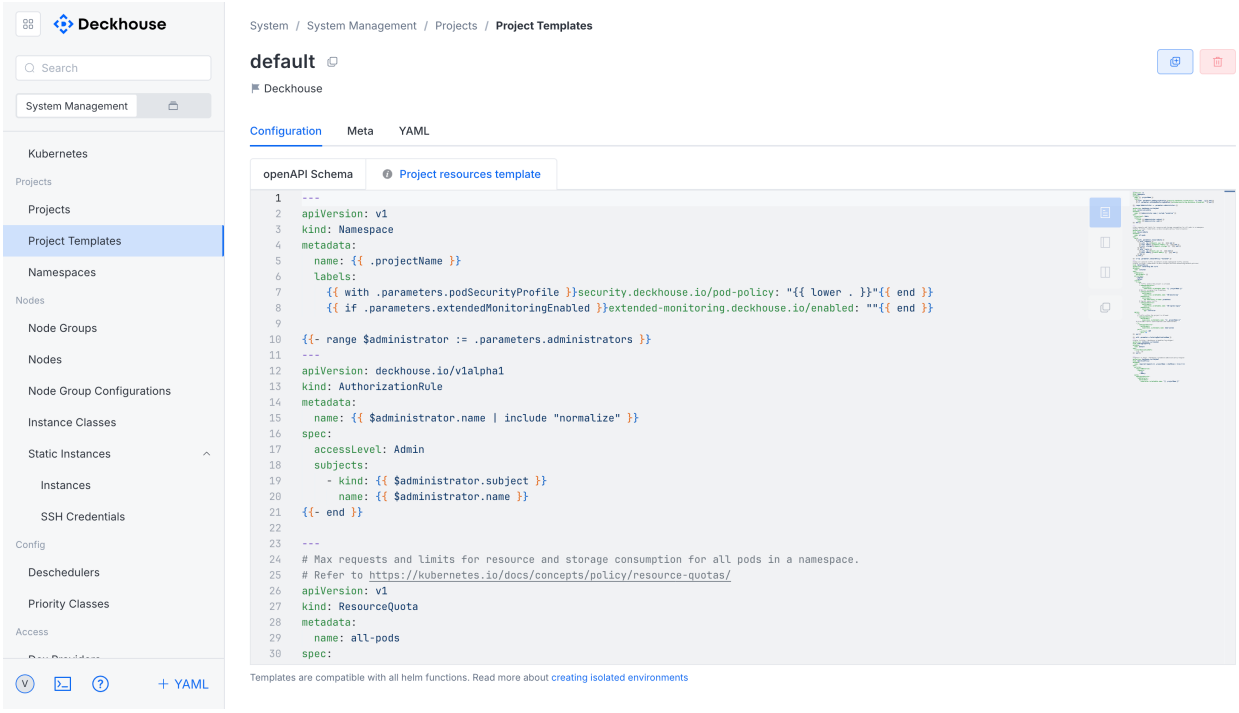


The “New project template” form lets you specify the template name and add labels and annotations for identification.

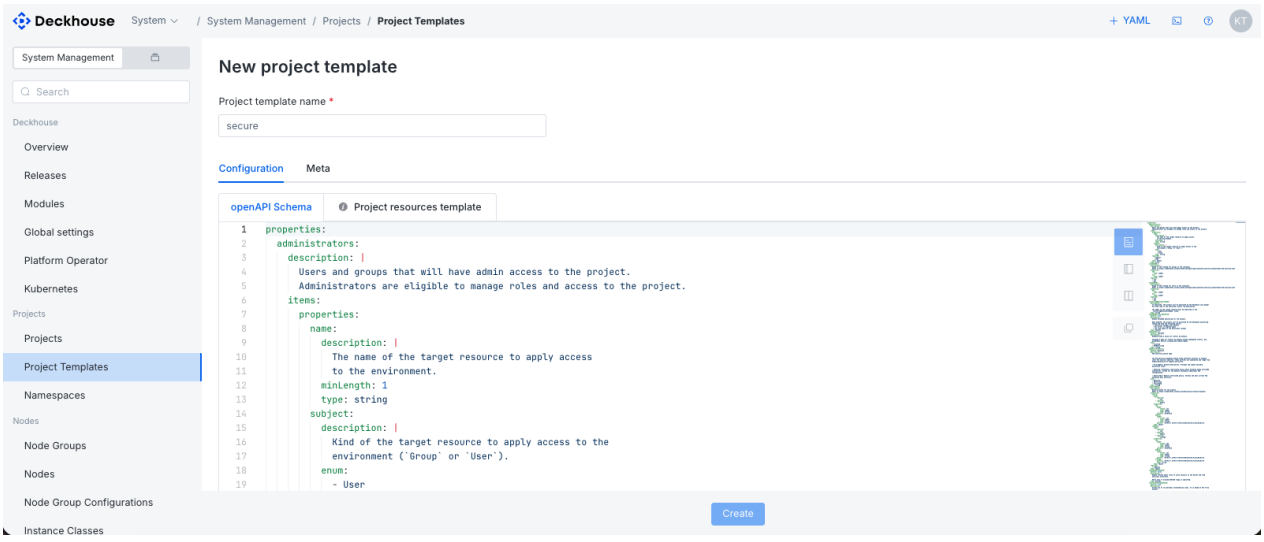
The form contains two tabs:

- “OpenAPI schema”: For describing the value specification in JSON format.
- “Project resource template”: For defining Helm-compatible resources and managing the project environment.





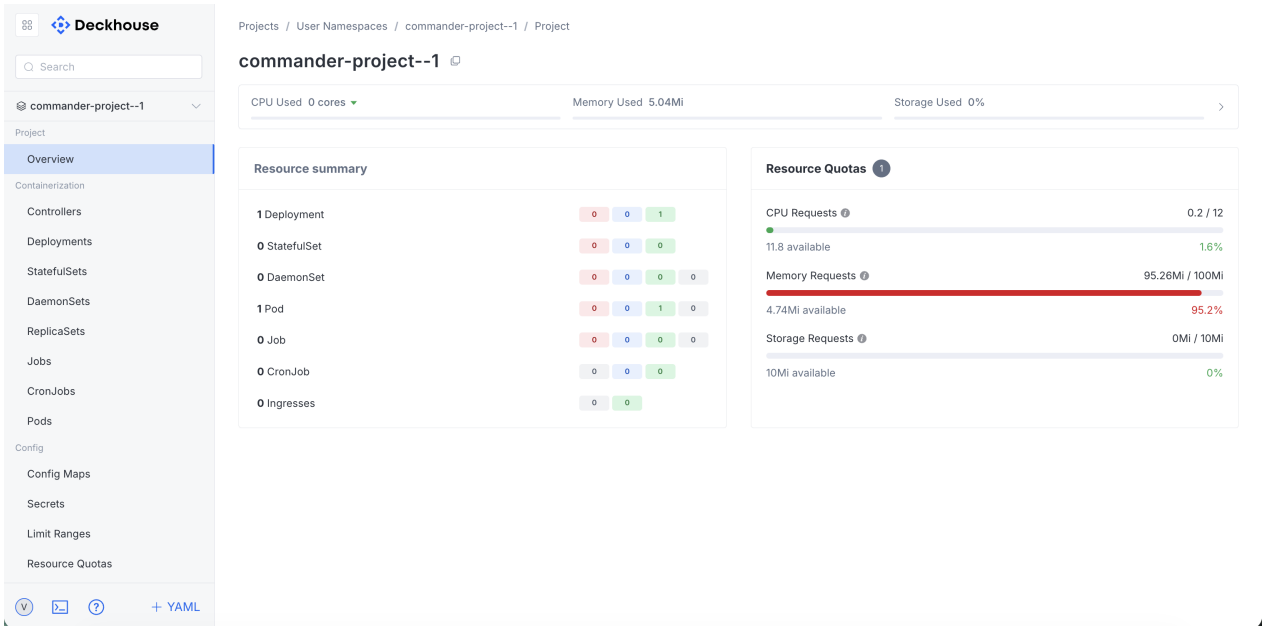
A new template can be created based on an existing one.



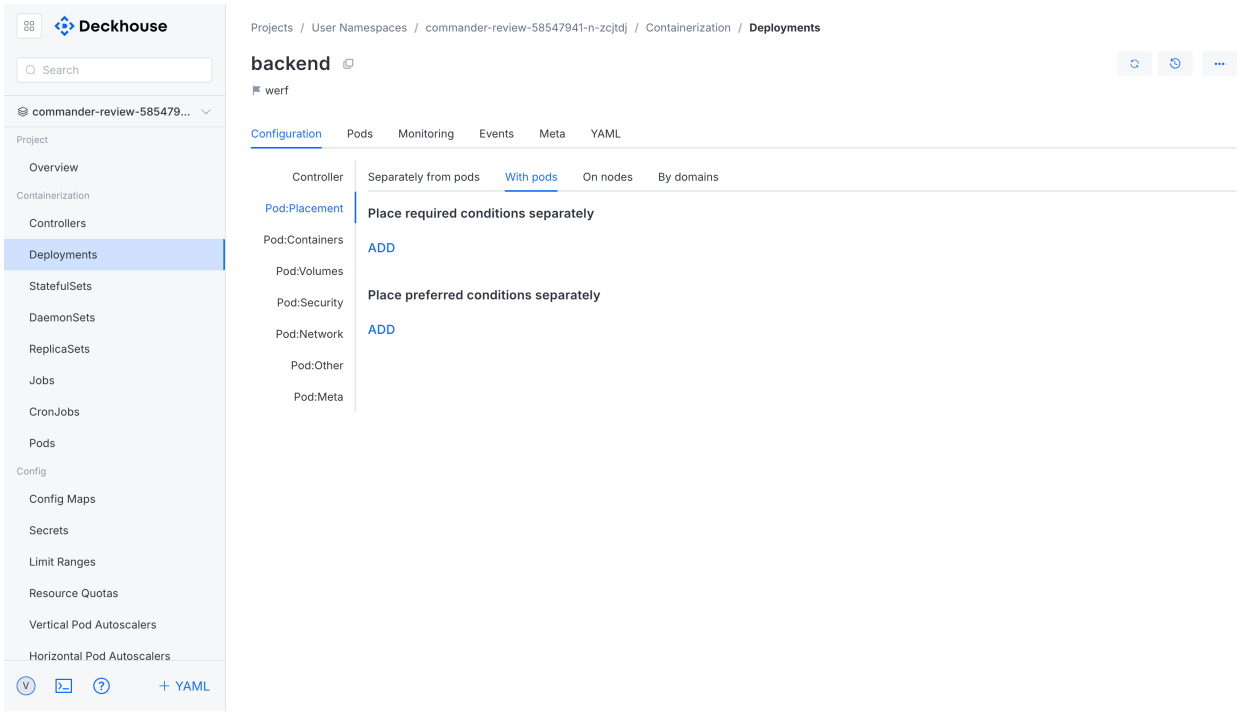
“User namespaces” subsection

This subsection is intended for managing user project namespaces.

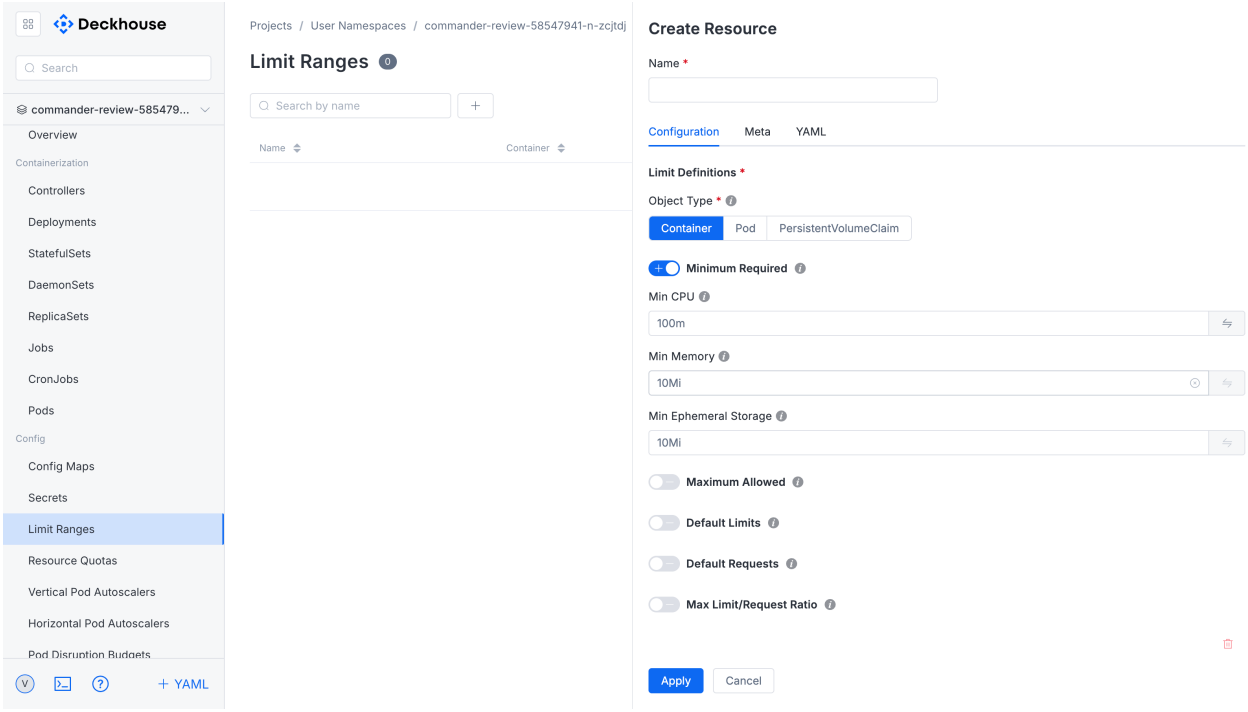
The “Overview” section provides information about the general namespace state.

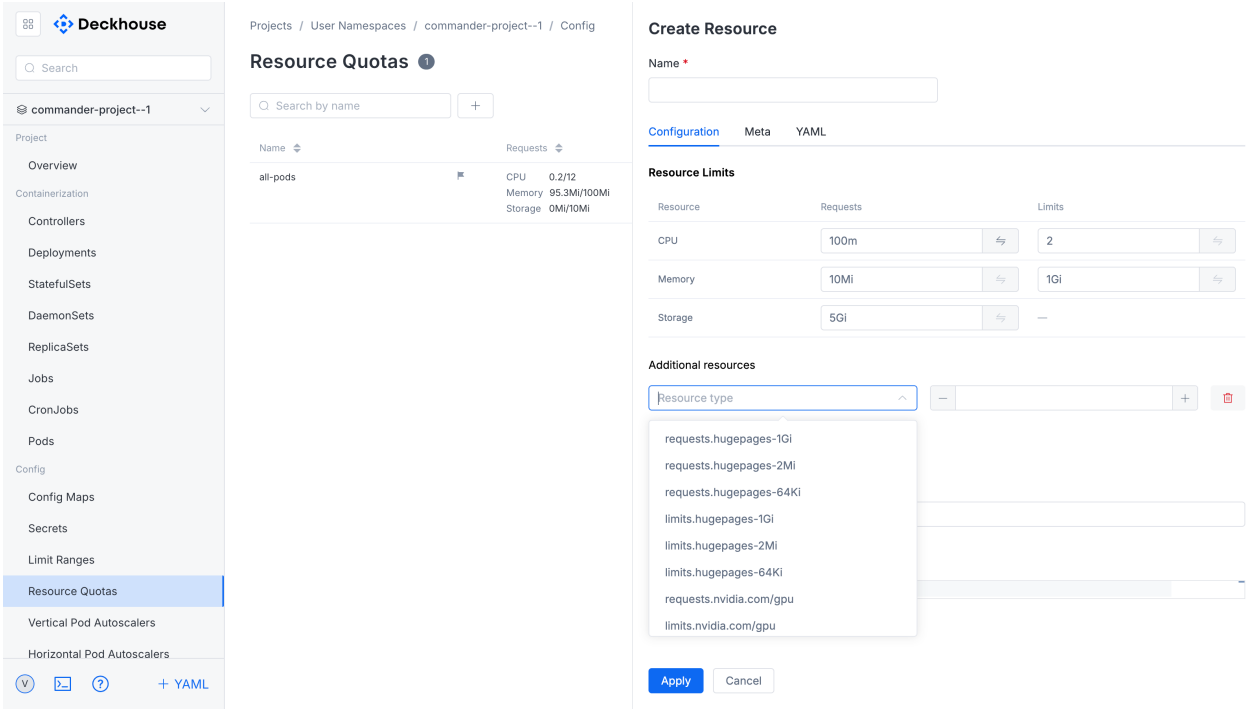


Here you can manage Deployment parameters and other cluster entities.



You can also configure resource limits and quotas.

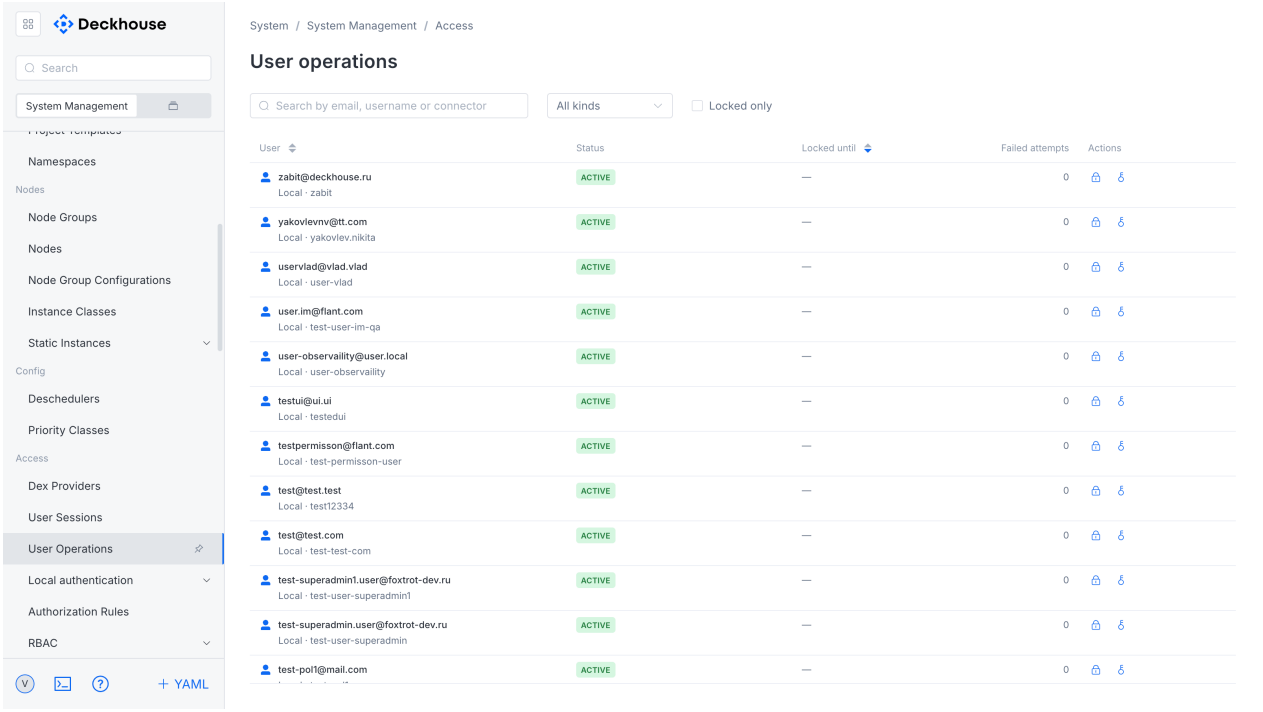




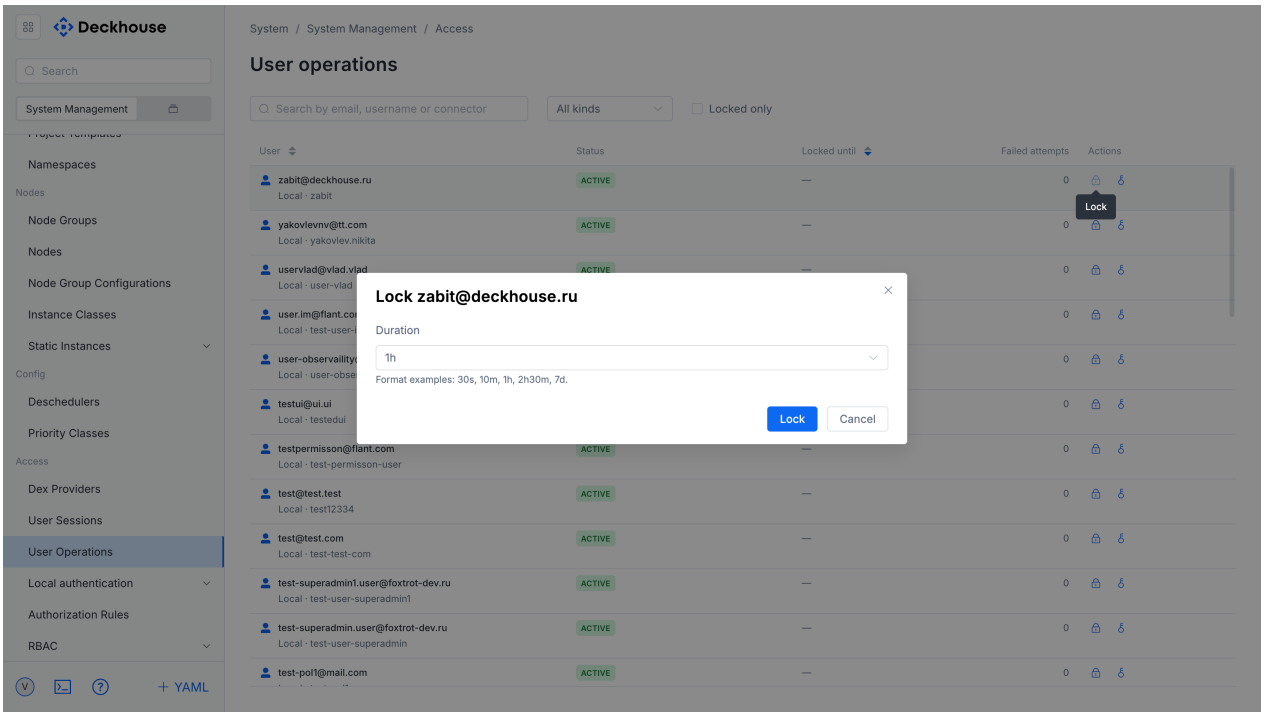
Access

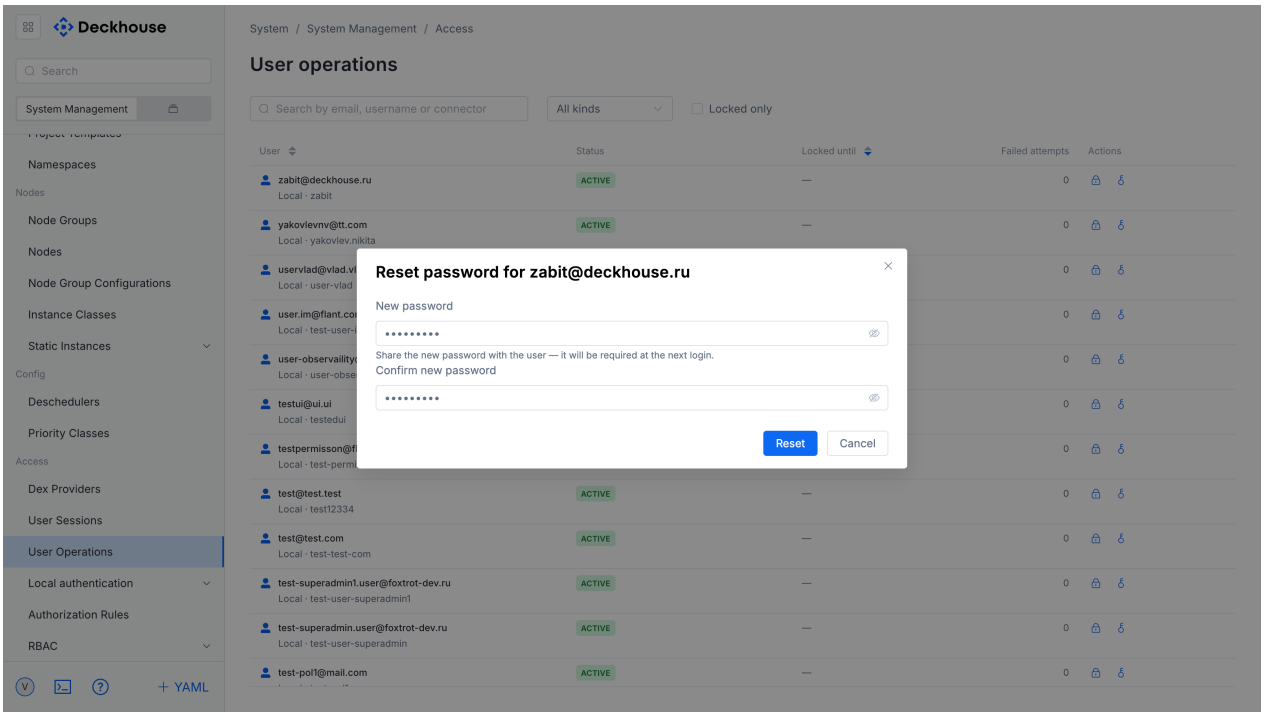
“User operations” subsection

This subsection provides an interface for user management.



If necessary, you can block a user or reset a password.





Network

“Ingress controllers” subsection

This subsection displays information about the current Ingress controllers responsible for traffic routing inside the DP cluster. If necessary, you can sort the list of controllers and add new ones.

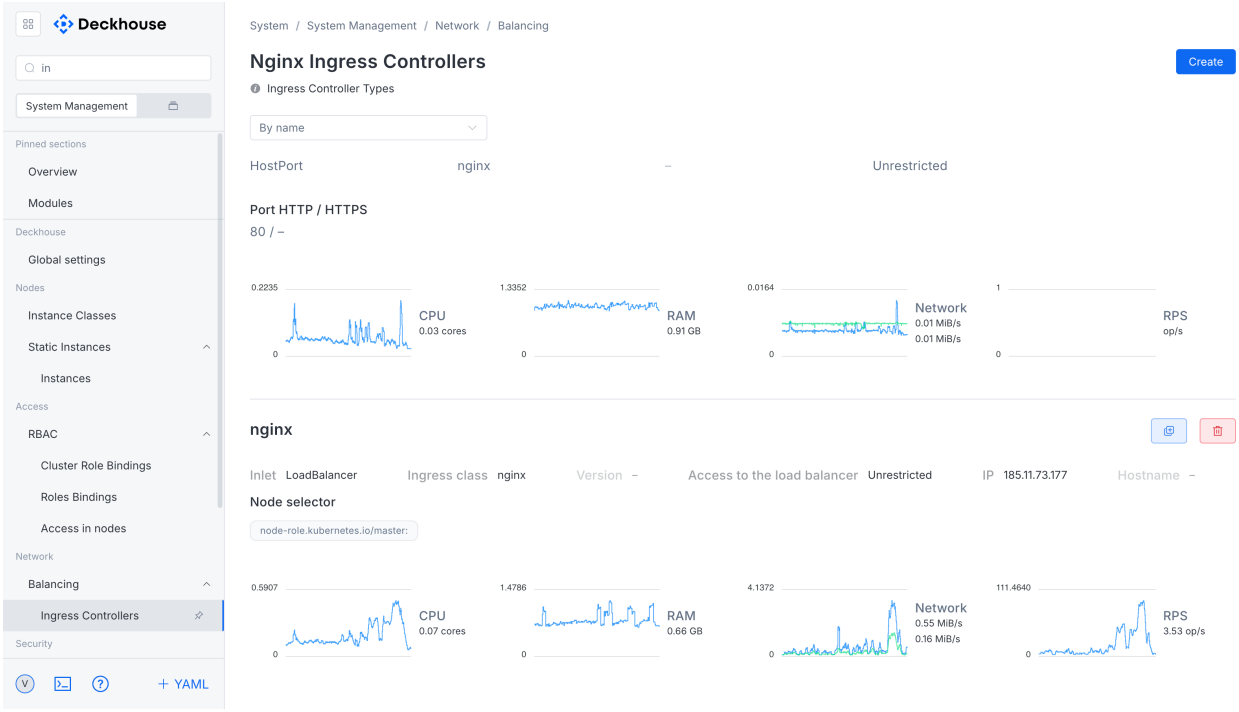
The “Nginx Ingress controllers” card displays its main parameters:

- Inbound connection type (LoadBalancer)
- IP address and Ingress class (`nginx`)
- Load balancer access level
- Node selector that determines on which nodes the controller runs

The lower part of the card contains monitoring charts for CPU usage, memory usage, network traffic, and requests per second (RPS), allowing you to monitor controller performance. The “Create from” option is available for cloning the configuration, and the “Delete” option is available for removing the Ingress controller.

The “Add” button in the upper-right corner allows you to select the type of a new inbound connection (inlet) for the Ingress controller. Available options:

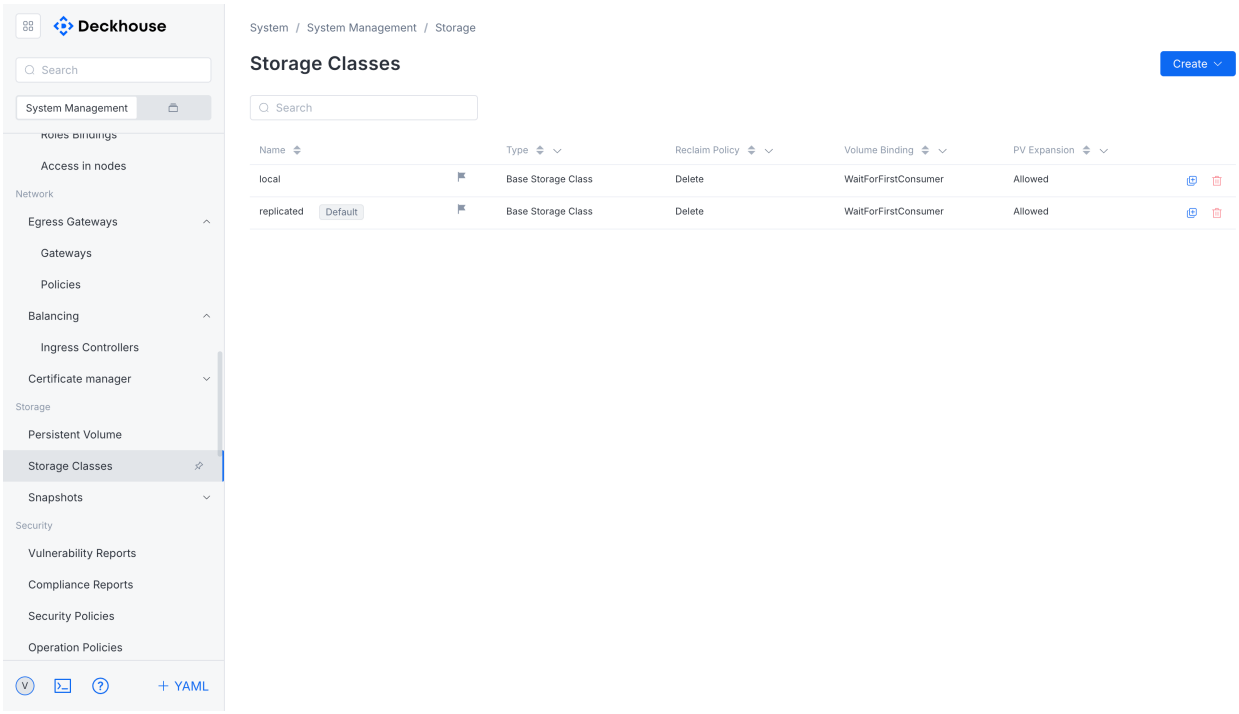
- HostPort
- HostPort with Proxy Protocol
- HostPort with a failover controller
- LoadBalancer
- LoadBalancer with Proxy Protocol



Storage

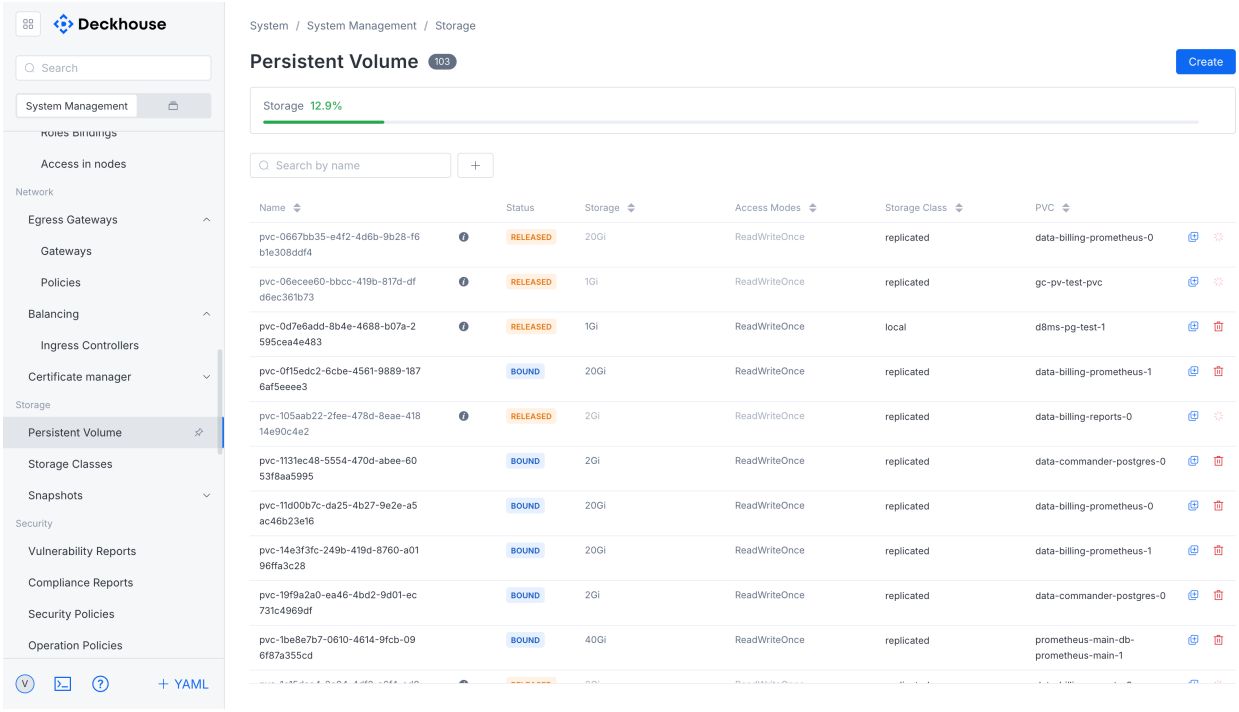
“Storage classes” subsection

This subsection is intended for managing storage classes.



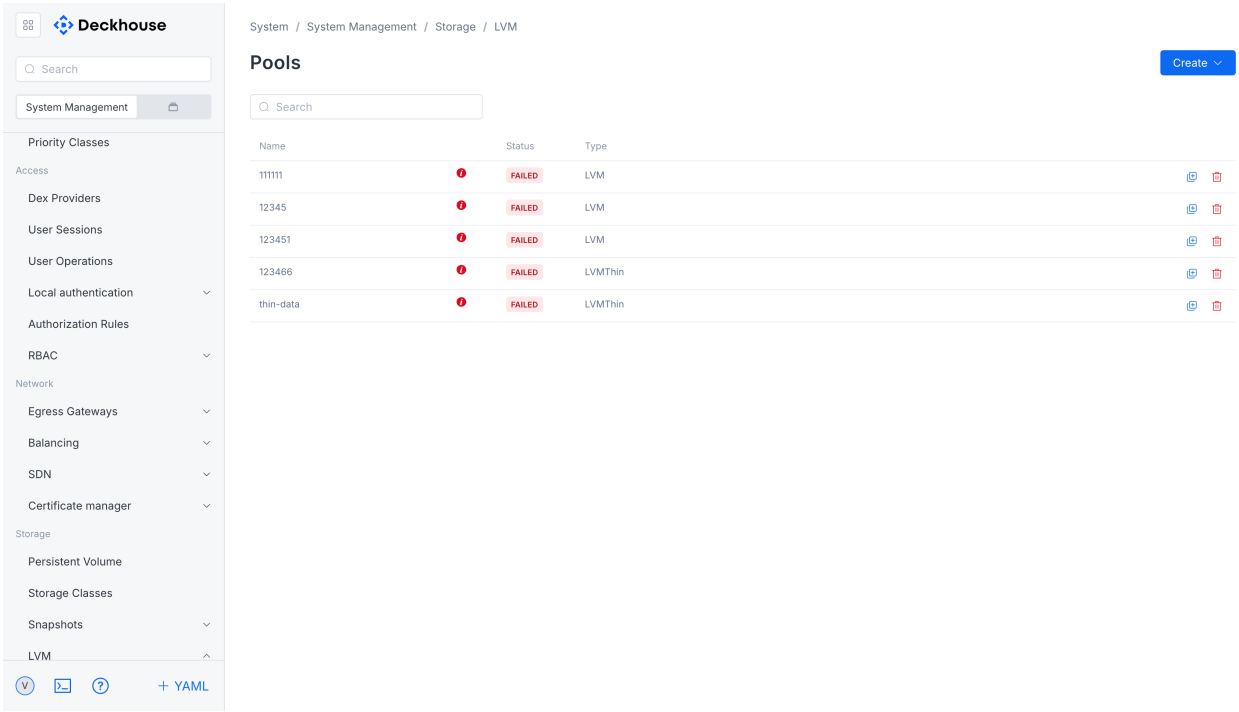
“Persistent volume” subsection

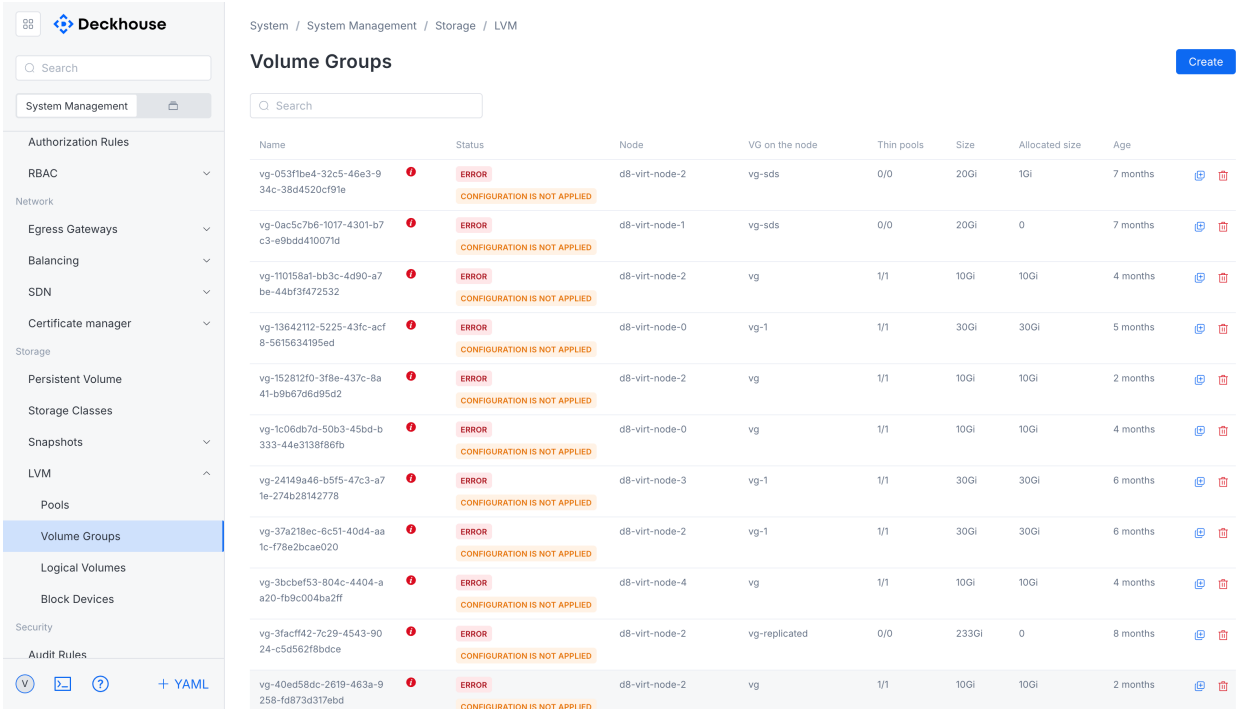
This subsection is intended for managing persistent volumes.

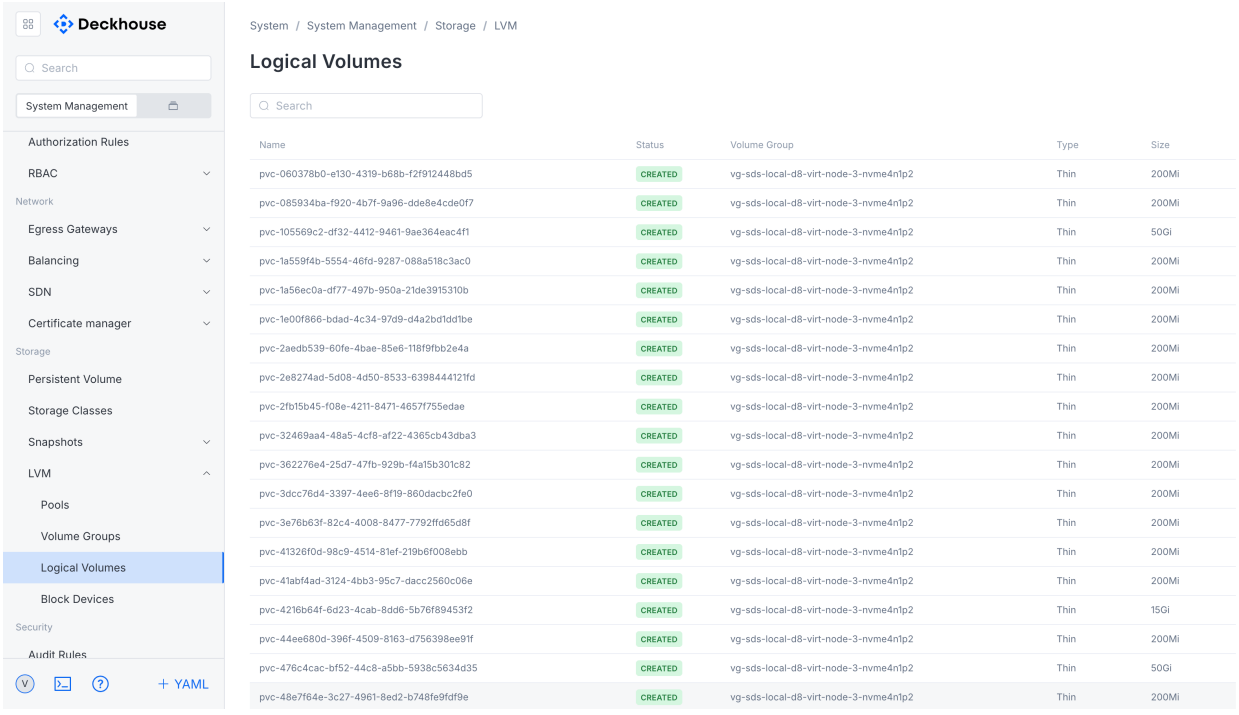


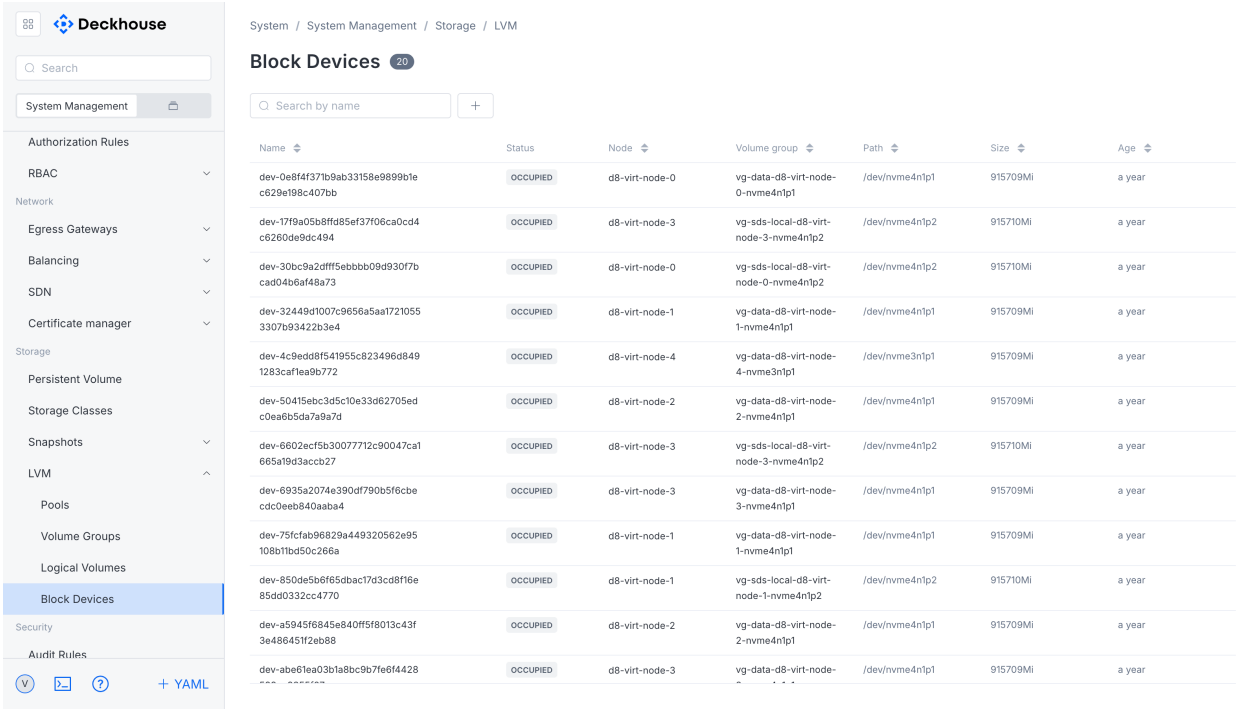
“Local volumes” subsection

This subsection is intended for managing local volumes. It contains four sections for managing pools, volume groups, logical volumes, and block devices.




Deckhouse
Search
System Management
Authorization Rules
RBAC
Network
Egress Gateways
Balancing
SDN
Certificate manager
Storage
Persistent Volume
Storage Classes
Snapshots
LVM
Pools
Volume Groups
Logical Volumes
Block Devices
Security
Audit Rules
+ YAML
System / System Management / Storage / LVM
Volume Groups
Create
Search
Name
Status
Node
VG on the node
Thin pools
Size
Allocated size
Age
vg-053f1be4-32c5-46e3-934c-38d4520cf91e
ERROR
CONFIGURATION IS NOT APPLIED
d8-virt-node-2
vg-sds
0/0
20Gi
1Gi
7 months
vg-0ac5c7b6-1017-4301-b7c3-e9bdd410071d
ERROR
CONFIGURATION IS NOT APPLIED
d8-virt-node-1
vg-sds
0/0
20Gi
0
7 months
vg-110158a1-bb3c-4d90-a7be-44bf3f472532
ERROR
CONFIGURATION IS NOT APPLIED
d8-virt-node-2
vg
1/1
10Gi
10Gi
4 months
vg-13642112-5225-43fc-acf8-5615634195ed
ERROR
CONFIGURATION IS NOT APPLIED
d8-virt-node-0
vg-1
1/1
30Gi
30Gi
5 months
vg-152812f0-3f8e-437c-8a41-b9b67d6d95d2
ERROR
CONFIGURATION IS NOT APPLIED
d8-virt-node-2
vg
1/1
10Gi
10Gi
2 months
vg-1c06db7d-50b3-45bd-b333-44e3138f86fb
ERROR
CONFIGURATION IS NOT APPLIED
d8-virt-node-0
vg
1/1
10Gi
10Gi
4 months
vg-24149a46-b5f5-47c3-a71e-274b28142778
ERROR
CONFIGURATION IS NOT APPLIED
d8-virt-node-3
vg-1
1/1
30Gi
30Gi
6 months
vg-37a218ec-6c51-40d4-aa1c-f78e2bcae020
ERROR
CONFIGURATION IS NOT APPLIED
d8-virt-node-2
vg-1
1/1
30Gi
30Gi
6 months
vg-3bcbef53-804c-4404-aa20-fb9c004ba2ff
ERROR
CONFIGURATION IS NOT APPLIED
d8-virt-node-4
vg
1/1
10Gi
10Gi
4 months
vg-3facff42-7c29-4543-9024-c5d562f8bdce
ERROR
CONFIGURATION IS NOT APPLIED
d8-virt-node-2
vg-replicated
0/0
233Gi
0
8 months
vg-40ed58dc-2619-463a-9258-fd873d317ebd
ERROR
CONFIGURATION IS NOT APPLIED
d8-virt-node-2
vg
1/1
10Gi
10Gi
2 months


Deckhouse
Search
System Management
Authorization Rules
RBAC
Network
Egress Gateways
Balancing
SDN
Certificate manager
Storage
Persistent Volume
Storage Classes
Snapshots
LVM
Pools
Volume Groups
Logical Volumes
Block Devices
Security
Audit Rules
+ YAML
System / System Management / Storage / LVM
Logical Volumes
Search
Name
Status
Volume Group
Type
Size
pvc-060378b0-e130-4319-b68b-f2f912448bd5
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-085934ba-f920-4b7f-9a96-dde8e4cde0f7
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-105569c2-df32-4412-9461-9ae364eac4f1
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
50Gi
pvc-1a559f4b-5554-46fd-9287-088a518c3ac0
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-1a56ec0a-df77-497b-950a-21de3915310b
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-1e00f866-bdad-4c34-97d9-d4a2bd1dd1be
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-2aedb539-60fe-4bae-85e6-118f9fbb2e4a
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-2e8274ad-5d08-4d50-8533-6398444121fd
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-2fb15b45-f08e-4211-8471-4657f755edae
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-32469aa4-48a5-4cf8-af22-4365cb43dba3
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-362276e4-25d7-47fb-929b-f4a15b301c82
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-3dcc76d4-3397-4ee6-8f19-860dacbc2fe0
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-3e76b63f-82c4-4008-8477-7792ffd65d8f
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-41326f0d-98c9-4514-81ef-219b6f008ebb
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-41abf4ad-3124-4bb3-95c7-dacc2560c06e
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-4216b64f-6d23-4cab-8dd6-5b76f89453f2
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
15Gi
pvc-44ee680d-396f-4509-8163-d756398ee91f
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi
pvc-476c4cac-bf52-44c8-a5bb-5938c5634d35
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
50Gi
pvc-48e7f64e-3c27-4961-8ed2-b748fe9fdf9e
CREATED
vg-sds-local-d8-virt-node-3-nvme4n1p2
Thin
200Mi



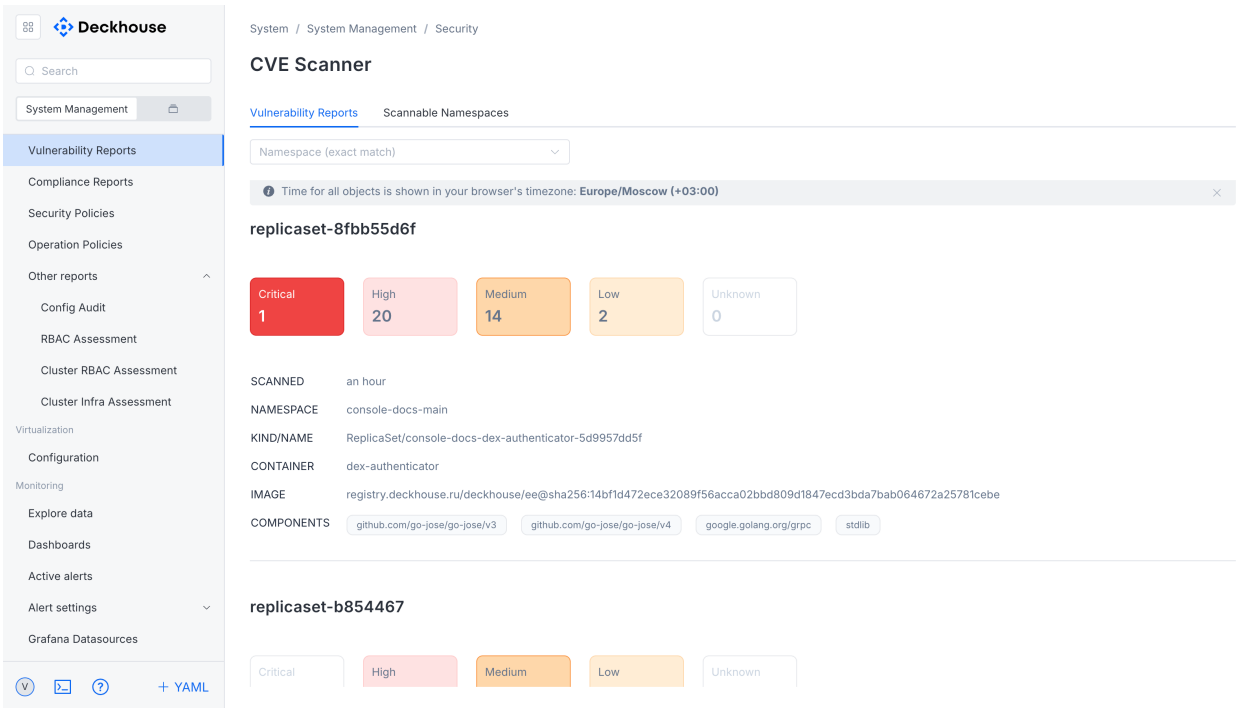
Security

“CVE scanner” subsection

This subsection is intended for scanning container images for vulnerabilities (CVEs) in the DP cluster.

The subsection contains the “Vulnerability reports” and “Scanned namespaces” tabs:

- “Vulnerability reports” tab displays the results of the latest scans. It provides information about the scanned object, including its name, namespace, type, and resource name, as well as the container and the image in use. If no vulnerabilities are found, a green indicator is displayed. The “Rescan” button allows you to perform a new scan.
- “Scanned namespaces” tab is intended for managing namespaces that are subject to scanning. The interface supports sorting by name and scan parameters. You can optionally hide system namespaces. The required namespaces can be selected for scanning, and scans can be started using the “Rescan” button. Reports for each object are also available for viewing.



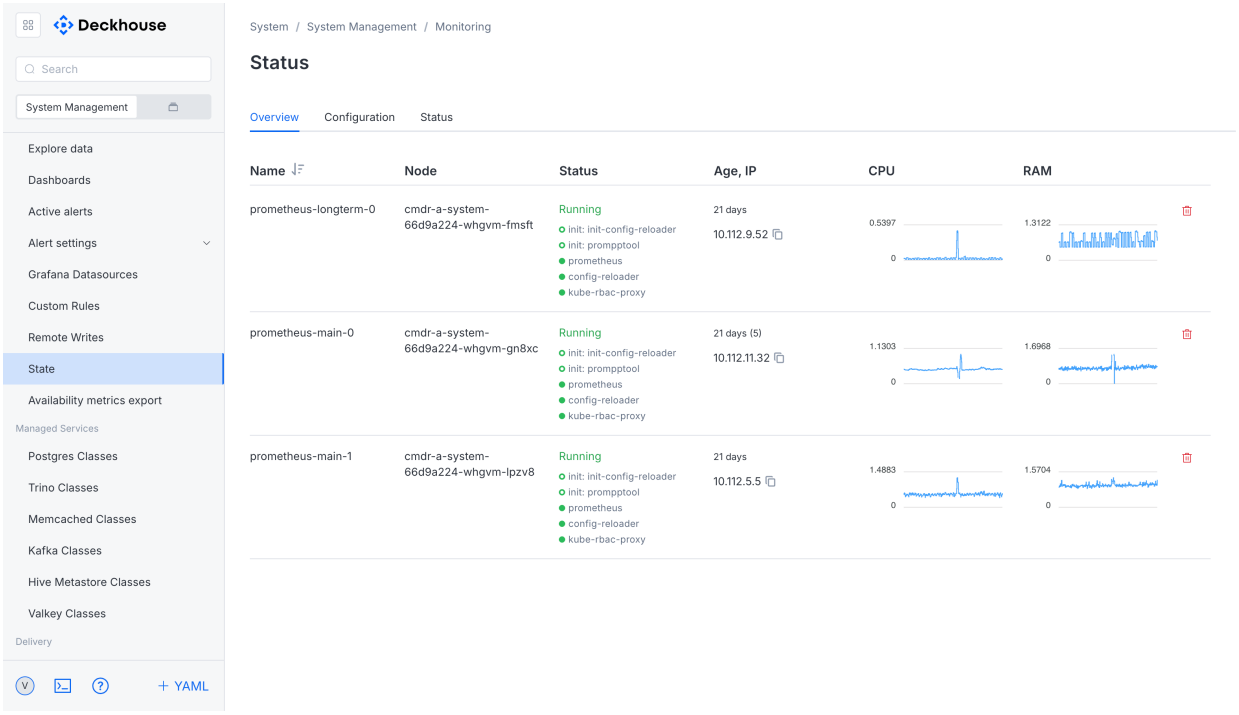
Monitoring

“Overview” subsection

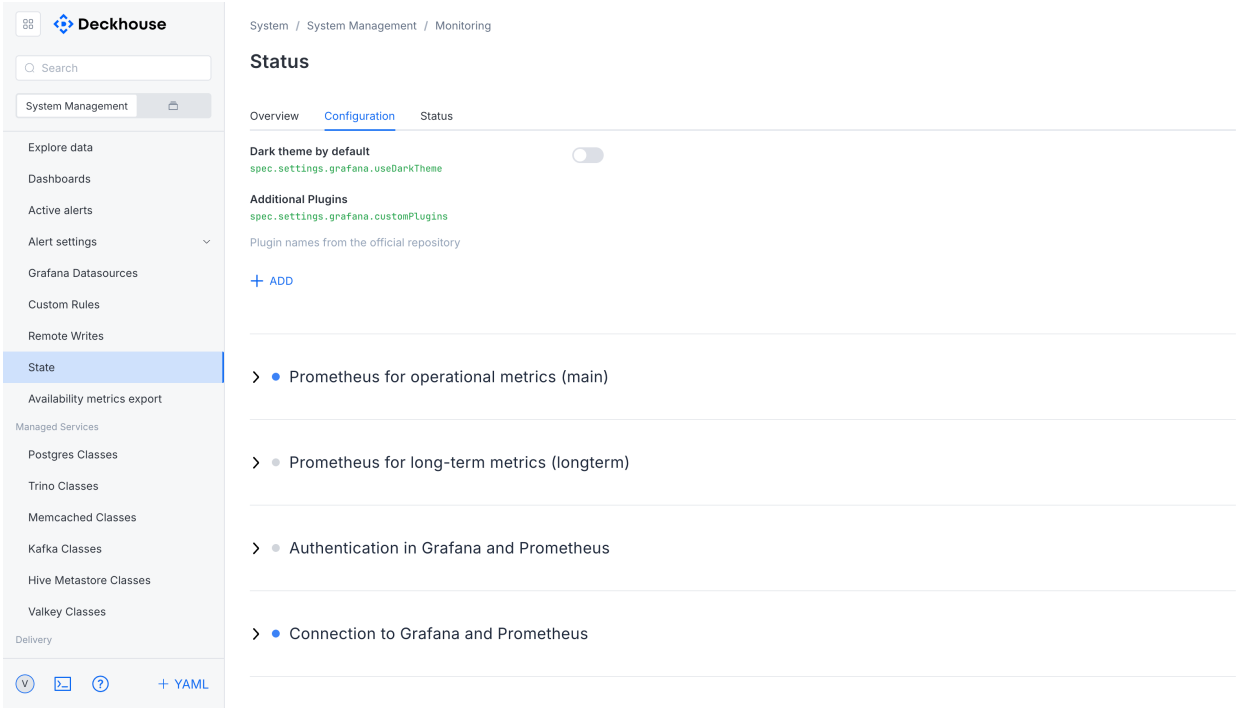
This subsection displays the state of Prometheus instances, load metrics, and a section with settings.

The “Overview” subsection contains the “Status” and “Configuration” tabs intended for monitoring and configuring Prometheus instances.

The “Status” tab displays a list of running pods with their names, assigned nodes, statuses, IP addresses, ages, and CPU and memory usage. Each pod includes components such as `init-config-reloader`, `prometheus`, `config-reloader`, and `kube-rbac-proxy`, which ensure its operation. The button on the right allows you to delete the selected pod.

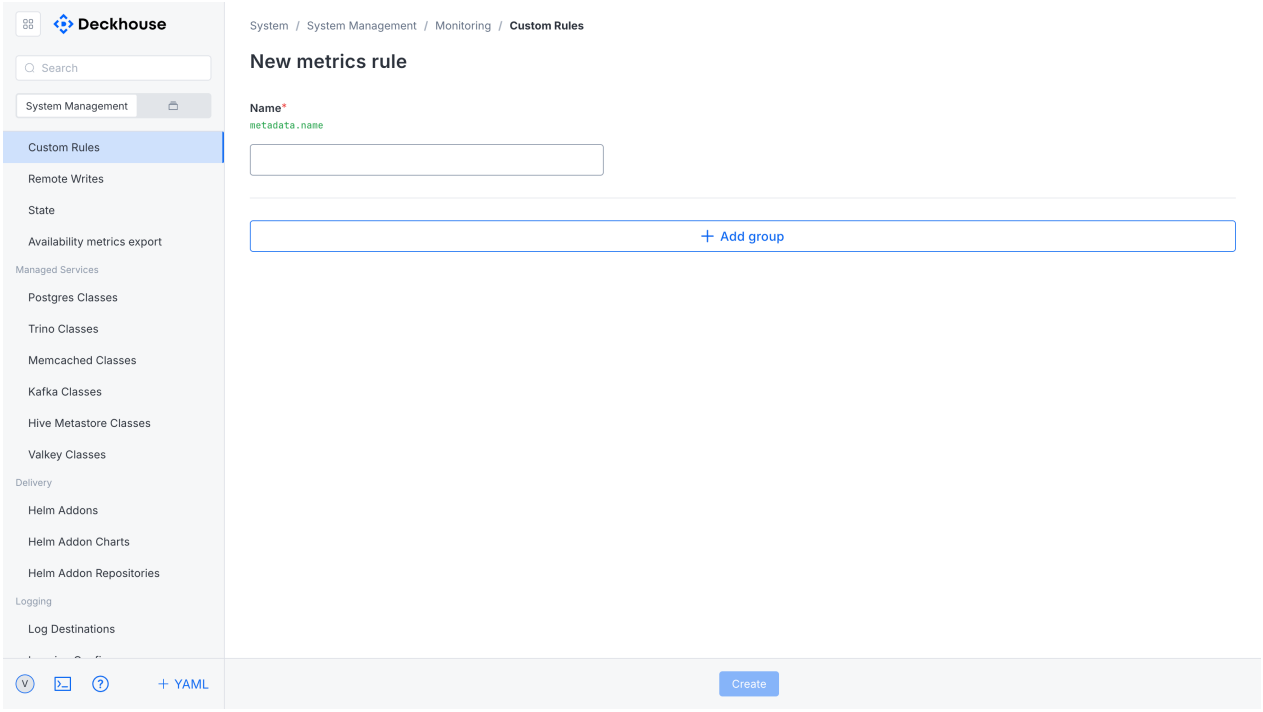


The “Configuration” tab contains expandable sections for configuring various aspects of Prometheus operation, including real-time and retrospective metrics, authentication and Grafana integration, as well as resource management. This interface is intended for monitoring metric states in real time and for flexible integration with other services.



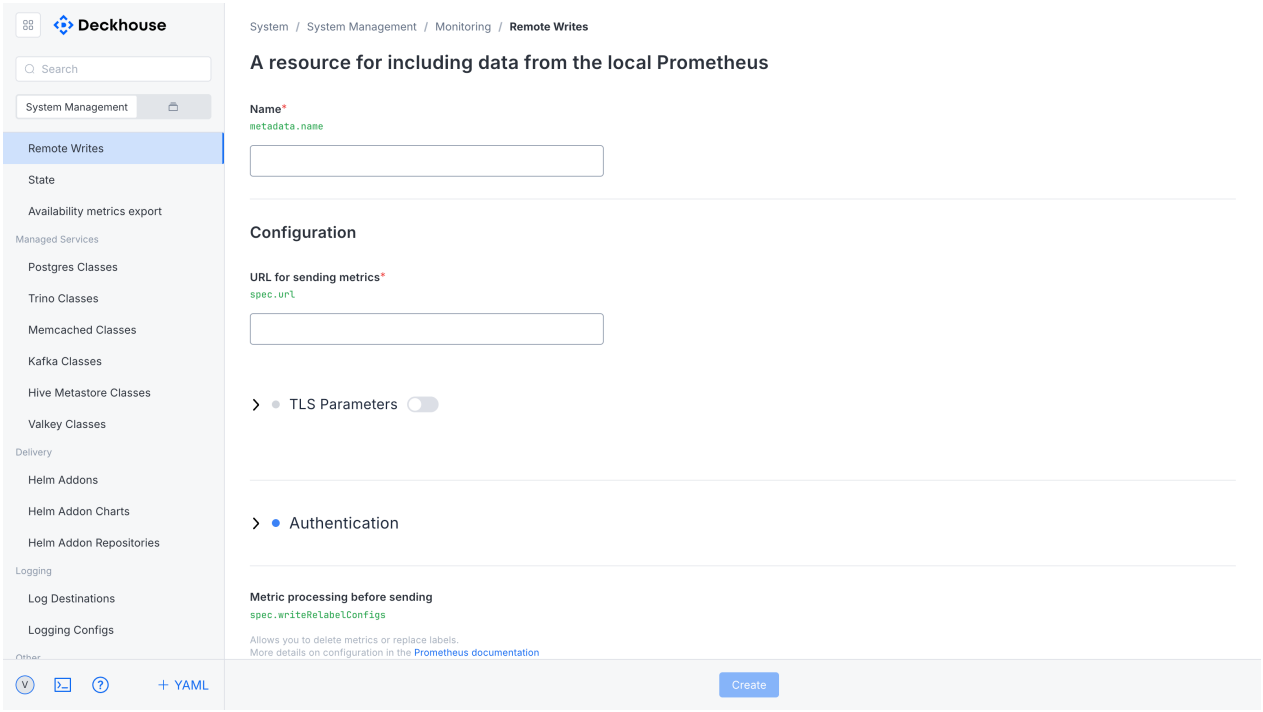
“Metric processing” subsection

This subsection allows you to create and manage metric processing rules. When adding a new rule, specify its name and processing group. This makes it possible to organize and modify incoming metrics before they are forwarded further.



“Metric delivery” subsection

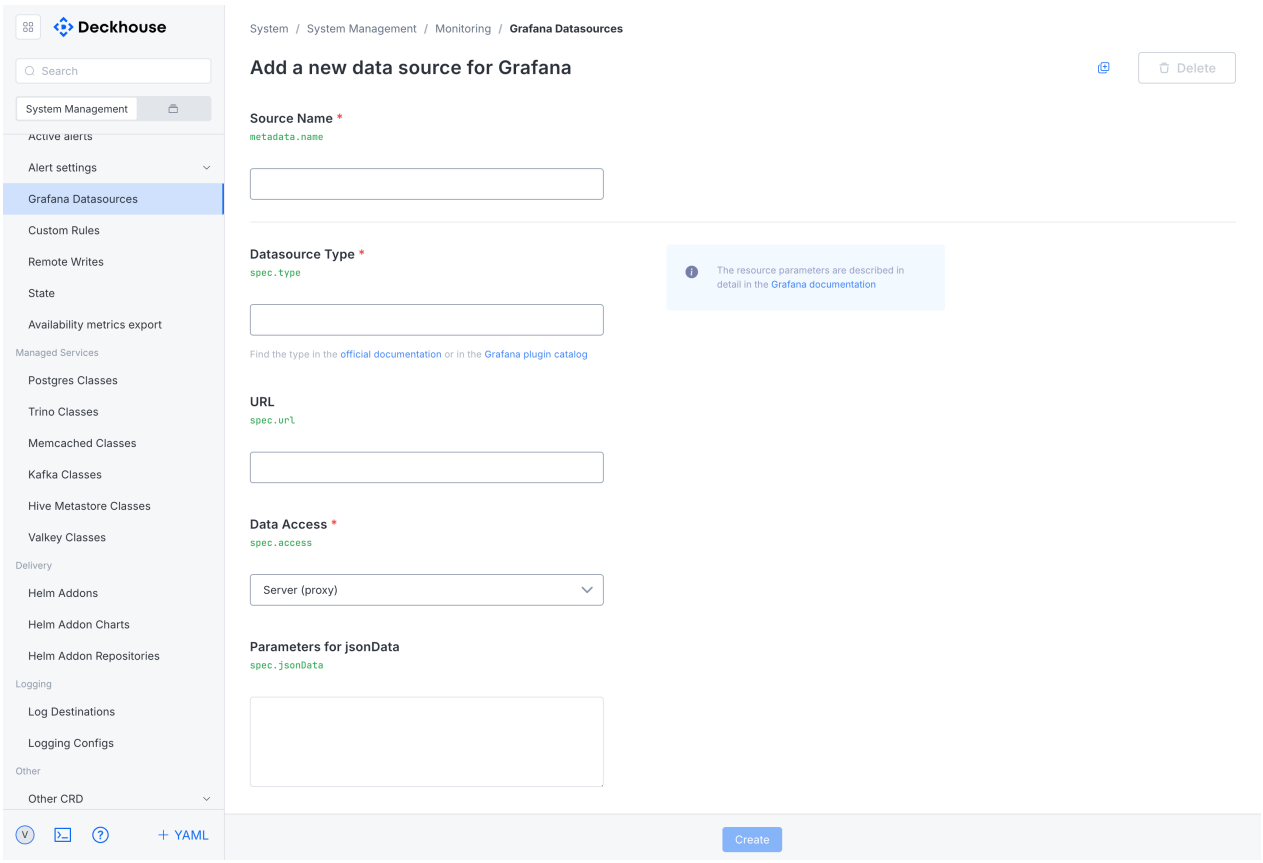
This subsection is intended for configuring data export to a local or external Prometheus server. When adding a new resource, specify the URLs for metric delivery, and configure TLS parameters, authentication, and optional metric preprocessing before delivery.



“Grafana data sources” subsection

In this subsection, you can configure Grafana integration with various data sources used in dashboards and visualize data.

To create a new data source, specify its name, type, URL, access parameters, and authentication settings.



“Grafana dashboards” subsection

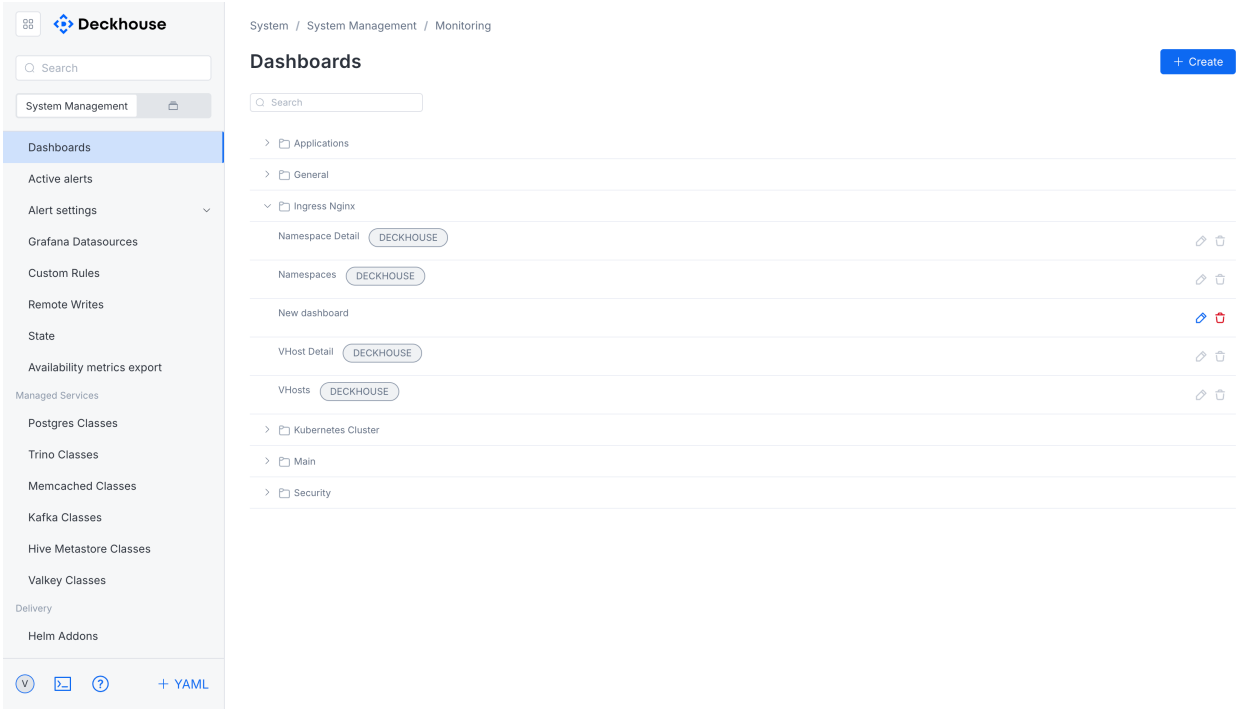
This subsection is intended for managing dashboards used for metric visualization.

The main interface displays a list of available dashboards. You can sort by creation time and filter by name or folder.

The dashboard row displays its name, associated folder, and the “Create from” option for creating a new dashboard based on it, as well as the “Delete” option.

To add a new dashboard:

1. Click the “Add” button.
2. Specify the new dashboard name and the folder in which it will be stored (if the folder does not exist, it will be created automatically).
3. Enter a JSON manifest containing the dashboard configuration description. It is important that the manifest does not contain a local ID (except for the UID), otherwise the dashboard may not be displayed correctly in Grafana.

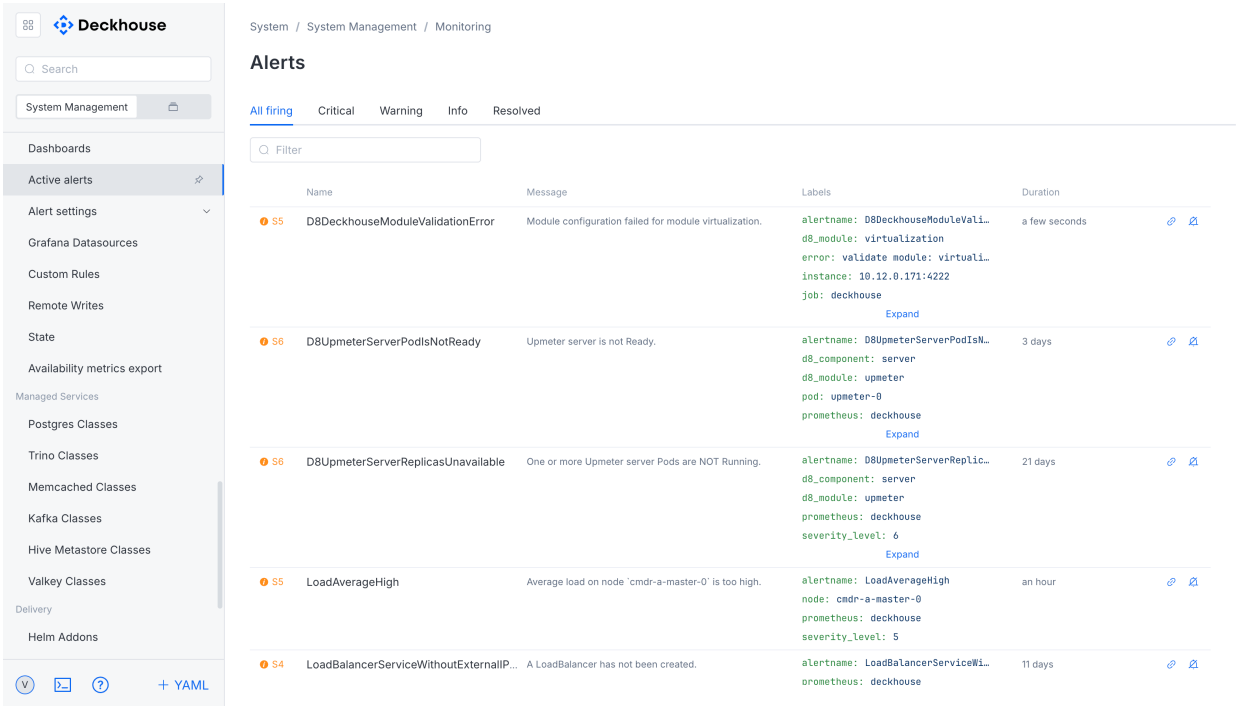


“Active alerts” subsection

This subsection displays a list of current monitoring system alerts. Alerts can be sorted by name and status.

The alert row displays the alert name, severity level, creation time, and last update time. Related components and modules are also displayed to help identify the source of the issue.

To get detailed information about an alert, click “Read description”. At the bottom, an explanation of the alert trigger reason is displayed.



A list of all monitoring system alerts available in DP is provided on a separate documentation page.

Logging

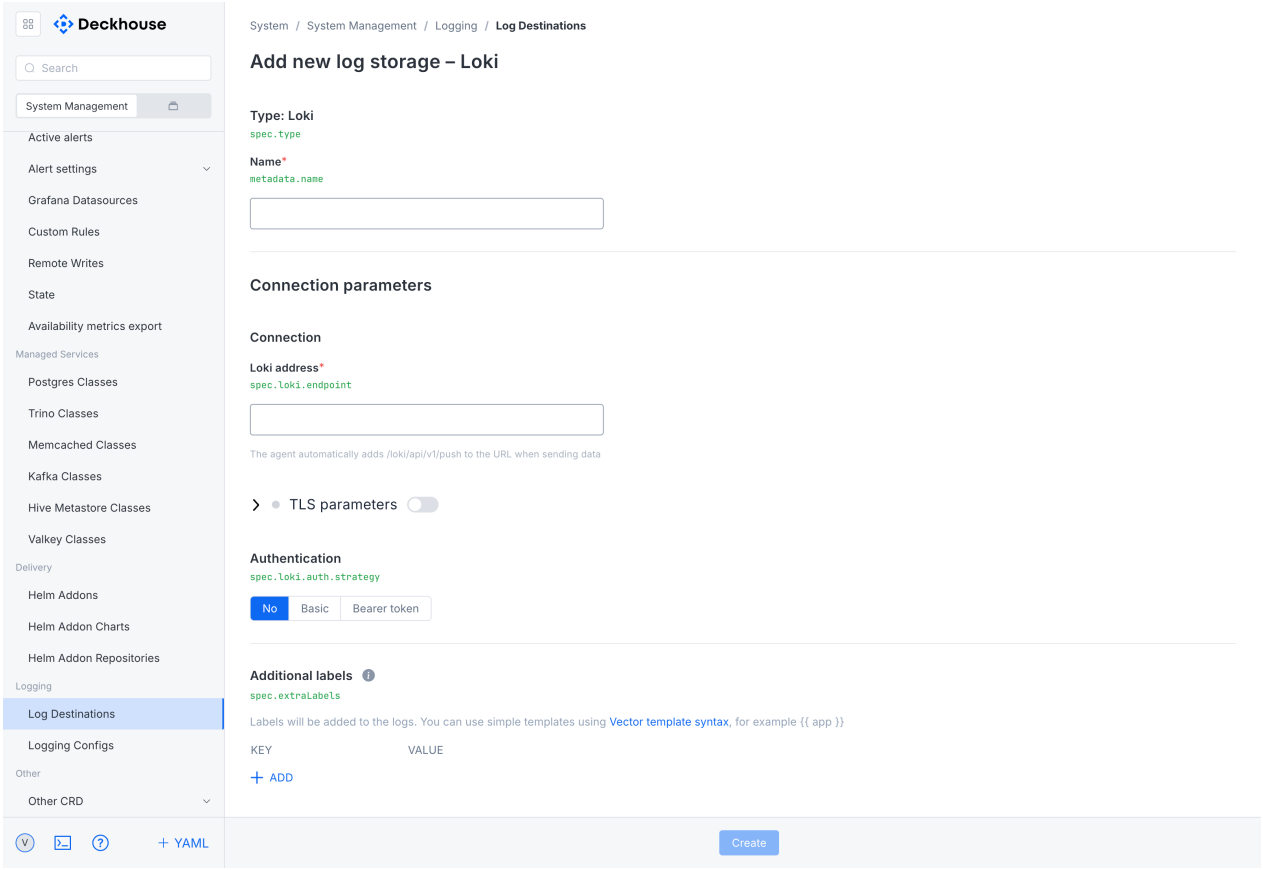
This section is intended for configuring log sources and managing log delivery.

“Log delivery” subsection

This subsection is intended for managing logging and configuring log delivery to various storage systems.

To configure log delivery:

1. Click the “Add” button.
2. Select one of the target storage systems:
   - Loki
   - ElasticSearch
   - Logstash
   - Vector
   - Kafka
   - Splunk
3. Specify the storage system name and connection address.
4. If necessary, specify additional settings:
   - TLS parameters for a secure connection.
   - Authentication (“Basic” or “Bearer token”).
   - Additional labels for filtering and organizing logs.
   - Buffer parameters that define how logs are stored before delivery (on disk or in memory).
   - Delivery limits that allow you to configure the log record delivery rate.
   - Exclusions that allow you to filter specific logs.

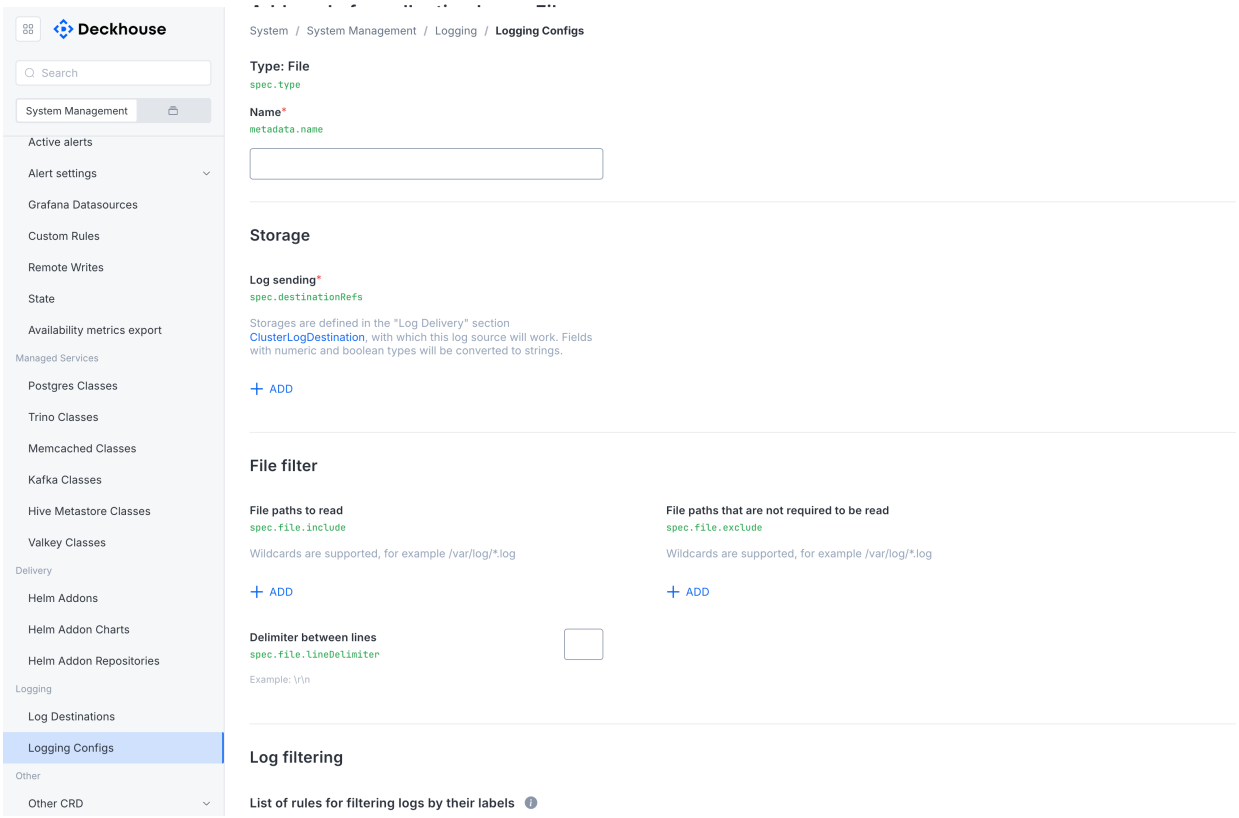


“Log collection” subsection

This subsection is intended for configuring log sources that can then be delivered to target storage systems. You can sort and filter existing log collection rules and add new sources.

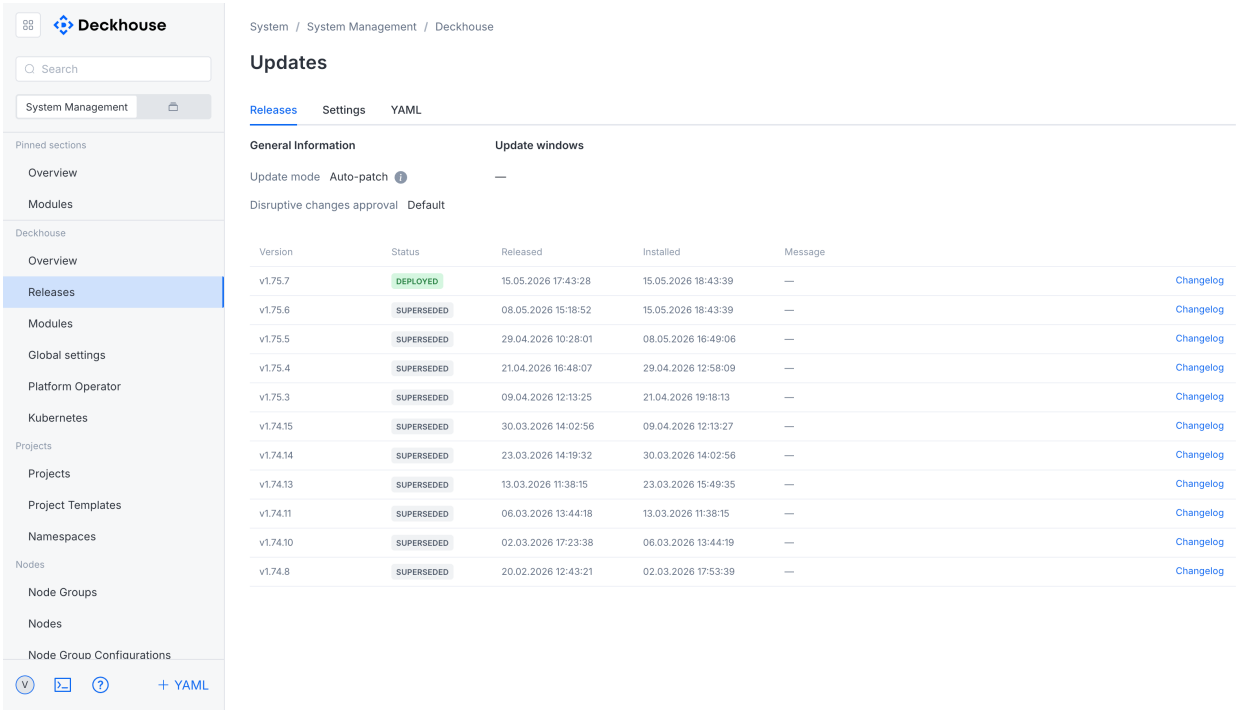
To configure collection sources:

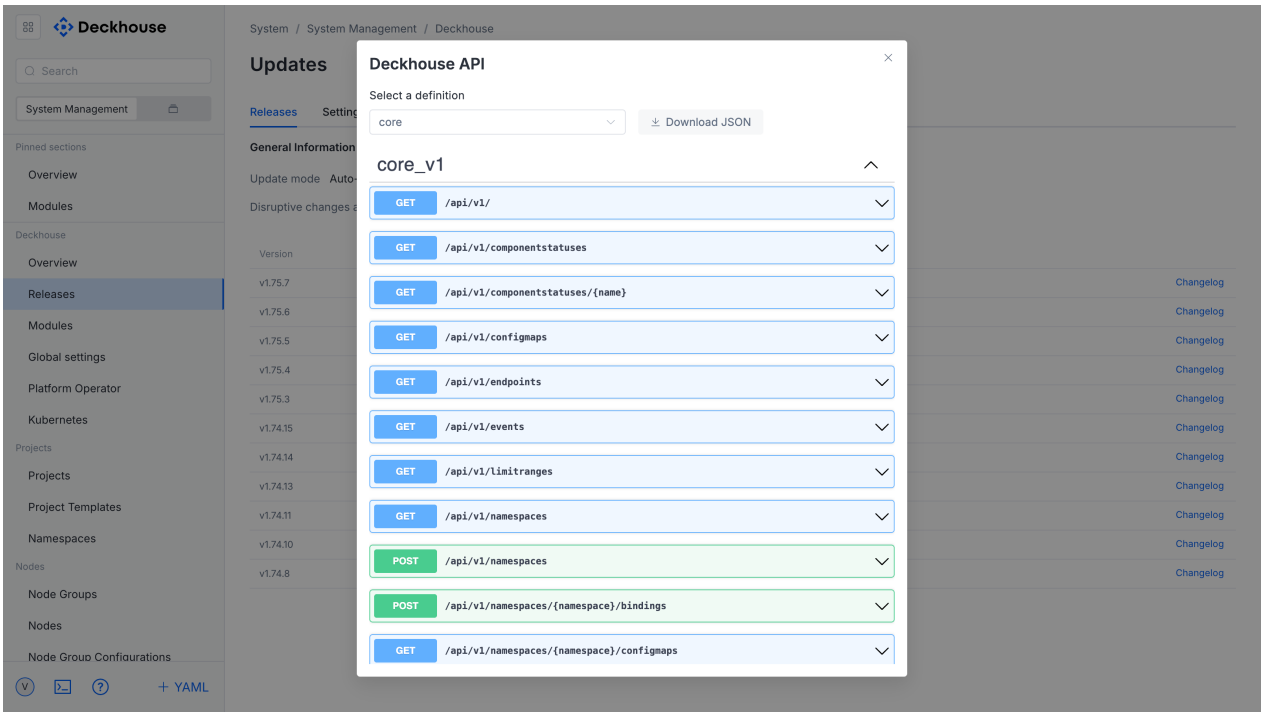
1. Click the “Add” button.
2. Select one of the sources:
   - File (collecting logs from the file system)
   - KubernetesPods (collecting logs from Kubernetes pods)
3. Specify the name and configure the parameters:
   - “Storage”: Specify where the collected logs will be delivered.
   - “File filters”: Specify the paths to log files that should or should not be read. Wildcards are supported.
   - “Line delimiter” (optional): Specify the character separating records in the file;
   - “Log filtering” (optional): Add label filtering rules and filters to keep only the required records.

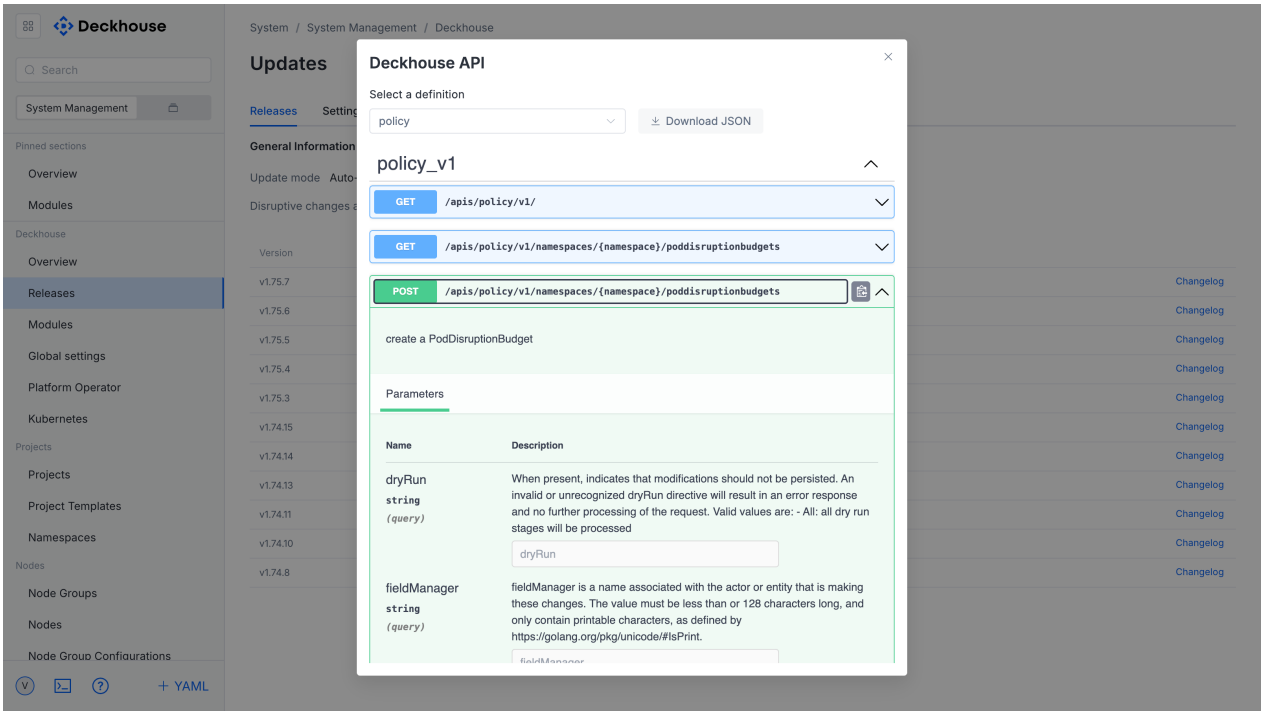


Viewing API information

To view API information in Swagger format, click the question mark button in the lower-left corner of the interface in the drop-down user menu.







Additional resources

- Global parameters
- Module `prometheus` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Grafana web interface

Monitoring system web UI

Grafana is used as the web UI for the monitoring system.

Main page

The main page of Grafana displays key information about the cluster and its main components.

In the top left corner, you'll find details about the main cluster components: the `containerd` version, the Linux distribution the cluster is running on, and the versions of Grafana, Prometheus, and other components.

The top right corner features convenient graphical indicators for key metrics including the number of nodes in the cluster, the number of running Pods, and other cluster entities.

For more detailed information, hover the mouse cursor over any element to display a tooltip.

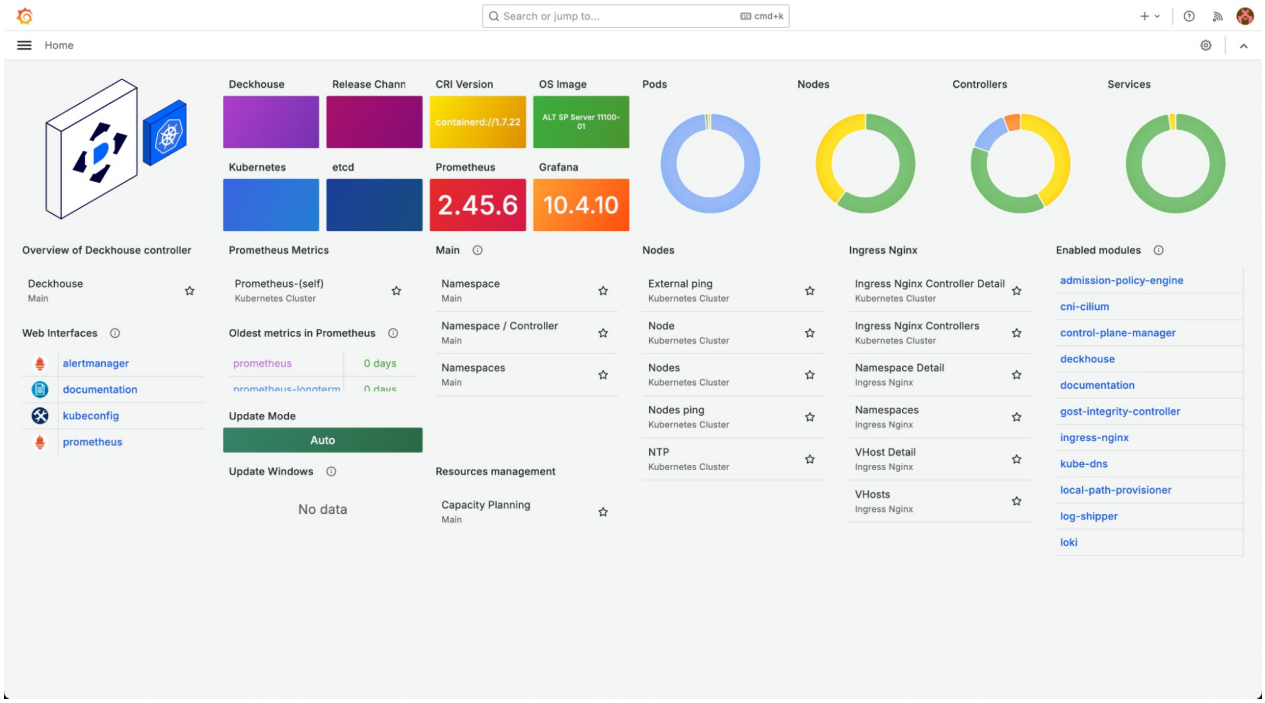
Below the component info blocks, there are quick links to dashboards for specific cluster components, such as node monitoring, system resource usage, and network interaction statistics.

On the left side, you'll find a section with links to the cluster's available web interfaces, as well as a block showing the cluster update method and the scheduled update windows (if configured).

In the top left corner of the main page is a button that opens the side menu, which contains links to Grafana's core sections.

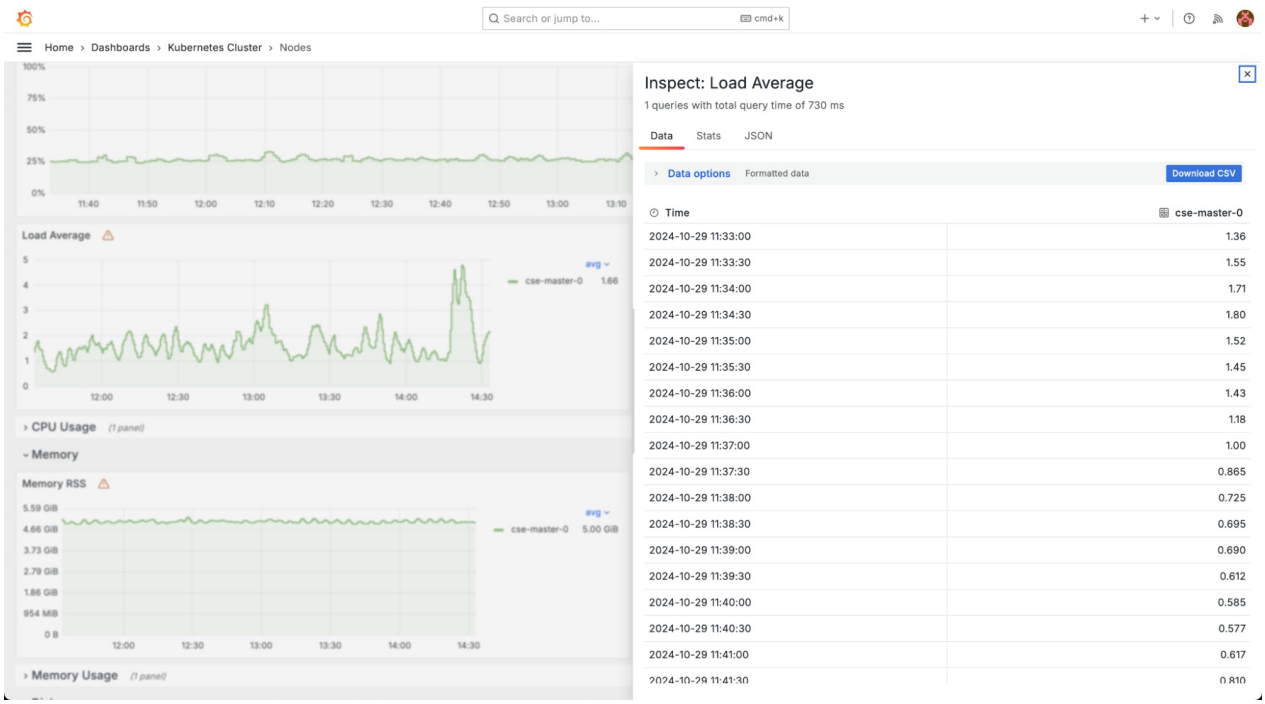
Clicking the **Dashboards** tab opens a list of all available Deckhouse dashboards, grouped by category.

These dashboards have a nested structure and are grouped by purpose: cluster applications (**Applications**), network interaction (**Ingress Nginx**), cluster parameters (**Kubernetes Cluster**), **Main**, and **Security**.



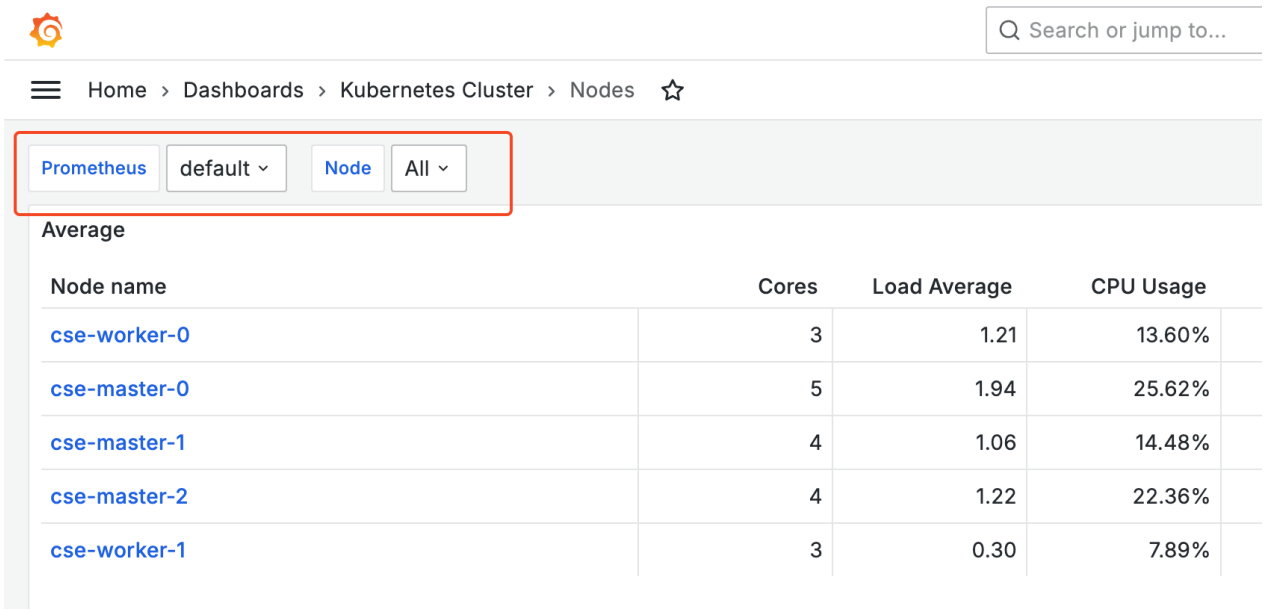
Working with dashboards

A dashboard is a page with tables and graphs that display information about a selected cluster component.

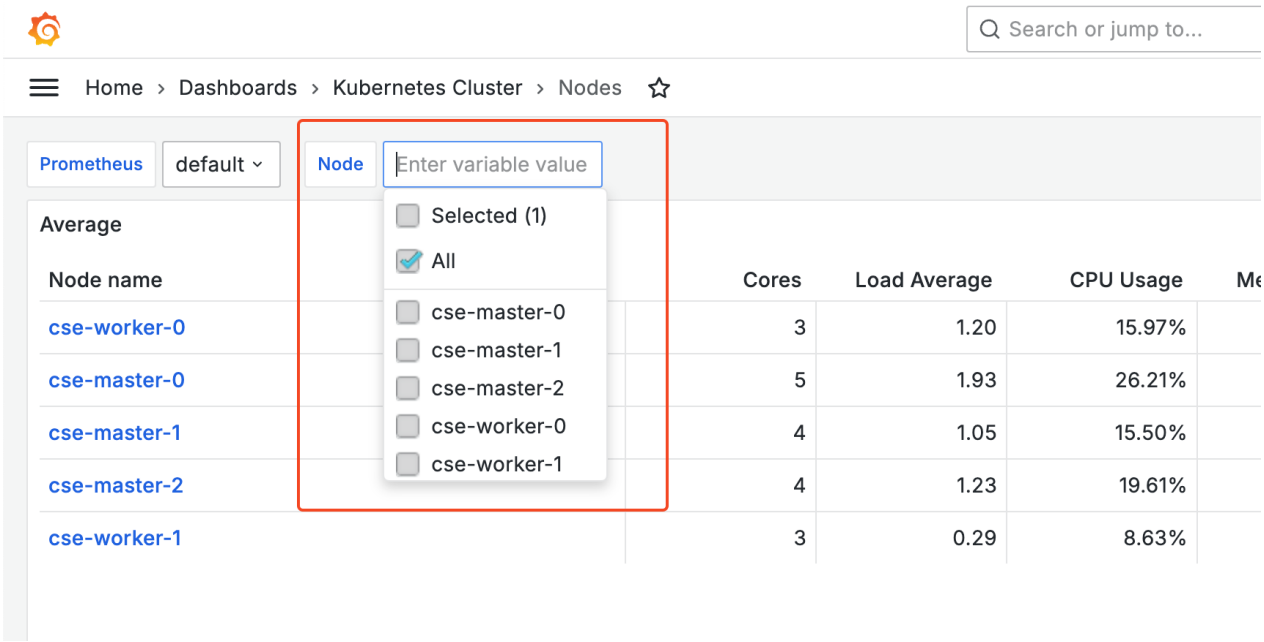


Data filtering

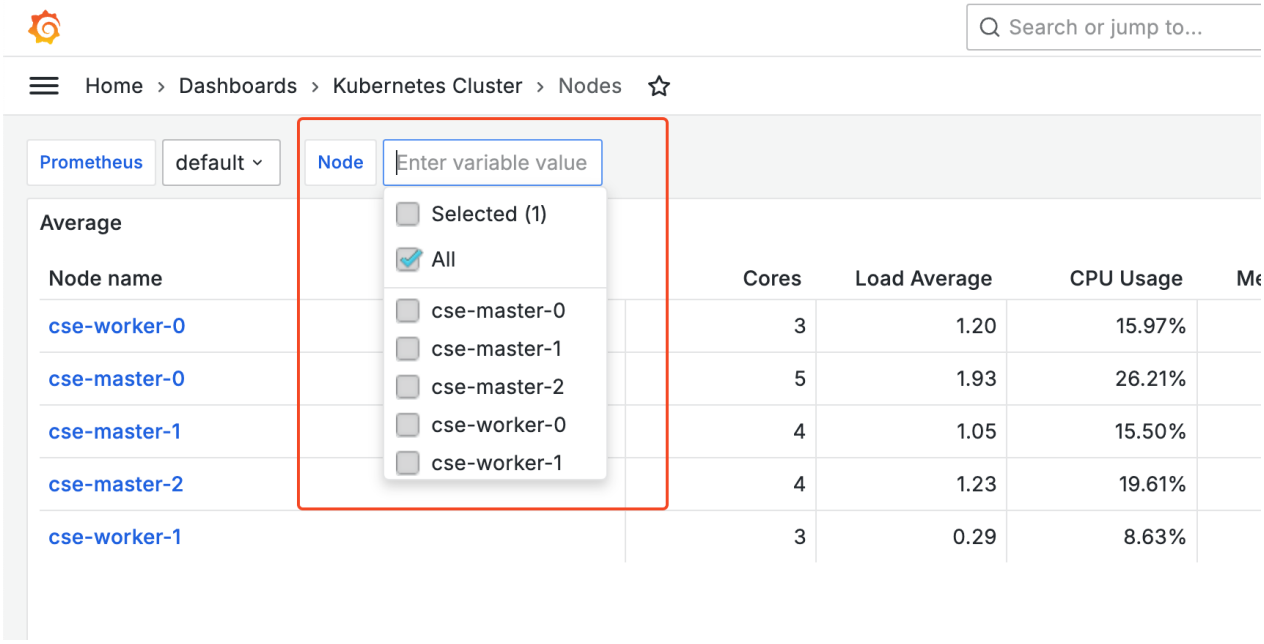
At the top of the page, below the title and the quick link to the parent category, you can find a filter block. This block lets you customize the displayed data by excluding irrelevant information or focusing on a specific component.



For example, in the dashboard showing cluster nodes, you can use the filter to display only one of the three nodes, excluding the others.

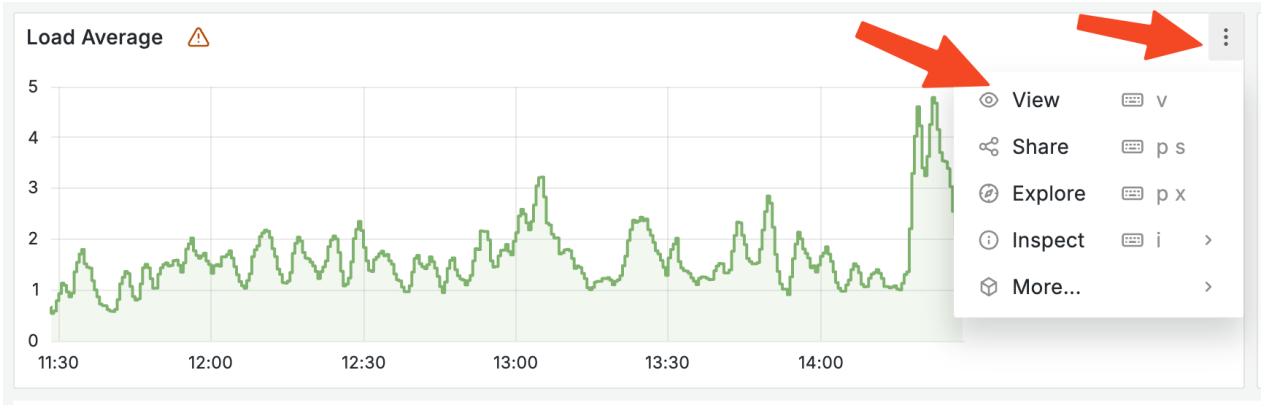


Once a filter is selected, the dashboard will update immediately to show data that matches the chosen filter parameters.

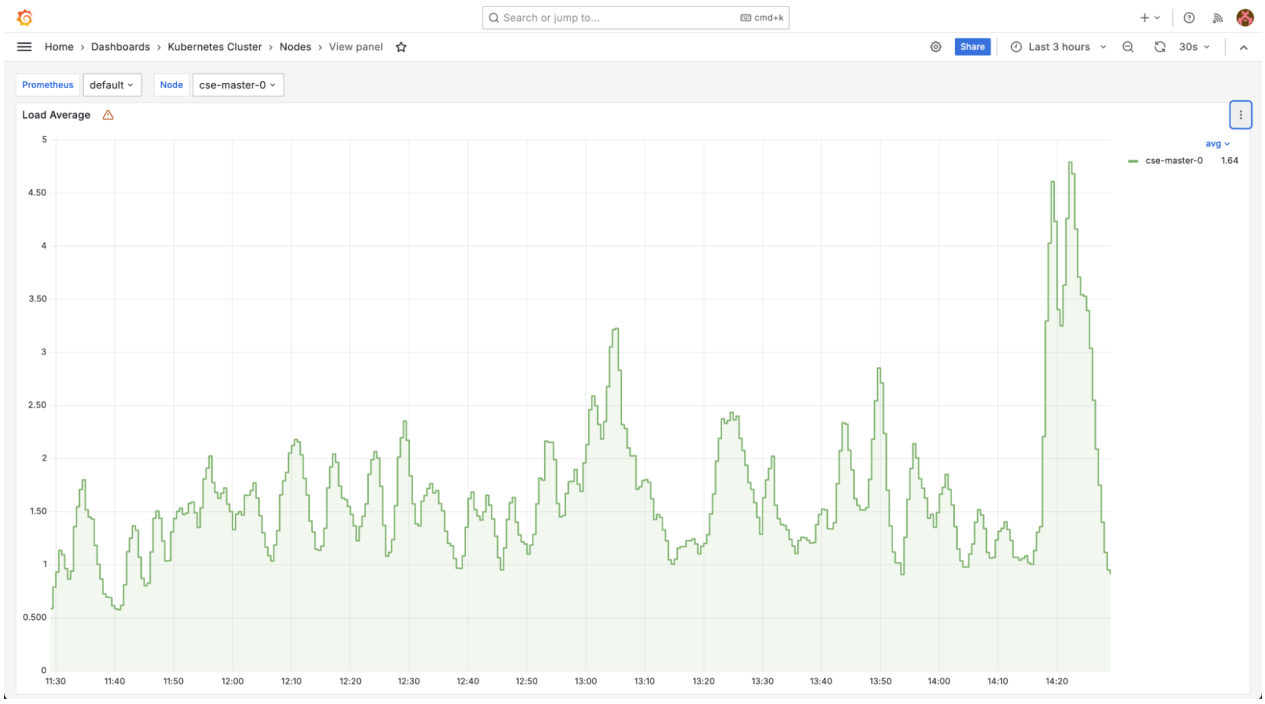


Working with data

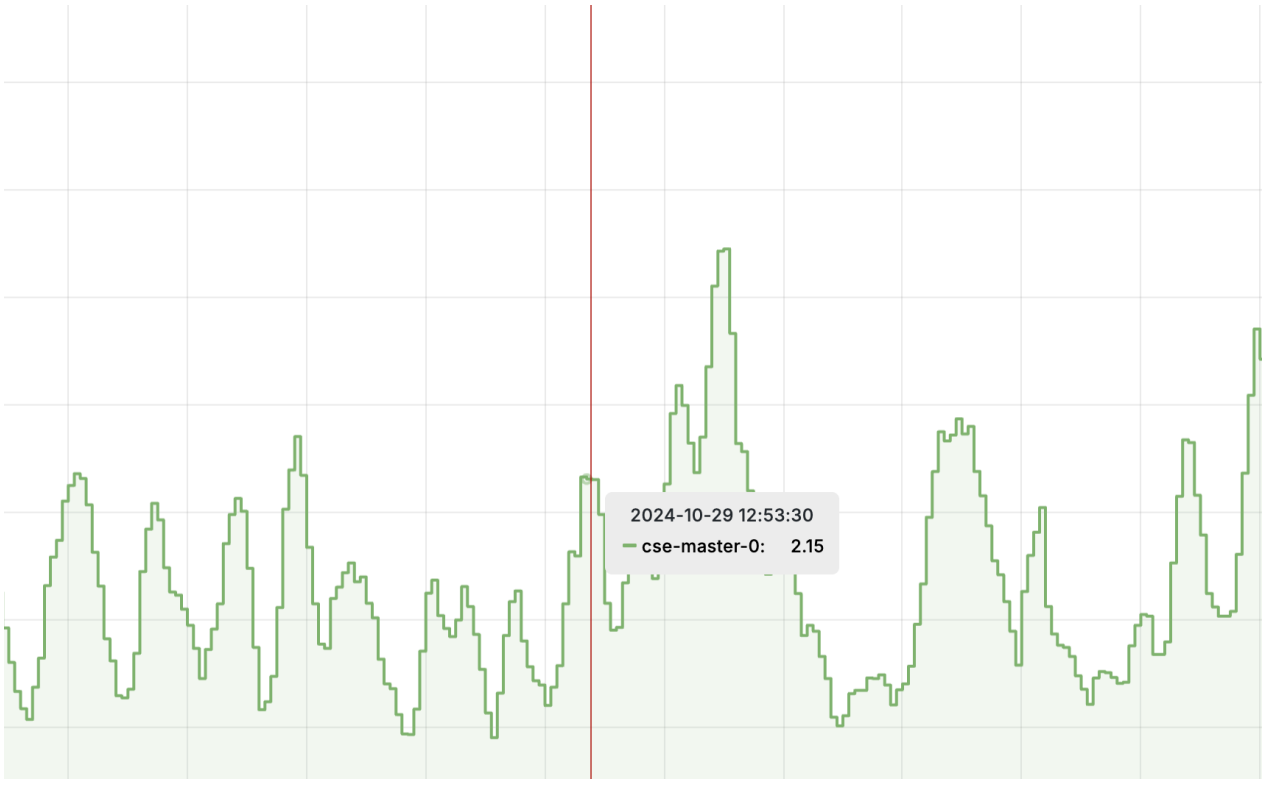
Each chart on a dashboard can be viewed in more detail. To do this, hover the mouse cursor over the top right corner of the chart block, click the three-dot menu button, and select **View**.



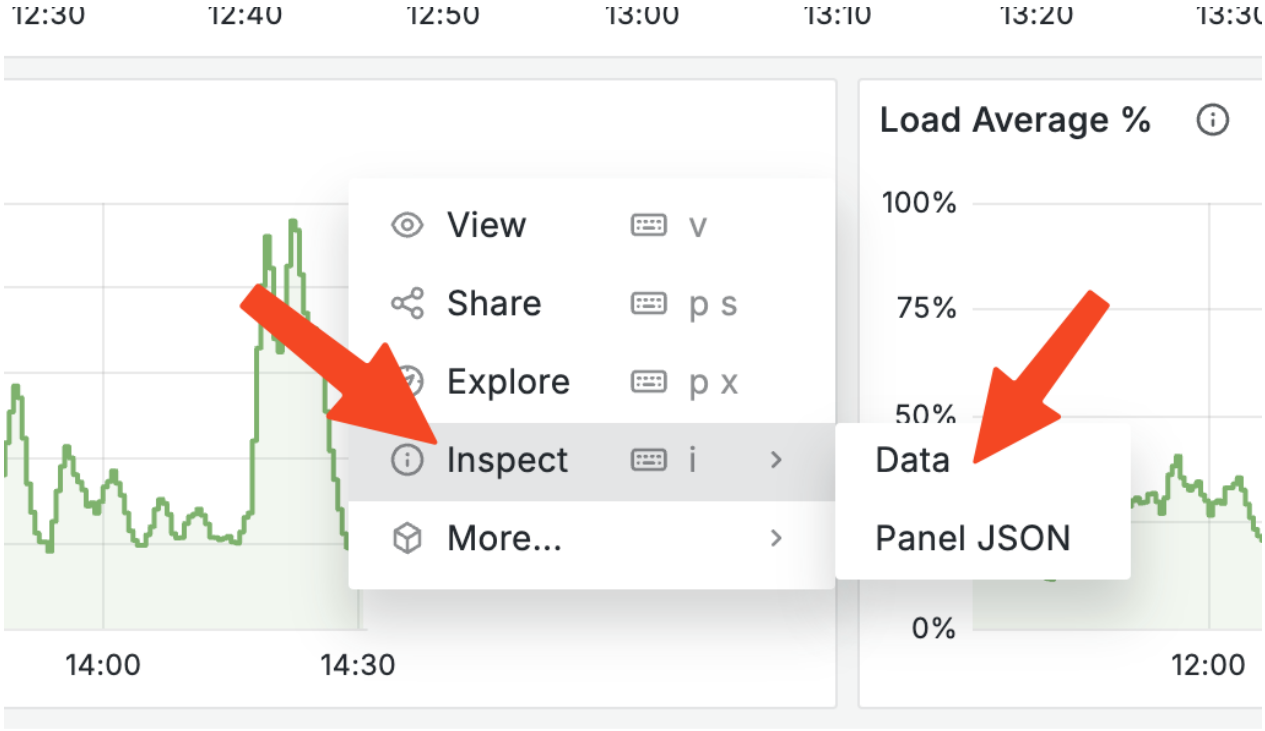
The selected chart will expand to full-screen view.



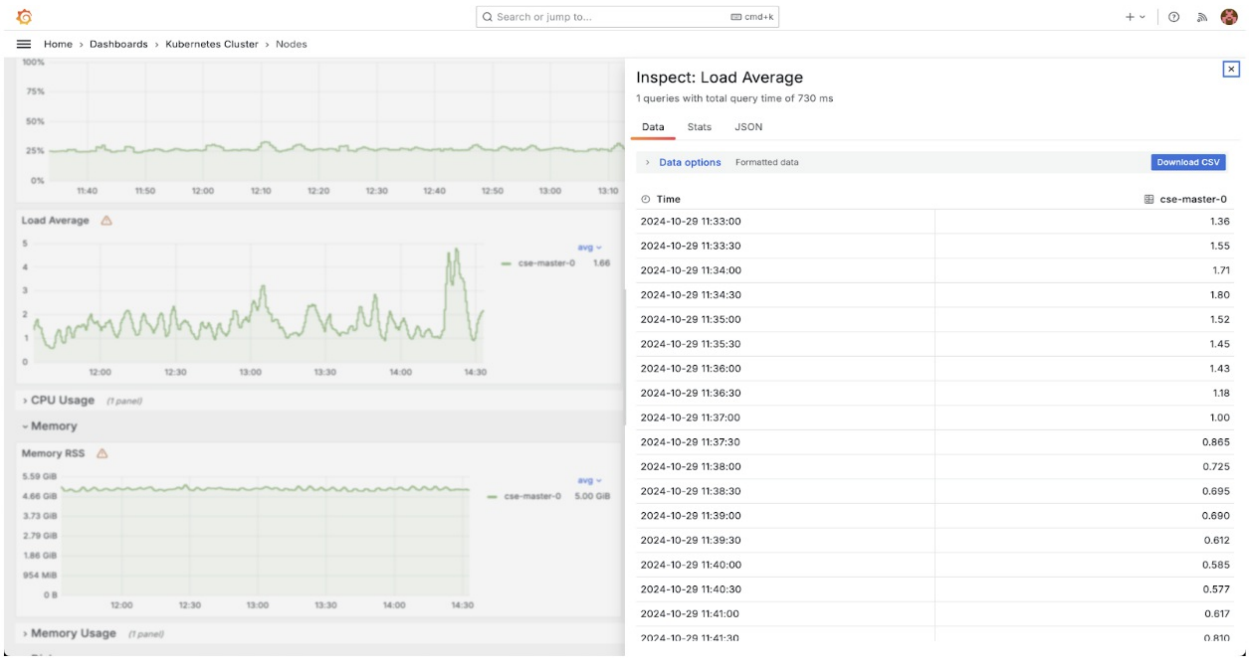
Here you can explore detailed data for a specific point in time. Hover over the graph. A red horizontal line will appear, and a tooltip will display the timestamp and value at that moment:



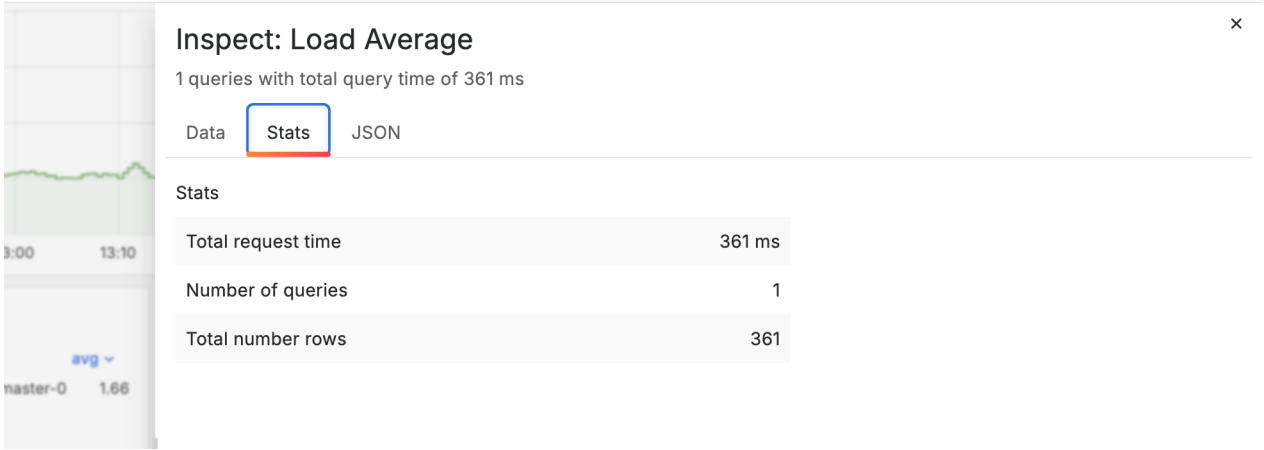
To return to the previous page, press **Esc**. To view the raw data behind the graph, click the three-dot menu in the top-right corner of the chart block and select **Inspect**, then choose the **Data** tab that corresponds to the query.



A panel with detailed record content will open on the right side of the page.

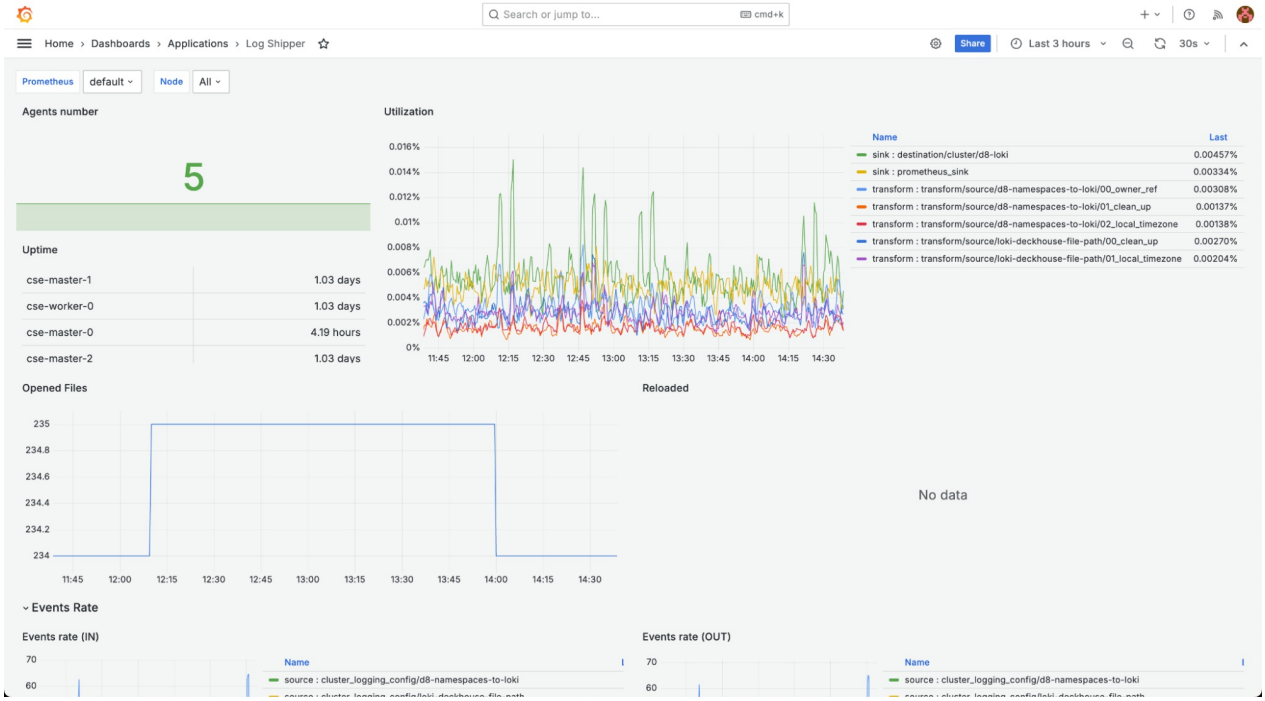


The panel displays all the data used to build the graph. You can also download it as a CSV file and view basic statistics (such as the total number of records) by switching to the **Stats** tab in the data panel.

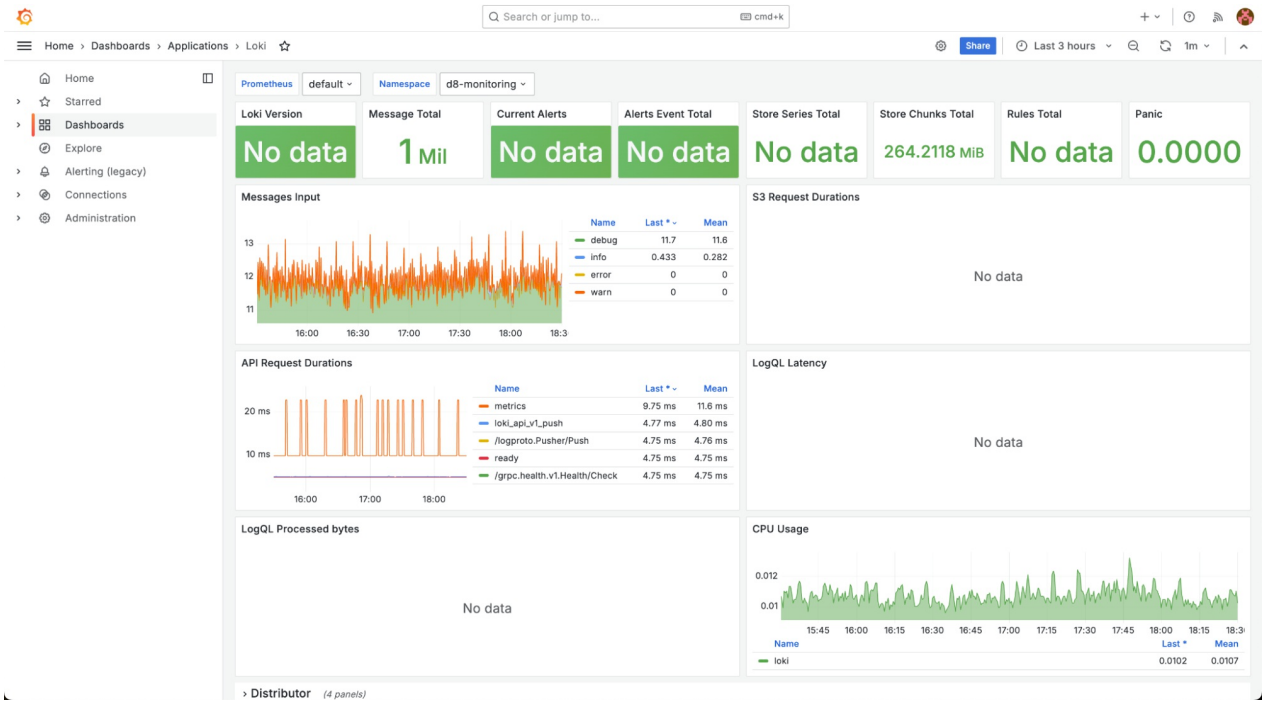


Dashboard descriptions

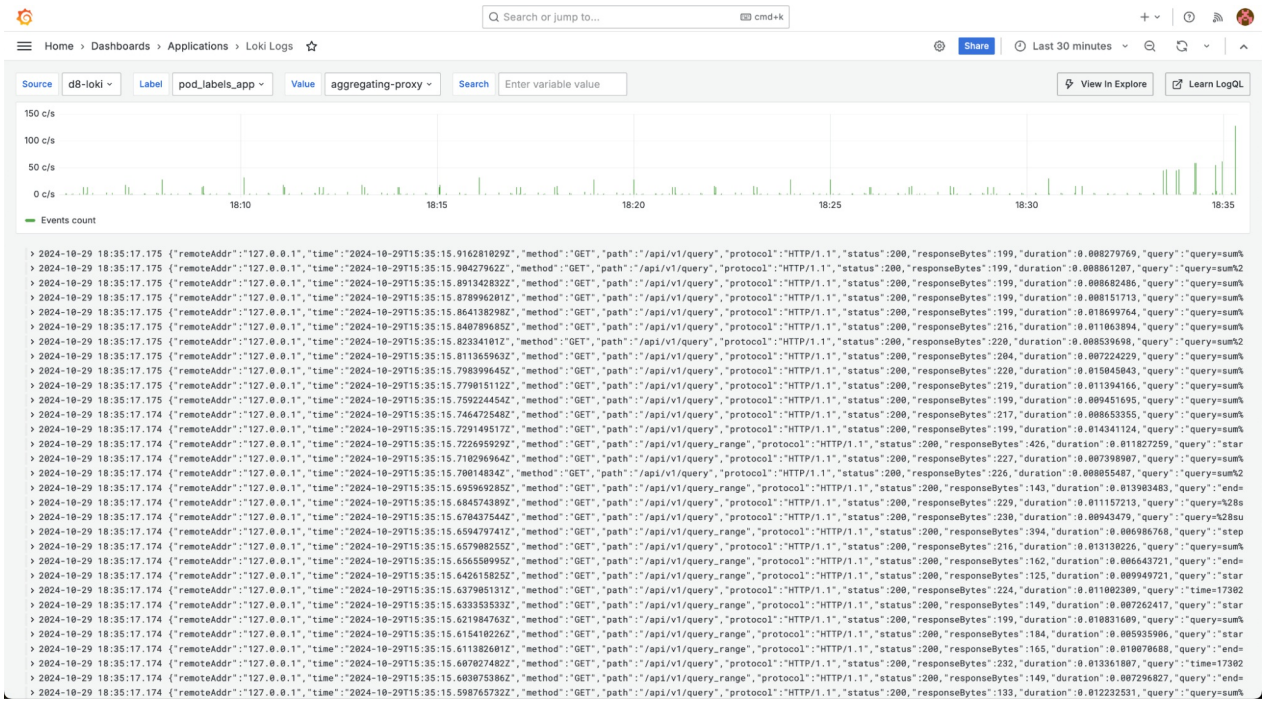
“Applications – Log Shipper” dashboard shows the status of the log-shipper module, including the number of agents running on nodes and their load.



“Applications – Loki” dashboard shows the status of the loki module.

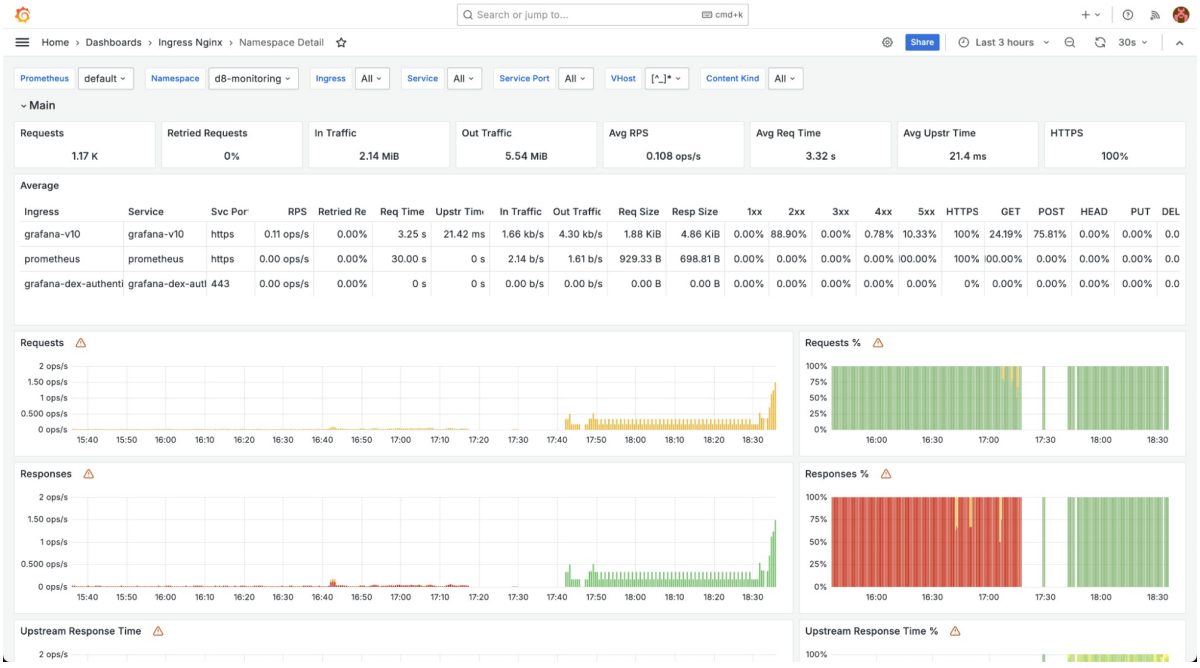


“Applications – Loki Logs” dashboard displays logs from the loki module.

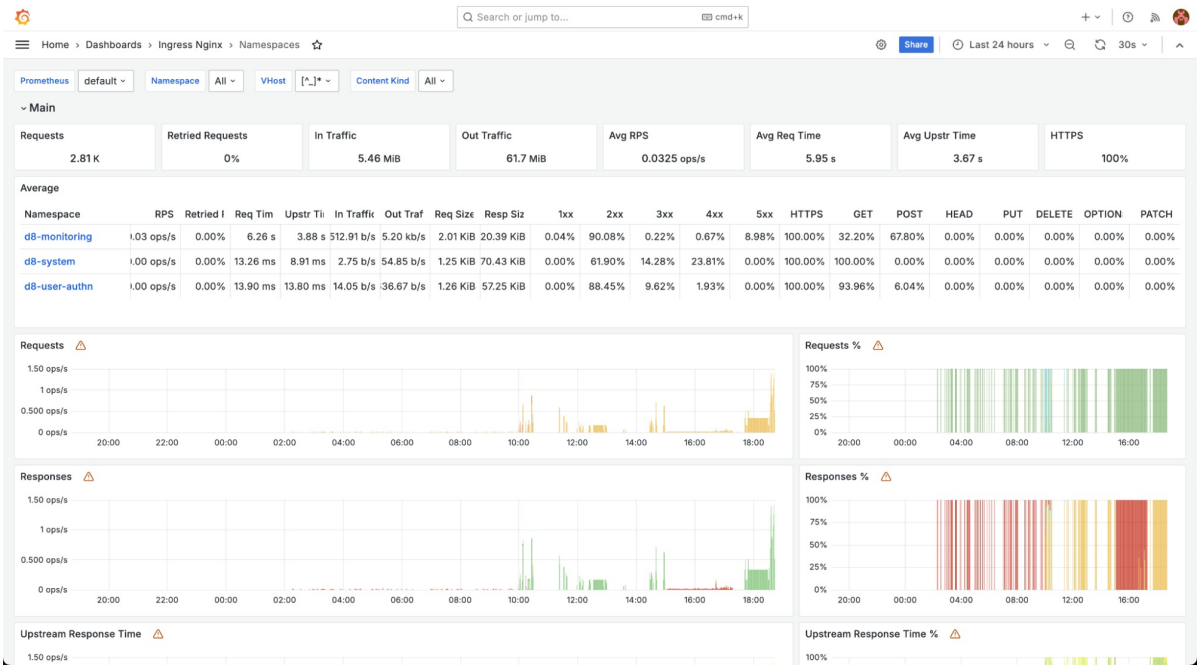


The Ingress Nginx group contains dashboards related to Ingress controllers:

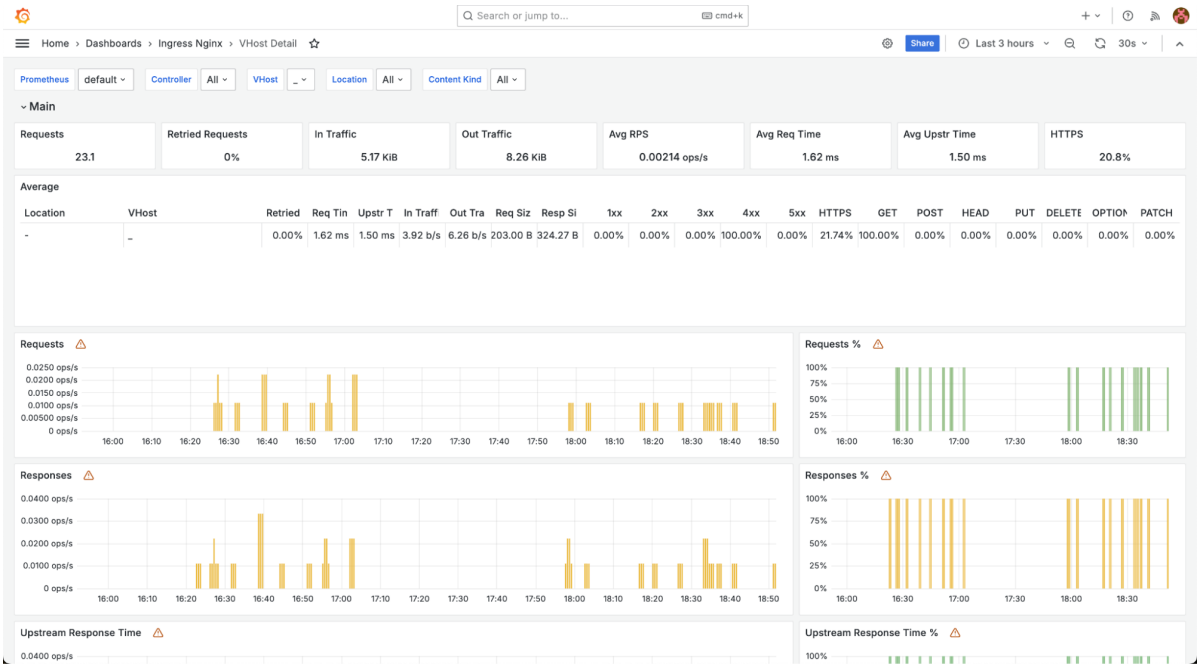
- “Namespace Detail” displays a breakdown of components within a specific namespace. You can filter by namespace, Ingress, Service, and other parameters.



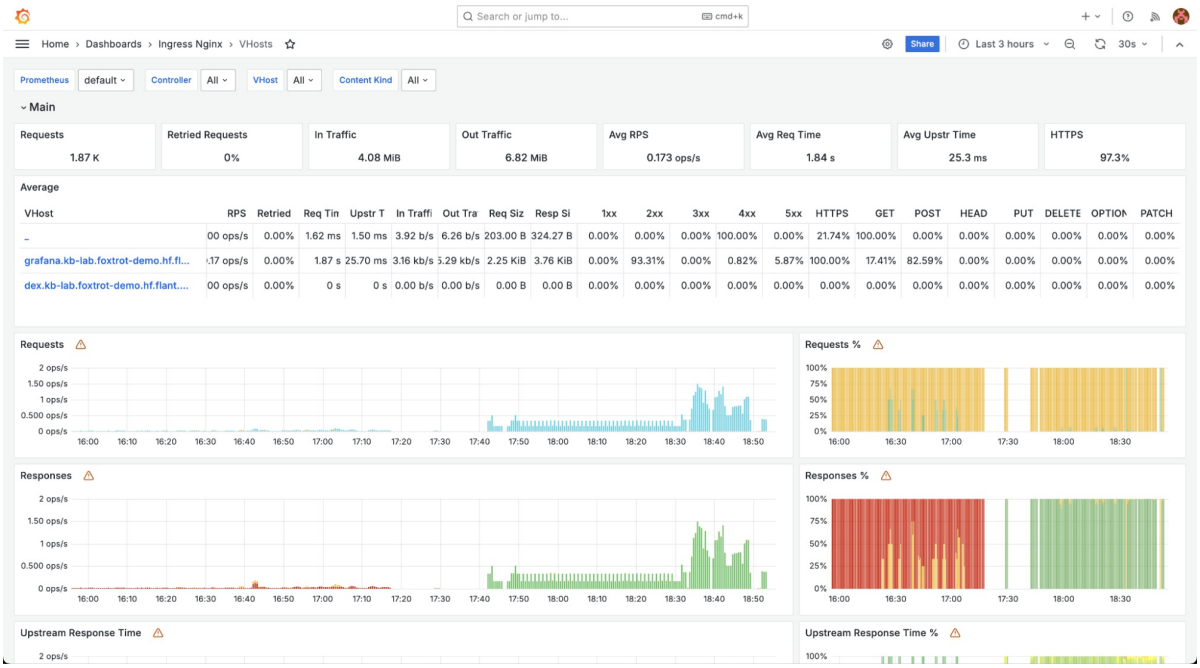
- “Namespaces” shows Ingress controller data grouped by cluster namespaces. Filters allow you to select a specific namespace, virtual hosts, and content types.



- “VHost Detail” shows detailed information on the Ingress controller by virtual hosts.

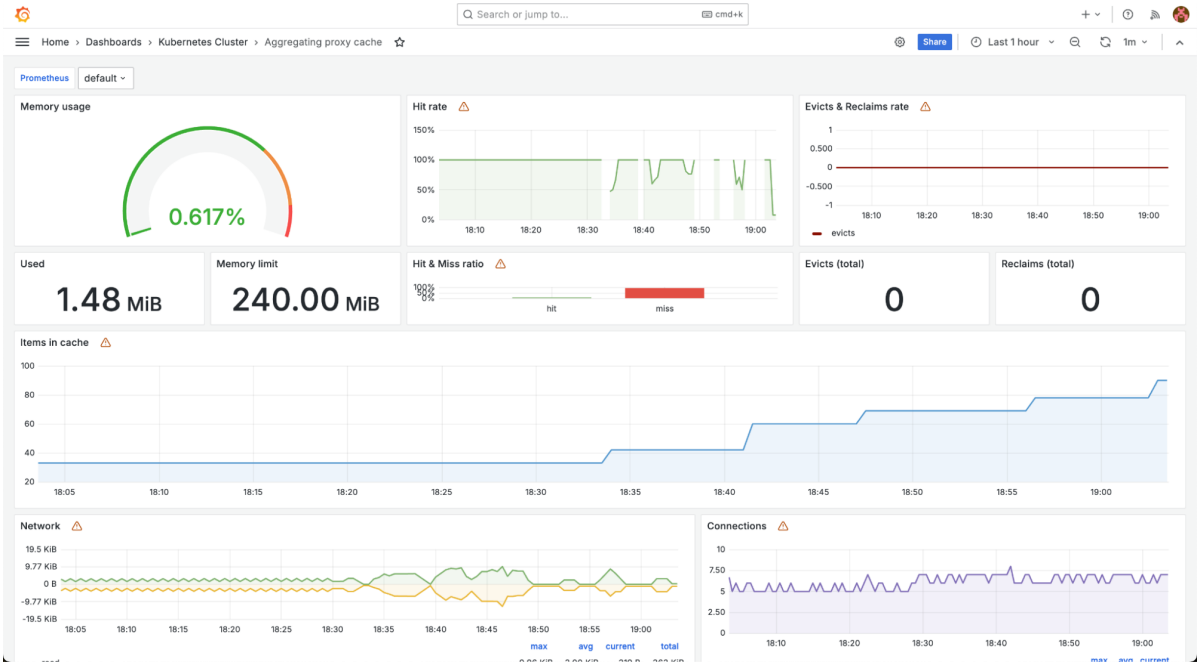


- “VHost” displays summary data on the Ingress controller by virtual hosts. You can filter by a specific virtual host.

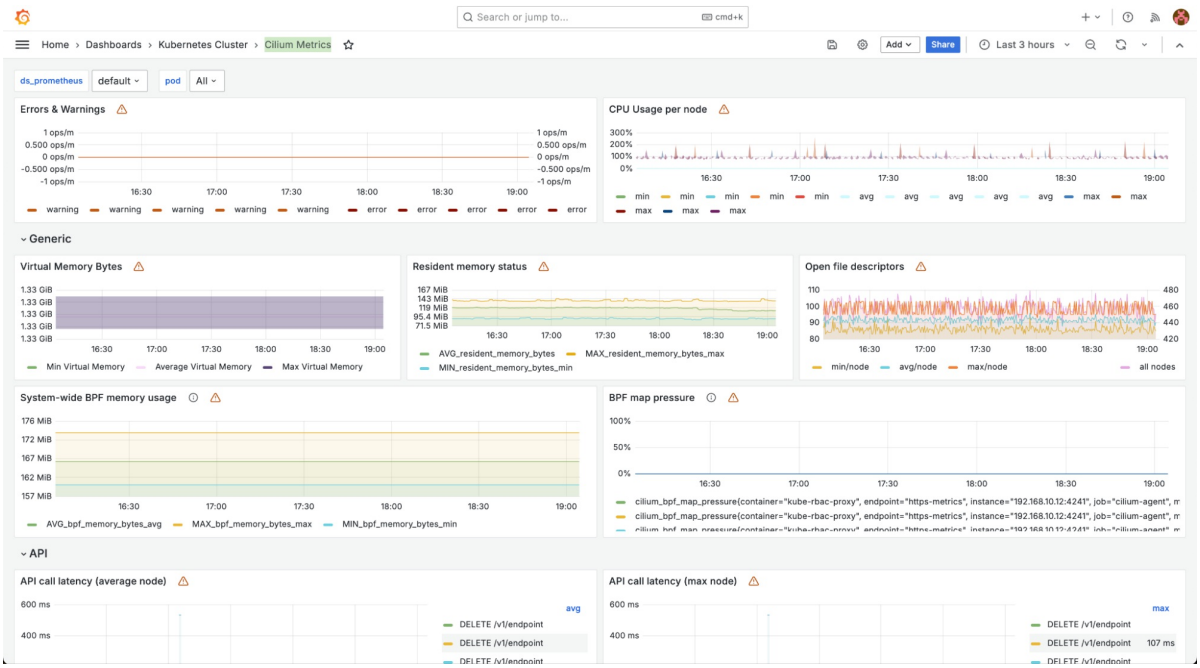


The Kubernetes Cluster group contains dashboards related to the Kubernetes cluster:

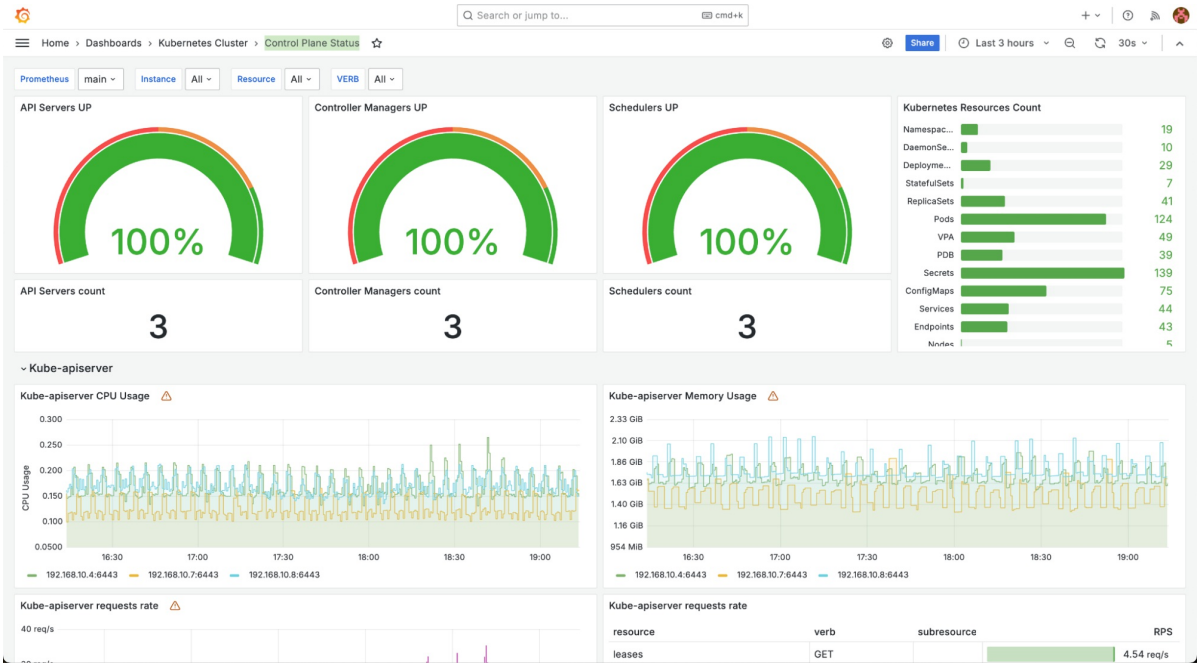
- “Aggregating Proxy Cache” displays a summary of the resources consumed by the proxy server.



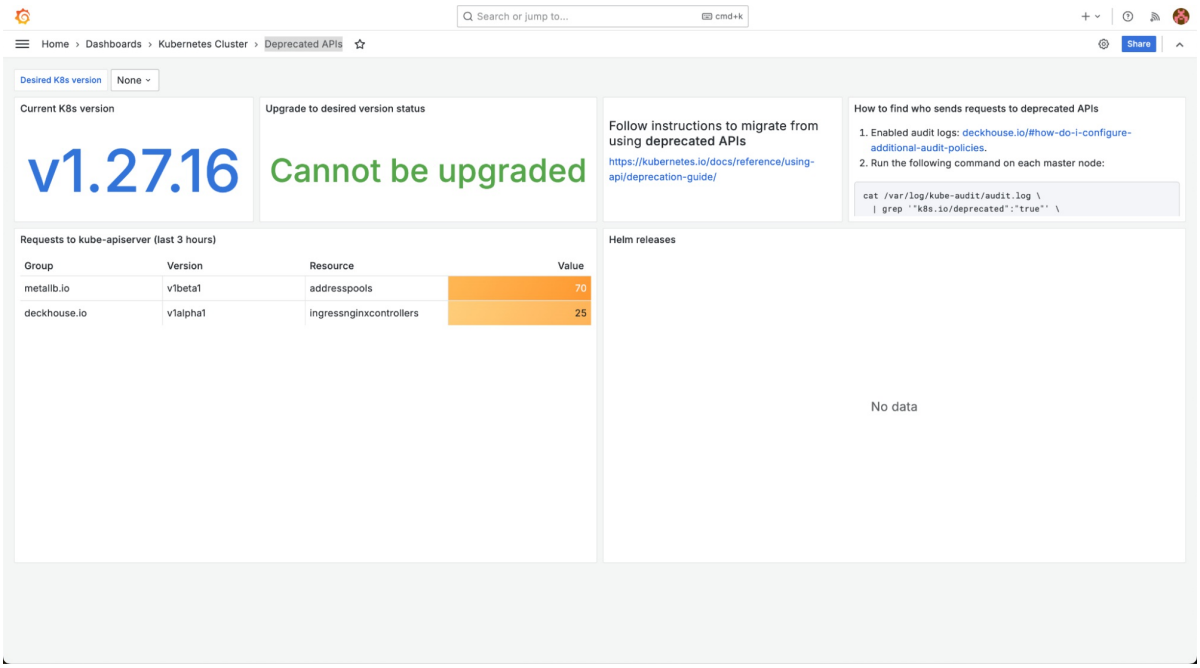
- “Cilium Metrics” displays metrics from the cni-cilium module.



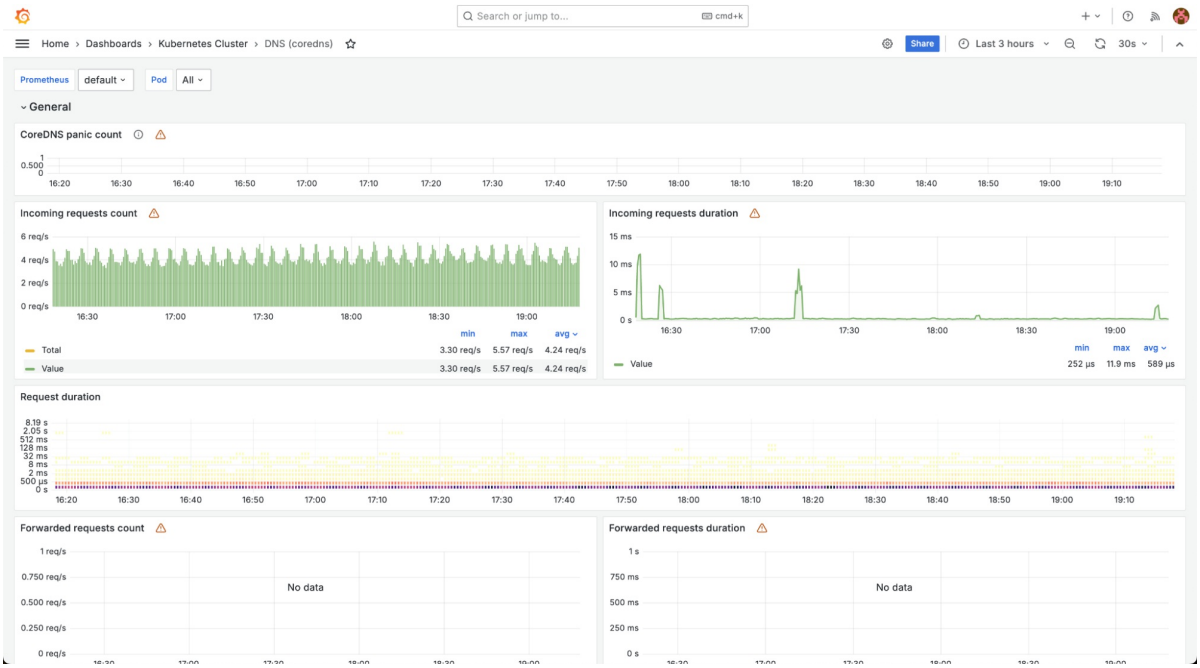
- “Control Plane Status” shows the status of the cluster control plane.



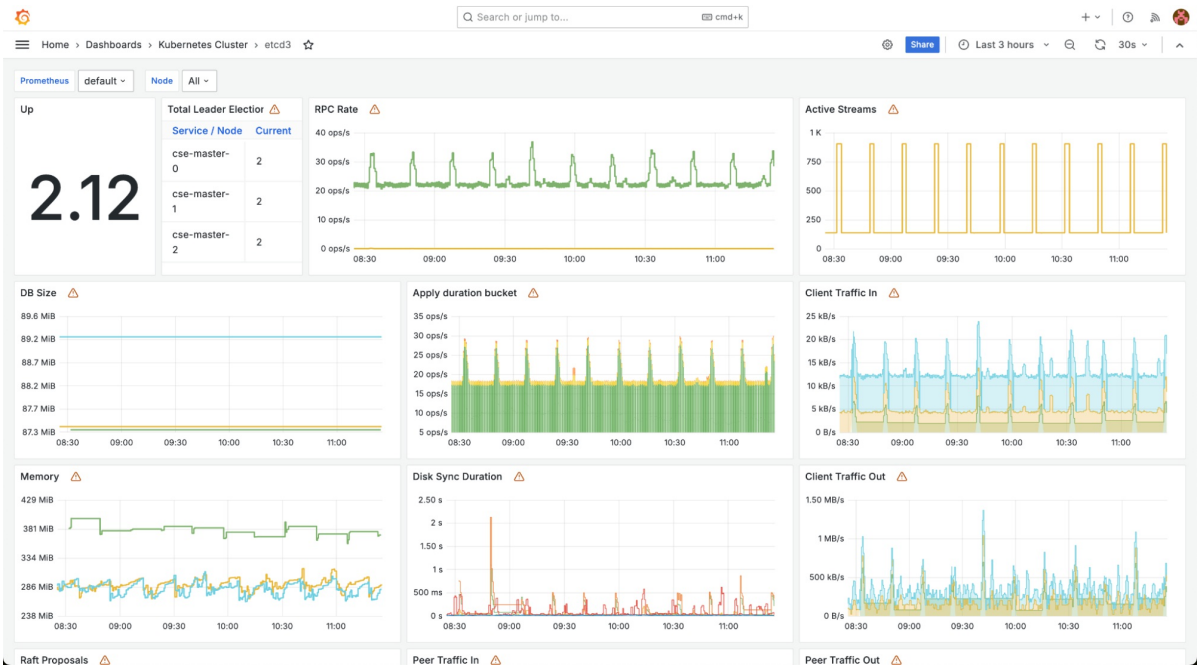
- “Deprecated APIs” displays the state of Kubernetes APIs that are currently deprecated. The dashboard also includes migration instructions and requests made to these API endpoints.



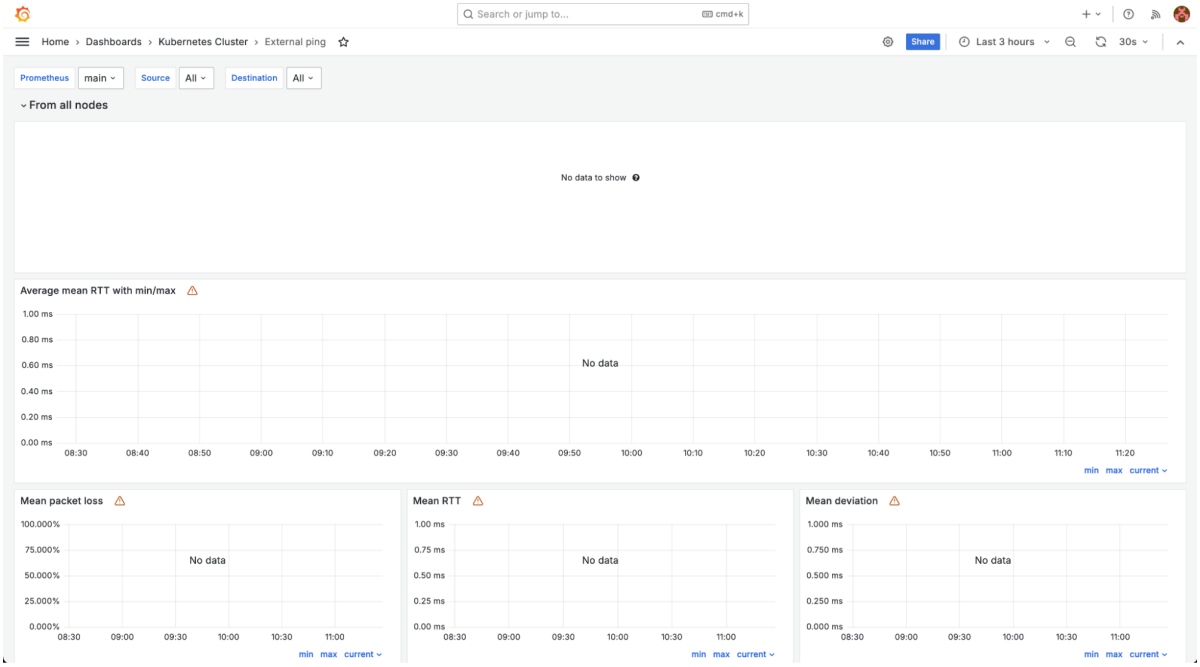
- “DNS (coredns)” dashboard displays data on the operation of the CoreDNS component.



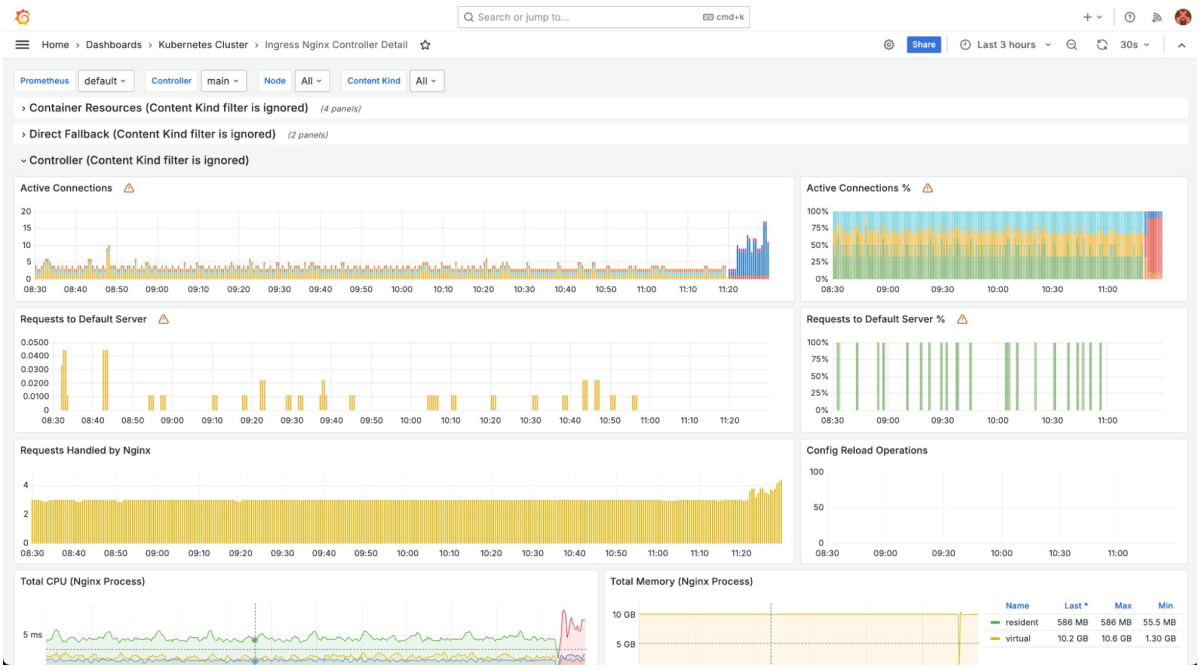
- “etcd3” dashboard shows the status of the etcd database.



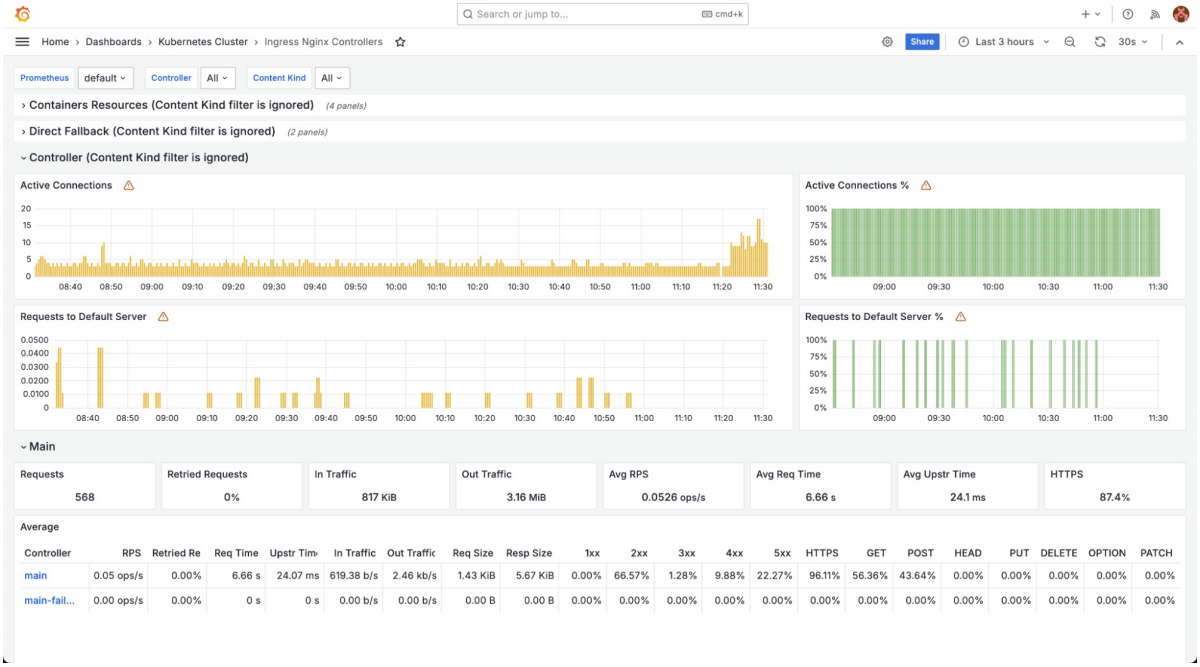
- “External ping” dashboard provides statistics on external requests.



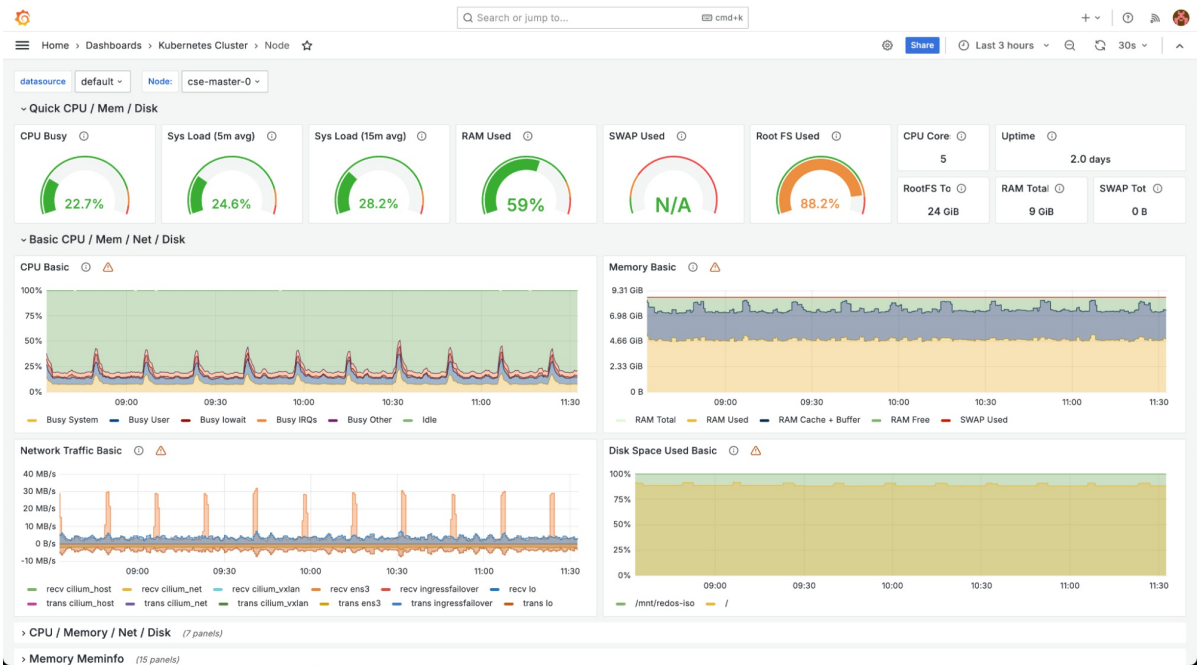
- “Ingress Nginx Controller Detail” dashboard displays parameters of the Ingress controller.



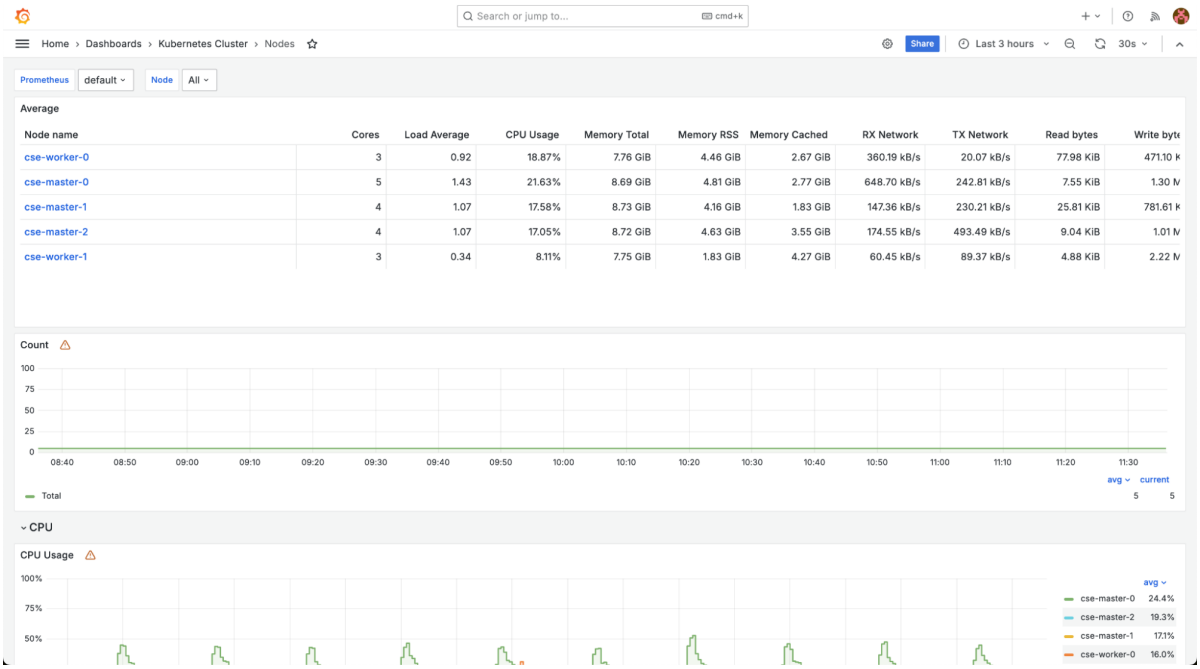
- “Ingress Nginx Controllers” dashboard shows detailed data on the cluster’s Ingress controllers.



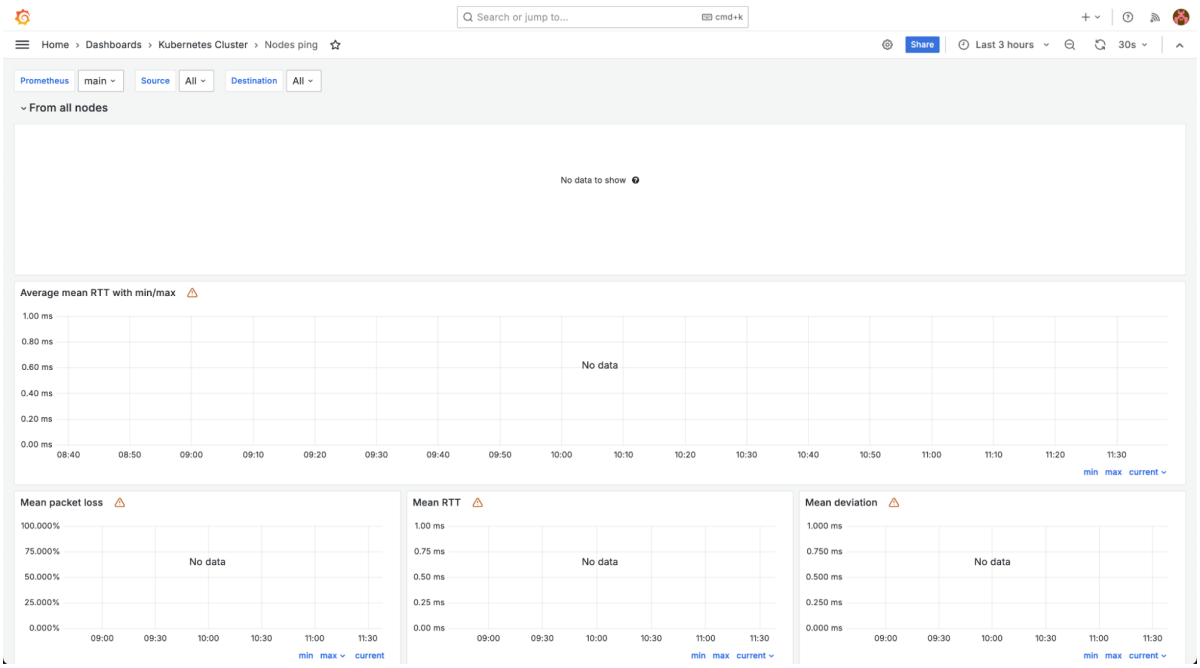
- “Node” dashboard displays information about individual node performance. You can select a specific node in the filters.



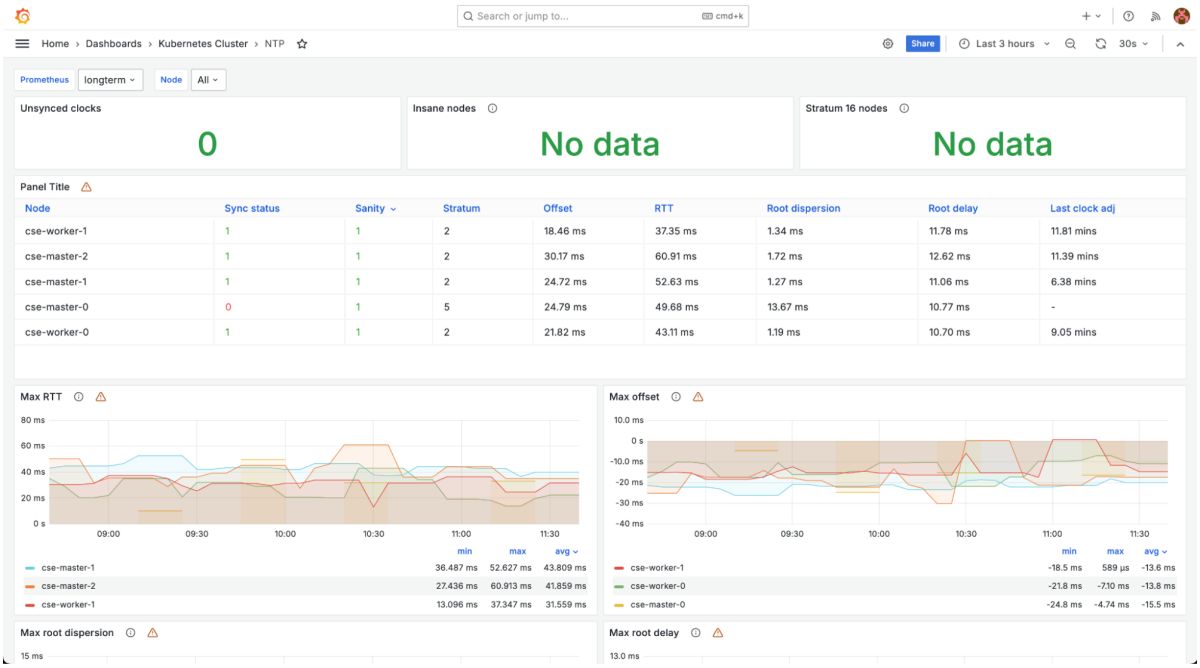
- “Nodes” dashboard shows aggregated data on all cluster nodes. The filters let you select a specific node.



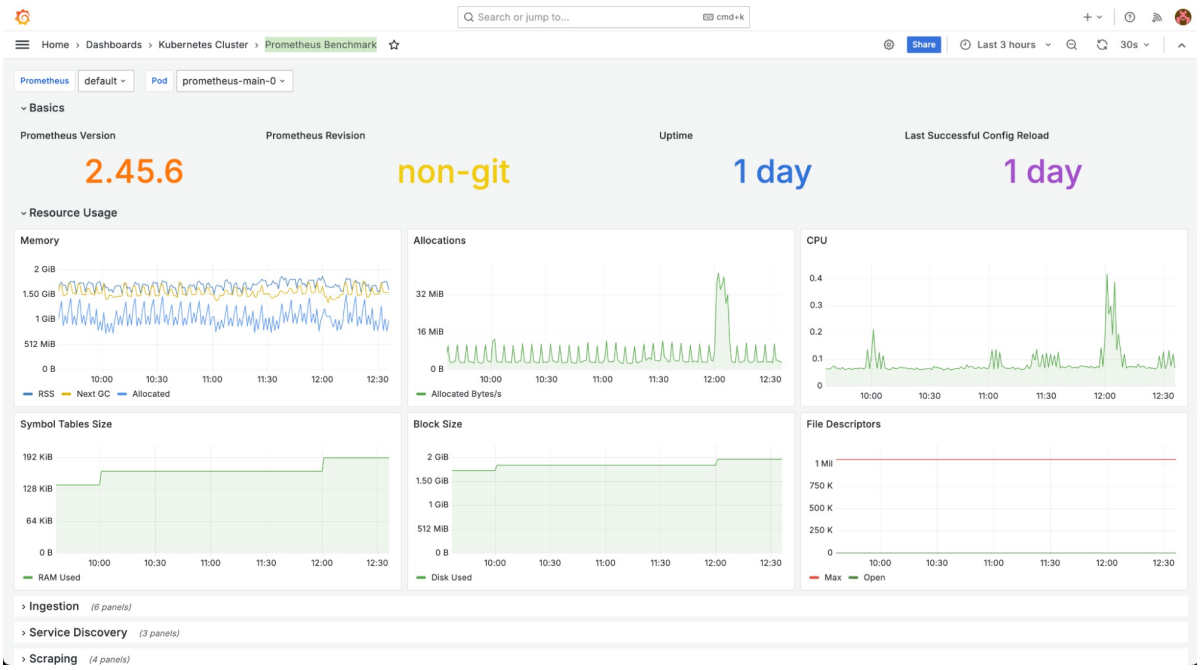
- “Nodes ping” dashboard displays ping results for cluster nodes.



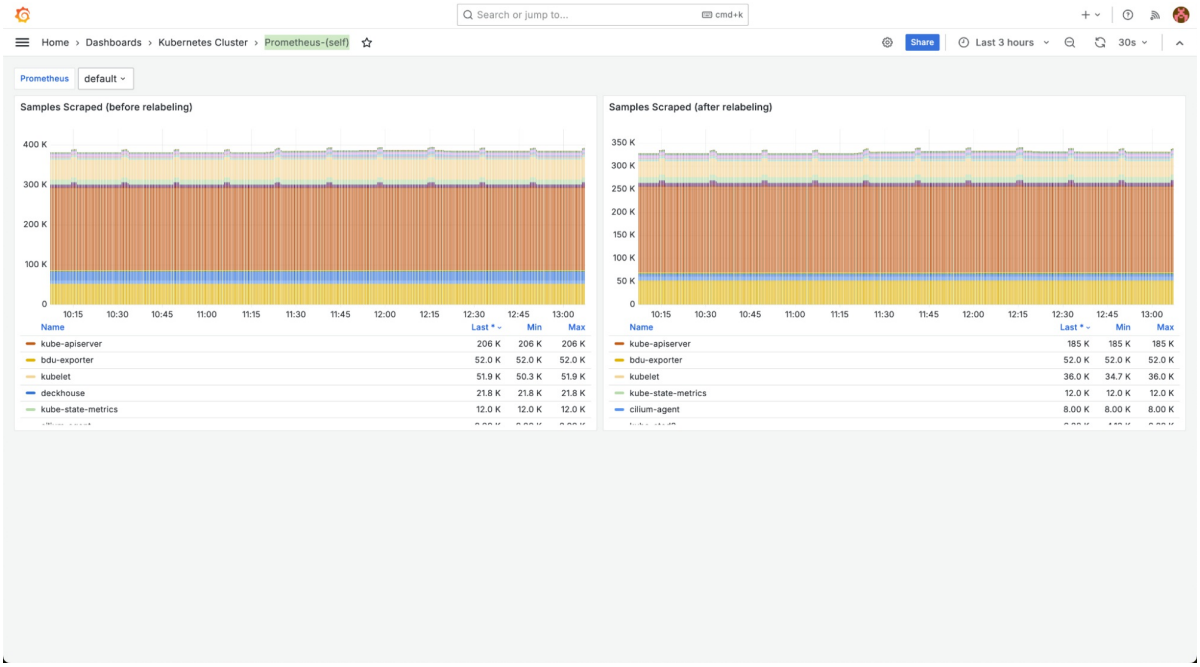
- “NTP” dashboard shows a status of the time synchronization server.



- “Prometheus Benchmark” dashboard shows a status of the prometheus module.

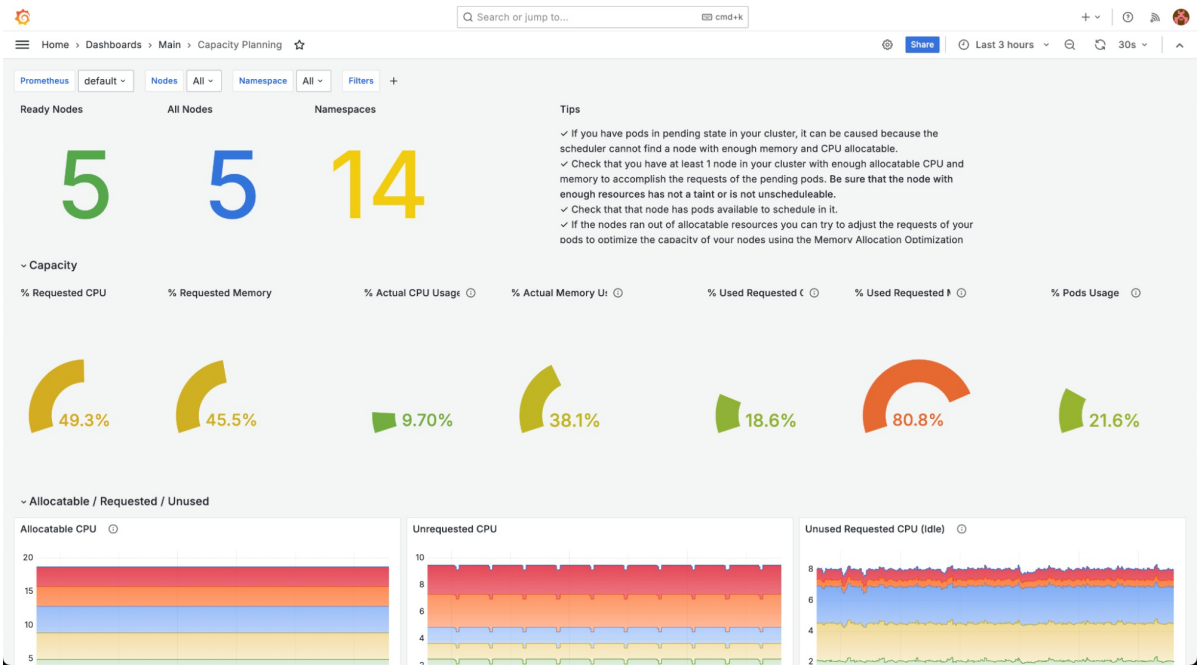


- “Prometheus-(self)” dashboard — summary information about the state of the prometheus module.

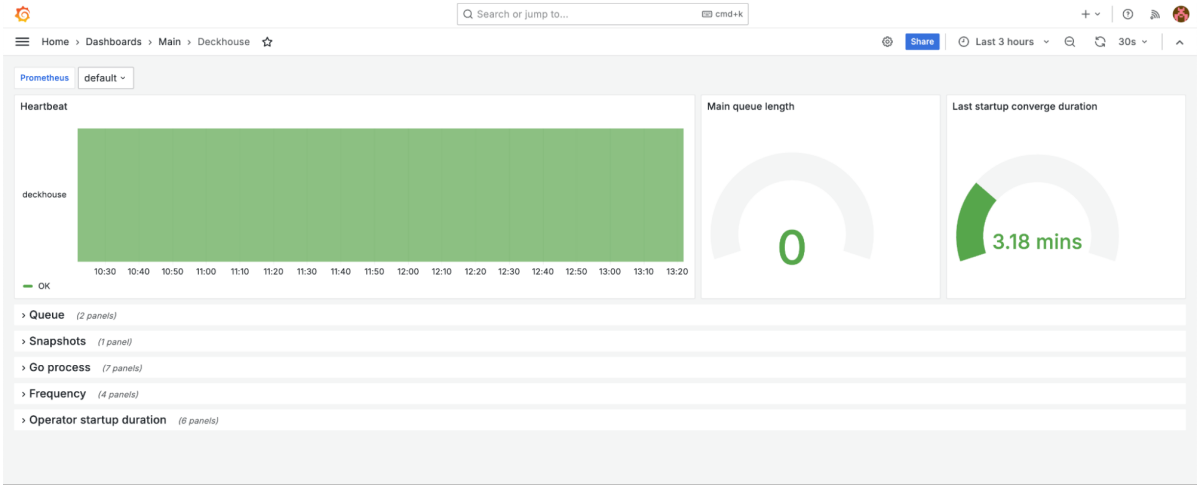


“Main” dashboard group provides general information about the state of the cluster:

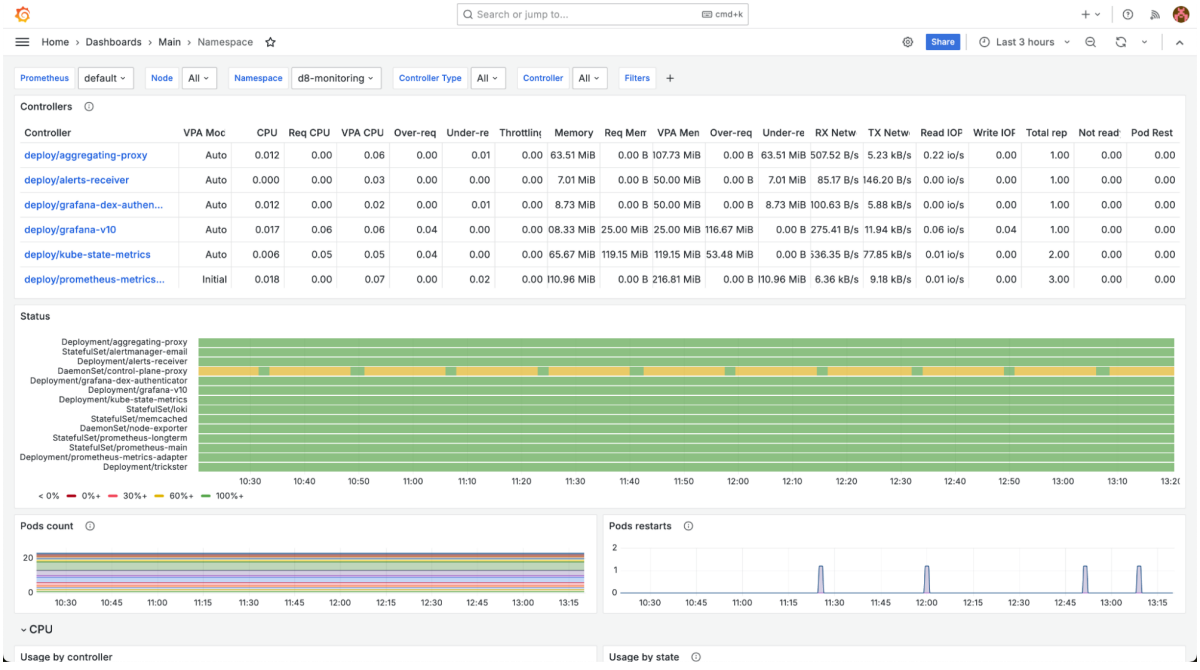
- “Capacity Planning” shows overview of the cluster’s performance and capacity usage.



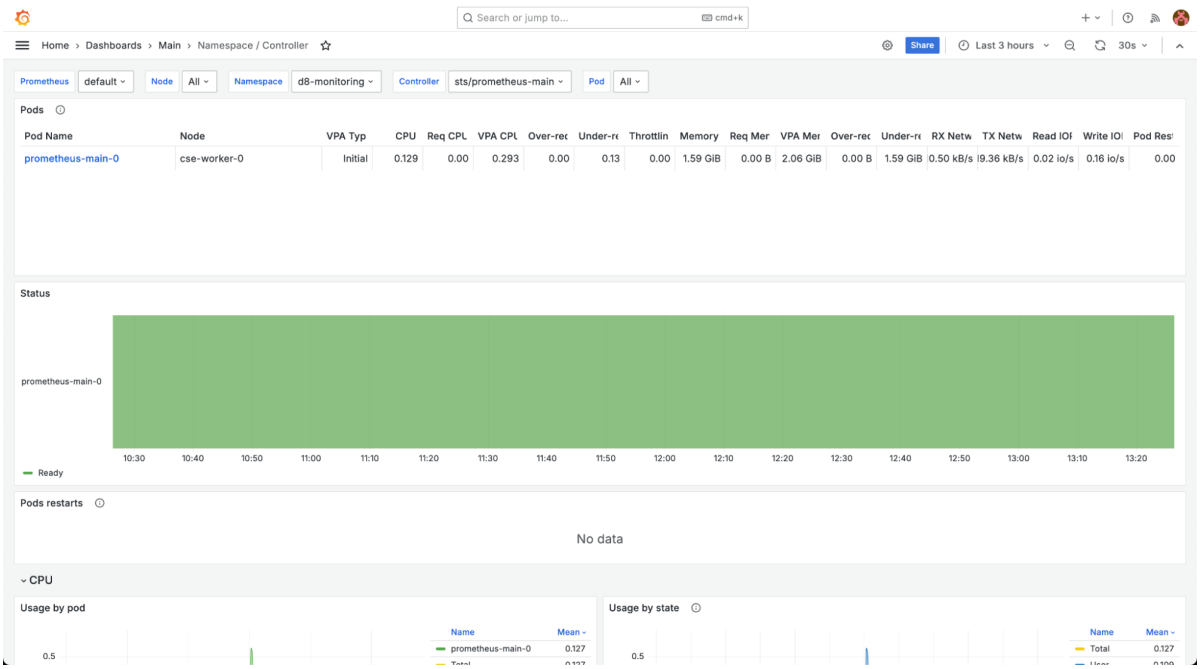
- “Deckhouse” shows summary information about the main deckhouse component.



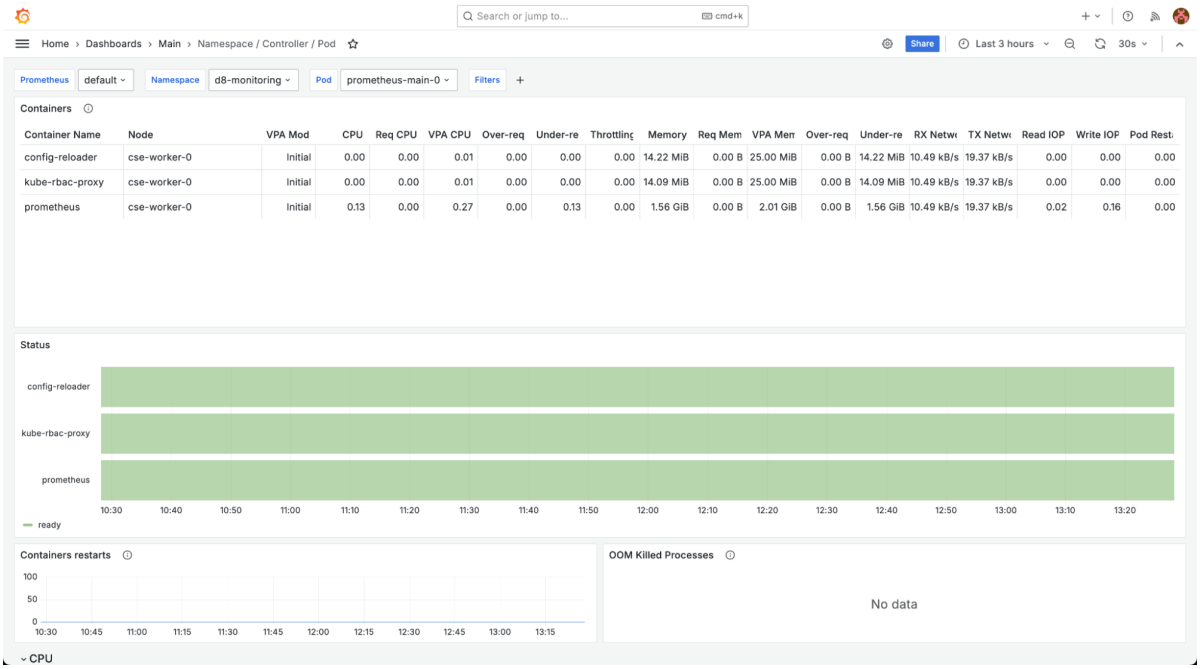
- “Namespace” displays data on a specific cluster namespace.



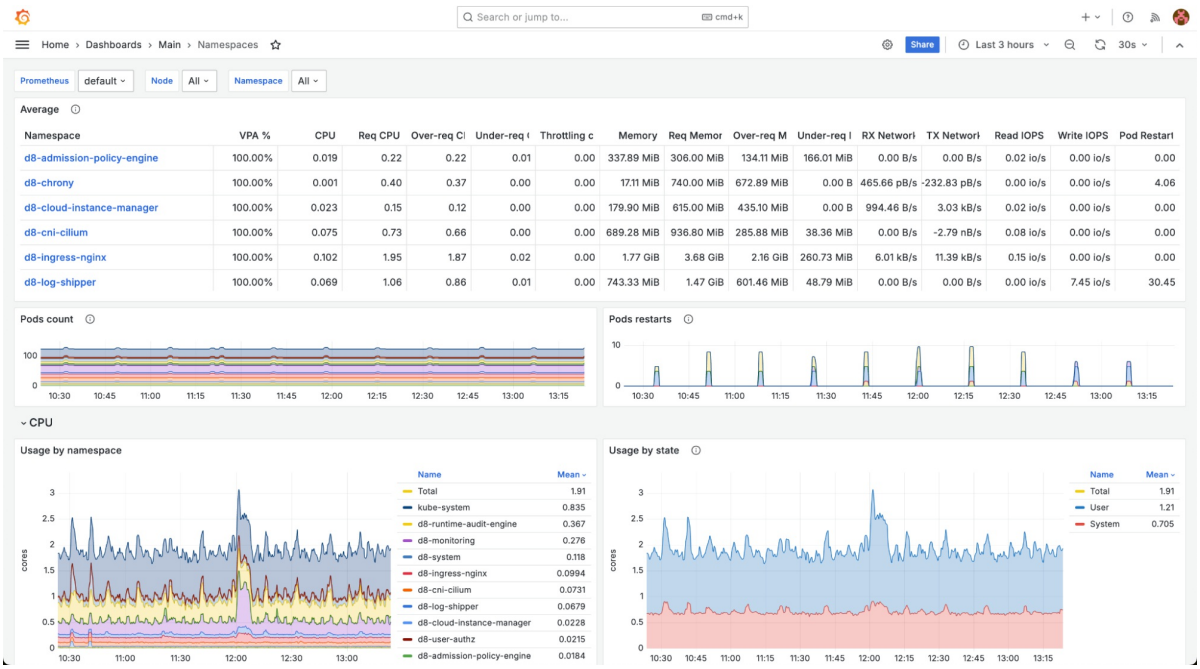
- “Namespace / Controller” displays data on controllers in namespaces. Filters allow you to select specific namespaces and controllers.



- “Namespace / Controller / Pod” show data on Pods in namespaces. Filters allow selection of specific namespaces and Pods.



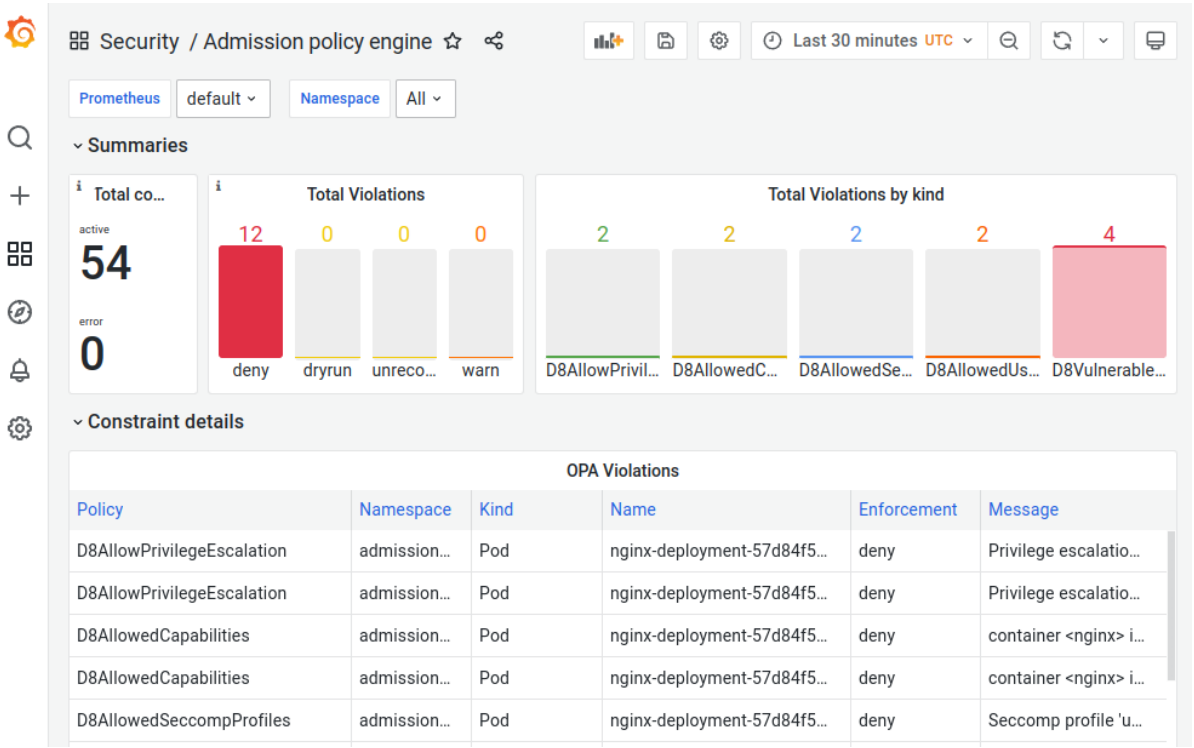
- “Namespaces” shows summary data grouped by cluster namespaces.



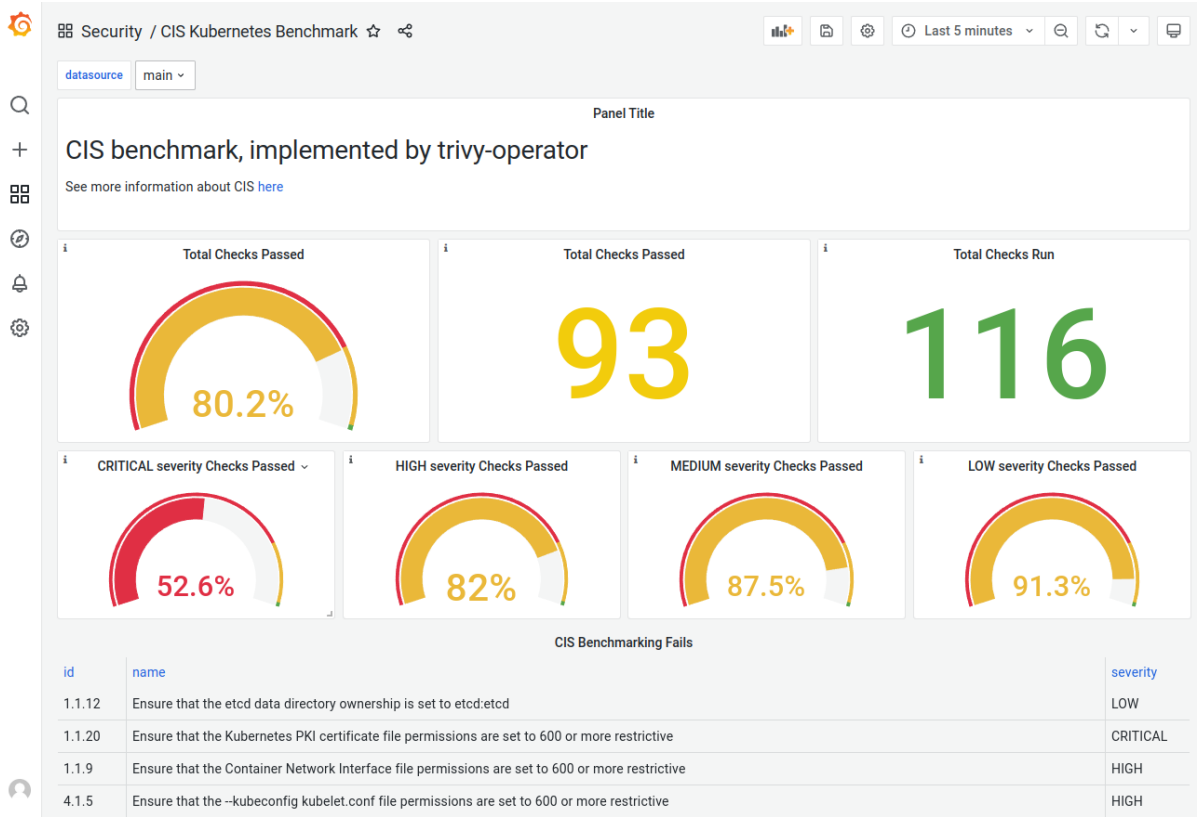
Viewing security event logs

Security event logs are available through the Grafana web UI. The relevant dashboards are grouped under the “Security” folder:

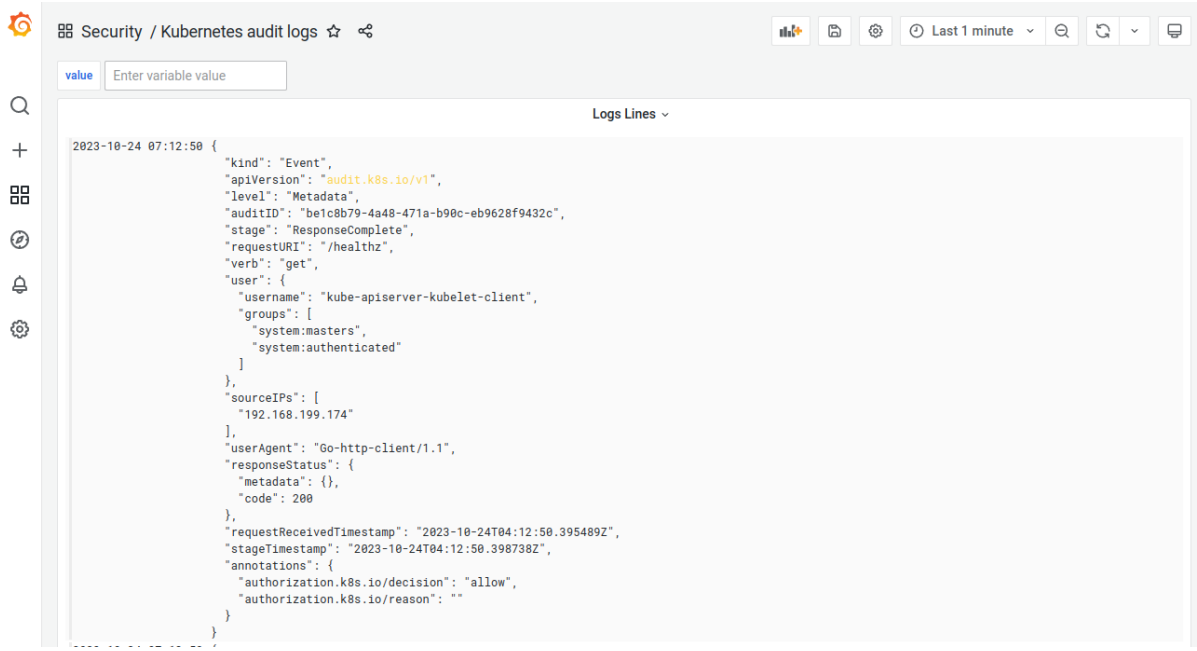
- Admission policy engine. Contains information related to the operation of security policies, including a number of denied actions due to policy violations, breakdown of denials by type, and event logs. The event log for policy-related security violations is available in the OPA Violations panel.



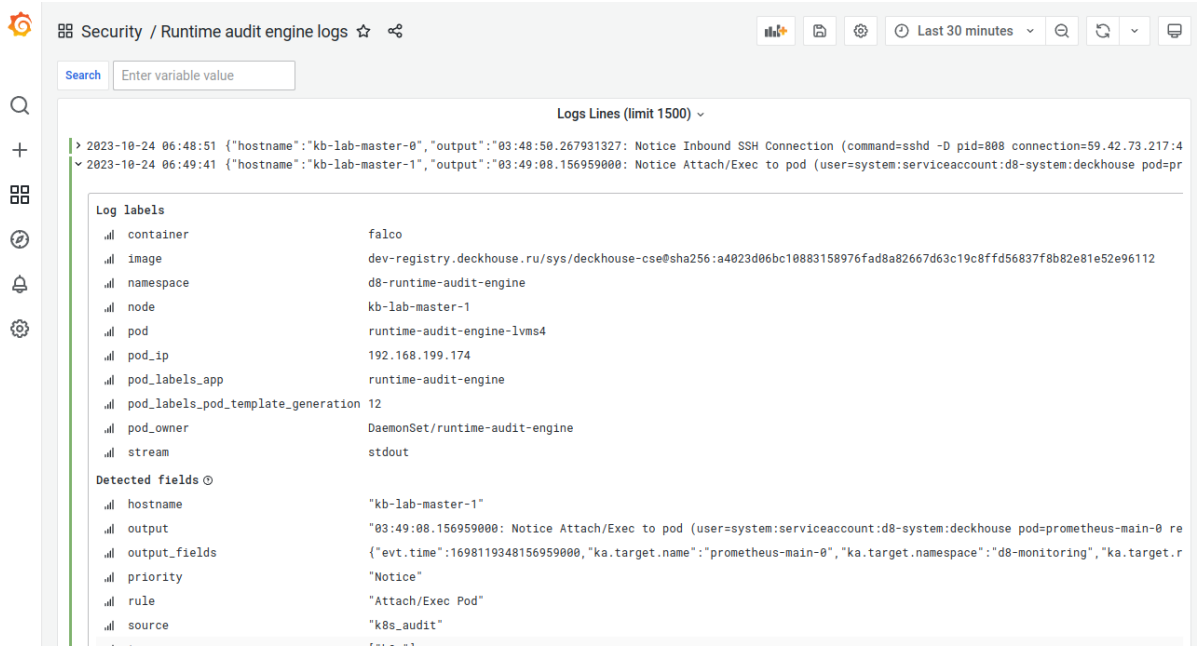
- CIS Kubernetes Benchmark. A dashboard with the results of a configuration audit for compliance with established best practices. It provides summary information with no detailed view. This dashboard is available when the operator-trivy module is enabled.



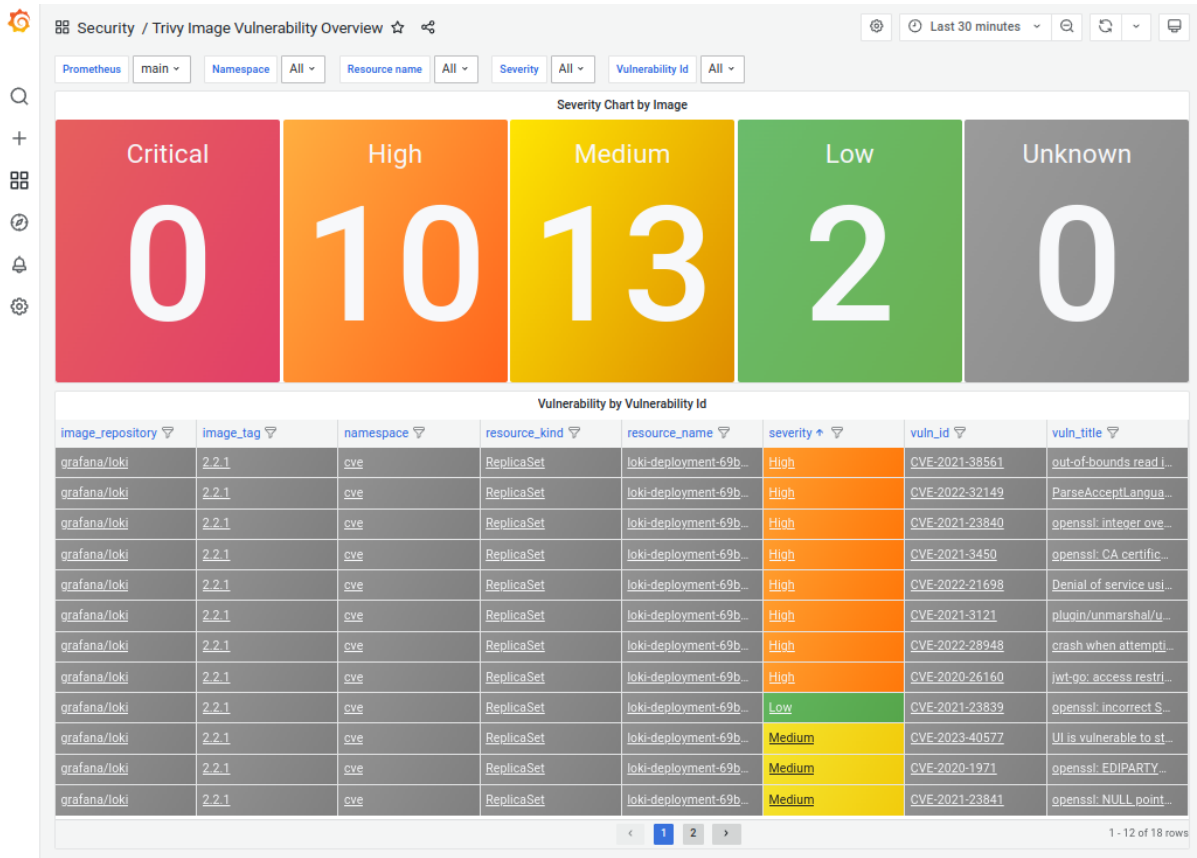
- Kubernetes audit logs. Includes logs of all interactions with the Kubernetes API server. Records are presented in JSON format.



- Runtime audit engine logs. Includes logs of Linux kernel and cluster API server security audit events.



- Trivy Image Vulnerability Overview. A dashboard with summary and detailed information on vulnerability scans of container images in namespaces annotated with `security-scanning.deckhouse.io/enabled`.



Additional resources

- Module `cni-cilium` documentation
- Module `deckhouse` documentation
- Module `log-shipper` documentation
- Module `loki` documentation
- Module `operator-trivy` documentation
- Module `prometheus` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

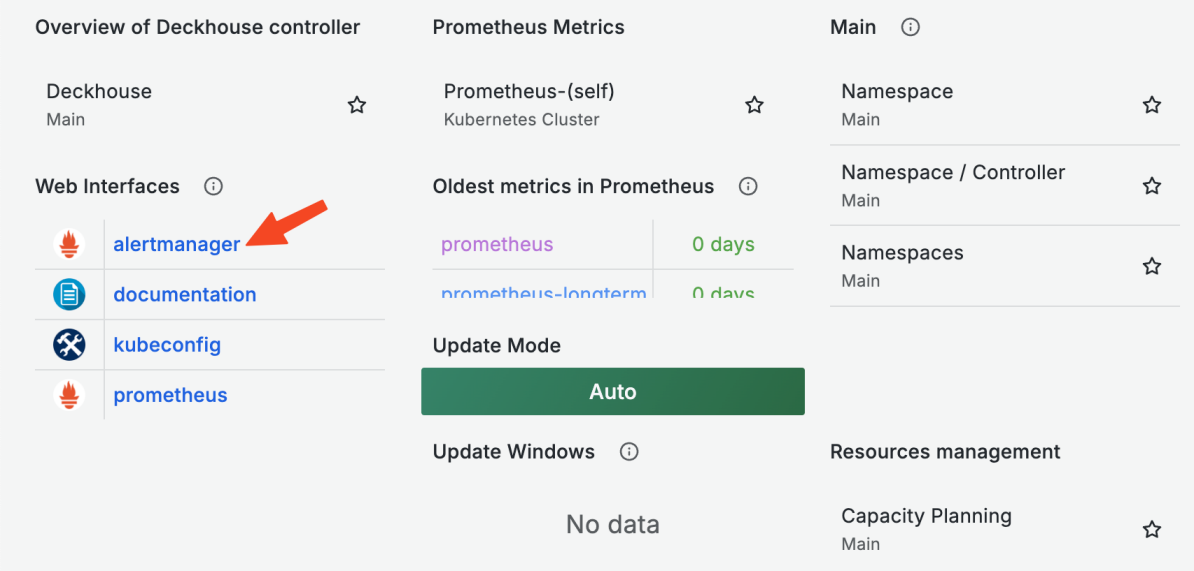
Web interface for managing alerts

The alert management web UI can be used for handling alerts in a Deckhouse cluster. It allows you to view detailed information about alerts, including the source, severity, and trigger time, and to temporarily silence alert notifications when needed.

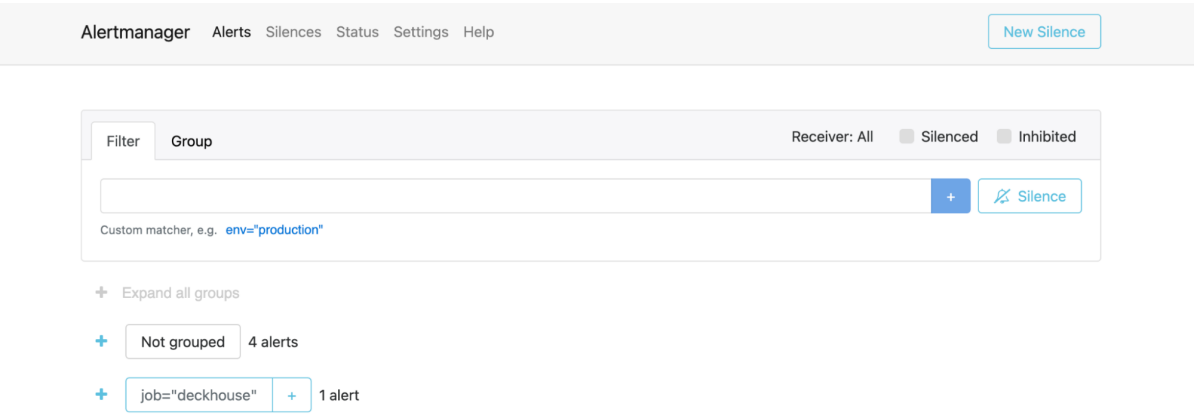
For the list of all available alerts in the Deckhouse Platform monitoring system, refer to the corresponding documentation page.

Accessing the web UI

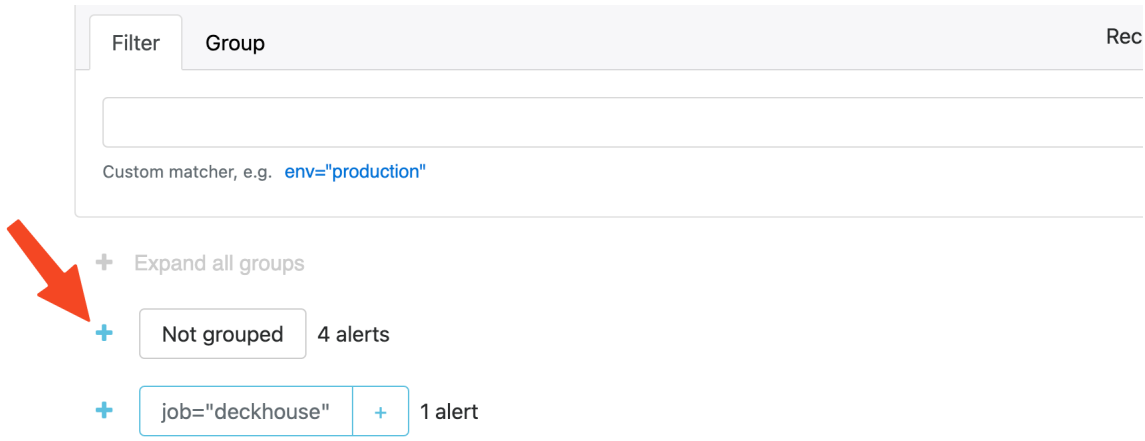
1. To open the alert management web UI, click the corresponding link in the side menu on the Grafana overview page.



2. If you are accessing the web UI for the first time, enter your user credentials. Once the authentication is complete, you will see the main web UI page displaying a summary of all alerts in the cluster.



3. Alerts are grouped by category. To expand a group, click the `+` icon to the left of the group name.



4. Once expanded, the group will display the full list of associated alerts.



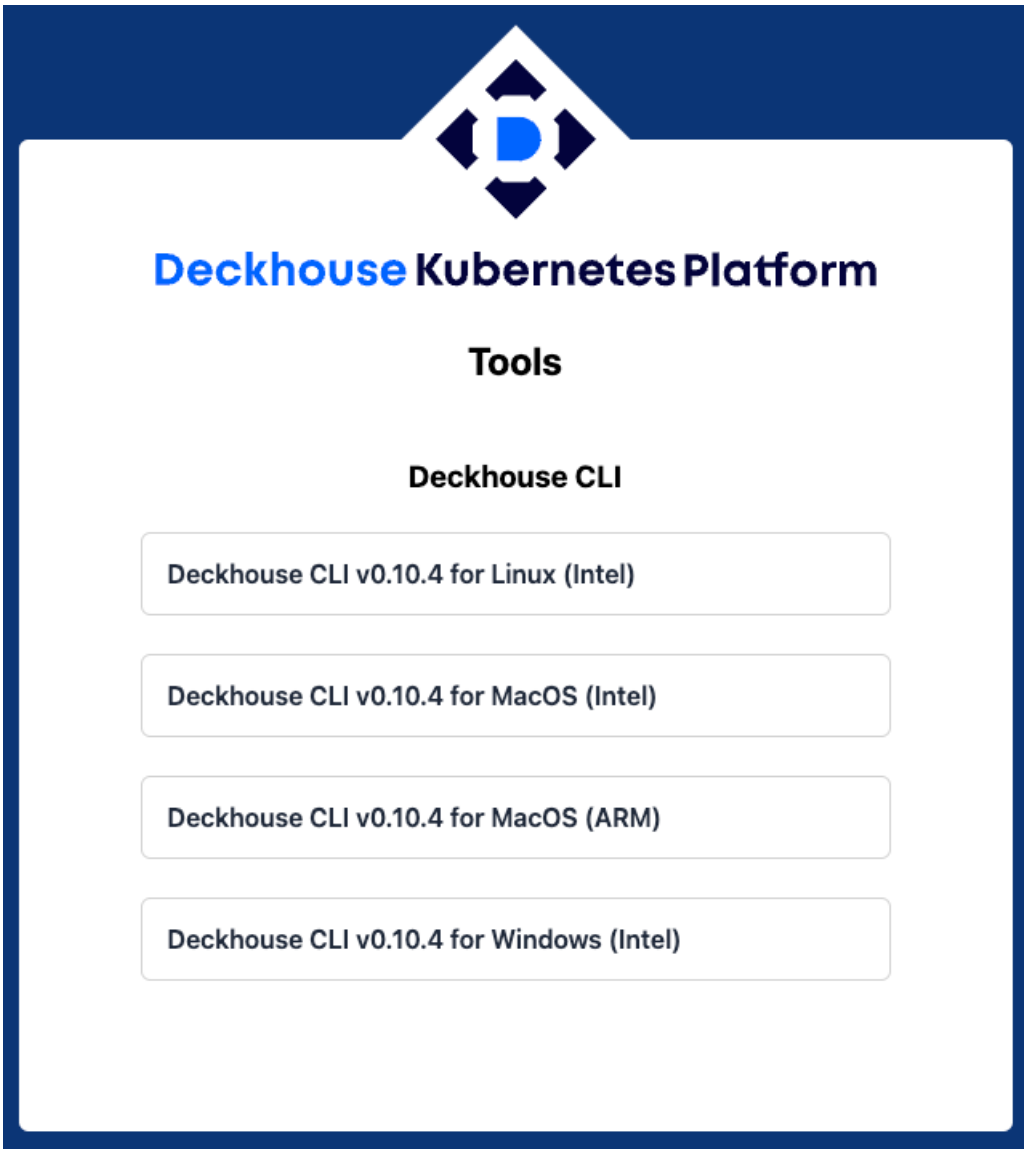
> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Web interface for downloading CLI utilities

Web interface for downloading Deckhouse utilities provides centralized access to CLI tools for popular operating systems (Linux, macOS, Windows). Using it, you can quickly locate and download executable binaries for the required architecture and version, as well as receive notifications about their updates. This functionality is implemented by the deckhouse-tools module.

Accessing the web interface

1. To open the web interface, enter `tools.<CLUSTER_NAME_TEMPLATE>` in your browser's address bar, where `<CLUSTER_NAME_TEMPLATE>` is the string matching the cluster's DNS name template, as specified by the global parameter `modules.publicDomainTemplate`. The exact URL format may vary depending on system configuration. Check with your administrator for the correct URL.
2. On first login, enter your user credentials. After successful authentication, the Deckhouse Tools download page will open, presenting links to download the Deckhouse CLI tool for various operating systems: Linux, macOS, and Windows.



3. Select the required version, download, and install the corresponding executable file.

Additional resources

- Module `deckhouse-tools` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Secret management web interface

The web interface for managing secrets (Stronghold web interface) is designed for managing secret mechanisms, authentication, and access control in a cluster. Operation is provided by the stronghold module.

It allows you to:

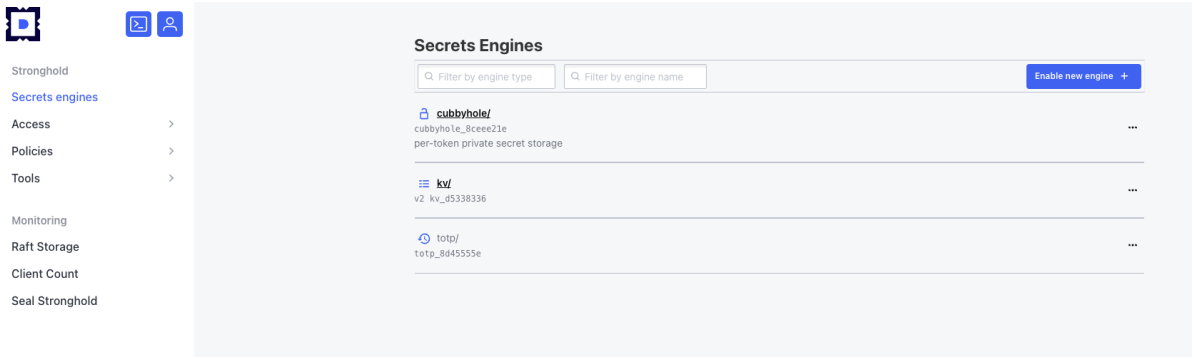
- View and configure secrets engines, add and edit secrets.
- Manage authentication methods, user groups, and entities.
- Control permissions, leases, and security policies.

Accessing the stronghold web UI

To open the web UI, enter `stronghold.<CLUSTER_NAME_TEMPLATE>` in your browser’s address bar, where `<CLUSTER_NAME_TEMPLATE>` is the DNS name template of the cluster defined in the global

`modules.publicDomainTemplate` parameter.

1. On your first login, enter the user credentials.
2. After successful authentication, the main stronghold page will open:



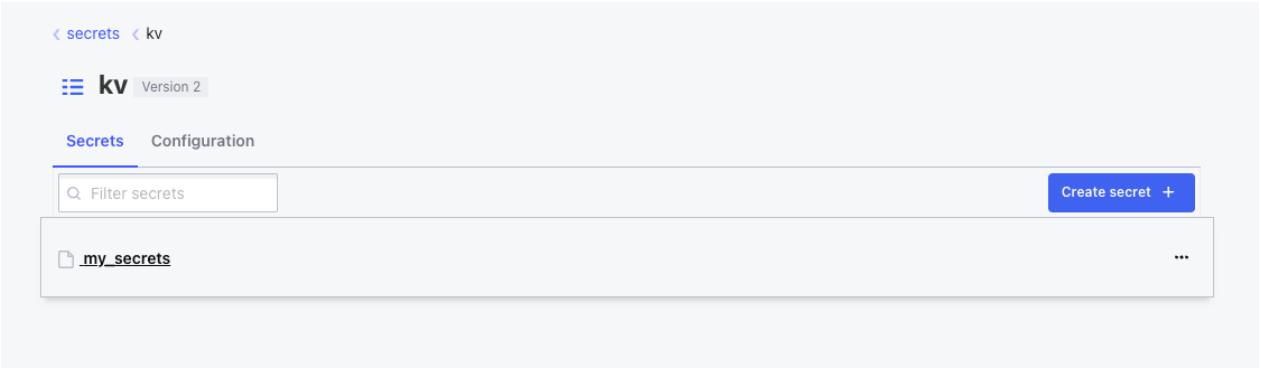
Managing secrets engines

Configuration guides for common engine types are available in the Deckhouse Stronghold documentation: KV secrets engine and PKI secrets engine.

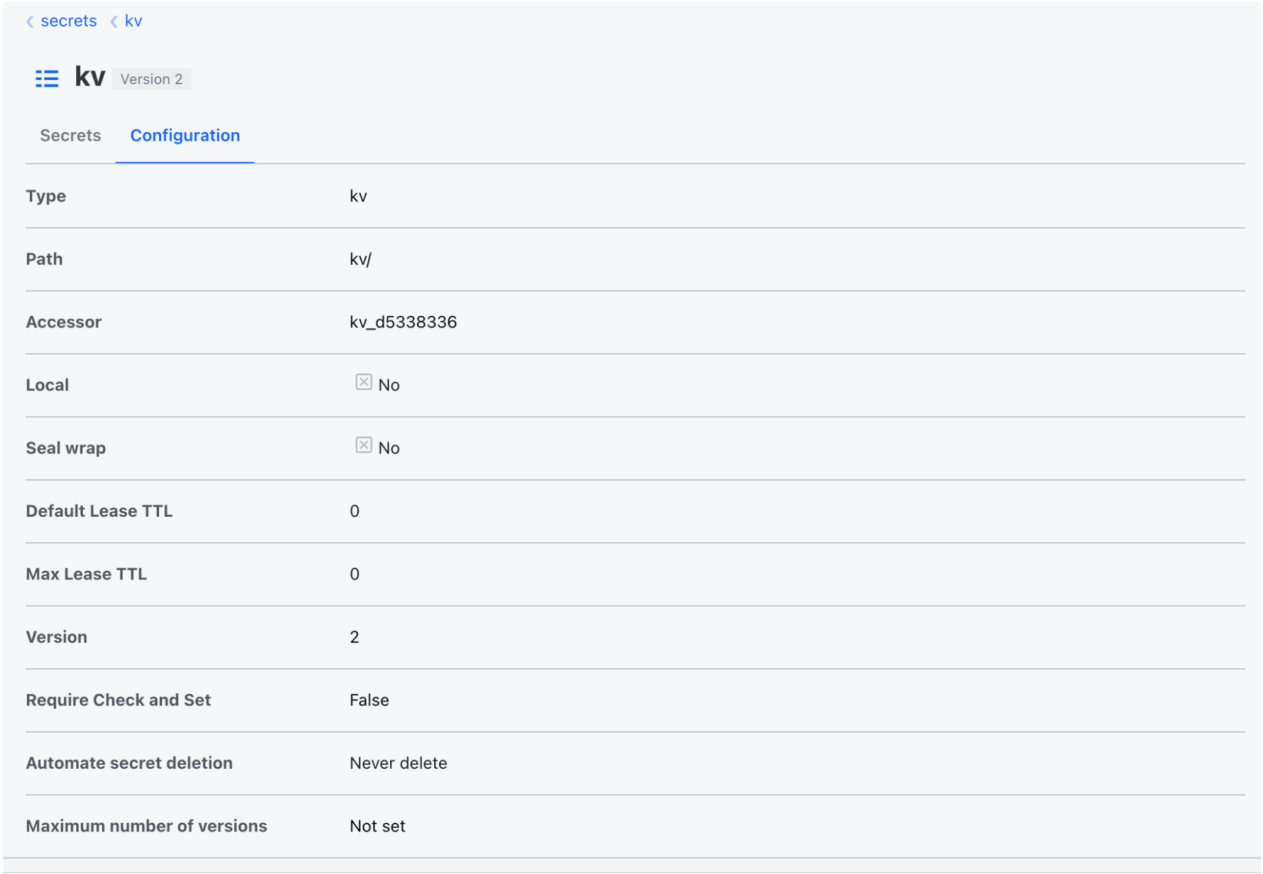
Viewing a secrets engine

To view a secrets engine:

1. Click on its name.
2. The following tabs will be displayed:
   - “Secrets” with a list of secrets (roles, keys, etc.).
   - “Configuration” with engine configuration.
   - A button to add a new secret.

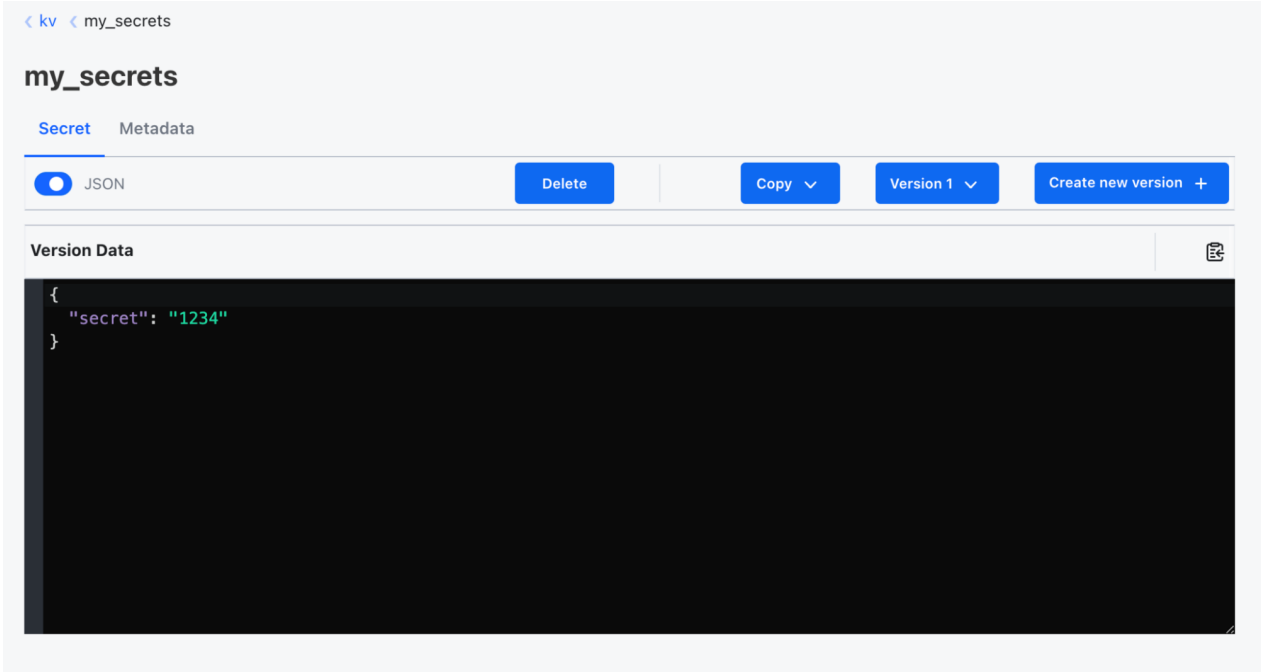


To view the configuration of an engine, go to the “Configuration” tab. The content of this tab depends on the selected engine.



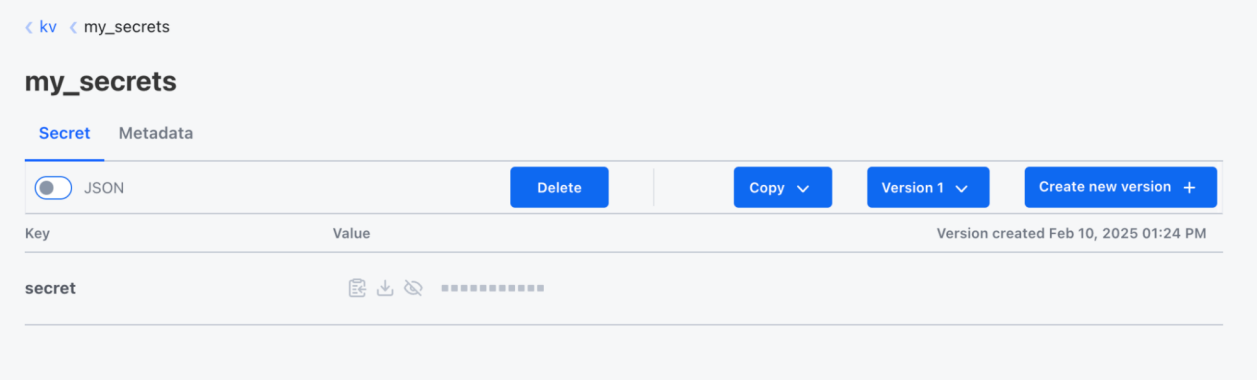
To view secret information and versions (for example, for a “key-value” engine):

1. Click the secret’s name in the list.
2. You will see the following tabs:
   - “Secret” with general information about the secret and version history.
   - “Metadata” with metadata of the secret.

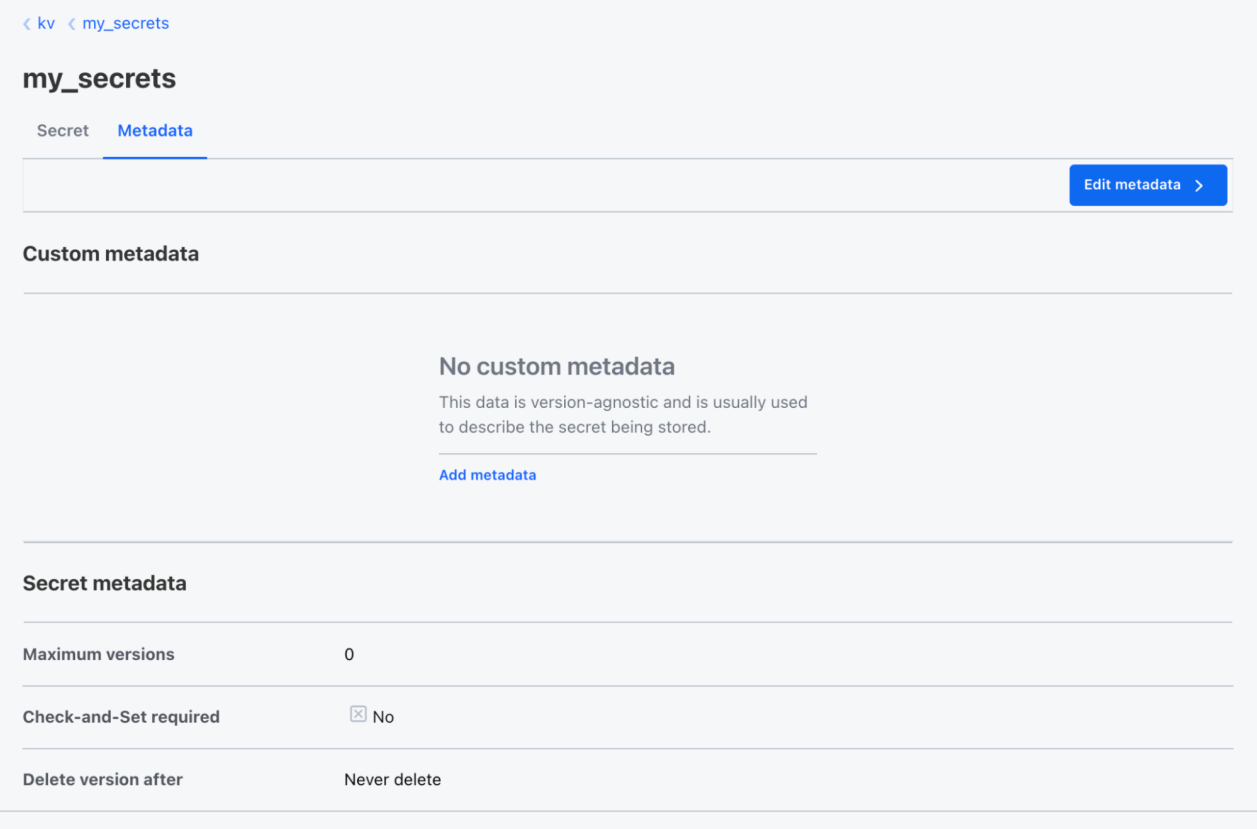


The “Secret” tab includes:

- A toggle to switch to JSON view.
- Buttons for:
  - Deleting the secret.
  - Copying the secret.
  - Selecting a version (if available).
  - Adding a new version.

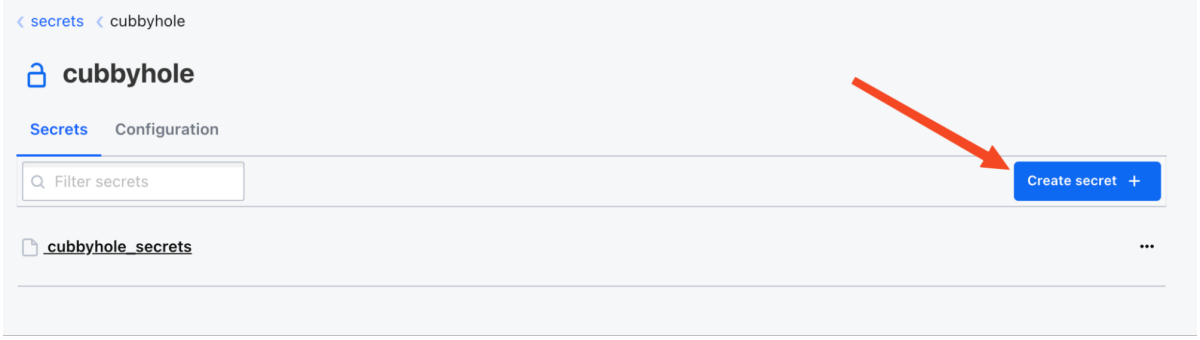


To view metadata, switch to the “Metadata” tab.

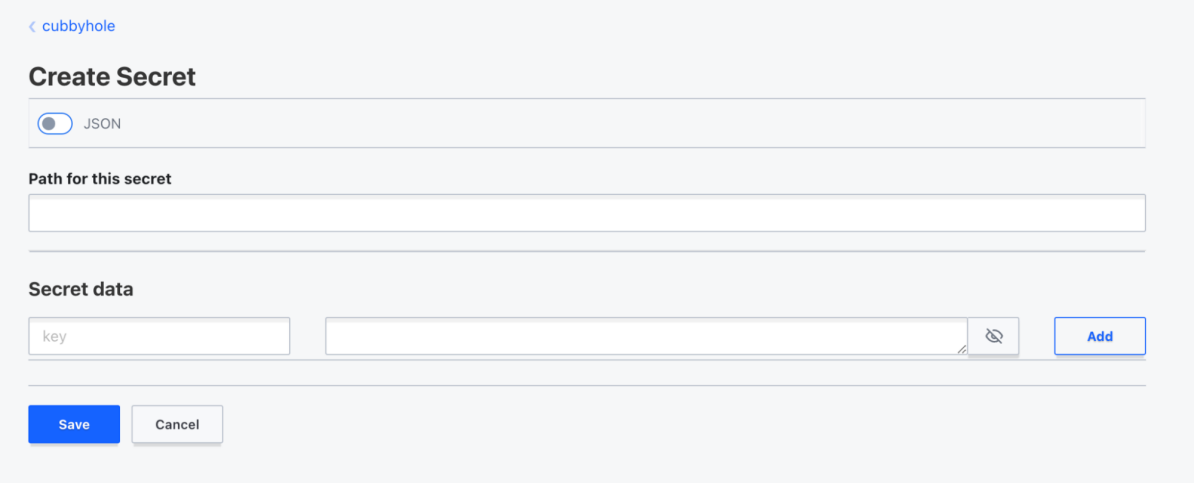


To add a new secret, follow these steps:

1. Click the button for adding a secret (button may vary depending on the engine: role, key, etc.).

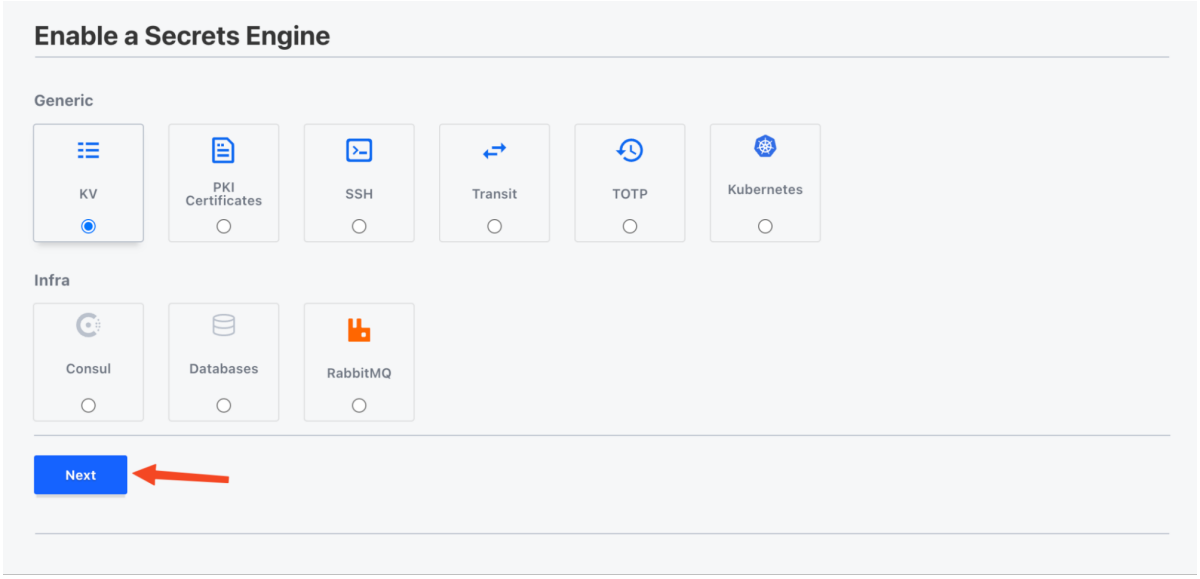


2. In the form that opens, provide the required parameters (fields vary by engine). For example, the form in a “Cubbyhole” engine includes the following fields:
   - A JSON view and edit toggle.
   - A path field.
   - A key field.
   - A value field.
   - A button to add additional key-value pairs under the same path.
3. Click **Save** to finish.

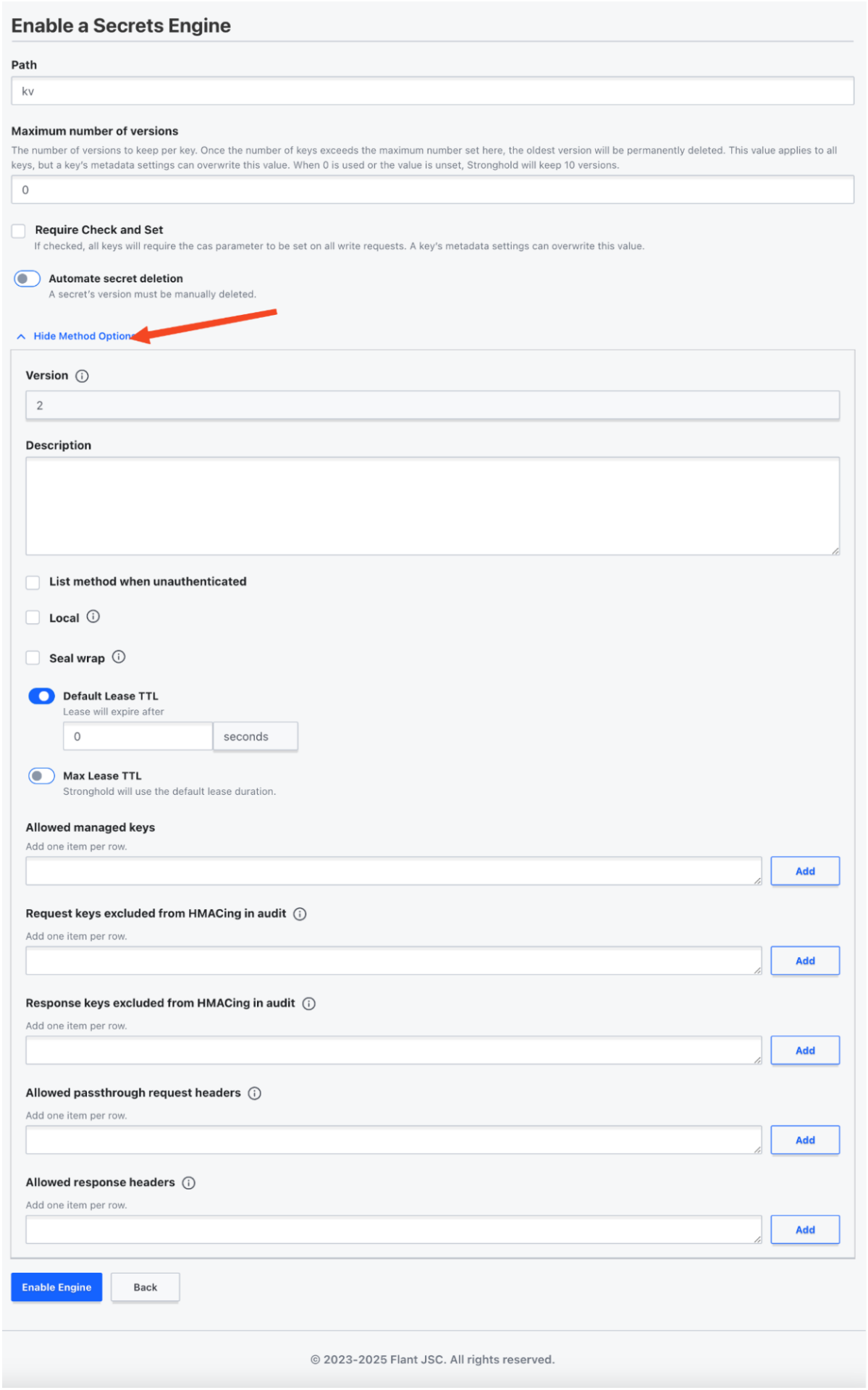


To add a new secrets engine:

1. Click **Enable new engine**.



2. Choose an engine type and click **Next**.
3. In the setup window, provide:
   - Basic engine configuration (depends on type)
   - Optional method settings (click the header to expand)
4. Click **Enable Engine** to save the engine. Alternatively, click **Back** to cancel and return.



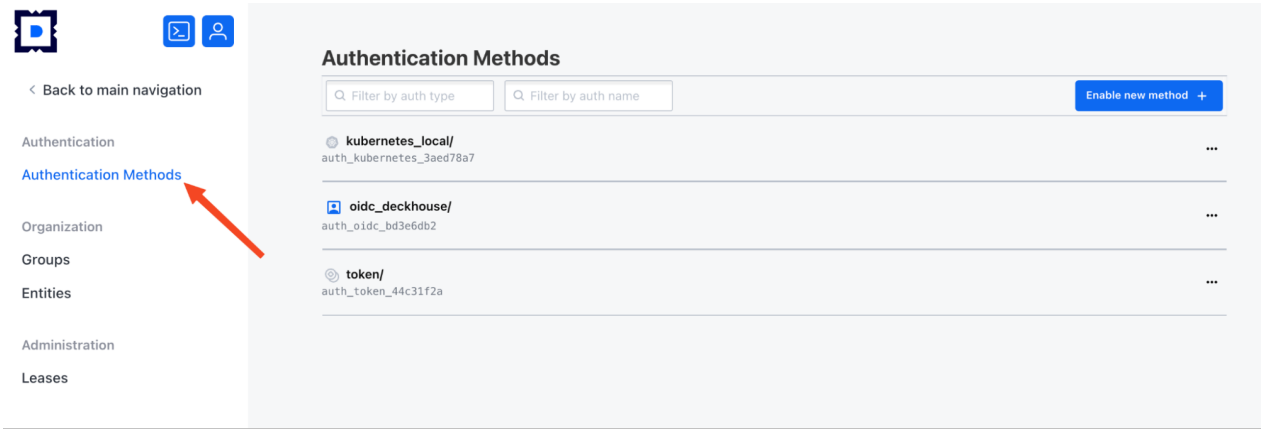
Managing access to data and stronghold features

Access control is managed in the “Access” section.

To open it, click **Access** in on the main page of stronghold web UI. The left pane contains a navigation menu with a link to return to the main page. The central pane displays content for the selected subsection (by default it's "Authentication Methods").

Working with authentication methods

This subsection opens by default when entering the "Access" section. You can also click "Authentication Methods" in the left menu.



This subsection shows:

- A list of authentication methods configured in the cluster.
- Filters for the list.
- A button to enable new method.

Available actions for each method:

- View configuration.
- Edit configuration.
- Delete method.

To view method details, click its name or select "View configuration". A configuration tab will appear with a button to configure the method.

For example, for the `oidc_deckhouse` method, the "Configuration" tab and a "Configure" button are shown.

To add a new authentication method:

1. Click **Enable new method** in the "Authentication Methods" section.
2. Choose a method from the list and click **Next**.
3. In the opened window, provide:
   - Path
   - Optional method options (click the header to expand)
4. Click **Enable Method** to finish. Alternatively, Click **Back** to cancel and return.

User groups

To manage user groups, click **Groups** in the side menu.

This subsection provides:

- A list of user groups in the cluster.
- Filters for the list.
- A button to add a new group.

Available actions for listed groups:

- View group details.
- Edit group settings.
- Delete a group.

To view group details, click its name or select **Details**.

To add a new user group:

1. Click **Create group** in the "Groups" section.
2. Fill out the group creation form.
3. Click **Create** to save. Alternatively, click **Back** to cancel and return.

Entities and aliases

Entities in stronghold are logical representations of users or applications. They allow linking multiple authentication methods under a single identity.

To manage entities and aliases, select **Entities** from the side menu of the “Access” section.

The center panel has two tabs:

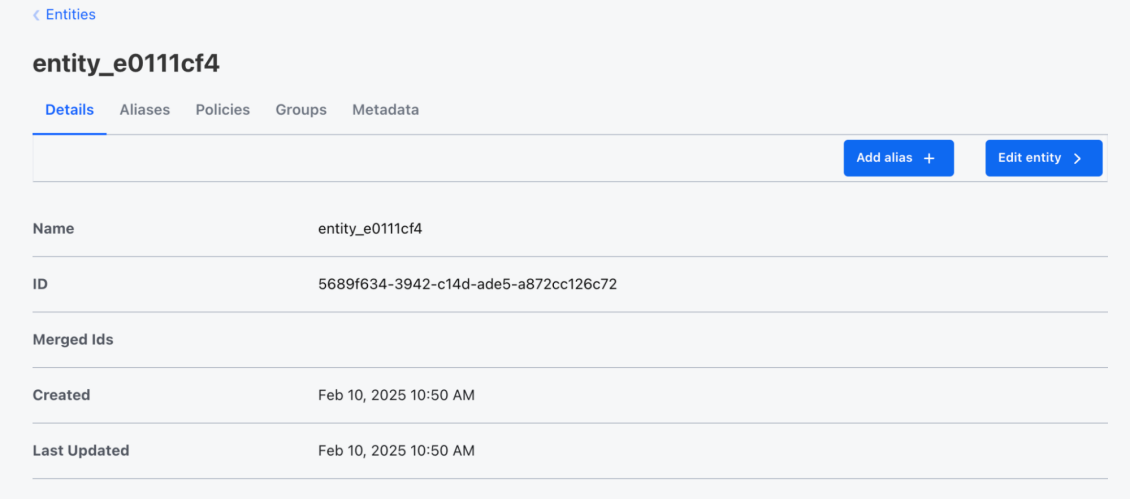
- **Entities** with a list of entities.
- **Aliases** with a list of aliases.

Also available:

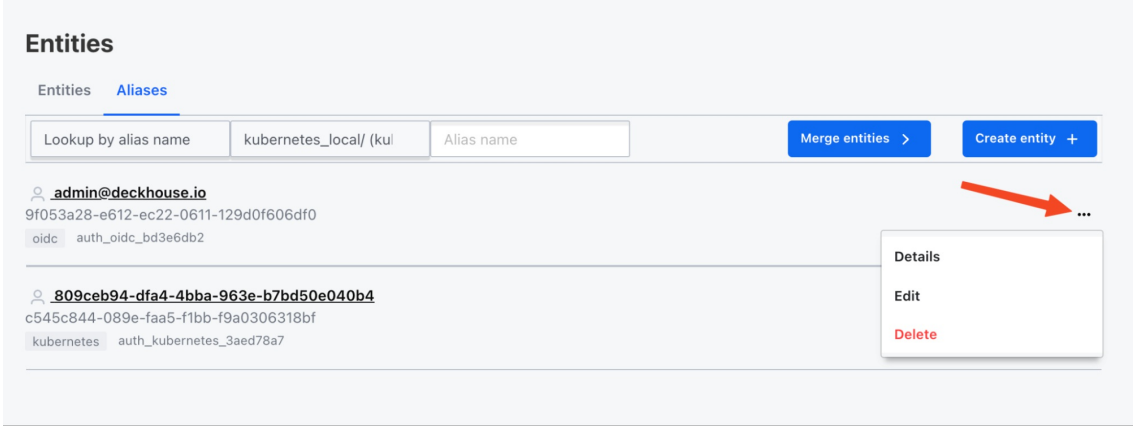
- Filters.
- **Merge entities** button to combine identities.
- **Create entity** button.

Available actions:

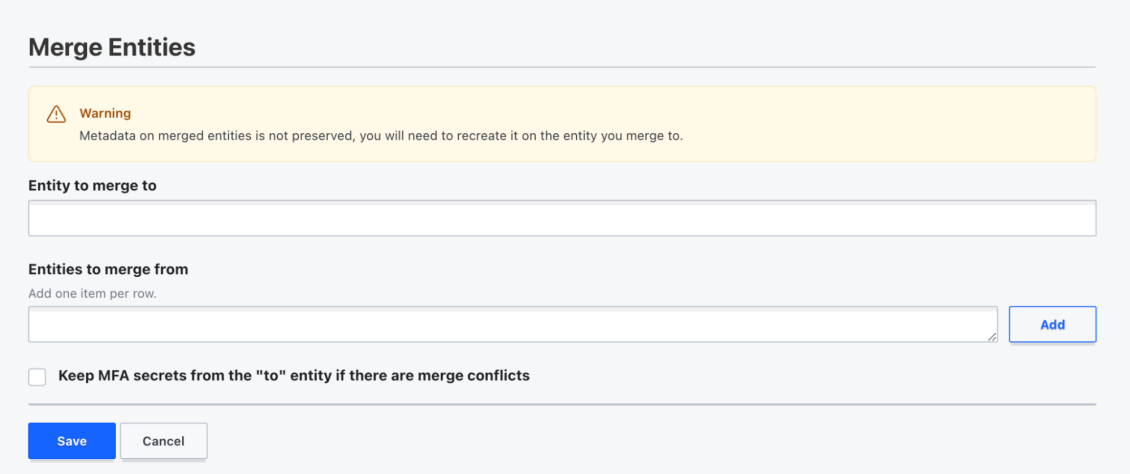
- For entities:
  - View details.



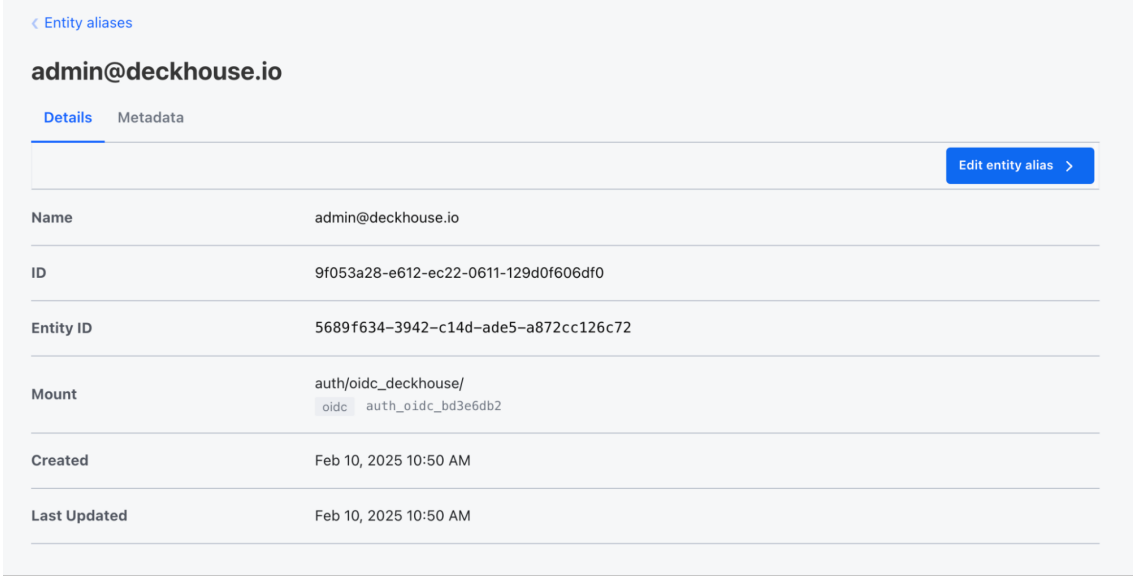
  - Create an entity.



  - Delete an entity.



- For Aliases:
  - View details.



  - Edit an alias.



  - Create an alias.
  - Delete an alias.

Managing temporary access to secrets and resources (leases)

To manage leases (temporary access to secrets and resources), click **Leases**. A search box will open that lets you find lease details by ID.



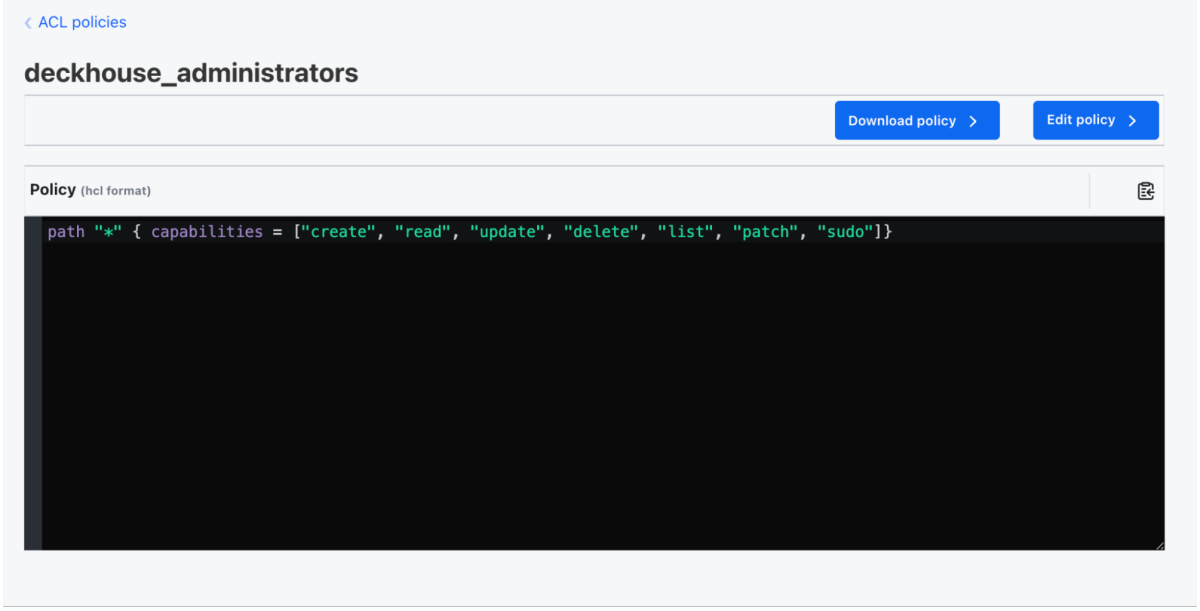
Managing access control policies

To manage access control policies in stronghold, click **Policies** from the “Access” menu. The left pane shows navigation menu. The central pane contains a list of policies, a search filter, and a button to add a new policy.

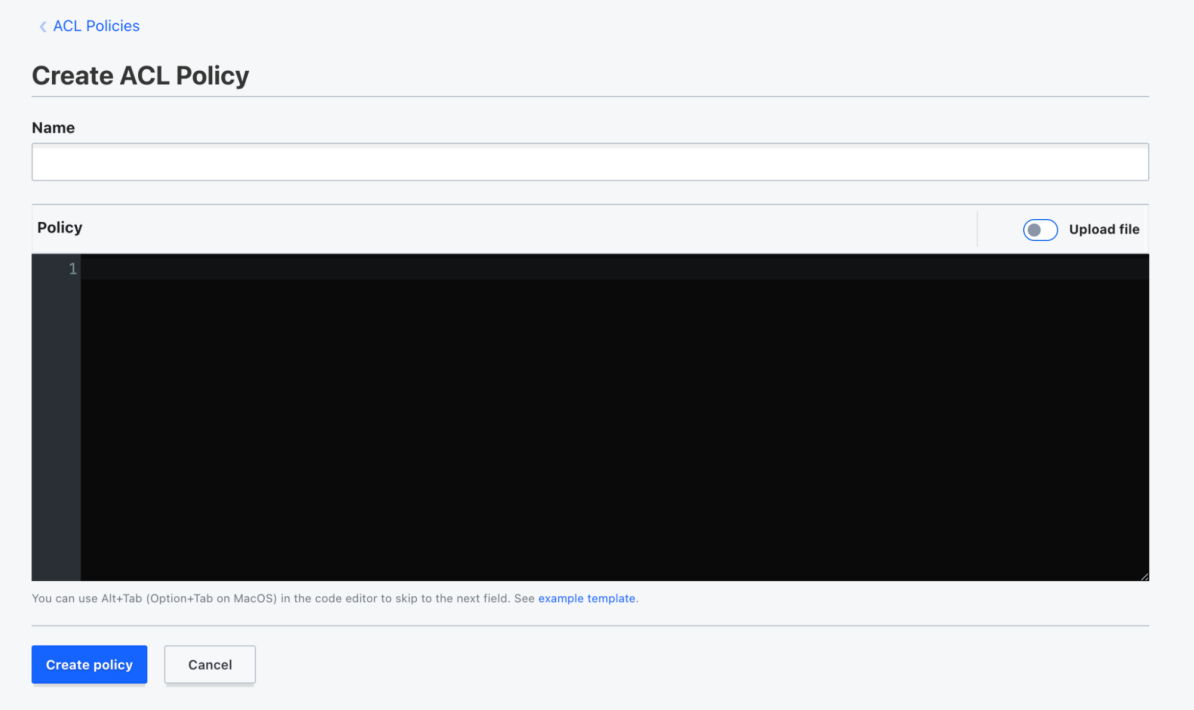
For actions with a policy, use the three-dot menu at the end of the row with the policy’s name.

Available actions for listed policies:

- **View policy details**. To view information about a policy, click its name or select “Details”. The policy details window shows information in HCL format, a button to download the data, and a button to edit the policy.



- **Edit policy**. To add a policy, click **Create ACL policy** and enter its name and HCL configuration.



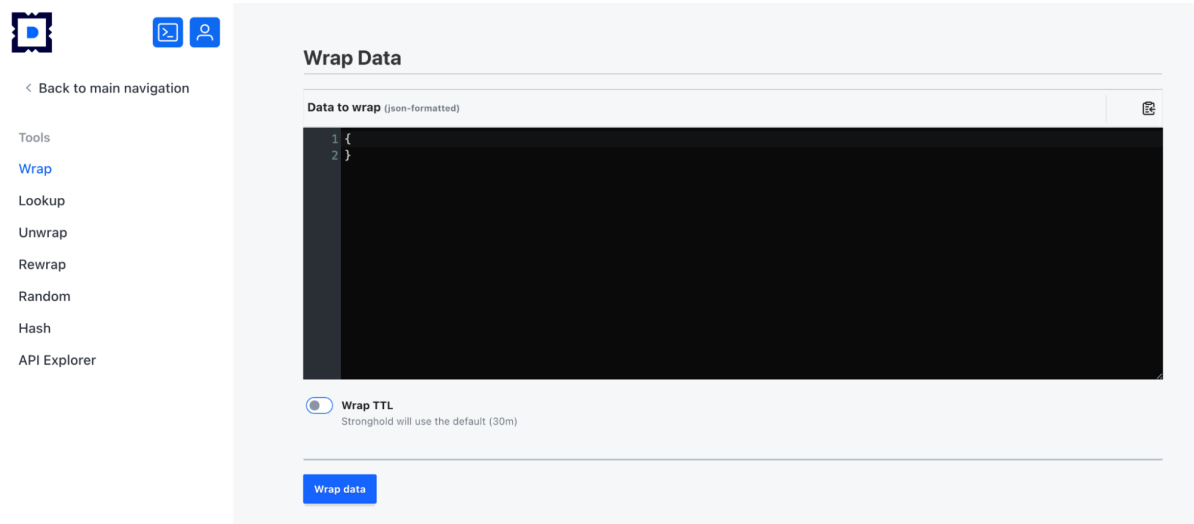
- **Delete policy**.

Using additional tools

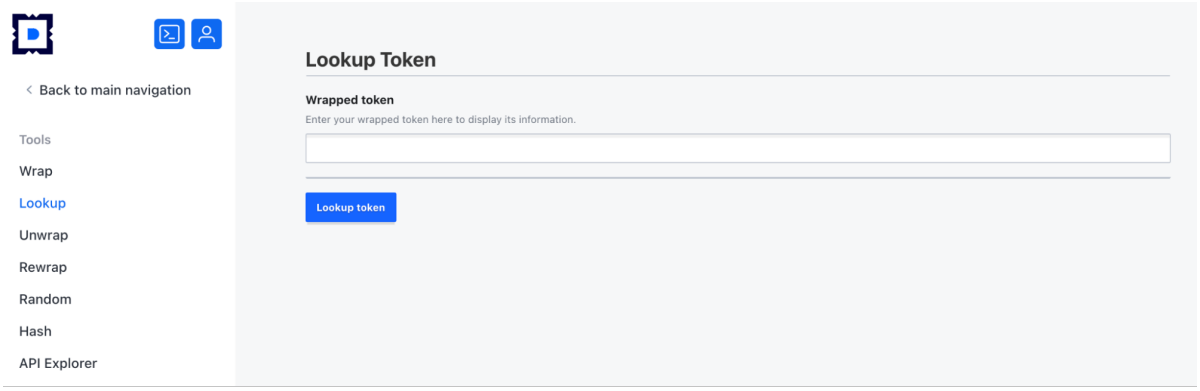
The “Tools” section provides access to extra tools in stronghold. The left pane displays tool navigation. The center shows fields for the selected tool.

Available tools:

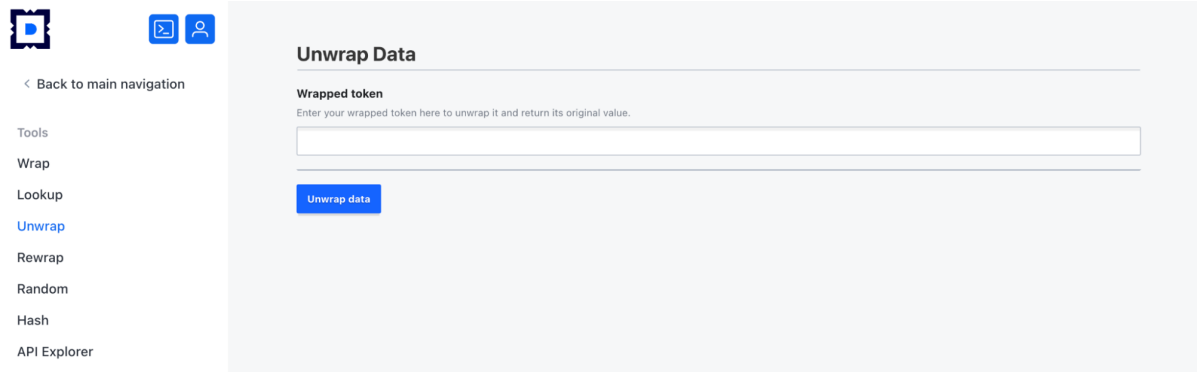
- **Wrap**: Creates a wrapping token for secure transfer of confidential data and secrets. This token can be transferred to another user or an application for subsequent unwrapping and accessing the data.



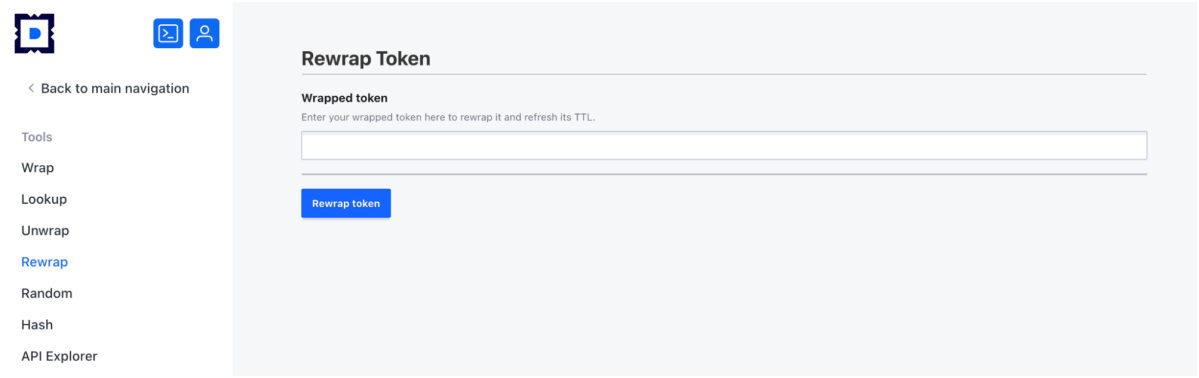
- **Lookup**: Displays details about tokens, secrets, leases, and other objects in stronghold. It can be used to view metadata, expiration dates, access policies, and other details associated with objects.



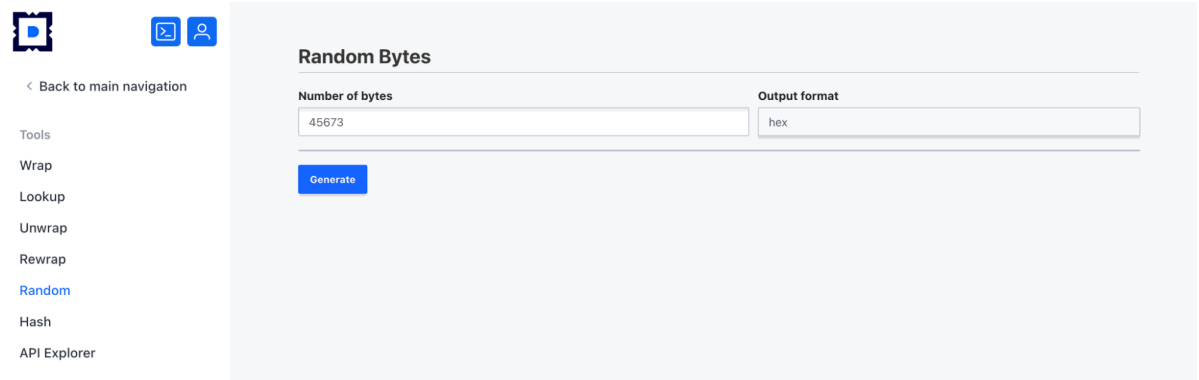
- **Unwrap**: Unwraps a wrapping token to access the wrapped data.



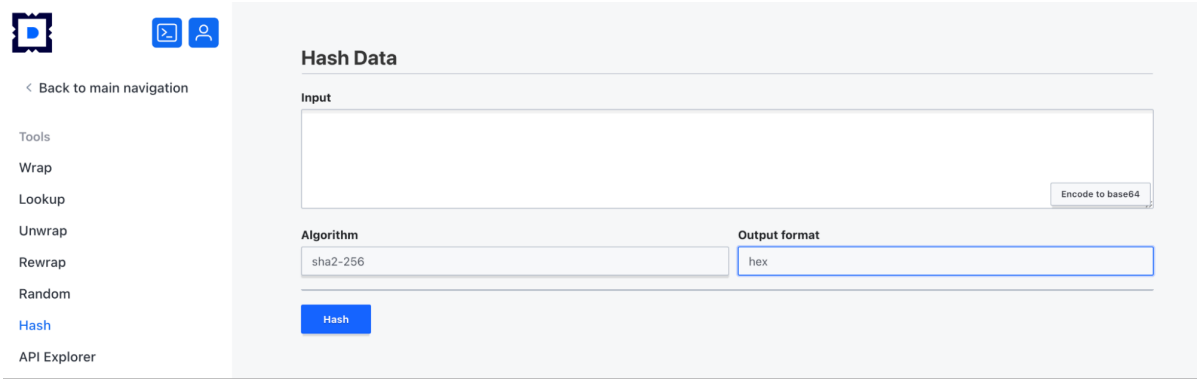
- **Rewrap**: Reissues a wrapping token based on an existing one. This lets you extend the token's TTL or modify its parameters without disclosing protected data.



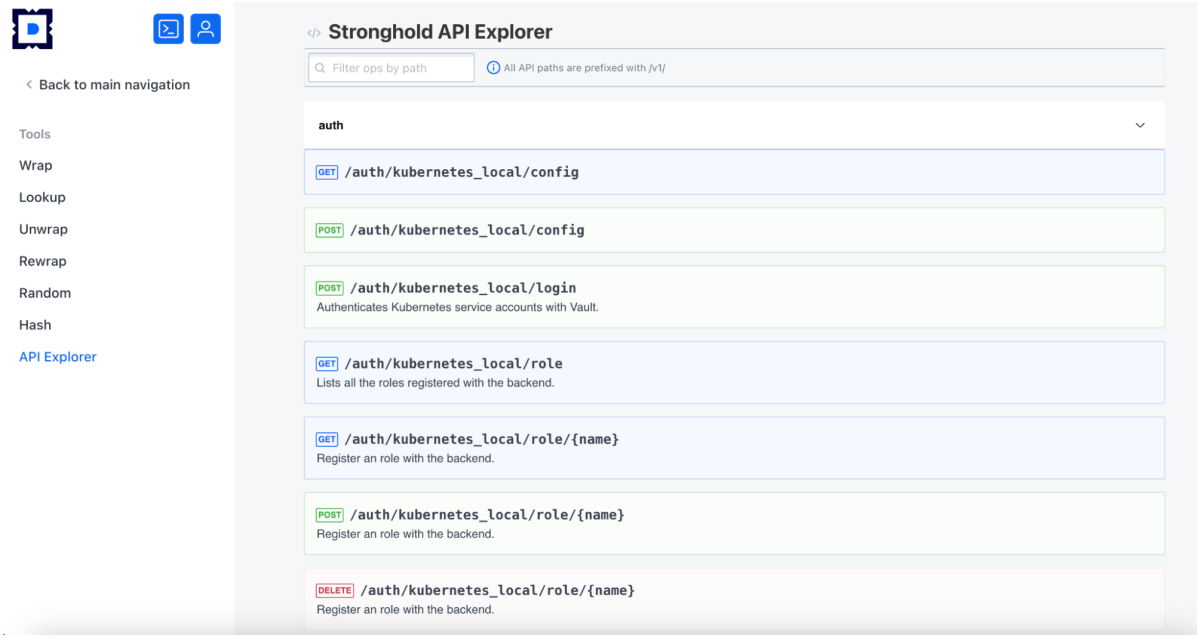
- **Random**: Generates cryptographically secure random data to create unique IDs, tokens, passwords and other data that requires a higher grade of randomness and security.



- **Hash**: Generates hashes for different data. Supports multiple caching algorithms.

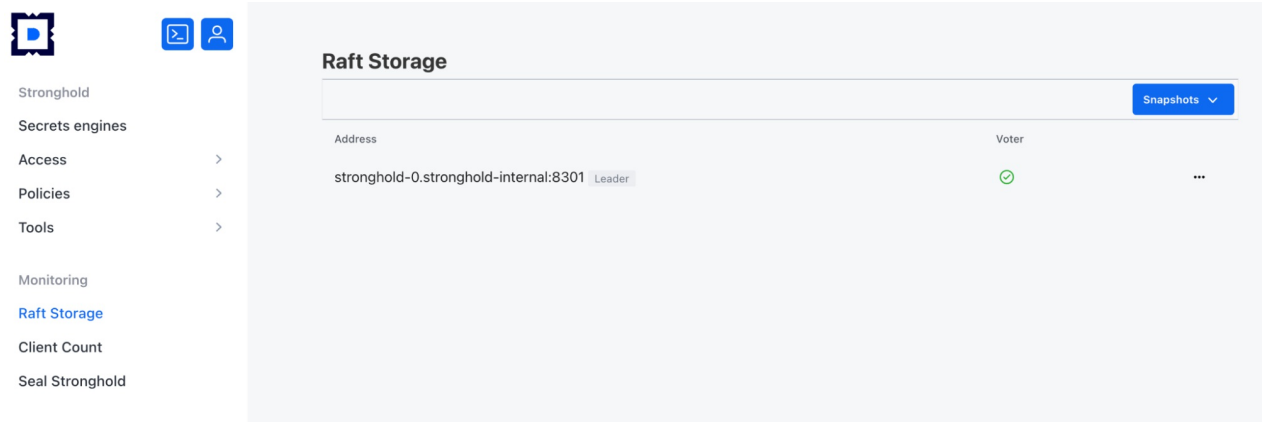


- **API Explorer**: Used for interacting with stronghold API via GUI.



Monitoring stronghold Raft cluster

Use the “Raft Storage” section to monitor the stronghold Raft cluster. The left menu contains a navigation menu; the central pane shows the cluster leader, nodes, and a **Snapshots** button for backing up or restoring Raft data from a snapshot.

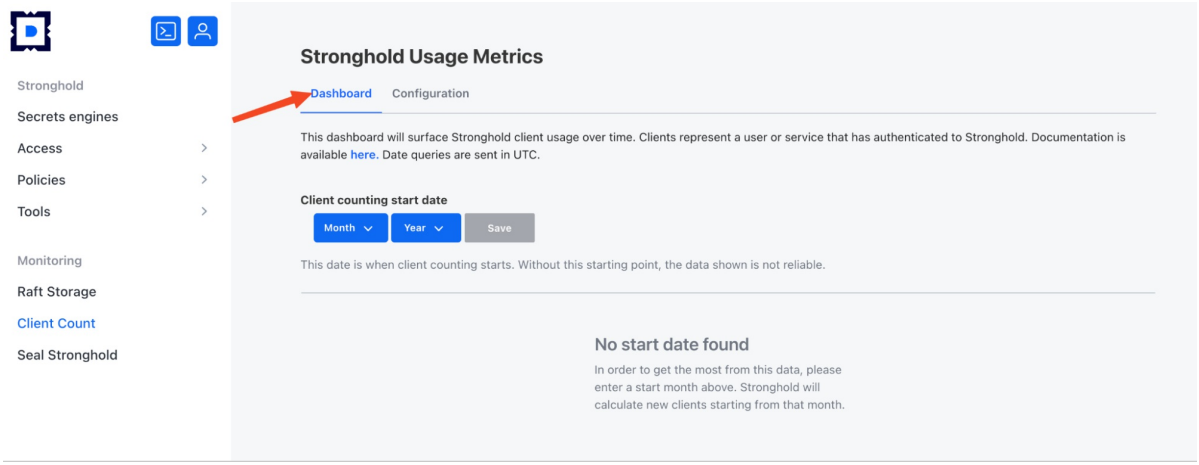


Monitoring activity and load

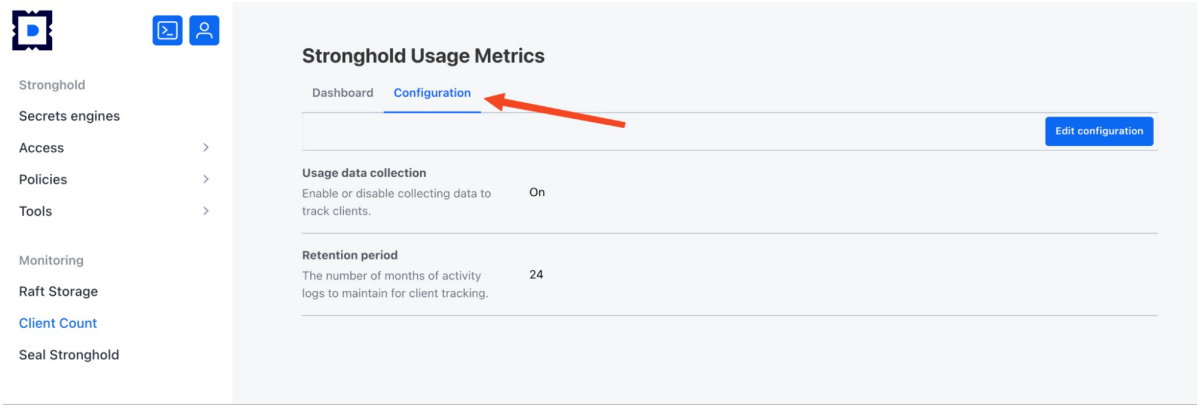
To monitor activity and assess stronghold workload, go to the “Client Count” section. The left pane contains a navigation menu.

The cental pane contains the following two tabs:

- **Dashboard** with a number of unique clients for the current month and buttons to change the period.



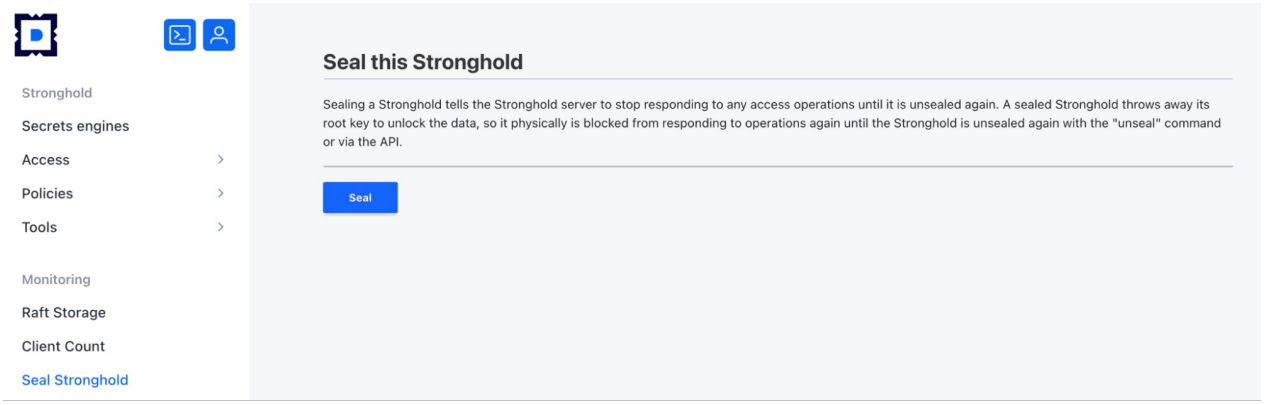
- **Configuration** that allows editing of metric collection settings (use the **Edit configuration** button for that).



Sealing and unsealing the secrets storage

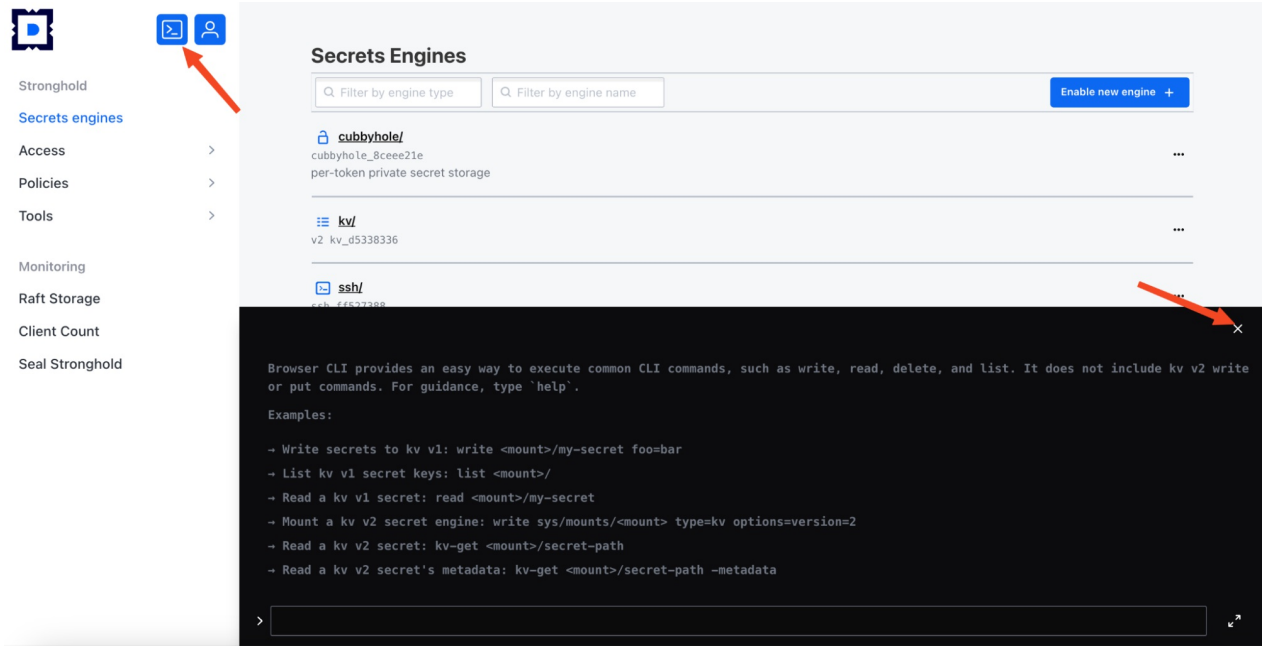
The “Seal Stronghold” section provides means to seal or unseal the secrets storage. The left pane contains a navigation menu. The cental pane contains a button for sealing or unsealing the secret storage (depending on its current state).

When the storage is sealed, stronghold can’t read or write secrets.



Using the stronghold CLI

The stronghold CLI tool can perform operations related to secrets, policies, users, and more. You can launch it from any interface section via the CLI button in the upper-left corner. Click again to close it.



Additional resources

- Module `stronghold` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

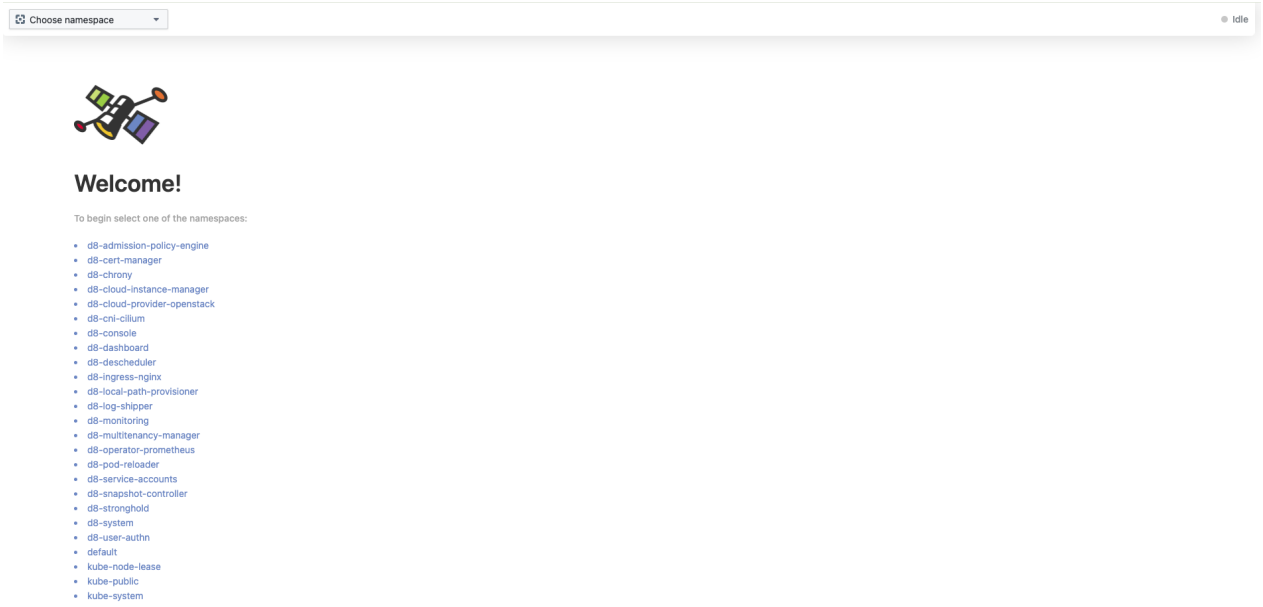
Web interface for visualizing the cluster network stack

The web interface for cluster network stack visualization is intended for operational diagnostics and analysis of network interactions. This interface enables tracking of communications between pods, services, and external resources, analyzing network activity and pinpointing network issues. The interface is implemented by the `cilium-hubble` module.

Accessing the web interface

To access the web interface for visualizing the cluster network stack, open the URL `hubble.<CLUSTER_NAME_TEMPLATE>`, where `<CLUSTER_NAME_TEMPLATE>` is the string matching the cluster's DNS name template, as specified by the global parameter `modules.publicDomainTemplate`.

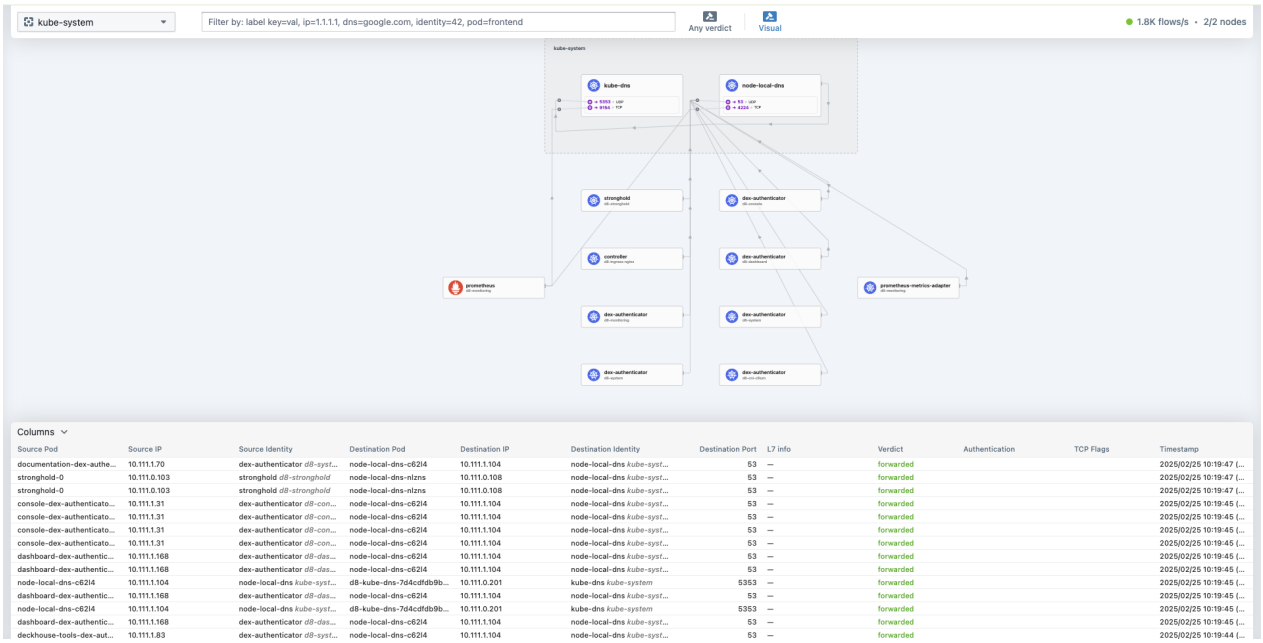
When you navigate to `hubble.<CLUSTER_NAME_TEMPLATE>`, you will see a namespace selection screen to choose which namespace's network stack will be displayed.



Select a namespace using the dropdown in the top-left corner or by clicking the desired namespace name in the list at the center of the screen.

After selecting a namespace, you will see the network stack visualization and analysis tools. The interface consists of the following parts:

- Top panel with filters and a brief cluster summary (number of flows and number of nodes). For more details on filtering what is displayed, see the section Filtering data for display.
- Network flow diagram. For more information on working with the visual flow diagram, see Working with the network flow diagram.
- Network flows and events table. For details on using the table, see Working with the network flows and events table.

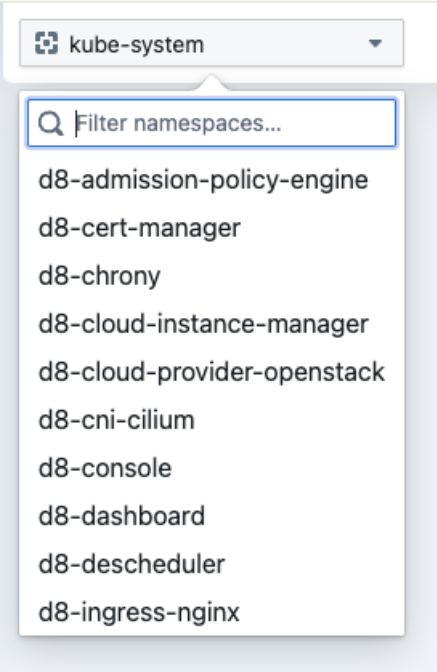


Data in the diagram and the network flows table is displayed in real time.

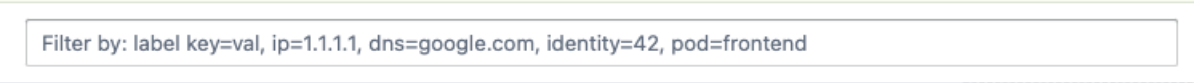
Filtering data for display

To filter the displayed information about the network stack and flows, use the top panel with filters. This panel contains:

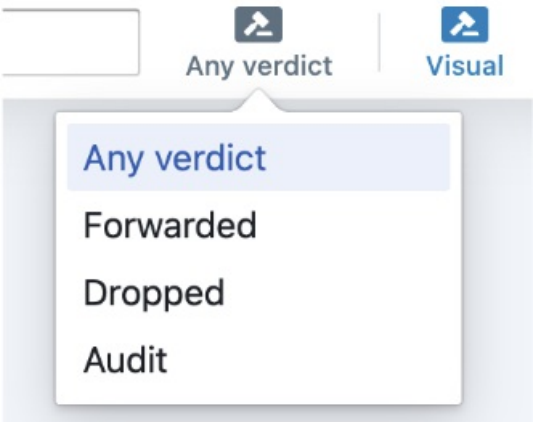
- Namespace selector (dropdown on the left side of the panel).



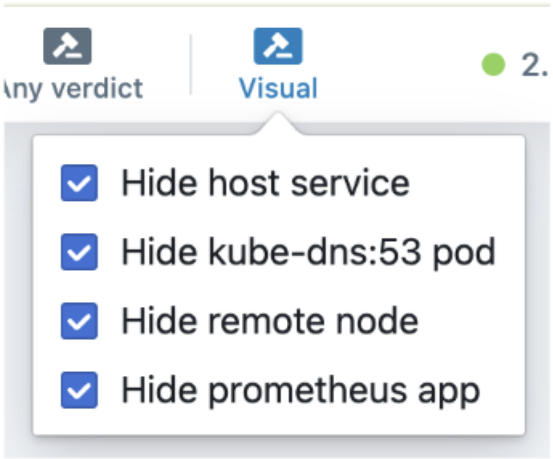
- Resource selector (input field in the center of the panel) to choose which namespace resources' flows to display.



- Flow verdict selector, based on the decision ("verdict") made by Cilium for each flow.



- Diagram element selector for choosing which elements in the namespace to show on the diagram.

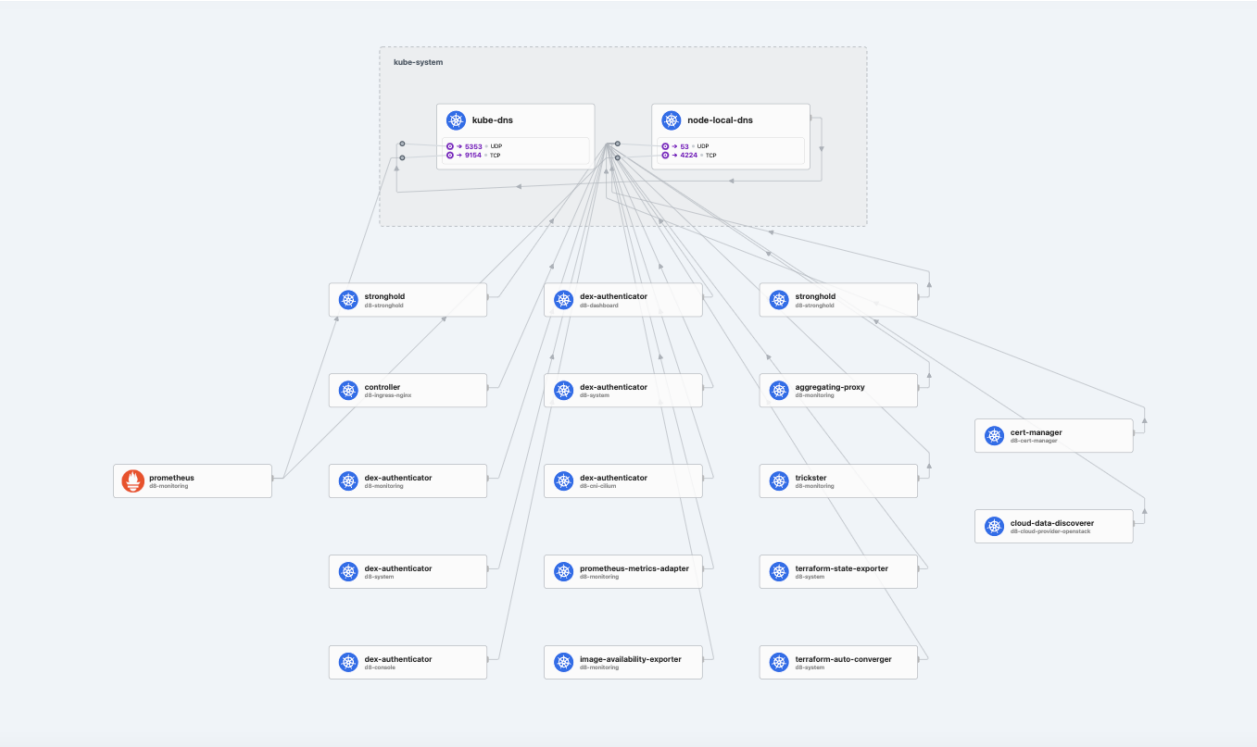


Working with the network flow diagram

The network flow diagram for the selected namespace appears in the middle of the screen, showing the network stack visualization and analysis tools. The diagram displays the resources of the chosen

namespace inside a rectangle labeled with that namespace’s name, as well as external elements they interact with.

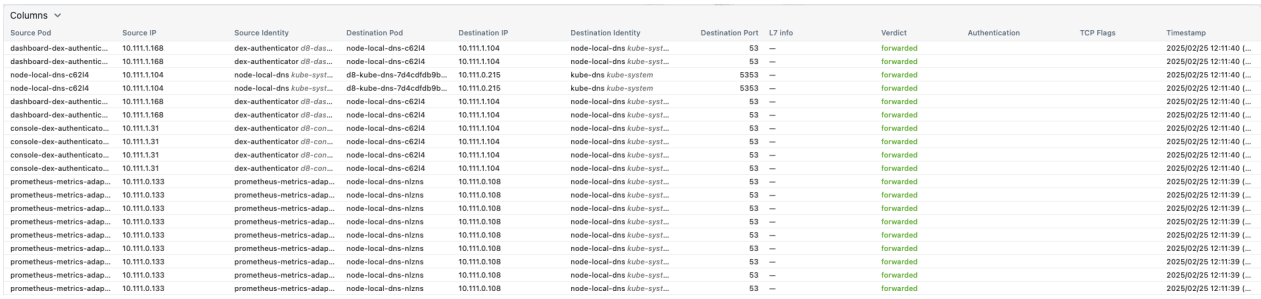
To view detailed information (labels, network interactions, etc.) for a specific resource on the diagram, click on that resource.



Working with the network flows and events table

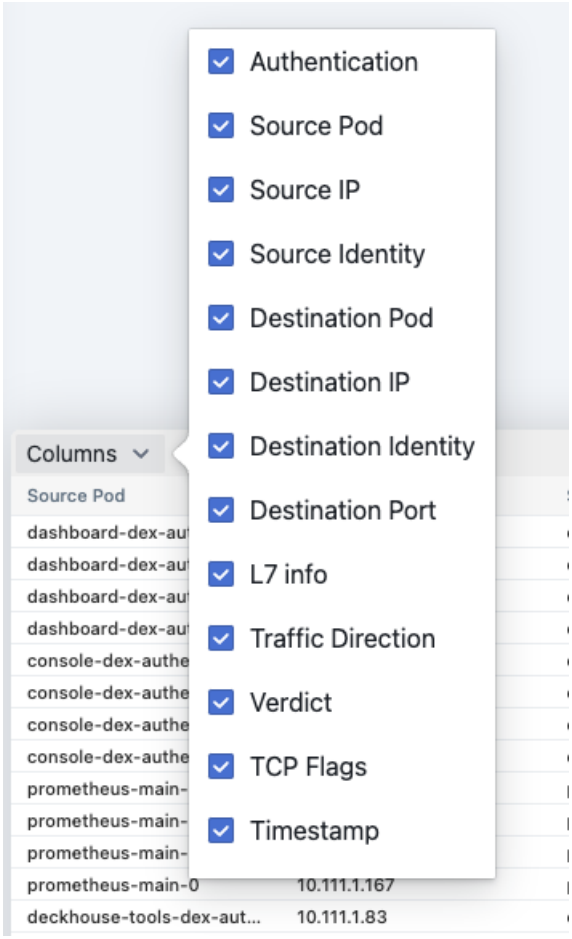
Each row in the table contains the following information about a network flow:

- Source Pod (pod name of the flow source)
- Source IP (IP address of the source pod)
- Source Identity (entity identifier for the source)
- Destination Pod (pod name of the flow destination)
- Destination IP (IP address of the destination pod)
- Destination Identity (entity identifier for the destination)
- Destination Port (destination port number)
- L7 Info (application-layer information, if the flow uses HTTP)
- Verdict (the decision made by Cilium for the flow)
- Authentication (authentication results for the flow, if applicable)
- TCP Flags (TCP flags associated with the flow)
- Timestamp (time when the flow occurred)



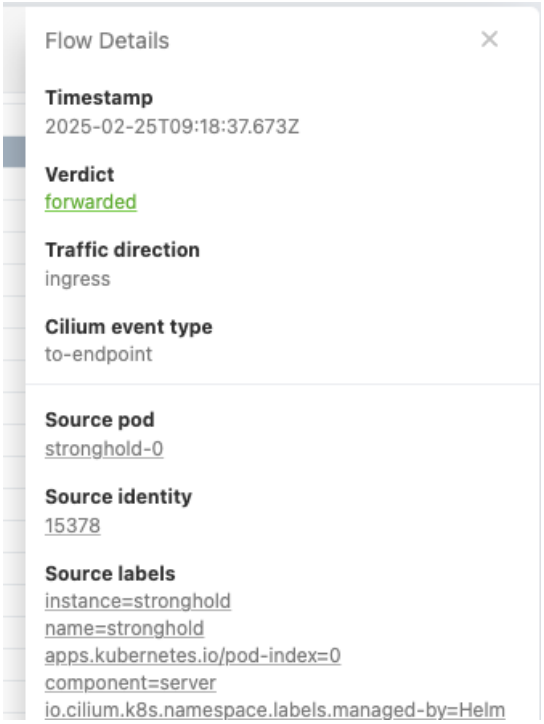
Configuring the column set

To configure which columns are shown in the table, click the Columns button in the top-left corner of the table and select the columns you need.



Viewing row details

To view detailed information for a specific row in text form, click anywhere within that row. The information will appear on the right side of the table. These details are shown regardless of which columns are currently selected for display.



> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Using multitenancy

Multitenancy is the ability to create isolated environments (projects) within a Kubernetes cluster. Projects are similar to namespaces but offer more capabilities. While namespaces are used for logical separation of resources in Kubernetes, they do not, for example, restrict network communication, pod resource consumption, or host directory mounts. These limitations make namespaces insufficient for modern development needs. By default, namespaces also do not include logging, auditing, and vulnerability scanning.

Using projects helps address these limitations and offers the following benefits:

- For platform administrators:
  - **Consistency**: Administrators can create projects using a shared template, which ensures consistency and simplifies management.
  - **Security**: Projects ensure isolation for resources and access policies between different tenants, supporting a secure multitenant environment.
  - **Resource consumption**: Administrators can easily set resource quotas and limits for each project to prevent resource overconsumption.
- For platform users:
  - **Immediate start**: Developers can request projects created per templates from administrators to quickly start developing new applications.
  - **Isolation**: Each project provides an isolated environment, allowing developers to deploy and test their applications without affecting others.

> ⚠ Secret copying across all namespaces is incompatible with projects in multitenancy mode.
>
> This mode creates isolated environments for users within their projects, while `secret-copier` automatically distributes secrets to all namespaces. If sensitive data is present in a user's private environment, it could lead to a data leak and a security model breach.

Additional namespaces

A project is not limited to a single namespace: if an application needs several namespaces (for example, a separate one for a cache or a queue), a cluster administrator, a project administrator or a project manager (`d8:project:manager`) can add them to the project as additional namespaces. Access grants and namespaced template policies (network isolation, log shipping) automatically apply to every namespace of the project, not just the main one. For details, refer to the Administration section.

Creating a project

1. To create a project, create a Project custom resource and specify the project template name in the `.spec.projectTemplateName` field.
2. Set the standard fields — `.spec.administrators` and `.spec.quota` — which are managed directly by the Project resource regardless of the template.
3. In the `.spec.parameters` field, specify parameter values for the `.spec.parametersSchema.openAPIV3Schema` section of the `ProjectTemplate` custom resource.

   Example of creating a project using Project from the `default` ProjectTemplate:

```
apiVersion: deckhouse.io/v1alpha3
kind: Project
metadata:
  name: my-project
spec:
  description: This is an example from the Deckhouse documentation.
  projectTemplateName: default
  # Standard fields, managed by the Project resource itself, independently of
the template.
  administrators:
    - kind: Group
      name: k8s-admins
  quota:
    requests.cpu: "5"
    requests.memory: 5Gi
    requests.storage: 1Gi
    limits.cpu: "5"
    limits.memory: 5Gi
  # Template-specific parameters.
  parameters:
    networkPolicy: Isolated
    podSecurityProfile: Restricted
    extendedMonitoringEnabled: true
```

4. To check the project status, run the following command:

   ```
   d8 k get projects my-project
   ```

   A successfully created project should have the `Deployed` (synchronized) status. If the `Error` status is displayed, add the `-o yaml` flag to the command (for example, `d8 k get projects my-project -o yaml`) to get more details about the cause of the error.

Automatically creating a project from a namespace

A namespace created directly (for example, `d8 k create ns test`) automatically becomes a project with the same name — no annotation is required. The project parameters are filled in from the current state of the namespace, so nothing inside it changes; from then on the project is the source of truth for it.

Users can create a namespace directly only while the `allowNamespacesWithoutProjects` parameter of the `multitenancy-manager` module is `true` (the default). With `false`, only cluster administrators and system components create namespaces directly, and other users create a Project or a ProjectNamespace instead.

The namespace of such a project cannot be deleted directly. To remove the environment, delete the Project: `d8 k delete project <NAME>` deletes the namespace with it.

For example:

1. Create a new namespace:

   ```
   d8 k create ns test
   ```

2. Check that the project was created:

   ```
   d8 k get projects
   ```

   In the output list of projects, you should see the newly created project corresponding to the namespace:

```
NAME        STATE      PROJECT TEMPLATE   DESCRIPTION
AGE
deckhouse   Deployed   virtual            This is a virtual project
181d
default     Deployed   virtual            This is a virtual project
181d
test        Deployed   simple
1m
```

To change the template of such a project, set another template in `.spec.projectTemplateName`.

⚠ If the namespace already has an object of the same kind and name as one that the new template creates, for example the `isolated` NetworkPolicy, the project switches to `Error`, and the `ResourcesUpgraded` condition names the object. Delete or rename the object first.

For details on project templates and their creation, refer to the Administration section.

Viewing available cluster-wide resources

A cluster administrator can restrict the use of cluster-wide resources in projects. For example, the administrator can define which StorageClasses, ClusterIssuers, and ClusterRoles are available to a project.

To view the cluster-wide resources available to a project, run the following command:

```
d8 k get available -n <PROJECT_NAME>
```

To view available resources of a specific type and the value used by default, specify the corresponding AvailableClusterResource. For example, for StorageClass:

```
d8 k get available storageclasses -n <PROJECT_NAME> -o yaml
```

If you see a message such as `[multitenancy] <KIND> "<OBJECT NAME>" references "<RESOURCE_NAME>" which is not available to project "<PROJECT_NAME>"` when creating or modifying a resource, the specified cluster-wide resource is unavailable to the project. Select an available resource from the corresponding AvailableClusterResource or ask the cluster administrator to add it.

For some fields, automatic default value assignment can be configured. For example, if `storageClassName` is not specified when creating a PersistentVolumeClaim, the StorageClass used by default in the project can be automatically assigned to this field.

For more information about using available cluster-wide resources, refer to the `multitenancy-manager` module documentation.

Additional resources

- Module `multitenancy-manager` documentation
- Module `secret-copier` documentation

ℹ **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**

The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Overview

Managed services allow users to create and use ready-to-run application services by using Kubernetes custom resources.

How it works

A user creates a custom resource of a managed service in their namespace (project), for example Postgres, and describes the desired configuration in it: resources, deployment mode, topology, users, and other parameters. The resource references a managed service class (for example, PostgresClass) prepared by the cluster administrator — the class defines the allowed parameters and limits.

The operator watches these resources and creates or updates the managed service's running instances according to the specified configuration. The resource status reflects the current state of the managed service.

Use this section to:

- create a managed service in your namespace (project);
- configure managed service instance parameters;
- connect to the created managed service;
- perform basic maintenance operations;
- check the resource status and configuration result.

For PostgreSQL creation and usage, see Managed PostgreSQL.

Details about infrastructure preparation, managed service class configuration, and other administrative settings are provided in the Administration section.

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Managed PostgreSQL

Users can create and configure PostgreSQL if this capability has been enabled in the DP cluster by the administrator.

The user sets the required configuration using a Postgres object that points to a specific service class (PostgresClass), which defines the available parameters and limits. The PostgresClass is created and configured by the cluster administrator.

This guide uses two examples:

- `app-postgres`: the main example for creating and operating PostgreSQL — resources, `Cluster` mode, replication, users, databases, PostgreSQL parameters, TLS, and observability;
- `snapshot-pg`: a separate example for creating and restoring snapshots, since it requires a StorageClass with CSI snapshot support.

> The examples use two worker nodes so that PostgreSQL instances in `Cluster` mode can be placed on different nodes. The cluster is located in a single zone, `default`.

Check available resources

Before creating Postgres, check the available resources of the worker nodes. This lets you choose CPU and memory values for the example based on the actual load on the cluster.

First, list the nodes:

```
d8 k get nodes -o wide
```

The example uses two available worker nodes.

Example output:

```
NAME       STATUS   ROLES    AGE   VERSION
worker-1   Ready    worker   25d   v1.34.9
worker-2   Ready    worker   43m   v1.34.9
```

Check the resources allocated on the first worker node:

```
d8 k describe node worker-1 | grep -A 5 "Allocated resources"
```

Example output:

```
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests          Limits
  --------           --------          ------
  cpu                1104m (28%)       500m (12%)
  memory             4096854330 (53%)  390Mi (5%)
```

Check the second worker node:

```
d8 k describe node worker-2 | grep -A 5 "Allocated resources"
```

Example output:

```
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests      Limits
  --------           --------      ------
  cpu                472m (12%)    500m (12%)
  memory             1004Mi (13%)  256Mi (3%)
```

Check storage

Before creating Postgres, check the available StorageClass options and choose the storage class where the PostgreSQL data will be placed:

```
d8 k get storageclass
```

Example output from the test bench:

```
NAME                    PROVISIONER            RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION
local                   csi.dvp.deckhouse.io   Delete          WaitForFirstConsumer
true
replicated (default)    csi.dvp.deckhouse.io   Delete          WaitForFirstConsumer
true
```

⚠ The `spec.instance.persistentVolumeClaim.storageClassName` parameter is set only when Postgres is created. It can’t be changed afterwards.

Main example of creating Postgres

Create a namespace:

```
d8 k create namespace my-postgres
```

To create a Postgres object, complete the following steps:

- Select PostgresClass;
- Configure resources;
- Select deployment mode;
- Configure topology and replication mode;
- Create logical database and user;
- Configure PostgreSQL parameters;
- Configure TLS;
- Configure observability.

The following example shows an `app-postgres` Postgres manifest. You can apply it as is and then

customize it using the steps below.

```
apiVersion: managed-services.deckhouse.io/v1alpha1
kind: Postgres
metadata:
  name: app-postgres
  namespace: my-postgres
spec:
  postgresClassName: default

  configuration:
    maxConnections: 120

  instance:
    cpu:
      cores: 1
      coreFraction: 50
    memory:
      size: 1Gi
    persistentVolumeClaim:
      size: 10Gi
      storageClassName: replicated

  type: Cluster
  cluster:
    topology: Ignored
    replication: Consistency

  users:
    - name: app-rw
      role: rw
      storeCredsToSecret: app-postgres-rw

  databases:
    - name: app

  tls:
    mode: K8s

  observability: Enabled
```

Save the manifest to `postgres.yaml` and apply it:

```
d8 k apply -f postgres.yaml
```

Check the status of the created Postgres:

```
d8 k get postgres app-postgres -n my-postgres -o wide
```

Once the rollout is complete, the main conditions (`status.conditions`) should change to `True` . For details about each condition, see Check status.

Example output:

```
NAME           AVAILABLE   CONFIGURATIONVALID   LASTVALIDCONFIGURATIONAPPLIED
SCALEDTOLASTVALIDCONFIGURATION   DATABASESSYNCED   USERSSYNCED
app-postgres   True        True                 True                              True
True             True
```

To change a parameter of `app-postgres` , edit the corresponding fragment of `postgres.yaml` and reapply the file.

Select PostgresClass

The `spec.postgresClassName` parameter defines the PostgresClass that sets the available parameters and limits for Postgres. To list the PostgresClass resources available in the cluster, run:

```
d8 k get postgresclass
```

Example output:

```
NAME      AGE
default   13d
```

This example uses the `default` PostgresClass with standard limits. If a different PostgresClass is selected, you can view its limits in the configuration:

```
d8 k get postgresclass <CLASS_NAME> -o yaml
```

Where `<CLASS_NAME>` is the name of the selected PostgresClass.

> ⚠ When choosing a PostgresClass, consider the allowed CPU, memory, and `coreFraction` values, available topologies, and PostgreSQL parameters that can be overridden. If the Postgres configuration doesn’t meet the limits of the selected class, the API rejects it on apply.

The PostgresClass settings and limits are described in Limit CPU and memory resources, Manage fault tolerance across availability zones, and Automatically validate PostgreSQL settings.

Placement restrictions

A PostgresClass can also define PostgreSQL instance placement rules using `nodeSelector`, `nodeAffinity`, and `tolerations`. These rules apply automatically once the class is selected and aren’t specified in the Postgres object.

Configure resources

The `spec.instance` parameter is used to configure the resources of each PostgreSQL instance — the number of CPUs, the guaranteed CPU share, and the amount of memory.

In the example, this fragment is responsible for resources and storage:

```
spec:
  instance:
    cpu:
      cores: 1
      coreFraction: 50
    memory:
      size: 1Gi
```

In the example, the instance is allocated one CPU core and `1Gi` of memory. The `coreFraction` parameter defines the ratio of the CPU request to the CPU limit. For `cores: 1` and `coreFraction: 50`, the module produced:

Example output:

```
limits.cpu:   1
requests.cpu: 500m
```

For more information, see Limit CPU and memory resources.

Change resources of an existing Postgres

You can change Postgres resources by reapplying the manifest, provided the new values are allowed by the selected PostgresClass. First, check the current resource values:

```
d8 k get pods -n my-postgres \
  -l managed-services.deckhouse.io/managed-service-name=app-postgres \
  -o custom-
columns='NAME:.metadata.name,CPU_REQUEST:.spec.containers[0].resources.requests.cpu,M
EMORY_REQUEST:.spec.containers[0].resources.requests.memory'
```

Example output:

```
NAME                       CPU_REQUEST   MEMORY_REQUEST
d8ms-pg-app-postgres-1     500m          1Gi
d8ms-pg-app-postgres-2     500m          1Gi
```

You can apply changes to both memory and CPU at once, but for clarity, first increase memory from `1Gi` to `2Gi`:

```
spec:
  instance:
    memory:
      size: 2Gi
```

Apply the updated manifest:

```
d8 k apply -f postgres.yaml
```

After the update completes, the CPU request stays at `500m`, and the memory request of the instances changes to `2Gi`.

Example output:

```
NAME                       CPU_REQUEST   MEMORY_REQUEST
d8ms-pg-app-postgres-1     500m          2Gi
d8ms-pg-app-postgres-2     500m          2Gi
```

Next, change `coreFraction` from `50` to `100`, leaving `cores: 1`:

```
spec:
  instance:
    cpu:
      cores: 1
      coreFraction: 100
```

Reapply the manifest:

```
d8 k apply -f postgres.yaml
```

After the update completes, the CPU request of the instances changes from `500m` to `1`, and the memory request stays at `2Gi`.

Example output:

```
NAME                       CPU_REQUEST   MEMORY_REQUEST
d8ms-pg-app-postgres-1     1             2Gi
d8ms-pg-app-postgres-2     1             2Gi
```

For a running Postgres, you can change memory and `coreFraction` within the limits allowed by the selected PostgresClass.

Check CPU and memory limits via PostgresClass

CPU and memory values must comply with the limits of the selected PostgresClass. If the specified resources or their combination don’t match the allowed values, the API rejects the configuration.

The `default` PostgresClass isn’t well suited for a clear demonstration of these limits. So this example uses a separate `check` PostgresClass, which allows memory from `512Mi` to `2Gi` in `512Mi` steps for `1–2` CPUs.

With `cores: 1` and `coreFraction: 50`, a memory value of `700Mi` doesn’t match the configured step, so the manifest is rejected:

```
spec:
  postgresClassName: check
  instance:
    cpu:
      cores: 1
      coreFraction: 50
    memory:
      size: 700Mi
```

Apply the manifest:

```
d8 k apply -f postgres.yaml
```

The API rejects the request. Example output:

```
spec.instance.memory.size: Invalid value: 734003200: memory setting does not fit
Step 536870912 of the selected PostgresClass
```

Select deployment mode

The set of PostgreSQL instances depends on the selected deployment mode: `Cluster` creates a primary instance and replicas, whose composition depends on the selected replication mode; `Standalone` creates a single PostgreSQL instance without replicas.

Cluster mode

To work with a primary instance and replicas, use `Cluster` mode, set by the `spec.type` parameter.

```
spec:
  type: Cluster
  cluster:
    topology: Ignored
    replication: Consistency
```

The replication mode and its parameters are configured separately. The available modes and usage examples are described in Configure replication.

Standalone mode

`Standalone` mode runs PostgreSQL as a single instance without replication. Unlike `Cluster` mode, it doesn’t use topology or replication parameters.

To use this mode, specify:

```
spec:
  type: Standalone
```

After Postgres is created, a single PostgreSQL instance starts. Check the created PostgreSQL instances:

```
d8 k get pods -n my-postgres \
  -l managed-services.deckhouse.io/managed-service-name=app-postgres \
  -o wide
```

Example output:

```
NAME                     STATUS    NODE
d8ms-pg-app-postgres-1   Running   worker-1
```

Check the Services created for connecting to PostgreSQL:

```
d8 k get svc -n my-postgres | grep app-postgres
```

Example output:

```
d8ms-pg-app-postgres-r    ClusterIP   10.223.234.52    <none>   5432/TCP
d8ms-pg-app-postgres-ro   ClusterIP   10.223.70.248    <none>   5432/TCP
d8ms-pg-app-postgres-rw   ClusterIP   10.223.120.250   <none>   5432/TCP
```

Check which instances the Services point to, via endpoints:

```
d8 k get endpoints -n my-postgres | grep app-postgres
```

Example output:

```
d8ms-pg-app-postgres-r    10.112.2.31:5432   42h
d8ms-pg-app-postgres-ro   <none>             42h
d8ms-pg-app-postgres-rw   10.112.2.31:5432   42h
```

The Services with the `-r` and `-rw` suffixes route connections to the only instance. The Service with the `-ro` suffix is also created but has no endpoint, since `Standalone` mode has no replicas.

Configure topology and replication mode

In `Cluster` mode, you can manage the fault tolerance of PostgreSQL instances by setting their placement across nodes and availability zones in the `spec.cluster.topology` parameter.

Configure topology

The following values are supported:

- `Ignored`: placement follows the standard Kubernetes scheduling rules, spreading instances across different nodes;
- `Zonal`: instances are placed within one of the allowed zones;
- `TransZonal`: instances are placed across different availability zones.

The available topology values and zones are determined by the selected PostgresClass. For `Zonal` and `TransZonal`, the cluster infrastructure must provide the corresponding availability zones. For more information, see Manage fault tolerance across availability zones.

Placement without zone selection

With `topology: Ignored`, instance placement is managed by the Kubernetes scheduler. This mode spreads instances across different nodes without any additional configuration from the user. The main example uses this mode:

```
spec:
  cluster:
    topology: Ignored
```

Check instance placement:

```
d8 k get pods -n my-postgres \
  -l managed-services.deckhouse.io/managed-service-name=app-postgres \
  -o wide
```

The instances should be on different nodes.

Example output:

```
NAME                        STATUS    NODE
d8ms-pg-app-postgres-1      Running   worker-1
d8ms-pg-app-postgres-2      Running   worker-2
```

Placement within a single zone

With `topology: Zonal`, one of the zones allowed by the selected PostgresClass is chosen for placing Postgres. All cluster instances are placed within this zone.

```
spec:
  cluster:
    topology: Zonal
```

To use `Zonal`, nodes must have the `topology.kubernetes.io/zone` label set to the corresponding zone value. The zone must be allowed by the selected PostgresClass.

For example, if two available nodes belong to the `default` zone:

Example output:

```
NAME       ZONE
worker-1   default
worker-2   default
```

The instances can be placed as follows.

Example output:

```
NAME                        STATUS    NODE
d8ms-pg-app-postgres-1      Running   worker-1
d8ms-pg-app-postgres-2      Running   worker-2
```

In this example, both instances are placed in the `default` zone.

Configure replication

In `Cluster` mode, replication transfers data from the primary PostgreSQL instance to the replicas. The replication mode is set in `spec.cluster.replication`.

The following modes are supported:

- `Availability`: a primary instance and one asynchronous replica;
- `Consistency`: a primary instance and one synchronous replica;
- `ConsistencyAndAvailability`: a primary instance, one synchronous replica, and one asynchronous replica.

Check replication mode

Replication status is checked through the `pg_stat_replication` view on the primary instance. Use this procedure for any `Cluster` mode, including after changing `spec.cluster.replication`.

First, identify the current primary instance:

```
PRIMARY="$(d8 k get clusters.cnpg.internal.managed.deckhouse.io d8ms-pg-app-postgres \
  -n my-postgres \
  -o jsonpath='{.status.targetPrimary}')"
```

Then run the query:

```
d8 k exec -n my-postgres "$PRIMARY" -- \
  psql -U postgres -d postgres -c \
  "SELECT application_name, state, sync_state FROM pg_stat_replication;"
```

The expected `sync_state` values depend on the mode:

Mode	Number of replicas	Expected `sync_state`
`Availability`	1	`async`
`Consistency`	1	`quorum`
`ConsistencyAndAvailability`	2	`quorum` and `async`

A `state: streaming` value means the replica is receiving changes from the primary instance. Instance roles can change while the mode is being switched, so always identify the primary instance again via `status.targetPrimary` rather than by Pod number.

Availability mode

`Availability` mode creates a primary PostgreSQL instance and one asynchronous replica.

To use this mode, specify:

```
spec:
  type: Cluster
  cluster:
    topology: Zonal
    replication: Availability
```

After Postgres is created, two PostgreSQL instances start:

```
d8 k get pods -n my-postgres \
  -l managed-services.deckhouse.io/managed-service-name=app-postgres \
  -o wide
```

Example output:

```
NAME                     READY   STATUS
d8ms-pg-app-postgres-1   1/1     Running
d8ms-pg-app-postgres-2   1/1     Running
```

Check the replication mode as described in Check replication mode. For `Availability`, expect one replica with `sync_state = async`:

```
     application_name      |   state   | sync_state
---------------------------+-----------+------------
 d8ms-pg-app-postgres-2    | streaming | async
(1 row)
```

You can check how Service traffic is distributed between the primary instance and the replica through EndpointSlice:

```
d8 k get endpointslice -n my-postgres | grep app-postgres
```

Example output:

```
d8ms-pg-app-postgres-r-v8kcv    IPv4   5432   10.112.2.249,10.112.2.155
d8ms-pg-app-postgres-ro-696bp   IPv4   5432   10.112.2.155
d8ms-pg-app-postgres-rw-8nx8s   IPv4   5432   10.112.2.249
```

The `-rw` Service routes connections to the primary instance, `-ro` to the replica, and `-r` to both instances.

Consistency mode

`Consistency` mode, used in the main example, creates a primary PostgreSQL instance and one synchronous replica:

```
spec:
  type: Cluster
  cluster:
    topology: Ignored
    replication: Consistency
```

After Postgres is created, two PostgreSQL instances start:

```
d8 k get pods -n my-postgres \
  -l managed-services.deckhouse.io/managed-service-name=app-postgres \
  -o wide
```

Example output:

```
NAME                       STATUS    NODE
d8ms-pg-app-postgres-1     Running   worker-1
d8ms-pg-app-postgres-2     Running   worker-2
```

Check the replication mode as described in Check replication mode. For `Consistency`, expect one replica with `sync_state = quorum`:

```
      application_name      |   state   | sync_state
----------------------------+-----------+------------
 d8ms-pg-app-postgres-2     | streaming | quorum
(1 row)
```

You can additionally verify synchronous replication by checking the actual data transfer. Identify the primary instance the same way as in Check replication mode:

```
PRIMARY="$(d8 k get clusters.cnpg.internal.managed.deckhouse.io d8ms-pg-app-postgres \
  -n my-postgres \
  -o jsonpath='{.status.targetPrimary}')"
```

Create a check table on the primary instance and insert a row:

```
d8 k exec -n my-postgres "$PRIMARY" -- \
  psql -U postgres -d postgres -c "
    CREATE TABLE consistency_check (
      id integer PRIMARY KEY,
      value text
    );
    INSERT INTO consistency_check VALUES (1, 'replicated');
  "
```

Identify the replica:

```
REPLICA="$(d8 k get pods -n my-postgres \
  -l managed-services.deckhouse.io/managed-service-name=app-postgres \
  -o jsonpath='{range .items[*]}{.metadata.name}{"\n"}{end}' | \
  grep -v "^${PRIMARY}$" | head -n1)"
```

Check that the row exists directly on the replica:

```
d8 k exec -n my-postgres "$REPLICA" -- \
  psql -U postgres -d postgres -c \
  "SELECT pg_is_in_recovery(), * FROM consistency_check;"
```

Example output:

```
 pg_is_in_recovery | id |   value
-------------------+----+------------
 t                 |  1 | replicated
(1 row)
```

A `pg is in recovery() = t` value shows that the query ran on the replica. The presence of the `replicated` row confirms that data was transferred from the primary instance to the synchronous replica.

ConsistencyAndAvailability mode

`ConsistencyAndAvailability` mode creates a primary PostgreSQL instance, one synchronous replica, and one asynchronous replica.

To use this mode, specify:

```
spec:
  type: Cluster
  cluster:
    topology: Zonal
    replication: ConsistencyAndAvailability
```

After Postgres is created, three PostgreSQL instances start:

```
d8 k get pods -n my-postgres \
  -l managed-services.deckhouse.io/managed-service-name=app-postgres \
  -o wide
```

Example output:

```
NAME                     READY   STATUS
d8ms-pg-app-postgres-1   1/1     Running
d8ms-pg-app-postgres-2   1/1     Running
d8ms-pg-app-postgres-3   1/1     Running
```

Check the replication mode as described in Check replication mode. For `ConsistencyAndAvailability`, expect two replicas — with `sync_state = quorum` and `sync_state = async`:

```
     application_name      |   state   | sync_state
---------------------------+-----------+------------
 d8ms-pg-app-postgres-2    | streaming | quorum
 d8ms-pg-app-postgres-3    | streaming | async
(2 rows)
```

Change replication mode of an existing cluster

You can change the replication mode of an existing Postgres in `Cluster` mode. To do this, change `spec.cluster.replication` in the `app-postgres` manifest and reapply it.

For example, to switch from `Availability` to `Consistency`, specify:

```
spec:
  cluster:
    replication: Consistency
```

Apply the changes:

```
d8 k apply -f postgres.yaml
```

During the update, `ScaledToLastValidConfiguration` may temporarily switch to `False`. After the update completes, the object's conditions (`status.conditions`) should return to `True`.

Check the new mode as described in Check replication mode. After switching to `Consistency`, the replica should work in synchronous mode:

```
d8ms-pg-app-postgres-1 | streaming | quorum
```

When switching back to `Availability`, the same check should show asynchronous replication:

```
d8ms-pg-app-postgres-2 | streaming | async
```

When switching to `ConsistencyAndAvailability`, the number of instances increases from two to three. Check the running instances:

```
d8 k get pods -n my-postgres \
  -l managed-services.deckhouse.io/managed-service-name=app-postgres \
  -o wide
```

After the update completes, the `pg_stat_replication` check should show a synchronous and an asynchronous replica:

```
     application_name      |   state   | sync_state
---------------------------+-----------+------------
 d8ms-pg-app-postgres-3    | streaming | async
 d8ms-pg-app-postgres-2    | streaming | quorum
(2 rows)
```

When switching back from `ConsistencyAndAvailability` to `Consistency`, the number of instances decreases from three to two, and the remaining replica works in `streaming | quorum` mode.

Create logical database and user

The main example creates the `app-rw` user and the `app` logical database:

```
spec:
  users:
    - name: app-rw
      role: rw
      storeCredsToSecret: app-postgres-rw

  databases:
    - name: app
```

After applying the manifest, wait for the users and databases to synchronize. The `USERSSYNCED` and `DATABASESSYNCED` conditions should be `True`:

```
d8 k get postgres app-postgres -n my-postgres -o wide
```

PostgreSQL user

User credentials are stored in the Secret specified in `storeCredsToSecret`.

Check the created Secret:

```
d8 k get secret app-postgres-rw -n my-postgres
```

Example output:

```
NAME              TYPE                       DATA
app-postgres-rw   kubernetes.io/basic-auth   4
```

The Secret contains the parameters required for connecting:

```
app-dsn
host
password
username
```

Get the connection parameters as follows:

```
echo "host: $(d8 k get secret app-postgres-rw -n my-postgres -o jsonpath='{.data.host}' |
base64 --decode)"
echo "username: $(d8 k get secret app-postgres-rw -n my-postgres -o
jsonpath='{.data.username}' | base64 --decode)"
echo "password: $(d8 k get secret app-postgres-rw -n my-postgres -o
jsonpath='{.data.password}' | base64 --decode)"
echo "app-dsn: $(d8 k get secret app-postgres-rw -n my-postgres -o jsonpath='{.data.app-dsn}'
| base64 --decode)"
```

Example output:

```
host: d8ms-pg-app-postgres-rw
username: app-rw
password: <PASSWORD>
app-dsn: postgresql://app-rw:<PASSWORD>@d8ms-pg-app-postgres-rw:5432/app
```

Where `<PASSWORD>` is the user's password from the Secret.

You can use the values from the Secret to configure your application or PostgreSQL client connection.

> **i** For application connections, use the Secret named in `storeCredsToSecret`. Don't use the internal Secrets named `d8ms-pg-...` for this purpose.

Declarative user management

The list of users in `spec.users` describes the required PostgreSQL state. When the list changes, the module synchronizes user roles and their related Secrets.

For example, remove the `app-rw` user from the manifest:

```
spec:
  users: []
```

Apply the updated manifest:

```
d8 k apply -f postgres.yaml
```

After synchronization completes, the `USERSSYNCED` condition should return to `True`:

```
d8 k get postgres app-postgres -n my-postgres -o wide
```

Check that the role is gone directly in PostgreSQL:

```
PRIMARY="$(d8 k get clusters.cnpg.internal.managed.deckhouse.io d8ms-pg-app-postgres \
  -n my-postgres \
  -o jsonpath='{.status.targetPrimary}')"

d8 k exec -n my-postgres "$PRIMARY" -- \
  psql -U postgres -d postgres -Atc \
  "SELECT rolname FROM pg_roles WHERE rolname = 'app-rw';"
```

The command shouldn't return a role name.

When running `d8 k exec`, a message about the selected container may appear:

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
```

The `app` logical database, which remains in `spec.databases`, isn't deleted when the user is removed. Check that it still exists:

```
d8 k exec -n my-postgres "$PRIMARY" -- \
  psql -U postgres -d postgres -Atc \
  "SELECT datname FROM pg_database WHERE datname = 'app';"
```

Expected output:

```
app
```

⚠ Removing a user from `spec.users` deletes the corresponding PostgreSQL role. Before removing a user, make sure it's no longer used by any applications.

To recreate the user, add it back to `spec.users`:

```
spec:
  users:
    - name: app-rw
      role: rw
      storeCredsToSecret: app-postgres-rw
```

After you reapply the manifest, the module recreates the PostgreSQL role and the Secret.

Logical database

The logical databases that the module must create and maintain are defined in `spec.databases`:

```
spec:
  databases:
    - name: app
```

After the database is created, the `DATABASESSYNCED` condition should be `True`.

⚠ Removing a database from `spec.databases` deletes the corresponding logical PostgreSQL database along with its data.

Connect to PostgreSQL

After Postgres is created, the module creates the `-r`, `-ro`, and `-rw` Services, used to connect to the PostgreSQL instances depending on their role:

- `-rw`: the primary instance;
- `-ro`: the replicas;
- `-r`: all available instances.

For clarity, the `app-postgres` example creates the following Services:

```
NAME                     TYPE        PORT(S)
d8ms-pg-app-postgres-r   ClusterIP   5432/TCP
d8ms-pg-app-postgres-ro  ClusterIP   5432/TCP
d8ms-pg-app-postgres-rw  ClusterIP   5432/TCP
```

By default, these Services have type `ClusterIP` and are available inside the cluster. The user's connection credentials and parameters are stored in the Secret specified in `storeCredsToSecret`.

You can connect to PostgreSQL both from applications inside the cluster and from an external network. For an external connection, the corresponding Service needs to be published separately.

Connect from within the cluster

For a connection from within the cluster, use the corresponding Service and the credentials from the user's Secret. In the main example, an application with write access connects to the `d8ms-pg-app-postgres-rw` Service as the `app-rw` user to the `app` database.

To test the connection, you don't need to install `psql` on a control plane node. You can use a temporary client Pod instead:

```
d8 k run postgres-client \
  -n my-postgres \
  --rm -it \
  --restart=Never \
  --image=postgres:17 \
  --env="PGPASSWORD=$(d8 k get secret app-postgres-rw -n my-postgres -o
jsonpath='{.data.password}' | base64 --decode)" \
  -- \
  psql \
    -h d8ms-pg-app-postgres-rw \
    -U app-rw \
    -d app \
    -c 'SELECT current_database(), session_user, current_user;'
```

Example output:

```
 current_database | session_user | current_user
------------------+--------------+--------------
 app              | app-rw       | rw
(1 row)
```

`session_user` shows the user the connection was made as (`app-rw`), and `current_user` shows the current access role (`rw`).

External connection to PostgreSQL

To work with PostgreSQL from an external network, you can use graphical clients and other applications that support connecting to PostgreSQL. For this, the Service through which PostgreSQL is published must be reachable from outside the cluster, and the client needs to specify the server address, port, database, and user credentials.

> **i** This section covers external connections using DBeaver as an example. You can use other PostgreSQL clients and applications the same way.

In this example, the connection is made to the previously created `app` database as the `app-rw` user in the `app-postgres` Postgres.

Publish PostgreSQL for external access

The publishing method depends on the cluster's network infrastructure. In this example, an external load balancer accepts connections on `<EXTERNAL_IP>:5432` and forwards them to `NodePort` `30001` on a cluster node. A separate Service routes this traffic to the primary PostgreSQL instance.

Don't modify the `d8ms-pg-app-postgres-rw` Service created by the module. Create a separate Service for external access:

```
apiVersion: v1
kind: Service
metadata:
  name: app-postgres-external
  namespace: my-postgres
spec:
  type: NodePort
  selector:
    cnpg.internal.managed.deckhouse.io/cluster: d8ms-pg-app-postgres
    cnpg.internal.managed.deckhouse.io/instanceRole: primary
  ports:
    - name: postgres
      protocol: TCP
      port: 5432
      targetPort: 5432
      nodePort: 30001
```

Apply the manifest:

```
d8 k apply -f app-postgres-external.yaml
```

Check the created Service:

```
d8 k get svc app-postgres-external -n my-postgres -o wide
```

Example output:

```
NAME                     TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE   SELECTOR
app-postgres-external    NodePort   10.223.111.45   <none>        5432:30001/TCP   4s
cnpg.internal.managed.deckhouse.io/cluster=d8ms-pg-app-
postgres,cnpg.internal.managed.deckhouse.io/instanceRole=primary
```

On the external load balancer, configure it to accept TCP connections on port `5432` and forward them to `NodePort` `30001` on the cluster node. In this example, the following chain is set up:

```
<EXTERNAL_IP>:5432
        |
external load balancer
        |
<NODE_IP>:30001
        |
NodePort
        |
primary PostgreSQL :5432
```

⚠ When publishing PostgreSQL to an external network, make sure access to the database port is restricted to trusted sources only. Use a firewall, allowed IP lists, a VPN, or other network infrastructure controls for this. It isn’t recommended to leave PostgreSQL accessible from the internet without restrictions.

Before testing the external connection, you can confirm that the created `NodePort` routes traffic to the primary PostgreSQL instance. To do this, get the user’s password:

```
PGPASSWORD="$(d8 k get secret app-postgres-rw -n my-postgres \
  -o jsonpath='{.data.password}' | base64 --decode)"
```

Start a temporary client Pod and connect via the node IP and `NodePort`:

```
d8 k run nodeport-test \
  -n my-postgres \
  --rm -i \
  --restart=Never \
  --image=postgres:17 \
  --env="PGPASSWORD=$PGPASSWORD" \
  -- \
  psql \
    -h <NODE_IP> \
    -p 30001 \
    -U app-rw \
    -d app \
    -c "SELECT current_database(), pg_is_in_recovery(), inet_server_addr();"
```

Example of a successful result:

```
 current_database | pg_is_in_recovery | inet_server_addr
------------------+-------------------+------------------
 app              | f                 | <POD_IP>
(1 row)
```

A `pg_is_in_recovery = f` value confirms that the connection is directed to the primary PostgreSQL instance.

Connect by IP address

You can connect directly by IP address, for example, to verify external access to PostgreSQL. For a permanent connection, it’s recommended to use a DNS name and TLS with server certificate verification, as described below.

Get the user’s password:

```
d8 k get secret app-postgres-rw -n my-postgres \
  -o jsonpath='{.data.password}' | base64 --decode; echo
```

In DBeaver, create a PostgreSQL connection and specify:

```
Host:     <EXTERNAL_IP>
Port:     5432
Database: app
Username: app-rw
Password: <PASSWORD>
```

Where `<PASSWORD>` is the password from the `app-postgres-rw` Secret.

After connecting, open the SQL Editor and run:

```
SELECT
   current_database(),
   session_user,
   inet_server_addr(),
   inet_server_port(),
   pg_is_in_recovery();
```

On the tested bench, the query returned:

```
 current_database | session_user | inet_server_addr | inet_server_port |
pg_is_in_recovery
------------------+--------------+------------------+------------------+------------
-------
 app              | app-rw       | <POD_IP>         |             5432 | f
```

A `pg_is_in_recovery = f` value confirms the connection to the primary PostgreSQL instance.

Connect with TLS verification

For a permanent external connection, it’s recommended to use TLS with server certificate verification.

In the example, `app-postgres` uses `K8s` mode, so PostgreSQL TLS certificates are issued automatically. The server certificate is signed by `cluster-selfsigned-ca`.

Save the automatically created server certificate to a file to determine the DNS name from the SAN:

```
d8 k get secret d8ms-pg-app-postgres-server-cert \
  -n my-postgres \
  -o jsonpath='{.data.tls\.crt}' | \
  base64 --decode > /tmp/app-postgres-server.crt
```

View the certificate details and its Subject Alternative Name (SAN):

```
openssl x509 \
  -in /tmp/app-postgres-server.crt \
  -noout \
  -subject -issuer -dates -ext subjectAltName
```

For `app-postgres`, the certificate contains the DNS name of the `-rw` Service:

```
d8ms-pg-app-postgres-postgres-rw.<EXTERNAL_IP>.sslip.io
```

With `verify-full` mode, the client checks that the server name matches the certificate, so use the DNS name from the SAN for the connection.

Get the CA certificate:

```
d8 k get secret selfsigned-ca-key-pair \
  -n d8-cert-manager \
  -o jsonpath='{.data.tls\.crt}' | \
  base64 --decode > /tmp/app-postgres-ca.crt
```

Verify the trust chain:

```
openssl verify \
  -CAfile /tmp/app-postgres-ca.crt \
  /tmp/app-postgres-server.crt
```

Example of a successful result:

```
/tmp/app-postgres-server.crt: OK
```

Transfer the CA certificate to the machine you’re connecting from. For example, if the cluster node is reachable via SSH, copy the certificate using `scp`:

```
scp user@<NODE_IP>:/tmp/app-postgres-ca.crt ~/app-postgres-ca.crt
```

In DBeaver, specify the connection parameters:

```
Host:     d8ms-pg-app-postgres-postgres-rw.<EXTERNAL_IP>.sslip.io
Port:     5432
Database: app
Username: app-rw
Password: <PASSWORD>
```

Where `<PASSWORD>` is the password from the `app-postgres-rw` Secret.

In the SSL settings, specify the CA certificate and `verify-full` mode:

```
CA Certificate: <CA_CERT_PATH>
SSL mode:       verify-full
```

Where `<CA_CERT_PATH>` is the path to the `app-postgres-ca.crt` file.

After connecting, run:

```
SELECT
   current_database(),
   session_user,
   inet_server_addr(),
   inet_server_port(),
   pg_is_in_recovery();
```

A successful query execution and a `pg_is_in_recovery = f` value confirm the connection to the primary PostgreSQL instance.

If you use `verify-full` and specify the `<EXTERNAL_IP>` address instead of the DNS name from the SAN, server name verification fails:

```
The hostname <EXTERNAL_IP> could not be verified by hostnameverifier PgjdbcHostnameVerifier.
```

When using `verify-full`, connect using a DNS name listed in the server certificate’s SAN.

Configure PostgreSQL parameters

PostgreSQL parameters can be changed via `spec.configuration`, if the selected PostgresClass allows overriding them.

Whether a parameter can be changed is determined by the PostgresClass settings:

- the parameter must be allowed to be overridden;
- the parameter value must comply with the configured validation rules.

If a parameter cannot be overridden or its value is outside the allowed limits, the API rejects the request.

Change an allowed parameter

In the main example, `app-postgres` uses the `default` PostgresClass, which allows changing the `maxConnections` parameter.

Change the value:

```
spec:
  configuration:
    maxConnections: 100
```

Apply the changes:

```
d8 k apply -f postgres.yaml
```

After the update completes, check the applied value directly in PostgreSQL:

```
PRIMARY="$(d8 k get clusters.cnpg.internal.managed.deckhouse.io d8ms-pg-app-postgres \
  -n my-postgres \
  -o jsonpath='{.status.targetPrimary}')"

d8 k exec -n my-postgres "$PRIMARY" -- \
  psql -U postgres -d postgres -c \
  "SHOW max_connections;"
```

Example output:

```
 max_connections
-----------------
 100
```

The parameter was changed because it’s allowed to be overridden by the selected PostgresClass.

Restrictions on parameter overrides

A PostgresClass can restrict the list of PostgreSQL parameters a user can change via `spec.configuration`.

For example, if the PostgresClass only allows overriding:

```
overridableConfiguration:
  - maxConnections
  - sharedBuffers
  - walKeepSize
```

an attempt to change a parameter not on this list is rejected.

Apply, for example:

```
spec:
  configuration:
    workMem: 16Mi
```

```
d8 k apply -f postgres.yaml
```

The API returns an error. Example output:

```
Configuration field workmem restricted to override by administrator in selected
postgresClass
```

In this case, the Postgres configuration doesn’t change, since the parameter is restricted by the selected PostgresClass.

Validate parameter values

In addition to specifying which parameters can be overridden, a PostgresClass can define validation rules for their values.

For example, if `maxConnections` has the following constraint:

```
configuration.maxConnections >= 100
```

the following change is rejected:

```
spec:
  configuration:
    maxConnections: 50
```

Apply the manifest:

```
d8 k apply -f postgres.yaml
```

The API returns an error. Example output:

```
Rule: configuration.maxConnections >= 100
```

The existing Postgres continues running with the last successfully applied configuration.

Configure TLS

The `spec.tls` parameter is used to manage PostgreSQL TLS certificates. The `CertManager`, `CustomCertificate`, and `K8s` modes are supported.

To use certificates issued by cert-manager, specify `CertManager` mode:

```
spec:
  tls:
    mode: CertManager
    certManager:
      clusterIssuerName: postgres-ca
```

The corresponding Issuer or ClusterIssuer must be prepared in advance. The administrative dependencies are described in Dependencies for specific features.

To use existing certificates from a Secret, select `CustomCertificate` mode:

```
spec:
  tls:
    mode: CustomCertificate
    customCertificate:
      serverCASecret: postgres-ca
      serverTLSSecret: postgres-tls
```

K8s mode

In `K8s` mode, PostgreSQL certificates are issued automatically:

```
spec:
  tls:
    mode: K8s
```

Once the object reaches the ready state, the module creates a Secret with the CA, server, and replication certificates.

Check TLS usage on the PostgreSQL side through the `pg_stat_ssl` view. Identify the primary instance the same way as in Check replication mode, and run the query:

```
PRIMARY="$(d8 k get clusters.cnpg.internal.managed.deckhouse.io d8ms-pg-app-postgres \
  -n my-postgres \
  -o jsonpath='{.status.targetPrimary}')"

d8 k exec -n my-postgres "$PRIMARY" -- \
  psql -U postgres -d postgres -c "
    SELECT
      a.pid,
      a.usename,
      a.client_addr,
      a.client_port,
      s.ssl,
      s.version,
      s.cipher
    FROM pg_stat_activity a
    LEFT JOIN pg_stat_ssl s USING (pid)
    WHERE a.usename = 'app-rw';
  "
```

For a TLS connection, the `ssl` field is `t`, and `version` and `cipher` show the TLS version and cipher used.

Configuring a client connection with server certificate verification is described in Connect with TLS verification.

Monitoring and alerts

For Postgres, you can enable monitoring with alerts, fully disable monitoring, or keep monitoring without alerts. The observability mode is set by the `spec.observability` parameter.

The main example enables monitoring and alerts:

```
spec:
  observability: Enabled
```

To fully disable monitoring, use:

```
spec:
  observability: Disabled
```

To keep monitoring but disable alerts, use:

```
spec:
  observability: EnabledWithoutAlerts
```

Check the applied mode by the Pod labels:

```
d8 k get pod -n my-postgres \
  -l managed-services.deckhouse.io/managed-service-name=app-postgres \
  -o json | \
  jq '.items[].metadata.labels | with_entries(select(.key |
test("observability|prometheus")))'
```

The `observability.deckhouse.io/servicemonitoring` label value depends on the selected mode:

```
Enabled               → enabled
Disabled              → disabled
EnabledWithoutAlerts  → no-alerts
```

With monitoring enabled, the output also contains the label:

```
"prometheus.deckhouse.io/custom-target": "managed-postgres"
```

Backup and restore

The PostgresSnapshot object is used to create snapshots. The StorageClass where Postgres is placed must use a CSI driver with snapshot support, and a corresponding VolumeSnapshotClass must be available in the cluster.

The main example uses the `replicated` StorageClass, for which the provider in this configuration doesn’t support creating snapshots. So for this demonstration, a separate `snapshot-local` StorageClass on `sds-local-volume` with LVM Thin is used.

Check the available snapshot classes:

```
d8 k get volumesnapshotclass
```

The following snapshot class is available for `snapshot-local`:

```
NAME                              DRIVER                              DELETIONPOLICY
sds-local-volume-snapshot-class   local.csi.storage.deckhouse.io   Delete
```

Create a snapshot

To test this, create a separate `snapshot-pg` Postgres in the `snapshot-local` StorageClass:

```
apiVersion: managed-services.deckhouse.io/v1alpha1
kind: Postgres
metadata:
  name: snapshot-pg
  namespace: my-postgres
spec:
  postgresClassName: default
  instance:
    cpu:
      cores: 1
      coreFraction: 50
    memory:
      size: 1Gi
    persistentVolumeClaim:
      size: 2Gi
      storageClassName: snapshot-local
  type: Standalone
  users:
    - name: snapshot-rw
      role: rw
      storeCredsToSecret: snapshot-pg-rw
  databases:
    - name: snapshotdb
```

To clearly verify data recovery at the moment the snapshot was created, use a check table: add the `BEFORE_SNAPSHOT` row before creating the snapshot, and `AFTER_SNAPSHOT` after.

Create the check table and insert the first row:

```
PGPASSWORD="$(d8 k get secret snapshot-pg-rw -n my-postgres \
  -o jsonpath='{.data.password}' | base64 --decode)"

d8 k run snapshot-client \
  -n my-postgres \
  --rm -i \
  --restart=Never \
  --image=postgres:17 \
  --env="PGPASSWORD=$PGPASSWORD" \
  -- \
  psql \
    -h d8ms-pg-snapshot-pg-rw \
    -U snapshot-rw \
    -d snapshotdb \
    -c "
       CREATE TABLE snapshot_check (
         id integer PRIMARY KEY,
         value text NOT NULL
       );
       INSERT INTO snapshot_check VALUES (1, 'BEFORE_SNAPSHOT');
       SELECT * FROM snapshot_check;
     "
```

Example output:

```
 id |      value
----+-----------------
  1 | BEFORE_SNAPSHOT
(1 row)
```

Create the PostgresSnapshot object:

```
apiVersion: managed-services.deckhouse.io/v1alpha1
kind: PostgresSnapshot
metadata:
  name: snapshot-pg-backup
  namespace: my-postgres
spec:
  postgresName: snapshot-pg
```

Apply the manifest:

```
d8 k apply -f snapshot-pg-backup.yaml
```

Check the snapshot status:

```
d8 k get postgressnapshot snapshot-pg-backup -n my-postgres \
  -o jsonpath='{.status.phase}{"\n"}'
```

After the snapshot is created successfully, the command returns:

```
completed
```

Check the created VolumeSnapshot:

```
d8 k get volumesnapshot -n my-postgres
```

Example output:

```
NAME                          READYTOUSE   SOURCEPVC               RESTORESIZE
SNAPSHOTCLASS
d8ms-pg-snapshot-pg-backup    true         d8ms-pg-snapshot-pg-1   2Gi           sds-
local-volume-snapshot-class
```

`READYTOUSE=true` confirms the snapshot is ready for recovery.

After the snapshot is created, add a second check row to the original database:

```
PGPASSWORD="$(d8 k get secret snapshot-pg-rw -n my-postgres \
  -o jsonpath='{.data.password}' | base64 --decode)"

d8 k run snapshot-client \
  -n my-postgres \
  --rm -i \
  --restart=Never \
  --image=postgres:17 \
  --env="PGPASSWORD=$PGPASSWORD" \
  -- \
  psql \
    -h d8ms-pg-snapshot-pg-rw \
    -U snapshot-rw \
    -d snapshotdb \
    -c "
       INSERT INTO snapshot_check VALUES (2, 'AFTER_SNAPSHOT');
       SELECT * FROM snapshot_check ORDER BY id;
     "
```

Example output:

```
 id |      value
----+-----------------
  1 | BEFORE_SNAPSHOT
  2 | AFTER_SNAPSHOT
(2 rows)
```

Restore from a PostgresSnapshot

To restore, create a new Postgres object and specify the created PostgresSnapshot in `spec.dataSource.objectRef`. You don't need to delete the original Postgres:

```
apiVersion: managed-services.deckhouse.io/v1alpha1
kind: Postgres
metadata:
  name: snapshot-pg-restored
  namespace: my-postgres
spec:
  dataSource:
    objectRef:
      kind: PostgresSnapshot
      name: snapshot-pg-backup
  type: Standalone
  instance:
    cpu:
      cores: 1
      coreFraction: 50
    memory:
      size: 1Gi
    persistentVolumeClaim:
      size: 2Gi
      storageClassName: snapshot-local
```

The `type` and `instance` fields must be specified explicitly — they aren't inherited from the original Postgres. After that, apply the manifest:

```
d8 k apply -f snapshot-pg-restored.yaml
```

Wait for the restored Postgres to become ready:

```
d8 k get postgres snapshot-pg-restored -n my-postgres -o wide -w
```

After the restored PostgreSQL starts, check the check table:

```
PGPASSWORD="$(d8 k get secret snapshot-pg-rw -n my-postgres \
  -o jsonpath='{.data.password}' | base64 --decode)"

d8 k run snapshot-restore-check \
  -n my-postgres \
  --rm -i \
  --restart=Never \
  --image=postgres:17 \
  --env="PGPASSWORD=$PGPASSWORD" \
  -- \
  psql \
    -h d8ms-pg-snapshot-pg-restored-rw \
    -U snapshot-rw \
    -d snapshotdb \
    -c "SELECT * FROM snapshot_check ORDER BY id;"
```

Example output:

```
 id |      value
----+-----------------
  1 | BEFORE_SNAPSHOT
(1 row)
```

The presence of only `BEFORE_SNAPSHOT` confirms that the database state was restored to the moment the snapshot was created.

Check status

The current state of Postgres is reflected in `status.conditions` of this object.

For a quick check, use:

```
d8 k get postgres app-postgres -n my-postgres -o wide
```

Main conditions (`status.conditions`):

Condition	What it shows
`ConfigurationValid`	The configuration passed the checks of the related PostgresClass
`LastValidConfigurationApplied`	The last valid configuration was applied
`ScaledToLastValidConfiguration`	The instances match the last valid configuration
`Available`	Postgres is available
`UsersSynced`	Users are synchronized
`DatabasesSynced`	Logical databases are synchronized

While resources or PostgreSQL parameters are being changed, some conditions may temporarily be `False`, while `Available` stays `True`.

To watch the status change:

```
d8 k get postgres app-postgres -n my-postgres -o wide -w
```

To see the details:

```
d8 k get postgres app-postgres -n my-postgres -o yaml
```

If Postgres doesn’t reach the ready state, see the Frequently Asked Questions section for diagnostics.

Additional resources

- Frequently Asked Questions
- Module `managed-postgres` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Application and Infrastructure Monitoring

This section is intended for project users of the Deckhouse Platform (DP).

DP includes a built-in monitoring system that provides convenient tools for observing the state of the infrastructure and applications.

By default, DP provides a predefined set of dashboards and alerts that help track key application health metrics. They are available through the “Monitoring” section of the Deckhouse web interface.

In addition, users can:

- Collect metrics from their own applications.
- Create custom dashboards to visualize required metrics.
- Configure custom alerts and override alert thresholds.

For more information about advanced monitoring capabilities, including dashboard and metrics management, refer to the documentation for the `observability` module.

Available by default

After installing DP, users have access to a basic set of tools for monitoring the cluster out of the box.

Dashboards

Dashboards display charts with data on CPU usage, memory consumption, disk activity, and network traffic, broken down by pods, nodes, or namespaces.

In the “Monitoring” → “Dashboards” section of the Deckhouse web interface, users can access the following dashboard groups:

- **Ingress Nginx**: Metrics related to the Ingress NGINX Controller operation, including virtual host status, response codes, and request latency.
- **Resource consumption (Main)**: Key cluster and application metrics, including resource utilization and status of pods, controllers, and namespaces.
- **Security**: Metrics related to cluster security.

Alerts

Alerts are automated notifications that report events requiring attention, such as metric threshold violations or component availability issues. For most alerts, trigger thresholds can be overridden if necessary.

By default, a DP cluster includes alerts for the following types of events:

- Certificate expiration, as well as errors during certificate issuing or renewal (`cert-manager`, `extended-monitoring`, and `ingress-nginx` modules).
- Container image pull failures, including authentication and authorization issues, invalid image names, missing images in a registry, or registry unavailability (`extended-monitoring` module).
- Workload execution errors for resources such as CronJob, Deployment, DaemonSet, and StatefulSet, including pod creation failures, unavailable replicas, and scheduling errors (`extended-monitoring` module).
- Metric exporter unavailability, preventing Prometheus from scraping metrics (`extended-monitoring` module).
- Disk space issues, including insufficient storage space or inode exhaustion on PVCs (`extended-monitoring` module).
- Ingress NGINX Controller errors, including a high rate of `5xx` responses from backends (`extended-monitoring` module).
- Network performance issues (`monitoring-ping` module).

Monitoring configuration

The following monitoring configuration options are available to DP users:

- **Monitoring user applications**: You can configure metrics collection from your application by following the instructions.
- **Creating custom dashboards**: You can add specialized dashboards using the GrafanaDashboardDefinition resource.
- **Configuring custom alerts**: You can define new alerting rules in one of two ways:
  - **Recommended**: Using the ClusterObservabilityMetricsRulesGroup and ObservabilityMetricsRulesGroup resources of the `observability` module.
  - **Outdated**: Using the CustomPrometheusRules resource of the `prometheus` module. To see an example, refer to the module documentation.

Additional resources

- Module `console` documentation
- Module `observability` documentation
- Module `prometheus` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Monitoring dashboards

In this section, you will learn how to work with dashboards to analyze the state of the Deckhouse Platform (DP) and the applications running in it.

Dashboards are collections of charts and tables that display application performance data. They provide information about CPU usage, memory consumption, disk and network activity, as well as the status of Pods, controllers, nodes, and namespaces.

Dashboard types

DP features preinstalled and custom dashboards, which can be created in several ways.

Dashboard type	Description
Preinstalled	Ready-to-use dashboards that are already installed in DP. Designed to monitor the state of running applications.
Custom dashboards created using the `observability` module	Custom dashboards created using the ObservabilityDashboard resource at the namespace level, with support for access control. This is the recommended way to work with dashboards.
Custom dashboards created using GrafanaDashboardDefinition	Custom dashboards created using the GrafanaDashboardDefinition resource at the cluster level. They require elevated privileges and do not support access control. This is a legacy approach that will be deprecated in future DP versions.

Preinstalled dashboards

DP users have access to a basic set of dashboards for monitoring the state of running applications. Dashboards are available in the Deckhouse web UI under “Monitoring” → “Dashboards”.

> Preinstalled dashboards are not available for editing.

Ingress Nginx

Dashboards for monitoring the operation of the Ingress controller. They include metrics reflecting the state of virtual hosts, HTTP response statistics, and request processing latency data.

Available dashboards:

- **Namespaces**: Aggregated Ingress resource metrics by namespace.
- **Namespace Detail**: Detailed information about Ingress resources in a selected namespace.
- **VHosts**: An overview of the state of virtual hosts.
- **VHost Detail**: Detailed information about a selected virtual host.

Resource consumption (Main)

A set of dashboards for analyzing application resource consumption. They are intended for load assessment, resource issue detection, and workload health analysis.

Available dashboards:

- **Namespaces**: Summary information for all namespaces.

- **Namespace**: Key resource usage metrics for a selected namespace.
- **Namespace / Controller**: Resource usage statistics for controllers within a selected namespace.
- **Namespace / Controller / Pod**: Detailed metrics for individual Pods.

Security

Dashboards that contain metrics related to the security of cluster components.

Available dashboards:

- **Admission policy engine** — metrics of the `admission-policy-engine` module, including information about policy checks and enforcement.

Custom dashboards

DP users can create custom dashboards in several ways, depending on the requirements for access control and the dashboard scope.

Using the observability module

The `observability` module extends the functionality of the `prometheus` module and the Deckhouse web UI by providing additional capabilities for flexible metric visualization and access control.

The module introduces new dashboard types, including namespace-scoped resources. This allows users to create and manage their own dashboards without requiring permissions for cluster-level objects. The module also simplifies dashboard editing by allowing users to configure dashboards directly in the web UI, without manually managing resources.

> **i** Before using these resources, make sure that the `observability` module is enabled in the cluster. If necessary, contact your DP administrator.

The following resources are available for creating dashboards:

- ObservabilityDashboard: Dashboards scoped to a namespace. They are displayed in the Deckhouse web UI under “Monitoring” → “Projects”.

  Example:

  ```
  apiVersion: observability.deckhouse.io/v1alpha1
  kind: ObservabilityDashboard
  metadata:
    name: example-dashboard
    namespace: my-namespace
    annotations:
      metadata.deckhouse.io/category: "Apps"
      metadata.deckhouse.io/title: "Example dashboard"
  spec:
    definition: |
      {
        "title": "Example dashboard",
        ...
      }
  ```

- ClusterObservabilityDashboard: Dashboards for visualizing cluster components. They are displayed in the Deckhouse web UI under “Monitoring” → “System”.

  Example:

```
apiVersion: observability.deckhouse.io/v1alpha1
kind: ClusterObservabilityDashboard
metadata:
  name: example-dashboard
  annotations:
    metadata.deckhouse.io/category: "Apps"
    metadata.deckhouse.io/title: "Example dashboard"
spec:
  definition: |
    {
      "title": "Example dashboard",
      ...
    }
```

- ClusterObservabilityPropagatedDashboard: Dashboards that extend the list of dashboards from the two categories above. Such dashboards are automatically added to the Deckhouse web UI and are displayed under “Monitoring” → “System and Monitoring” → “Projects”. They become available to users who have permissions for the corresponding namespace or system section.

  Example:

```
apiVersion: observability.deckhouse.io/v1alpha1
kind: ClusterObservabilityPropagatedDashboard
metadata:
  name: example-dashboard
  annotations:
    metadata.deckhouse.io/category: "Apps"
    metadata.deckhouse.io/title: "Example dashboard"
spec:
  definition: |
    {
      "title": "Example dashboard",
      ...
    }
```

Access control

Access to dashboards is configured using the mechanisms of the current role-based access control (RBAC) model.

Depending on the dashboard type (system or user), permissions are granted for the following resources:

- `observabilitydashboards.observability.deckhouse.io`: Namespace-scoped dashboards.
- `clusterobservabilitydashboards.observability.deckhouse.io`: System dashboards.
- `clusterobservabilitypropagateddashboards.observability.deckhouse.io`: Dashboards propagated to all users.

The following permissions are required to perform operations on dashboards:

- reading: `get`
- creating and editing: `create`, `update`, `patch`, `delete`

Access to metrics used in dashboards is also controlled via RBAC. Depending on the granted permissions, metric filtering is performed automatically.

The following access scenarios are supported:

- Namespace users can access only the metrics of their own namespace. RBAC access to the `metrics.observability.deckhouse.io` resource is checked.
- DP administrators have access to all system metrics:
  - Deckhouse metrics (`d8-*`)
  - Kubernetes metrics (`kube-*`)
  - metrics without the `namespace` label

  RBAC access to the `clustermetrics.observability.deckhouse.io` resource is used.

- Metrics from user namespaces can also be available to administrators if they have the appropriate permissions for the `metrics.observability.deckhouse.io` resource.

Example configuration of ClusterRole and RoleBinding resources for read and edit access to metrics and dashboards:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: observability-editor
rules:
  - apiGroups: ["observability.deckhouse.io"]
    resources: ["metrics", "observabilitydashboards"]
    verbs: ["get", "list", "watch"]
  - apiGroups: ["observability.deckhouse.io"]
    resources: ["observabilitydashboards"]
    verbs: ["create", "update", "delete"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-observability-editor
  namespace: my-namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: observability-editor
subjects:
  - kind: User
    name: user@example.com
    apiGroup: rbac.authorization.k8s.io
```

Converting dashboards from GrafanaDashboardDefinition

To migrate dashboards created using the legacy GrafanaDashboardDefinition resource to one of the formats supported by the `observability` module, edit each dashboard manifest manually. Pay attention to the following key differences:

GrafanaDashboardDefinition format	`observability` module format
The Grafana folder is specified in the `spec.folder` field.	The folder is specified using the `observability.deckhouse.io/category` annotation.
The dashboard title is specified in the `title` field of the JSON manifest.	The title is specified using the `observability.deckhouse.io/title` annotation. If the annotation is missing, the `title` field from the JSON manifest is used.

Conversion example:

- Old format:

  ```
  apiVersion: deckhouse.io/v1
  kind: GrafanaDashboardDefinition
  metadata:
    name: example-dashboard
  spec:
    folder: "Apps"
    json: '{
      "title": "Example Dashboard",
      ...
    }'
  ```

- New format:

```
apiVersion: observability.deckhouse.io/v1alpha1
kind: ObservabilityDashboard
metadata:
  name: example-dashboard
  namespace: my-namespace
  annotations:
    metadata.deckhouse.io/category: "Apps"
    metadata.deckhouse.io/title: "Example Dashboard"
spec:
  definition: |
    {
      "title": "Example Dashboard",
      ...
    }
```

Using GrafanaDashboardDefinition

> **i** This is a legacy approach and is not recommended for new dashboards. Support for this method will be removed in future DP versions.

To add a dashboard directly to Grafana, use the GrafanaDashboardDefinition resource.

Example:

```
apiVersion: deckhouse.io/v1
kind: GrafanaDashboardDefinition
metadata:
  name: my-dashboard
spec:
  folder: My folder # The folder in Grafana where the dashboard will be displayed.
  definition: |
    {
      "annotations": {
        "list": [
          {
            "builtIn": 1,
            "datasource": "-- Grafana --",
            "enable": true,
            "hide": true,
            "iconColor": "rgba(0, 211, 255, 1)",
            "limit": 100,
...
```

When using this method, consider the following limitations:

- Dashboards added via GrafanaDashboardDefinition cannot be modified through the Grafana UI.
- Alerts configured in the Dashboard panel do not work with datasource templates — such dashboards are considered invalid and are not imported. Starting with Grafana 9.0, legacy alerting has been deprecated and replaced with Grafana Alerting. Therefore, using legacy alerting (dashboard panel alerts) in dashboards is not recommended.
- If the dashboard does not appear in Grafana after applying the resource, the dashboard JSON file may contain an error. To view the logs of the component responsible for dashboard provisioning, use the following command:

```
d8 k logs -n d8-monitoring deployments/grafana-v10 dashboard-provisioner
```

Additional resources

- Module `admission-policy-engine` documentation
- Module `console` documentation

- Module `observability` documentation
- Module `prometheus` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Configuring Application Monitoring

Deckhouse Platform (DP) supports four ways to connect an application to the monitoring system:

Connection method	Description
Via labels and annotations	The simplest and fastest method, requiring only metadata to be added to a Service or Pod. Allows you to configure basic monitoring parameters.
Using PodMonitor or ServiceMonitor	An advanced monitoring configuration method for cases where Prometheus relabeling rules are required. Provides flexible control over metrics collection and label processing and lets you add custom labels into metrics. This approach is suitable for complex monitoring scenarios but requires a deeper understanding of Prometheus and its scraping mechanism.
Using ScrapeConfig	A monitoring configuration method that is as close as possible to the native Prometheus configuration structure. Provides full control over scrape settings, including relabeling, and allows collecting metrics both from Kubernetes and from targets located outside the cluster.
Availability monitoring using blackbox-exporter	A method for monitoring endpoint availability using probes. It is integrated using blackbox-exporter, which must be installed in the cluster separately.

Configuring metrics collection via labels and annotations

> This section describes a basic application integration scenario. For advanced configuration options, refer to additional annotations.

1. Make sure the `monitoring-custom` module is enabled. If necessary, contact your DP administrator.
2. Ensure that the application exposing metrics does so in the Prometheus format.
3. Add the `prometheus.deckhouse.io/custom-target` label to the Service or Pod that should be connected to monitoring. The label value defines the name in the Prometheus targets list.

   Example:

   ```
   labels:
     prometheus.deckhouse.io/custom-target: my-app
   ```

   It is recommended to use the application name as the value of the `prometheus.deckhouse.io/custom-target` label, which allows it to be uniquely identified in the cluster.

   The label format must comply with Kubernetes requirements: no more than 63 characters, which can include alphanumeric characters (`[a-z0-9A-Z]`), as well as hyphens (`-`), underscores (`_`), dots (`.`).

   If the application is deployed in the cluster more than once (staging, testing, etc.) or even

deployed several times in one namespace, one common name is sufficient, since all metrics will have `namespace`, `pod` labels anyway, and if access is through Service, the `service` label. This is the name that uniquely identifies the application in the cluster, not its single installation.

4. Specify the name `http-metrics` and `https-metrics` for the port from which metrics need to be collected for HTTP or HTTPS connection respectively.

   If this is not possible (for example, the port is already defined and named differently), use the following annotations:

   - `prometheus.deckhouse.io/port: port_number`: To specify the port.
   - `prometheus.deckhouse.io/tls: "true"`: If metrics collection will be over HTTPS.

   > **i** When specifying an annotation on a Service, you must use `targetPort` as the port value. That is, the port that is open and listened to by the application, not the Service port.

   - Example 1:

     ```
     ports:
     - name: https-metrics
       containerPort: 443
     ```

   - Example 2:

     ```
     annotations:
       prometheus.deckhouse.io/port: "443"
       prometheus.deckhouse.io/tls: "true"  # If metrics are provided over HTTP,
     do not specify this annotation.
     ```

5. When using service mesh Istio in STRICT mTLS mode, specify the following annotation for metrics collection on Service or Pod: `prometheus.deckhouse.io/istio-mtls: "true"`. It is important that application metrics should be exported over HTTP protocol without TLS.

   Example:

   ```
   annotations:
     prometheus.deckhouse.io/istio-mtls: "true"
   ```

Example of configuring metrics collection from a Service

Below is an example of setting up metrics collection from a Service:

```
apiVersion: v1
kind: Service
metadata:
  name: my-app
  namespace: my-namespace
  labels:
    prometheus.deckhouse.io/custom-target: my-app
  annotations:
    prometheus.deckhouse.io/port: "8061"                  # By default, the service
port with the name http-metrics or https-metrics will be used.
    prometheus.deckhouse.io/path: "/my_app/metrics"         # Set to /metrics by
default.
    prometheus.deckhouse.io/query-param-format: "prometheus"  # Set to '' by default.
    prometheus.deckhouse.io/allow-unready-pod: "true"       # By default, Pods NOT in
Ready state are ignored.
    prometheus.deckhouse.io/sample-limit: "5000"           # By default, no more than
5000 metrics are accepted from a single Pod.
spec:
  ports:
  - name: my-app
    port: 8060
  - name: http-metrics
    port: 8061
    targetPort: 8061
  selector:
    app: my-app
```

Example of configuring metrics collection from a Deployment

Below is an example of setting up metrics collection from a Deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
        prometheus.deckhouse.io/custom-target: my-app
      annotations:
        prometheus.deckhouse.io/sample-limit: "5000"  # By default, no more than 5000
metrics are accepted from a single Pod.
    spec:
      containers:
      - name: my-app
        image: my-app:1.7.9
        ports:
        - name: https-metrics
          containerPort: 443
```

Additional annotations for advanced configuration

For more precise application monitoring configuration, you can specify additional annotations for the Pod or Service for which monitoring is configured:

- `prometheus.deckhouse.io/path`: Path for metrics collection (default: `/metrics`).
- `prometheus.deckhouse.io/query-param-$name`: GET parameters that will be converted to a map of the form `$name=$value` (default: `''`). You can specify several such annotations. For example, `prometheus.deckhouse.io/query-param-foo=bar` and `prometheus.deckhouse.io/query-param-bar=zxc` will be converted to a request like `http://...?foo=bar&bar=zxc`.
- `prometheus.deckhouse.io/allow-unready-pod`: Allows metrics collection from pods in any state (by default, metrics are collected only from pods in `Ready` state). This option is useful in rare cases. For example, if your application starts very slowly (data is loaded into the database or caches are warmed up at startup), but useful metrics are already provided during startup that help monitor the application startup.
- `prometheus.deckhouse.io/sample-limit`: How many samples are allowed to be collected from a pod (`5000` by default). The default value protects against situations where the application suddenly starts providing too many metrics, which can disrupt the entire monitoring system. The annotation must be placed on the same resource where the `prometheus.deckhouse.io/custom-target` label is attached.

Configuring metrics collection using PodMonitor or ServiceMonitor resources

DP supports connecting applications using two functionally similar resources:

- PodMonitor (recommended): Discovers Pods directly and collects metrics from their containers. In most cases, this is the preferred option, as it works directly with Pods and does not depend on the presence of Services.
- ServiceMonitor: Discovers Services and collects metrics from the Pods behind them. Services are used as a source of metadata (such as labels), while the actual metrics scraping is performed against the Pod addresses included in the corresponding endpoints.

Both resources let you configure the scrape interval, paths, TLS settings, relabeling rules, and other parameters.

The difference between these resources lies in the source of the collected metrics. Use PodMonitor if you need to scrape metrics directly from Pods, and ServiceMonitor if your application exposes metrics via a Service.

To connect an application to the monitoring system using one of these resources, follow these steps:

1. Add the `prometheus.deckhouse.io/monitor-watcher-enabled: "true"` label to the namespace where the PodMonitor or ServiceMonitor will be created:

   ```
   apiVersion: v1
   kind: Namespace
   metadata:
     name: frontend
     labels:
       prometheus.deckhouse.io/monitor-watcher-enabled: "true"
   ```

2. Create a PodMonitor or ServiceMonitor resource, specifying the required `prometheus: main` label and the target endpoint parameters.

   The `spec.selector.matchLabels` field defines the objects to use for metrics collection. For PodMonitor, these are target pod labels and for ServiceMonitor, these are target service labels.

   PodMonitor example:

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: example-app
  namespace: frontend
  labels:
    prometheus: main
spec:
  selector:
    matchLabels:
      # Label of pods targeted for metrics collection.
      app: example-app
  podMetricsEndpoints:
    - port: metrics
      interval: 30s
      path: /metrics
```

ServiceMonitor example:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: example-app
  namespace: frontend
  labels:
    prometheus: main
spec:
  selector:
    matchLabels:
      # Label of a service that exposes application metrics.
      app: example-app
  endpoints:
    - port: web
      interval: 30s
      path: /metrics
```

If necessary, configure additional settings using the reference for available resource parameters: PodMonitor, ServiceMonitor.

Adding custom labels to metrics

If you need to add additional labels directly to the collected metrics, use metric relabeling rules (via the `metricRelabelings` parameter).

The example below shows a PodMonitor resource that adds the label `application` with the value `dotnet` to the collected metrics:

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: example-app
  namespace: frontend
  labels:
    prometheus: main
spec:
  selector:
    matchLabels:
      app: example-app
  podMetricsEndpoints:
    - port: metrics
      interval: 30s
      path: /metrics
      metricRelabelings:
        - targetLabel: application
          replacement: dotnet
          action: Replace
```

After applying this rule, an additional label will appear in the metrics:

```
http_requests_total{
  application="dotnet",
  code="404",
  container="example-app",
  instance="10.111.5.69:8080",
  namespace="frontend",
  pod="example-app-75476c469d-rqq4s"
}
```

Configuring metrics collection via scrape_configs using the ScrapeConfig resource

ScrapeConfig is a custom resource that lets you configure the `scrape_config` section of Prometheus configuration, providing full control over the metrics scraping process.

To connect an application to the monitoring system, follow these steps:

1. Add the `prometheus.deckhouse.io/scrape-configs-watcher-enabled: "true"` label to the namespace where the ScrapeConfig will be created:

   ```
   apiVersion: v1
   kind: Namespace
   metadata:
     name: frontend
     labels:
       prometheus.deckhouse.io/scrape-configs-watcher-enabled: "true"
   ```

2. Create a ScrapeConfig resource with the required `prometheus: main` label:

```
apiVersion: monitoring.coreos.com/v1alpha1
kind: ScrapeConfig
metadata:
  name: example-scrape-config
  namespace: frontend
  labels:
    prometheus: main
spec:
  honorLabels: true
  staticConfigs:
    - targets: ['example-app.frontend.svc.{{
.Values.global.discovery.clusterDomain }}.:8080']
  relabelings:
    - regex: endpoint|namespace|pod|service
      action: labeldrop
    - targetLabel: scrape_endpoint
      replacement: main
    - targetLabel: job
      replacement: kube-state-metrics
  metricsPath: '/metrics'
```

If necessary, configure additional settings using the reference for available resource parameters.

Configuring metrics collection using blackbox-exporter

DP supports availability metrics collection using blackbox-exporter, which is not included in DP and must be installed separately in the cluster. The Probe custom resource is used to define availability checks (probes) executed by Prometheus.

To connect a Probe to the DP monitoring system, follow these steps:

1. Add the `prometheus.deckhouse.io/probe-watcher-enabled: "true"` label to the namespace where the Probe will be created:

   ```
   apiVersion: v1
   kind: Namespace
   metadata:
     name: frontend
     labels:
       prometheus.deckhouse.io/probe-watcher-enabled: "true"
   ```

2. Create a Probe resource with the required `prometheus: main` label:

```
apiVersion: monitoring.coreos.com/v1
kind: Probe
metadata:
  labels:
    app: prometheus
    component: probes
    prometheus: main
  name: cdn-is-up
  namespace: frontend
spec:
  interval: 30s
  jobName: httpGet
  module: http_2xx
  prober:
    path: /probe
    scheme: http
    url: blackbox-exporter.blackbox-exporter.svc.cluster.local:9115
  targets:
    staticConfig:
      static:
      - https://example.com/status
```

If necessary, configure additional settings using the reference for available resource parameters.

Additional resources

- Module `monitoring-custom` documentation
- Module `operator-prometheus` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Application delivery with Argo CD

This section describes how to organize application delivery with Argo CD in Deckhouse Platform (DP).

Argo CD lets you describe applications declaratively and synchronize their state with the contents of a Git repository. The user specifies the manifest source, target cluster, namespace, and synchronization parameters, after which Argo CD deploys the application and keeps it in the target state.

In DP, Argo CD instances are deployed with the operator-argo module. Typical work with Argo CD includes:

- creating or using an existing AppProject object;
- preparing the target namespace for the application;
- creating an application through the Argo CD web interface;
- creating an application with the `argocd` CLI utility;
- creating an Application object that describes the application source and synchronization rules.

Prerequisites

Before you start, the following conditions must be met:

- the cluster administrator has enabled the operator-argo module;
- the administrator has deployed at least one Argo CD instance;
- the user has access to the required Argo CD instance and target namespaces.

If an Argo CD instance is not deployed yet, contact the administrator or follow the instructions in the Running Argo CD section.

AppProject projects

AppProject is an Argo CD custom resource that defines the logical boundaries of a project. With it, you can define:

- which Git repositories are allowed as application sources;
- which clusters and namespaces applications may be deployed to;
- which cluster-wide and namespaced resources are allowed;
- which roles and access policies apply within the project.

Every Application object must reference a project through the `spec.project` parameter.

By default, Argo CD includes the `default` project. You can use it for first experiments and simple scenarios. For production environments, create separate AppProject objects to restrict access to repositories, clusters, and namespaces.

⚠ By default, only the cluster administrator (a user with the `SuperAdmin` role) and the Argo CD instance administrator have permissions to create AppProject objects.

The manifest of the `default` AppProject object is shown below:

```
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: default
  namespace: argocd
spec:
  clusterResourceWhitelist:
  - group: '*'
    kind: '*'
  destinations:
  - namespace: '*'
    server: '*'
  sourceRepos:
  - '*'
```

Preparing a namespace

Before deploying an application, ensure that the target namespace has the `argocd.argoproj.io/managed-by` label that indicates which Argo CD instance manages resources in this namespace. If the label is missing, contact the administrator to configure it.

Example:

```
d8 k get ns demo --show-labels
NAME   STATUS   AGE     LABELS
demo   Active   5d18h   argocd.argoproj.io/managed-
by=argocd,kubernetes.io/metadata.name=demo
```

In this example, the `demo` namespace will be managed by the Argo CD instance deployed in the `argocd` namespace.

Deploying an application

You can create an application in Argo CD in several ways:

- interactively — through the Argo CD web interface;
- with the `argocd` CLI utility;
- declaratively — with an Application object.

i You can download the `argocd` CLI utility binary from the “Documentation” section of the Argo CD web interface. To get help for the CLI utility, run `argocd --help`.

Creating an application through the Argo CD web interface

You can create an application through the Argo CD web interface. To do this:

1. Open the web interface of the required Argo CD instance.
2. Go to the “Applications” section.
3. Click “New App”.
4. Specify the application name, project, repository, revision, path to manifests or chart, and the target cluster and namespace.
5. If needed, configure automatic synchronization and additional parameters.
6. Click “Create”.

The form fields in the web interface correspond to the main parameters of the Application object: project, application source, target cluster, namespace, and synchronization policy.

Creating an application with the argocd CLI utility

The `argocd` CLI utility lets you create and maintain applications from the command line. You can download the `argocd` binary from the “Documentation” section of the Argo CD web interface.

Before creating an application, authenticate with the following command:

```
argocd login <ARGOCD_DOMAIN>:443
```

> **i** When using SSO authentication, add the `--sso` flag to the login command.

Example of creating an application:

```
argocd app create guestbook \
  --repo https://github.com/argoproj/argocd-example-apps.git \
  --path guestbook \
  --dest-namespace demo \
  --dest-server https://kubernetes.default.svc \
  --directory-recurse \
  --sync-policy automated \
  --self-heal \
  --auto-prune
```

The command specifies the application source, target cluster, namespace, and synchronization policy.

To view the status of a deployed application, use the `argocd app get <APP_NAME>` command, for example:

```
argocd app get guestbook
Name:               argocd/guestbook
Project:            default
Server:             https://kubernetes.default.svc
Namespace:          demo
URL:                https://argocd.192.168.0.235.sslip.io/applications/guestbook
Source:
- Repo:             https://github.com/argoproj/argocd-example-apps.git
  Target:
  Path:             guestbook
SyncWindow:         Sync Allowed
Sync Policy:        Automated (Prune)
Sync Status:        Synced to  (8088f4c)
Health Status:      Healthy

GROUP  KIND        NAMESPACE  NAME          STATUS  HEALTH   HOOK  MESSAGE
apps   Deployment  demo       guestbook-ui  Synced  Healthy
deployment.apps/guestbook-ui unchanged
       Service     demo       guestbook-ui  Synced  Healthy
```

Creating an application with an Application object

⚠ Before creating an Application object, ensure that you have the required permissions in the namespace where the Argo CD instance runs (`argocd` in the examples below). By default, only the cluster administrator (a user with the `SuperAdmin` role) has permissions to create Application objects.

An Application object is used to describe an application. It specifies:

- the Argo CD project (`spec.project`);
- the manifest or chart source (`spec.source`);
- the target cluster and namespace (`spec.destination`);
- the synchronization policy (`spec.syncPolicy`).

Example of an Application object:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: demo
  namespace: argocd
spec:
  destination:
    namespace: demo
    server: https://kubernetes.default.svc
  project: default
  source:
    path: helm-guestbook
    repoURL: https://github.com/argoproj/argocd-example-apps
    targetRevision: HEAD
  syncPolicy:
    # Enable automatic synchronization.
    automated:
      # Delete outdated resources.
      prune: true
      # Enable self-healing if third-party changes occur.
      selfHeal: true
```

After you create the Application object, Argo CD starts tracking the application state and synchronizing it with the repository contents. As a result, resources related to the `demo` application should appear in the `demo` namespace:

```
d8 k -n demo get deployment,svc,pod
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/demo-helm-guestbook   1/1     1            1           15s

NAME                            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
service/demo-helm-guestbook   ClusterIP   10.222.177.84   <none>        80/TCP    15s

NAME                                      READY   STATUS    RESTARTS   AGE
pod/demo-helm-guestbook-66d5d69ccd-xfkb7   1/1     Running   0          15s
```

Additional resources

- Official Argo CD website
- Official Argo CD Operator website
- Module `operator-argo` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Marketplace

Marketplace lets you install ready-made applications into your namespace from the repositories connected by the cluster administrator. Each application is installed as an Application resource and can exist in multiple independent instances, for example, separate Redis instances for caching and for sessions in the same namespace.

> Marketplace and the Application resource are available in Deckhouse Platform starting from version 1.76.

To learn about connecting package repositories and managing repository scanning, refer to “Administration”.

For the Marketplace architecture, including core concepts, application development and configuration, templates, Nelm annotations, hooks, and the application lifecycle, refer to “Architecture”.

Prerequisites

Before you can install an application, the cluster administrator must connect at least one package repository. Check that ApplicationPackageVersion objects are available:

```
d8 k get apv
```

If the output is empty, the repository has not been scanned yet or contains no packages. If the command fails with a `Forbidden` error, you don’t have permission to view package versions. In both cases, contact your administrator.

For a description of how to browse available versions, install, update, and delete applications, refer to “Installing and managing applications”.

For a description of how to diagnose and resolve problems with installed applications, refer to “Troubleshooting”.

Additional resources

- Global custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Installing and managing applications

This section describes how to browse available package versions, as well as how to install, update, and delete applications.

Browsing available package versions

To list all available package versions, use the following command (you can use `apv` as the short name for ApplicationPackageVersion):

```
d8 k get apv
```

Example output:

```
NAME                           PACKAGE    REPOSITORY    METADATALOADED   USEDBY
AGE
my-registry-redis-v7.2.0       redis      my-registry   True             1        2d
my-registry-redis-v7.3.0       redis      my-registry   True                      5h
my-registry-postgres-v15.0.0   postgres   my-registry   True             2        2d
```

The name of an ApplicationPackageVersion object is formatted as `<REPOSITORY_NAME>-<PACKAGE_NAME>-<PACKAGE_VERSION>`. To also see when the metadata was loaded and the loading error, if any, add `-o wide` to the command.

To filter by package name, use the following command (the example filters versions of the `redis` package):

```
d8 k get apv -l packages.deckhouse.io/package=redis
```

To see which package repositories provide a package, use the following command (you can use `ap` as the short name for ApplicationPackage):

```
d8 k get ap redis \
  -o jsonpath='{.status.availableRepositories}'
```

i If you see `MetadataLoaded=True` in the output, it means that the package's OpenAPI schema, description, and requirements were successfully loaded from the repository. A package version with `MetadataLoaded=False` cannot be installed until the metadata is loaded.

Packages available in a project

If the `multitenancy-manager` module is enabled, each project namespace has an AvailableClusterResource named `applicationpackages` that lists the packages available for installation in the project. Unlike ApplicationPackage objects, it can be read with the roles of the namespace.

To view the `applicationpackages` contents, use the following command:

```
d8 k get available applicationpackages -n <NAMESPACE> -o yaml
```

Abridged example output:

```
status:
  grantedResourceKind: ApplicationPackage
  availableCount: 2
  available:
    - name: postgres
      fields:
        availableRepositories:
          - my-registry
        releaseChannels:
          my-registry:
            stable: v15.0.0
    - name: redis
      fields:
        availableRepositories:
          - my-registry
        releaseChannels:
          my-registry:
            alpha: v7.3.0
            stable: v7.2.0
```

The `fields.availableRepositories` field lists the package repositories that provide the package, and `fields.releaseChannels` lists the package versions on the release channels of each repository.

The list of package versions (`d8 k get apv`) also includes the packages that are not available to the

project.

An Application of a package that is not available to the project is rejected with the following message:

```
[multitenancy] Application "<APPLICATION_NAME>" references "<PACKAGE_NAME>" which is not available to
project "<PROJECT_NAME>". Ask the cluster administrator to grant it.
```

The cluster administrator selects the packages available for installation in a project with a ClusterResourceGrantPolicy for the `applicationpackages` resource (for details, refer to “Configuring package access”). By default, every package is available to every project. An application installed before its package became unavailable to the project keeps working.

Installing an application

To install an application, create an Application object in the target namespace.

Example manifest for installing Redis from the `redis` package version `v7.2.0` with the `maxmemory` setting:

```
apiVersion: deckhouse.io/v1alpha1
kind: Application
metadata:
  name: redis-cache
  namespace: my-app
spec:
  packageName: redis
  packageVersion: "v7.2.0"
  # Name of the PackageRepository that provides the package.
  packageRepositoryName: my-registry
  settings:
    replicas: 3
    maxmemory: "256mb"
```

i The `spec.settings` section is validated against the OpenAPI schema defined in the package. If the schema rejects your settings, the Application is not created. The schema is published in the `status.packageSchemas.settingsSchema` field of the package’s ApplicationPackageVersion object.

Naming constraints

The Application name (`metadata.name`) must be **at most 24 characters**. An Application with a longer name is rejected on creation. The instance name is a part of the names of all application objects. The objects are named in the format `d8a-<INSTANCE_NAME>-<SUFFIX>`, for example, `d8a-redis-cache-server`, and must fit the Kubernetes name length limits (for details, refer to “Architecture”).

Checking application status

To get a brief status of an application, use the following command:

```
d8 k get applications -n <NAMESPACE> <APPLICATION_NAME>
```

Example output:

```
NAME          PACKAGE   VERSION   STATE   MESSAGE   AGE
redis-cache   redis     v7.2.0    Ready             5m
```

To also see the repository and the `Installed` and `Ready` conditions, add `-o wide`.

To get the full status including conditions, use the following command:

```
d8 k get applications -n <NAMESPACE> <APPLICATION_NAME> -o yaml
```

Conditions

The application state is described in detail through a set of conditions:

Condition	Meaning
`Installed`	The initial installation is complete: the package is downloaded, the hooks have run, the manifests are applied, and the Deployments and StatefulSets of the application are ready
`UpdateInstalled`	The new version is installed: it is downloaded, the hooks have run, and the manifests are applied. The condition appears after the package version is changed for the first time
`ConfigurationApplied`	The current configuration is applied: the settings, the hooks, and the manifests
`Scaled`	All Deployments and StatefulSets of the application have rolled out and have the desired number of ready replicas
`Managed`	DP manages the application. `False` in maintenance mode or if DP can't keep the application in the managed state, for example, because of hook or manifest errors
`Ready`	The application is ready to work. During an update, the condition can stay `True` while the previous version keeps working

Until the initial installation completes, `Installed` is the only condition reported. The other conditions appear once it becomes `True` and are removed again while it is `False`, for example, when a module the application depends on is disabled. While the application is being deleted, all conditions are `False` with the `Deleting` reason.

To quickly view all conditions, use the following command:

```
d8 k get applications -n <NAMESPACE> <APPLICATION_NAME> \
  -o jsonpath='{range .status.conditions[*]}{.type}: {.status} ({.reason}){"\n"}
{end}'
```

Example output:

```
Installed: True (Installed)
UpdateInstalled: False (Pending)
ConfigurationApplied: True (ConfigurationApplied)
Managed: True (Managed)
Scaled: True (Scaled)
Ready: True (Ready)
```

Brief state description

The `status.summary` field provides a brief description of the current application state. Check it first when you diagnose issues:

```
status:
  summary:
    state: Updating
    message: "Update is waiting for dependent modules to converge; previous version
is still serving"
    tip: "Wait — the previous version is still working. The update will continue
automatically once dependent modules converge."
```

For `status.summary` field descriptions, refer to the Application documentation.

Installing multiple instances

The same package can be installed multiple times in the same or different namespaces, each with a

separate name and settings. For example, two Redis instances can be created: one for caching and one for sessions:

```
# Caching instance.
apiVersion: deckhouse.io/v1alpha1
kind: Application
metadata:
  name: redis-cache
  namespace: team-alpha
spec:
  packageName: redis
  packageRepositoryName: my-registry
  packageVersion: "v7.2.0"
  settings:
    maxmemory: "512mb"
---
# Session storage instance.
apiVersion: deckhouse.io/v1alpha1
kind: Application
metadata:
  name: redis-sessions
  namespace: team-alpha
spec:
  packageName: redis
  packageRepositoryName: my-registry
  packageVersion: "v7.2.0"
  settings:
    maxmemory: "128mb"
```

The objects of each instance are named in the format `d8a-<INSTANCE_NAME>-<SUFFIX>`. For example, `d8a-redis-cache-server` and `d8a-redis-sessions-server`. This prevents any name conflicts.

Configuring workload resources

The `spec.resourceRequests` parameter overrides the number of replicas, the container resources, and the disk sizes that the package sets for the application workloads. Each entry addresses a workload by its kind (`kind`) and its object name (`name`) in the format `d8a-<INSTANCE_NAME>-<SUFFIX>`. Workloads, containers, and disks that aren’t listed keep the package settings. For details on the parameters, refer to the Application reference.

Autoscaling

If the package deploys a HorizontalPodAutoscaler or a VerticalPodAutoscaler for a workload, you can fix the values it controls or set a range for them:

- HorizontalPodAutoscaler. If `replicas` is greater than zero, Deckhouse Platform sets this value in the workload and as both HPA bounds. The number of replicas is fixed, and the `ApplicationHorizontalPodAutoscalerPinned` alert is triggered. With `replicas: 0`, the value applies only to the workload, and the HPA bounds remain unchanged.

  To let HPA choose the number of replicas within a range, set `minReplicas` and `maxReplicas`.

- VerticalPodAutoscaler. If a `cpu` or `memory` request is set in `resources.requests` and VPA controls that resource, Deckhouse Platform sets the request in the workload and as both bounds of the VPA policy. The request is fixed, and the `ApplicationVerticalPodAutoscalerPinned` alert is triggered. If a `cpu` or `memory` limit is set, VPA does not change the container’s limits; if VPA controls the corresponding resource, the `ApplicationVerticalPodAutoscalerPinned` alert is also triggered.

  To let VPA choose the container’s `cpu` or `memory` requests within a range, set `resources.min` and `resources.max`.

Example manifest where HorizontalPodAutoscaler keeps from 2 to 10 replicas of the `d8a-shop-backend` Deployment, VerticalPodAutoscaler chooses the requests of the `backend` container within the range, and the `proxy` container has fixed requests and limits:

```
apiVersion: deckhouse.io/v1alpha1
kind: Application
metadata:
  name: shop
  namespace: my-app
spec:
  packageName: shop
  packageVersion: "v1.4.0"
  packageRepositoryName: my-registry
  resourceRequests:
    - kind: Deployment
      name: d8a-shop-backend
      minReplicas: 2
      maxReplicas: 10
      containers:
        - name: backend
          resources:
            min:
              cpu: 250m
              memory: 256Mi
            max:
              cpu: "2"
              memory: 2Gi
        - name: proxy
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              memory: 256Mi
```

A range always has both bounds and takes the place of the fixed values it bounds. An Application is rejected in the following cases:

- `minReplicas` is set without `maxReplicas`, or the other way round.
- `min` and `max` of a container are not set together for the same resources.
- `min` and `max` are set, but neither contains `cpu` or `memory`.
- `replicas` is set together with `minReplicas` and `maxReplicas`.
- `minReplicas` is greater than `maxReplicas`, or a replicas range is set for a workload other than Deployment, StatefulSet, or ReplicaSet.
- A container with `min` and `max` has `cpu` or `memory` requests or limits (`ephemeral-storage` is allowed, since VerticalPodAutoscaler doesn’t manage it).
- `min` of a resource is greater than its `max`.

The error names the workload and the container and explains how to fix the problem. For example:

```
Application 'my-app/shop' has invalid spec.resourceRequests: Deployment 'd8a-shop-backend', container 'backend': set both min and max: a range needs both bounds, but only min is set
```

To fix a resource of a container with a range at one value, specify the same value in `min` and `max`.

Specifics of ranges:

- Deckhouse Platform doesn’t change the number of replicas in the workload itself: within the range, HorizontalPodAutoscaler chooses it.
- Until VerticalPodAutoscaler makes its first recommendation, the pods run with the requests from the package.

If no autoscaler of the application applies a range, for example, because the package doesn’t deploy a HorizontalPodAutoscaler for the workload or the `vertical-pod-autoscaler` module is disabled, Deckhouse Platform applies the range as fixed values: the workload runs `minReplicas` replicas, and the container gets `min` as the request and `max` as the limit of the resource. In this case, the `ApplicationHorizontalPodAutoscalerMissing` or `ApplicationVerticalPodAutoscalerMissing` alert is triggered. To fix it, make the autoscaler available or replace the range with fixed values.

If the package doesn’t allow changing the number of replicas of a workload, Deckhouse Platform doesn’t apply the change: the Application conditions get the `ScalingForbidden` reason, and the message names the workload. To fix it, change or remove the replicas or the range of the workload in `spec.resourceRequests`.

Updating an application

You can change the application version manually or configure automatic updates. In both cases, transitions between versions are limited by the version compatibility rules of the package.

Manual update

To update an application, change `spec.packageVersion` to the target version and apply the change:

```
d8 k patch applications -n <NAMESPACE> <APPLICATION_NAME> --type=merge -p '{"spec":
{"packageVersion":"v7.3.0"}}'
```

While the update is in progress, the `UpdateInstalled` condition is `False` with the `Pending` reason, and then with the `ApplyingManifests` reason while the manifests of the new version are applied. Once the update succeeds, the condition becomes `True`. The previous version keeps working until the update completes.

If the specified version doesn’t exist in the repository, the change is rejected, and the current version keeps running. If Deckhouse Platform (DP) fails to download the new version, `UpdateInstalled` becomes `False` with the `DownloadFailed` reason, and the current version keeps running.

If the version compatibility rules don’t allow moving from the current version to the specified one in a single step, the change is also rejected, with the `incompatible version` error. In this case, update the application step by step, through intermediate versions.

Automatic updates

Update settings are set only in the application itself, in the `spec.update` parameter. There are no common update settings for all applications in the cluster.

The update mode is set in the `spec.update.mode` parameter:

- `Manual`: Automatic updates are disabled, and the version changes only manually. This mode is used if `spec.update` isn’t set.
- `AutoPatch`: Deckhouse Platform automatically installs new patch versions within the current minor version. The application is moved to a new minor or major version manually.
- `Auto`: Deckhouse Platform automatically updates the application to every new version that the version compatibility rules allow.

Example of a manifest in which Redis is automatically updated to new patch versions on Tuesdays and Saturdays from 8:00 to 15:00 UTC:

```
apiVersion: deckhouse.io/v1alpha1
kind: Application
metadata:
  name: redis-cache
  namespace: my-app
spec:
  packageName: redis
  packageVersion: "v7.2.0"
  packageRepositoryName: my-registry
  update:
    mode: AutoPatch
    windows:
      - from: "8:00"
        to: "15:00"
        days:
          - Tue
          - Sat
```

Automatic updates work as follows:

- In every update mode, Deckhouse Platform builds the update path to the newest package version

in the cluster that the version compatibility rules allow reaching, and publishes it in `status.update.plan`. In the `Auto` and `AutoPatch` modes, the version that the automatic update installs next is published in `status.update.automaticVersion`.
- The update goes step by step: the next version is chosen only after the previous one is installed and the application is ready (the `Ready` condition).
- Automatic updates don't change `spec.packageVersion`. In the `Auto` and `AutoPatch` modes, the application runs the newer of `spec.packageVersion` and `status.update.automaticVersion`.

The update windows set in `spec.update.windows` apply in the `Auto` and `AutoPatch` modes: a new version, including one set in `spec.packageVersion` manually, is installed only during an update window. While the application waits for a window, the `UpdateInstalled` condition is `False` with the `WaitingWindow` reason. If no windows are set, updates are installed at any time. In the `Manual` mode, update windows don't apply. The first installation of an application doesn't wait for a window.

Update notifications can't be configured for an application: the `update.notification` settings of the `deckhouse` module are used. If they set a webhook, in the `Auto` and `AutoPatch` modes Deckhouse Platform sends it a notification about the pending application update (`subject: Application`) and installs the update no earlier than `minimalNotificationTime` after the notification. A patch version update is announced only with `releaseType: All`. If the notification can't be sent, Deckhouse Platform retries, and the update waits. Until the notification is sent and `minimalNotificationTime` has passed, the `UpdateInstalled` condition is `False` with the `WaitingNotification` reason. The sent notification is shown in `status.update.notified`.

Rolling back to an earlier version

To roll back an application, specify an earlier version in `spec.packageVersion`. By default, the version compatibility rules allow rolling back only to patch versions within the current minor version. In the `Auto` and `AutoPatch` modes, the application runs the newer of `spec.packageVersion` and `status.update.automaticVersion`, so switch the application to the `Manual` mode before rolling it back.

> ⚠ Deckhouse Platform doesn't apply any migration logic on rollback. If necessary, verify that the settings are compatible with the target version before applying the change.

Version compatibility rules

Each package version can set version compatibility rules, `versionCompatibilityRules`, in its `package.yaml` file. They define which versions an application can be updated or rolled back to in a single step. The rules apply both to manual changes of `spec.packageVersion` and to automatic updates.

The limits are set separately for updates (`upgrade.step`) and rollbacks (`downgrade.step`):

- `patches`: How many patch versions the application can move within a minor version.
- `minor`: How many minor versions the application can move within a major version.
- `major`: How many major versions the application can move.

The `-1` value removes the limit, `0` forbids the transition, and a positive number sets the largest allowed difference between version numbers. Only the limit of the highest changed part of the version is checked: `major` when the major version changes, and `minor` when the minor version changes. The rules of the version being left apply, except for the `upgrade.from` constraint described below. The difference is counted by version numbers, not by the number of published versions: with `minor: 1`, the application can't move from `v1.2.3` to `v1.4.0` directly, even if there are no `v1.3.*` versions.

If a limit isn't set, its default value is used:

Limit	upgrade.step	downgrade.step
`patches`	`-1`: Any patch version within the minor version	`-1`: Any patch version within the minor version
`minor`	`1`: The next minor version	`0`: Rolling back the minor version is forbidden
`major`	`0`: Moving to another major version is forbidden	`0`: Rolling back the major version is forbidden

The effective rules of a version, with unset limits replaced by their default values, are published in `status.packageMetadata.versionCompatibilityRules` of the ApplicationPackageVersion object:

```
d8 k get apv my-registry-redis-v7.2.0 -o
jsonpath='{.status.packageMetadata.versionCompatibilityRules}'
```

Example of the rules in `package.yaml`:

```
versionCompatibilityRules:
  upgrade:
    step:
      # Update up to three minor versions ahead in a single step.
      minor: 3
      # Update to the next major version.
      major: 1
  downgrade:
    step:
      # Roll back up to five patch versions within a minor version.
      patches: 5
      # Roll back to the previous minor version.
      minor: 1
```

The other limits in the example (`upgrade.step.patches` and `downgrade.step.major`) take their default values.

upgrade.from constraint

The `step` limits are set in the version being left and don’t change after it’s published. Therefore, earlier versions can’t allow moving to a version released later, such as a new major version. A new version can specify which versions it can be updated from: its `package.yaml` sets the `upgrade.from` constraint in the SemVer format:

```
# package.yaml of version v2.0.0.
versionCompatibilityRules:
  upgrade:
    from: ">= 1.5.0"
```

If `upgrade.from` is set, it takes priority over the `upgrade.step` limits of the version being updated from: the application can be updated to this version only from versions that satisfy the constraint. The constraint allows transitions that the limits forbid and forbids transitions that they allow. In the example, the application is updated to `v2.0.0` from any version starting from `v1.5.0`, although moving to another major version is forbidden by default. From versions below `v1.5.0`, the application is updated through intermediate versions.

An invalid constraint forbids updating to this version. `upgrade.from` doesn’t affect rollbacks: a rollback follows the `downgrade.step` limits of the version being rolled back from.

Deleting an application

To delete an application, delete the Application object. For example:

```
d8 k delete applications -n <NAMESPACE> <APPLICATION_NAME>
```

When an Application is deleted, DP deletes the Kubernetes objects of the application, except for the objects that the package templates protect with the `helm.sh/resource-policy: keep` annotation. The objects that the application creates at runtime and the objects that stay after the Helm release is uninstalled (for example, Helm hooks), are deleted if the package declares them as orphan resources. The Application stays in the cluster until DP completes the deletion.

FAQ

Can an application be updated automatically?

Yes. Set the `Auto` or `AutoPatch` mode in the `spec.update.mode` parameter. For details, see Automatic updates.

Can an Application depend on another Application?

No. An Application can declare dependencies only on modules (via `requirements.modules` in `package.yaml`). This is an architectural constraint that ensures instance isolation.

Can I install the same application in different namespaces?

Yes. Create Application objects with the same `packageName` and `packageVersion` in different namespaces. Each of them is a fully independent instance.

Additional resources

- Global custom resources
- Module `deckhouse` parameters
- Module `multitenancy-manager` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Troubleshooting

In this section, you can learn how to solve typical issues when working with Marketplace applications.

Verifying Marketplace custom resources

If the `d8 k get applications` command returns an error saying that the resource type doesn’t exist, the Marketplace custom resources may not be installed. To check, run the following command:

```
d8 k get crd | grep -E 'application|package'
```

Expected output:

```
applicationpackages.deckhouse.io                    2026-02-10T14:54:41Z
applicationpackageversions.deckhouse.io             2026-02-10T14:54:41Z
applications.deckhouse.io                           2026-02-10T14:54:41Z
modulepackages.deckhouse.io                         2026-02-10T14:54:41Z
modulepackageversions.deckhouse.io                  2026-02-10T14:54:41Z
packagerepositories.deckhouse.io                    2026-02-10T14:54:41Z
packagerepositoryoperations.deckhouse.io            2026-02-10T14:54:41Z
```

If any resources are missing, contact your cluster administrator. Marketplace requires Deckhouse Platform (DP) version 1.76 or later.

Quick application diagnostics

The quickest way to understand why an application is not working is to check fields in the `status.summary` section of the Application resource. To run quick diagnostics, run the following command:

```
d8 k get applications -n <NAMESPACE> <APPLICATION_NAME> -o yaml | grep -A5 'summary:'
```

Example output:

```
summary:
  state: Updating
  message: "Update is waiting for dependent modules to converge; previous version is
still serving"
  tip: "Wait — the previous version is still working. The update will continue
automatically once dependent modules converge."
```

For `status.summary` field descriptions, refer to the Application documentation.

Reading individual conditions

To get a more detailed view of the application state, use the following command:

```
d8 k get applications -n <NAMESPACE> <APPLICATION_NAME> \
  -o jsonpath='{range .status.conditions[*]}{.type}: {.status} ({.reason}) -
{.message}{"\n"}{end}'
```

Example output for an update that is waiting:

```
Installed: True (Installed) -
UpdateInstalled: False (Pending) - waiting for processing
ConfigurationApplied: True (ConfigurationApplied) -
Managed: True (Managed) -
Scaled: True (Scaled) -
Ready: True (Ready) -
```

In this example, `Installed: True` means that the application is running the previously installed version, and `UpdateInstalled: False` with the `Pending` reason means that the update is waiting, for example, for the modules the application depends on. The installed version is shown in the `status.currentVersion.version` field.

Viewing DP logs

If the conditions don’t provide enough detail, ask the cluster administrator to check the DP log. The log is available only in the `d8-system` namespace:

```
d8 k -n d8-system logs svc/deckhouse-leader -c deckhouse | grep '<NAMESPACE>.
<APPLICATION_NAME>'
```

Viewing application pod logs

The objects of an application are named in the format `d8a-<APPLICATION_NAME>-<SUFFIX>`. To list the Deployments and StatefulSets of the application, run the following command:

```
d8 k get deployments,statefulsets -n <NAMESPACE> -l packages.deckhouse.io/instance=<APPLICATION_NAME>
```

The pods of these workloads have names with the same prefix. To list them, run:

```
d8 k get pods -n <NAMESPACE> | grep 'd8a-<APPLICATION_NAME>-'
```

To view logs for a specific pod, run:

```
d8 k logs -n <NAMESPACE> <POD_NAME>
```

To view logs for a Deployment of the application, run:

```
d8 k logs -n <NAMESPACE> deployments/d8a-<APPLICATION_NAME>-<SUFFIX>
```

Common condition reasons

The following table specifies the most common reasons for listed conditions.

Reason	Conditions	Meaning and what to check
`Pending`	`Installed`, `UpdateInstalled`	DP is waiting to install the version, for example, until the modules from `requirements.modules` of the package are enabled. Ask the administrator to check these modules

Reason	Conditions	Meaning and what to check
`RequirementsUnmet`	`Installed`	The cluster doesn’t meet the package requirements. The message names the unmet requirement
`DownloadFailed`	`Installed`, `UpdateInstalled`, `ConfigurationApplied`, `Managed`, `Ready`	DP can’t download the package version. Ask the administrator to check the PackageRepository and access to the repository
`LoadFromFilesystemFailed`	`Installed`, `UpdateInstalled`, `Ready`	DP can’t read the downloaded package. Contact the package developer
`SettingsInvalid`	`Installed`, `UpdateInstalled`, `ConfigurationApplied`	The settings didn’t pass validation. Check `spec.settings` against the settings schema of the ApplicationPackageVersion
`HookInitializationFailed`, `HookFailed`	`Installed`, `UpdateInstalled`, `ConfigurationApplied`, `Managed`, `Ready`	A hook of the package failed. The message contains the error
`ManifestsApplyFailed`	`Installed`, `UpdateInstalled`, `ConfigurationApplied`, `Managed`, `Ready`	DP can’t apply the manifests of the package. Check the message and the events in the namespace
`ApplyingManifests`, `SettingsChanged`	`UpdateInstalled`, `ConfigurationApplied`, `Managed`, `Ready`	Not an error. DP is applying the manifests or the changed settings
`Reconciling`	`Scaled`	A workload is rolling out. The message names the workload
`Degraded`	`Scaled`	A workload failed to roll out. Check the events and the pods of the workload with `d8 k describe`
`NoResourceReconciliation`	`Managed`	The application is in maintenance mode (the `spec.maintenance` field)
`Deleting`	All	The application is being deleted

Additional resources

- Global custom resources

> **i** **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Configuration of applications logs collection

Deckhouse Platform (DP) provides log collection and delivery from cluster nodes and pods to internal or external storage systems.

DP allows you to:

- Collect logs from all or specific pods and namespaces.
- Filter logs by labels, message content and other attributes.
- Send logs to multiple storage systems simultaneously (e.g., Loki and Elasticsearch).
- Enrich logs with Kubernetes metadata.
- Use log buffering to improve performance.

- Store logs in internal short-term storage based on Grafana Loki.

The general mechanism of log collection, delivery and filtering is described in detail in the “Architecture” section.

DP users can configure application log collection parameters using the PodLoggingConfig resource, which describes log sources within a specified namespace, including collection, filtering and parsing rules.

Configuring application log collection

1. Check with the DP administrator whether log collection and storage are configured in your cluster. Also ask them to provide you with the storage name that you will specify in the `clusterDestinationRefs` parameter.
2. Create a PodLoggingConfig resource in your namespace.

   In this example, logs are collected from all pods in the specified namespace and sent to short-term storage based on Grafana Loki:

   ```
   apiVersion: deckhouse.io/v1alpha1
   kind: PodLoggingConfig
   metadata:
     name: app-logs
     namespace: my-namespace
   spec:
     clusterDestinationRefs:
       - loki-storage
   ```

3. (**Optional**) Limit log collection by label.

   If you need to collect logs only from specific pods, for example, only from applications with the `app=backend` label, add the `labelSelector` parameter:

   ```
   apiVersion: deckhouse.io/v1alpha1
   kind: PodLoggingConfig
   metadata:
     name: app-logs
     namespace: my-namespace
   spec:
     clusterDestinationRefs:
       - loki-storage
     labelSelector:
       matchLabels:
         app: backend
   ```

4. (**Optional**) Configure log filtering.

   Using `labelFilter` and `logFilter` filters, you can set up filtering by metadata or message fields. For example, in this case, only logs that do not contain fields with the string `.*GET /status" 200$` will be sent to storage:

```
apiVersion: deckhouse.io/v1alpha1
kind: PodLoggingConfig
metadata:
  name: app-logs
  namespace: my-namespace
spec:
  clusterDestinationRefs:
    - loki-storage
  labelSelector:
    matchLabels:
      app: backend
  logFilter:
  - field: message
    operator: NotRegex
    values:
    - .*GET /status" 200$
```

5. Apply the created manifest using the following command:

```
d8 k apply -f pod-logging-config.yaml
```

Examples

Creating a source in a namespace and reading logs from all pods in it with direction to Loki

The following pipeline creates a source in the `test-whispers` namespace, reads logs from all pods in it and sends them to Loki:

```
apiVersion: deckhouse.io/v1alpha1
kind: PodLoggingConfig
metadata:
  name: whispers-logs
  namespace: tests-whispers
spec:
  clusterDestinationRefs:
    - loki-storage
---
apiVersion: deckhouse.io/v1alpha1
kind: ClusterLogDestination
metadata:
  name: loki-storage
spec:
  type: Loki
  loki:
    endpoint: http://loki.loki:3100
```

Reading pods in a specified namespace with a specific label

Example of configuring reading pods with the `app=booking` label in the `test-whispers` namespace:

```
apiVersion: deckhouse.io/v1alpha1
kind: PodLoggingConfig
metadata:
  name: whispers-logs
  namespace: tests-whispers
spec:
  labelSelector:
    matchLabels:
      app: booking
  clusterDestinationRefs:
    - loki-storage
---
apiVersion: deckhouse.io/v1alpha1
kind: ClusterLogDestination
metadata:
  name: loki-storage
spec:
  type: Loki
  loki:
    endpoint: http://loki.loki:3100
```

Additional resources

- Module `log-shipper` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Virtual machines

In a Deckhouse Platform (DP) project, virtual machines (VMs) run next to container workloads. A machine, its disks, and its images are described by Kubernetes resources, so you create, change, and delete them the same way as the other objects of the project.

Cluster-wide resources are prepared by an administrator: virtual machine classes, cluster images, and the storage and subnets available to the project. If the project lacks the class or image you need, contact the administrator.

To go all the way from creating an image to logging in to the guest system over SSH, and to clean up the created resources afterwards, refer to Quick start.

Images and disks

Machine data is stored on a disk, and the data source for that disk is an image. Which sources are supported and how to store an image so that disks are created from it faster is covered in Virtual machine images.

Working with disks is covered in three sections: Virtual machine disks for creating a disk and choosing its storage, Resizing and migrating virtual machine disks for expanding a disk and moving it to another storage, and Exporting virtual machine disk data for downloading disk contents outside the cluster.

Working with a virtual machine

The minimum set of parameters and the machine start are covered in Creating a virtual machine. Each group of parameters is then covered separately:

- CPU and memory of a virtual machine: The number of cores, the core fraction, and the memory size.
- Guest OS and bootloader of a virtual machine: The operating system type and the boot mode.
- Provisioning and the guest OS agent: cloud-init, Sysprep, and `qemu-guest-agent`.
- Placement of virtual machines across nodes: Node selectors and affinity rules.

- Attaching disks and images to a virtual machine: Devices in the machine spec and as a separate resource.

You can connect to a running machine over SSH, through the serial console, or over VNC, as covered in Connecting to a virtual machine. Which configuration changes apply right away and which require a restart is covered in Changing the configuration of a virtual machine. To move a running machine to another node, refer to Live migration of virtual machines; to collect the state of a machine and its related resources for a support request, refer to Collecting debug information about a virtual machine.

Snapshots, cloning, and pools

The state of a disk or of a whole machine is saved by a snapshot, which a new disk is later created from or the machine itself is restored from. This is covered in Snapshots of disks and virtual machines. To get a copy of a machine along with its disks without stopping it, refer to Cloning virtual machines; to keep a given number of identical machines and scale them, refer to Virtual machine pools.

Networking

A machine gets an address in the main cluster network automatically, but you can choose the address in advance and keep it reserved for the project, as covered in IP addresses of virtual machines. To open a machine application to other cluster workloads or to the outside, refer to Access to applications on a virtual machine. Besides the main network, a machine can be connected to project and cluster networks, which is covered in Additional network interfaces of virtual machines.

Device passthrough

If an administrator has prepared devices on the nodes, you can give them to a machine: GPU devices, USB devices, and PCI devices.

The full list of resources and their parameters is provided in the reference. DP configuration for virtualization is covered in the Virtualization section.

Additional resources

- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Quick start

This section walks through a minimal scenario: you create an Ubuntu 24.04 image, a disk from that image, and a virtual machine (VM), connect to it over the console, and then delete the created resources.

- Using the CLI
- Using the web interface

1. Create a VirtualImage from an external source:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualImage
metadata:
  name: ubuntu
spec:
  storage: ContainerRegistry
  dataSource:
    type: HTTP
    http:
      url: https://cloud-images.ubuntu.com/noble/current/noble-server-cloudimg-
amd64.img
EOF
```

2. Create a VirtualDisk from that image. Make sure the cluster has a default StorageClass, then apply the manifest:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualDisk
metadata:
  name: linux-disk
spec:
  dataSource:
    type: ObjectRef
    objectRef:
      kind: VirtualImage
      name: ubuntu
EOF
```

3. Create a VirtualMachine. The example uses a cloud-init script that creates the `cloud` user:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachine
metadata:
  name: linux-vm
spec:
  virtualMachineClassName: generic
  cpu:
    cores: 1
  memory:
    size: 1Gi
  provisioning:
    type: UserData
    userData: |
      #cloud-config
      ssh_pwauth: True
      users:
        - name: cloud
          passwd: <PASSWORD_HASH>
          shell: /bin/bash
          sudo: ALL=(ALL) NOPASSWD:ALL
          lock_passwd: False
  blockDeviceRefs:
    - kind: VirtualDisk
      name: linux-disk
EOF
```

Where `<PASSWORD_HASH>` is the hash of the user password, in quotes. Generate it with `mkpasswd --method=SHA-512 --rounds=4096`: the command prompts for the password and prints the ready

value. The script format is described in the cloud-init documentation.

4. Verify that the image and the disk are created and the VM is running. Resources don't become ready instantly, so wait for the expected values in the `PHASE` column:

   ```
   d8 k get vi,vd,vm
   ```

   Example output:

   ```
   NAME                                                  PHASE   CDROM   PROGRESS
   AGE
   virtualimage.virtualization.deckhouse.io/ubuntu       Ready   false   100%
   7h50m

   NAME                                                  PHASE   CAPACITY
   VIRTUALMACHINE   AGE
   virtualdisk.virtualization.deckhouse.io/linux-disk   Ready   4Gi        linux-
   vm         7h40m

   NAME                                                  PHASE     UPTIME   NODE
   IPADDRESS    AGE
   virtualmachine.virtualization.deckhouse.io/linux-vm  Running   7h30m
   virtlab-pt-2   10.66.10.2   7h46m
   ```

5. Connect to the VM over the console:

   ```
   d8 v console linux-vm
   ```

   Example output:

   ```
   Successfully connected to linux-vm console. The escape sequence is ^]

   linux-vm login: cloud
   Password:
   ...
   cloud@linux-vm:~$
   ```

   To exit the console, press `Ctrl+]`.

6. Delete the created resources:

   ```
   d8 k delete vm linux-vm
   d8 k delete vd linux-disk
   d8 k delete vi ubuntu
   ```

1. Create an image from an external source:

   1. Go to the **Projects** tab and select the project you need.
   2. Go to **Virtualization → Images**.
   3. Click **Create**.
   4. In the **Source** block, select **By link**.
   5. In the form that opens, enter `ubuntu` in the **Image name** field.
   6. In the **Storage** block, select `ContainerRegistry` in the **Storage type** field.
   7. In the **URL** field, paste `https://cloud-images.ubuntu.com/noble/current/noble-server-cloudimg-amd64.img`.
   8. Click **Create**.
   9. Check the image status on its page.

2. Create a disk from that image. You can skip this step and create the disk while creating the VM.

   1. Go to **Virtualization → Disks**.
   2. Click **Create**.
   3. In the form that opens, enter `linux-disk` in the **Disk name** field.
   4. In the **Source** field, select the `ubuntu` image from the drop-down list.
   5. If required, specify a larger size in the **Size** field, for example `5Gi`.
   6. In the **Storage class** field, select a StorageClass or keep the default one.
   7. Click **Create**.

8. Check the disk status on its page.

> **i** If the selected StorageClass uses the `WaitForFirstConsumer` mode, the disk waits for the VM that uses it. Until then, the disk shows the “CREATING 0%” status, but you can already select it when creating a VM.

3. Create a virtual machine:

   1. Go to **Virtualization → Virtual machines**.
   2. Click **Create**.
   3. In the form that opens, enter `linux-vm` in the **Name** field.
   4. In the **Platform** and **Resources** sections, keep the default settings.
   5. In the **Disks** section, click **Add**.

      If the disk is already created, select **Existing** in the **Disks / Images** window that opens and pick `linux-disk` from the list.

      If the disk isn’t created, select **Create from** in the same window and set the parameters:

      - In the **Name** field, enter `linux-disk`.
      - In the **Source** field, select the `ubuntu` image from the drop-down list. The list shows the resource type.
      - If required, specify a larger size in the **Size** field, for example `5Gi`.
      - In the **Storage** field, select a StorageClass or keep the default one.

      Click **Add**.
   6. Scroll down to the **Cloud-init** toggle and enable it.
   7. Paste the script into the field that appears, replacing `<PASSWORD HASH>` with the password hash in quotes, generated with `mkpasswd --method=SHA-512 --rounds=4096`:

      ```
      #cloud-config
      ssh_pwauth: True
      users:
        - name: cloud
          passwd: <PASSWORD_HASH>
          shell: /bin/bash
          sudo: ALL=(ALL) NOPASSWD:ALL
          lock_passwd: False
      ```
   8. Click **Create**.
   9. Check the VM status on its page.

4. Connect to the VM over the console:

   1. Go to **Virtualization → Virtual machines**.
   2. Select the VM from the list and click its name.
   3. In the form that opens, go to the **TTY** tab and log in to the console window.

5. Delete the created resources:

   1. Go to **Virtualization** and select the section you need, for example **Virtual machines**, **Disks**, or **Images**.
   2. In the resource row, click the ellipsis button and select **Delete**. In some lists, for example in the list of VM snapshots, deletion is a separate button.
   3. In the confirmation window, click **Delete** or cancel the action with **Don’t delete**.

> **i** **Important:** Deleting a resource is irreversible. A disk attached to a running virtual machine can’t be deleted, and the **Delete** item is inactive for it.

Additional resources

- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Virtual machine images

An image holds the contents of a disk that you use to create virtual machine disks. A VirtualImage is created in a project and available only in the project or namespace where it was created.

> To make the same image available to every project in the cluster, you need a cluster image, ClusterVirtualImage. Only an administrator can create it; the procedure is described in Cluster images of virtual machines.

A virtual machine accesses an attached image in read-only mode.

An image appears in a project in three steps:

1. You create a VirtualImage resource and specify a data source in it.
2. Deckhouse Platform (DP) downloads the image from that source to the storage, which is either DVCR or a PVC, depending on the selected type.
3. The downloaded image becomes available for creating disks.

Sources and storage options

The image source can be an HTTP server hosting the image file, a container image registry, or a file on your computer that you upload from the command line. You can also create an image from another image, from a virtual machine disk, or from a disk snapshot.

Image types, supported file formats, and compression algorithms are described in Image types and formats.

The downloaded image is stored in one of two ways, set by the `.spec.storage` parameter:

- `ContainerRegistry`: The default option, the image is stored in DVCR.
- `PersistentVolumeClaim`: The image is stored in a PVC. This option is preferable if the storage can clone PVCs quickly, because disks are created from such an image faster.

> ⚠ An image stored with `storage: PersistentVolumeClaim` can only be used to create disks in the same storage class.

The `PHASE` column in the `d8 k get vi` output shows the progress of image creation; for its values, see the `.status.phase` field. To follow the creation in real time, add the `-w` flag. If an image stays not ready for a long time, check the `.status.conditions` block and the `d8 k describe vi` output for the reason.

Until an image reaches the `Ready` phase, you can change its `.spec` block, and the download restarts after each change. For a ready image, the `.spec` block can no longer be changed. For all image parameters, see VirtualImage.

Creating an image from an HTTP server

The simplest way to create an image is to provide a link to a file hosted on an HTTP server.

- Using the CLI
- Using the web interface

1. Create a VirtualImage resource. In the example, the image is stored in DVCR:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualImage
metadata:
  name: ubuntu-24-04
spec:
  # Store the image in DVCR.
  storage: ContainerRegistry
  # Source for the image.
  dataSource:
    type: HTTP
    http:
      url: https://cloud-images.ubuntu.com/noble/current/noble-server-cloudimg-
amd64.img
EOF
```

2. Verify that the image is created:

   ```
   d8 k get virtualimage ubuntu-24-04

   # Short form of the command.
   d8 k get vi ubuntu-24-04
   ```

   Example output:

   ```
   NAME           PHASE   CDROM   PROGRESS   AGE
   ubuntu-24-04   Ready   false   100%       23h
   ```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization → Images**.
3. Click **Create**.
4. In the **Source** block, select **By link**.
5. In the form that opens, enter the image name in the **Image name** field.
6. In the **Storage** block, select `ContainerRegistry` in the **Storage type** field.
7. In the **URL** field, specify the link to the image.
8. Click **Create**.
9. Check the image status on its page.

Verifying the integrity of a downloaded image

The `checksum` block makes DP verify what it downloaded from the HTTP server. The image reaches the `Ready` phase only if the downloaded file matches every specified checksum, otherwise the resource ends up in the `Failed` phase:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualImage
metadata:
  name: ubuntu-24-04
spec:
  storage: ContainerRegistry
  dataSource:
    type: HTTP
    http:
      url: https://cloud-images.ubuntu.com/noble/current/noble-server-cloudimg-
amd64.img
      checksum:
        sha256: 78be890d71dde316c412da2ce8332ba47b9ce7a29d573801d2777e01aa20b9b5
EOF
```

Take the checksum from the mirror that publishes the image and put it in the field of the matching algorithm:

Field	Algorithm	Verification speed
`sha1`	SHA-1	~1.6 GB/s
`sha256`	SHA-256	~1.5 GB/s
`md5`	MD5	~700 MB/s
`sha512`	SHA-512	~570 MB/s
`streebog256`	GOST R 34.11-2012 (Streebog), 256-bit	~17 MB/s
`streebog512`	GOST R 34.11-2012 (Streebog), 512-bit	~17 MB/s

The speeds are an order of magnitude, not a promise. They were measured on an x86-64 CPU with the SHA instruction set, and a CPU without it computes SHA-1 and SHA-256 several times slower. What holds on any CPU is the distance between the rows. SHA-1 and SHA-256 are computed by dedicated instructions, MD5 and SHA-512 by hand-written assembly, and all four hash data faster than it arrives over the network, so their cost stays invisible against the download itself.

The Streebog algorithms have no hardware support anywhere and are about two orders of magnitude slower. For a 10 GiB image, that’s around ten minutes of hashing alone, and image creation becomes CPU-bound rather than network-bound. Use them only when a GOST checksum is actually required. Both lengths cost the same, because GOST R 34.11-2012 uses one compression function for 256 and 512 bits, and the shorter variant differs only in the initial value.

Checksums are computed in a single pass over the downloaded data, so specifying several checksums at once costs the sum of their times. Empty fields cost nothing.

The same block is available for the `Upload` source in the `dataSource.upload.checksum` parameter and works the same way. The uploaded data is verified against every specified checksum, and on a mismatch the resource stays in the `Failed` phase.

Storing an image in a PVC

To create disks from an image faster, store it in a PVC. DP can then clone the volume instead of unpacking the image again.

- Using the CLI
- Using the web interface

1. Create a VirtualImage resource with the `PersistentVolumeClaim` storage type:

   ```
   d8 k apply -f - <<EOF
   apiVersion: virtualization.deckhouse.io/v1alpha2
   kind: VirtualImage
   metadata:
     name: ubuntu-24-04-pvc
   spec:
     # Storage settings for the project image.
     storage: PersistentVolumeClaim
     persistentVolumeClaim:
       # Specify the name of your StorageClass.
       storageClassName: rv-thin-r2
     # Source for the image.
     dataSource:
       type: HTTP
       http:
         url: https://cloud-images.ubuntu.com/noble/current/noble-server-cloudimg-
   amd64.img
   EOF
   ```

   If the `.spec.persistentVolumeClaim.storageClassName` parameter isn’t set, DP uses the cluster-wide default StorageClass or the class set for images in the module settings.

2. Verify that the image is created:

```
d8 k get vi ubuntu-24-04-pvc
```

Example output:

```
NAME               PHASE   CDROM   PROGRESS   AGE
ubuntu-24-04-pvc   Ready   false   100%       23h
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Images**.
3. Click **Create**.
4. In the **Source** block, select **By link**.
5. In the form that opens, enter the image name in the **Image name** field.
6. In the **Storage** block, select `PersistentVolumeClaim` in the **Storage type** field.
7. In the **Storage class** field, select a StorageClass or keep the default one.
8. In the **URL** field, specify the link to the image.
9. Click **Create**.
10. Check the image status on its page.

Creating an image from a container image registry

DP can pull an image from an external container image registry, but the disk file must be located in the container image under the `/disk` path. The following steps show how to prepare such a container image and create a project image from it.

- Using the CLI
- Using the web interface

1. Download the image file to your local machine:

   ```
   curl -L https://cloud-images.ubuntu.com/minimal/releases/noble/release/ubuntu-24.04-minimal-cloudimg-amd64.img -o ubuntu2404.img
   ```

2. Create a `Dockerfile` with the following contents:

   ```
   FROM scratch
   COPY ubuntu2404.img /disk/ubuntu2404.img
   ```

3. Build the container image. The example uses the docker.com registry, which requires an account and a configured environment:

   ```
   docker build -t docker.io/<USERNAME>/ubuntu2404:latest
   ```

   Where `<USERNAME>` is the username you specified when registering in the registry.

4. Push the built container image to the registry:

   ```
   docker push docker.io/<USERNAME>/ubuntu2404:latest
   ```

5. Create a VirtualImage resource that points to the container image:

   ```
   d8 k apply -f - <<EOF
   apiVersion: virtualization.deckhouse.io/v1alpha2
   kind: VirtualImage
   metadata:
     name: ubuntu-2404
   spec:
     storage: ContainerRegistry
     dataSource:
       type: ContainerImage
       containerImage:
         image: docker.io/<USERNAME>/ubuntu2404:latest
   EOF
   ```

DP works only with registries that have TLS enabled. If the registry uses its own certificate authority,

provide the certificate chain in the `caBundle` parameter, and take the credentials for a private registry from the secret specified in the `imagePullSecret` parameter.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Images**.
3. Click **Create**.
4. In the **Source** block, select **From registry**.
5. In the form that opens, enter the image name in the **Image name** field.
6. In the **Storage** block, select `ContainerRegistry` in the **Storage type** field.
7. In the **Image in container registry** field, specify the path to the container image.
8. Click **Create**.
9. Check the image status on its page.

Uploading an image from the command line

If the image file is on your computer, upload it directly. DP creates a temporary upload endpoint for this and waits for the data.

- Using the CLI
- Using the web interface

1. Create a VirtualImage resource with the `Upload` source:

   ```
   d8 k apply -f - <<EOF
   apiVersion: virtualization.deckhouse.io/v1alpha2
   kind: VirtualImage
   metadata:
     name: some-image
   spec:
     # Storage settings for the project image.
     storage: ContainerRegistry
     # Image source settings.
     dataSource:
       type: Upload
   EOF
   ```

   The resource moves to the `WaitForUserUpload` phase and is ready to accept the file. Start the upload within 10 minutes, otherwise the resource moves to the `Failed` phase and you have to create it again.

2. Get the addresses that accept the file:

   ```
   d8 k get vi some-image -o jsonpath="{.status.imageUploadURLs}" | jq
   ```

   Example output:

   ```
   {
     "external": "https://virtualization.example.com/upload/<SECRET_URL>",
     "inCluster": "http://10.222.165.239/upload"
   }
   ```

   Use the `inCluster` address if you upload the file from one of the cluster nodes, and `external` in all other cases.

3. Upload the file to the selected address. The example first downloads the Cirros image and then sends it to the cluster:

   ```
   curl -L http://download.cirros-cloud.net/0.5.1/cirros-0.5.1-x86_64-disk.img -o cirros.img
   curl https://virtualization.example.com/upload/<SECRET_URL> --progress-bar -T cirros.img | cat
   ```

   Where `<SECRET_URL>` is the last part of the address from the previous step.

4. Verify that the image has reached the `Ready` phase:

   ```
   d8 k get vi some-image
   ```

Example output:

```
NAME         PHASE   CDROM   PROGRESS   AGE
some-image   Ready   false   100%       1m
```

You can also verify the uploaded file against a checksum. To do this, specify the `checksum` block in the data source. The checksums are computed over the bytes the client sends, and on a mismatch the resource stays in the `Failed` phase, so the upload has to be repeated on a recreated resource.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Images**.
3. Click **Create**, then select **Upload** in the **Source** block.
4. In the **Image name** field, enter the image name.
5. In the **Upload file** block, drag the file to the highlighted area or click **select on your computer**.
6. Select the file in the file manager that opens.
7. Click **Create**.
8. Wait until the image reaches the **Ready** state.

Creating an image from a disk

You can create an image from a disk if the disk isn't attached to any virtual machine, or if the machine it's attached to is powered off.

- Using the CLI
- Using the web interface

Create an image using the required disk as the source:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualImage
metadata:
  name: linux-vm-root
spec:
  storage: ContainerRegistry
  dataSource:
    type: ObjectRef
    objectRef:
      kind: VirtualDisk
      name: linux-vm-root
EOF
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Images**.
3. Click **Create**.
4. In the **Source** block, select **Create from**.
5. In the form that opens, enter `linux-vm-root` in the **Image name** field.
6. In the **Storage** block, select `ContainerRegistry` in the **Storage type** field.
7. In the **Source** field, select the disk you need from the drop-down list.
8. Click **Create**.
9. Check the image status on its page.

Creating an image from a disk snapshot

You can create an image from a disk snapshot if the snapshot is in the `Ready` phase.

- Using the CLI
- Using the web interface

Create an image using the disk snapshot as the source:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualImage
metadata:
  name: linux-vm-root
spec:
  storage: ContainerRegistry
  dataSource:
    type: ObjectRef
    objectRef:
      kind: VirtualDiskSnapshot
      name: linux-vm-root-snapshot
EOF
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Images**.
3. Click **Create**.
4. In the form that opens, enter the image name in the **Image name** field.
5. In the **Storage** block, select the image storage type in the **Storage type** field.
6. In the **Source** block, select **Create from**.
7. In the **Source** field, expand the list and select the snapshot you need in the **Disk snapshots** group.
8. Click **Create**.
9. Check the image status on its page.

Image properties are convenient to review in the web interface, in **Virtualization** → **Images**:

- The list shows the image name, status, size, format in the **Type** column, and visibility scope in the **Availability** column, and the filters narrow it down by status, image, and type.
- On the image page, the **Information** tab collects the creation parameters in the **Storage** and **Source** blocks, and the **State** block shows the average download speed, format, unpacked size, size in storage, creation time, the **CD-ROM** flag, and the image path in DVCR.
- The **Meta** and **YAML** tabs show labels with annotations and the full resource specification.

Additional resources

- Module `virtualization` custom resources
- Module `virtualization` parameters

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Virtual machine disks

A disk stores virtual machine (VM) data, including the operating system and application files. A disk is described by the VirtualDisk resource, and its specification consists of two blocks:

- `persistentVolumeClaim`: Storage parameters, that is, the StorageClass and the size.
- `dataSource`: The data source, which can be an image, another disk, or a snapshot.

Without the `dataSource` block, an empty disk is created, and then you have to specify at least the size in `persistentVolumeClaim`. If a source is set, you can omit the `persistentVolumeClaim` block, and Deckhouse Platform (DP) takes the size from the source and picks the storage class based on it too. When no class can be picked, DP uses the cluster-wide default StorageClass or the class set for disks in the module settings.

The `PHASE` column in the `d8 k get vd` output shows the progress of disk creation; for its values, see the `.status.phase` field. If a disk stays not ready for a long time, the `.status.conditions` block tells you the reason.

Until a disk reaches the `Ready` phase, you can change any field of the `.spec` block, and the creation restarts after the change. For a ready disk, only the size and the storage class remain editable, in the `.spec.persistentVolumeClaim.size` and `.spec.persistentVolumeClaim.storageClassName` parameters.

⚠ You can’t create a disk from an ISO image.

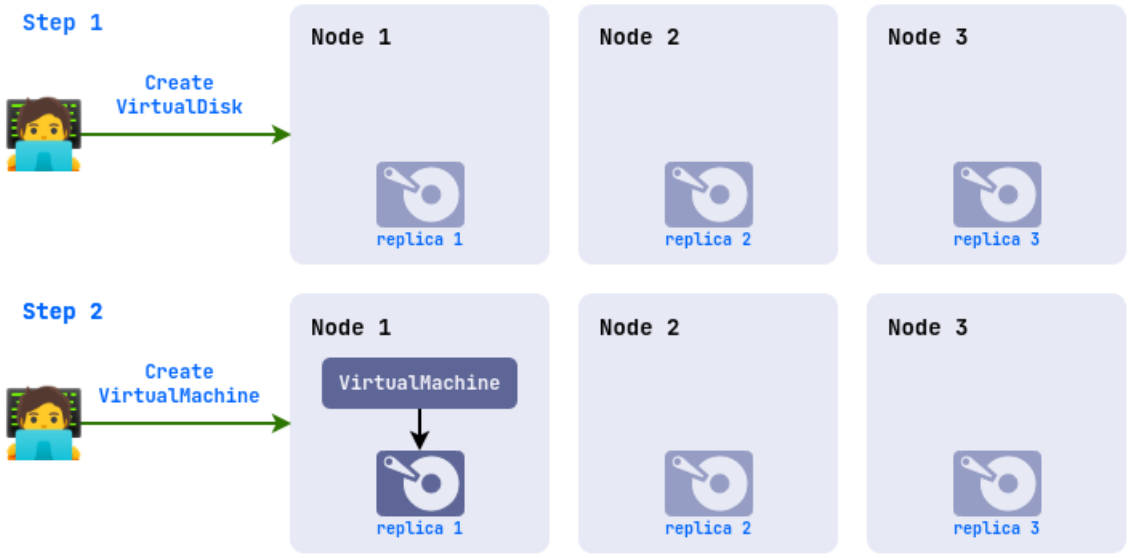
How storage affects a disk

Disk behavior depends on the storage behind the selected StorageClass. The differences show up in two properties.

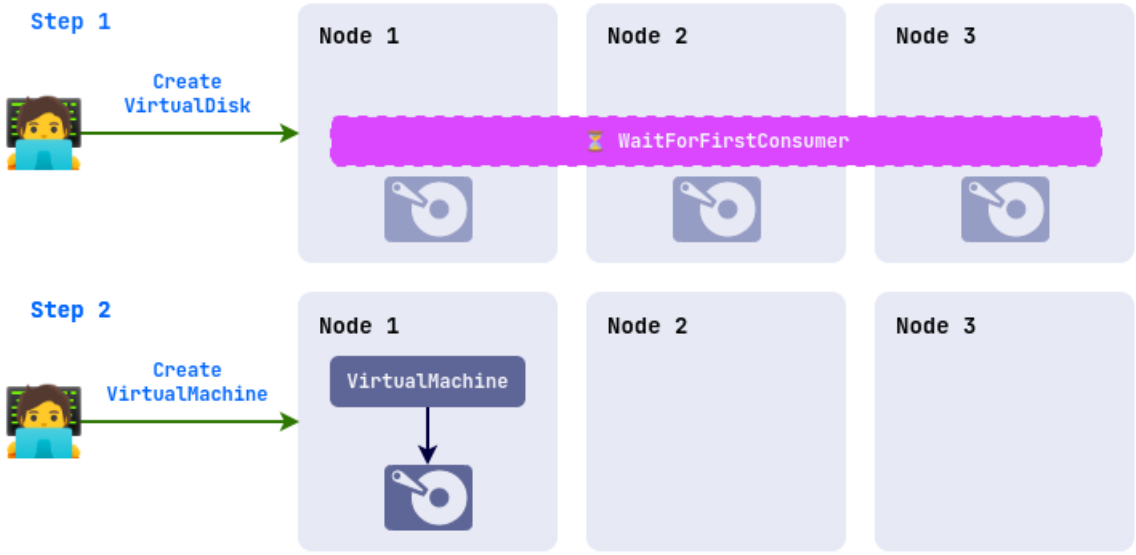
The volume type determines the format in which DP creates the disk. On file system volumes (`FileSystem`, for example NFS), the disk is created in the `qcow2` format, and on block devices (`Block`, for example iSCSI or Ceph RBD), data is written directly. Some storage types support both.

The volume binding mode determines when the disk is created:

- `Immediate`: The disk is created right away, independently of virtual machines, and you can attach it to a machine on any cluster node.



- `WaitForFirstConsumer`: The disk is created only after it’s attached to a virtual machine, and it’s placed on the node where that machine starts.



DP determines the remaining parameters, including the disk format, on its own from the capabilities of the selected StorageClass.

To view the available storage types, run the following command:

```
d8 k get storageclass
```

Example output:

```
NAME                    PROVISIONER                             RECLAIMPOLICY
VOLUMEBINDINGMODE       ALLOWVOLUMEEXPANSION   AGE
rv-thin-r1 (default)    replicated.csi.storage.deckhouse.io     Delete
Immediate               true                   48d
rv-thin-r2              replicated.csi.storage.deckhouse.io     Delete
Immediate               true                   48d
nfs-4-1-wffc            nfs.csi.k8s.io                          Delete
WaitForFirstConsumer    true                   30d
```

In the web interface, the same list is available on the **System** tab, in **Storage** → **Storage classes**.

Creating an empty disk

An empty disk is what you need to install an operating system on it or to store data separately from the system disk.

- Using the CLI
- Using the web interface

1. Create a VirtualDisk resource with the size and the storage class:

   ```
   d8 k apply -f - <<EOF
   apiVersion: virtualization.deckhouse.io/v1alpha2
   kind: VirtualDisk
   metadata:
     name: blank-disk
   spec:
     # Disk storage parameters.
     persistentVolumeClaim:
       # Specify the name of your StorageClass.
       storageClassName: rv-thin-r2
       size: 100Mi
   EOF
   ```

2. Verify that the disk is created:

   ```
   d8 k get vd blank-disk
   ```

   Example output:

   ```
   NAME         PHASE   CAPACITY   VIRTUALMACHINE   AGE
   blank-disk   Ready   100Mi                       1m2s
   ```

You can skip this step and create the disk while creating the VM.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Disks**.
3. Click **Create**.
4. In the form that opens, enter `blank-disk` in the **Disk name** field.
5. In the **Size** field, specify the size with units, for example `100Mi`.
6. In the **Storage class** field, select a StorageClass or keep the default one.
7. Click **Create**.
8. Check the disk status on its page.

Creating a disk from an image

You can fill a disk with data from an image created earlier, either a project VirtualImage or a cluster ClusterVirtualImage.

Specifying the disk size is optional. If you don’t set it, DP creates the disk exactly at the unpacked size of the image, and if you do set it, the size must be no smaller than the unpacked one.

- Using the CLI
- Using the web interface

1. Check the unpacked image size in the `UNPACKEDSIZE` column:

```
d8 k get vi ubuntu-24-04 -o wide
```

Example output:

```
NAME           PHASE   CDROM   PROGRESS   STOREDSIZE   UNPACKEDSIZE   REGISTRY
URL
TARGETPVC   AGE
ubuntu-24-04   Ready   false   100%       285.9Mi      2.5Gi          dvcr.d8-
virtualization.svc/vi/default/ubuntu-24-04:eac95605-7e0b-4a32-bb50-cc7284fd89d0
122m
```

2. Create a disk with a size larger than the unpacked one:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualDisk
metadata:
  name: linux-vm-root
spec:
  # Disk storage parameters.
  persistentVolumeClaim:
    # The size is larger than the unpacked image size.
    size: 10Gi
    # Specify the name of your StorageClass.
    storageClassName: rv-thin-r2
  # The source the disk is created from.
  dataSource:
    type: ObjectRef
    objectRef:
      kind: VirtualImage
      name: ubuntu-24-04
EOF
```

3. Create a second disk without specifying the size:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualDisk
metadata:
  name: linux-vm-root-2
spec:
  # Disk storage parameters.
  persistentVolumeClaim:
    # Specify the name of your StorageClass.
    storageClassName: rv-thin-r2
  # The source the disk is created from.
  dataSource:
    type: ObjectRef
    objectRef:
      kind: VirtualImage
      name: ubuntu-24-04
EOF
```

4. Compare the sizes of the created disks:

```
d8 k get vd
```

Example output:

```
NAME              PHASE   CAPACITY   VIRTUALMACHINE   AGE
linux-vm-root     Ready   10Gi                        7m52s
linux-vm-root-2   Ready   2590Mi                      7m15s
```

The first disk got the specified 10 GiB, and the second one got the unpacked image size.

You can skip this step and create the disk while creating the VM.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Disks**.
3. Click **Create**.
4. In the form that opens, enter `linux-vm-root` in the **Disk name** field.
5. In the **Source** field, select the image you need from the drop-down list.
6. If required, specify a larger size in the **Size** field or keep the default value.
7. In the **Storage class** field, select a StorageClass or keep the default one.
8. Click **Create**.
9. Check the disk status on its page.

Uploading a disk from the command line

If the image file is on your computer, upload it straight into a disk. DP creates a temporary upload endpoint for this and waits for the data.

- Using the CLI
- Using the web interface

1. Create a VirtualDisk resource with the `Upload` source:

   ```
   d8 k apply -f - <<EOF
   apiVersion: virtualization.deckhouse.io/v1alpha2
   kind: VirtualDisk
   metadata:
     name: uploaded-disk
   spec:
     dataSource:
       type: Upload
   EOF
   ```

   The resource moves to the `WaitForUserUpload` phase and is ready to accept the file. Start the upload within 10 minutes, otherwise the resource moves to the `Failed` phase and you have to create it again.

2. Get the addresses that accept the file:

   ```
   d8 k get vd uploaded-disk -o jsonpath="{.status.imageUploadURLs}" | jq
   ```

   Example output:

   ```
   {
     "external": "https://virtualization.example.com/upload/<SECRET_URL>",
     "inCluster": "http://10.222.165.239/upload"
   }
   ```

   Use the `inCluster` address if you upload the file from one of the cluster nodes, and `external` in all other cases.

3. Upload the file to the selected address:

   ```
   curl https://virtualization.example.com/upload/<SECRET_URL> --progress-bar -T <IMAGE_FILE> |
   cat
   ```

   Where `<SECRET_URL>` is the address from the previous step, and `<IMAGE_FILE>` is the path to the image file on your computer.

4. Verify that the disk has reached the `Ready` phase:

```
d8 k get vd uploaded-disk
```

Example output:

```
NAME            PHASE   CAPACITY   VIRTUALMACHINE   AGE
uploaded-disk   Ready   3Gi                         7d23h
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Disks**.
3. Click **Create**.
4. In the form that opens, enter the disk name in the **Disk name** field.
5. In the **Disk** block, select **Upload** in the **Data source** field.
6. Drag the file to the highlighted area, or click it and select the file on your computer.
7. In the **Size** field, specify the disk size, and in the **Storage class** field, select a StorageClass.
8. Click **Create**.

> **i**
>
> If the selected storage class uses the `WaitForFirstConsumer` volume binding mode, the **Upload** option isn't available. Without a consumer, the disk isn't created and there's nowhere to upload the file, so select a storage class with the `Immediate` mode or upload the data as an image.
>
> For the same reason, an empty disk created in advance with such a storage class stays in the "WAITING FOR VM" status and isn't offered in the **Disks / Images** window when attaching to a virtual machine, because only a ready disk can be selected. Create such disks right from the virtual machine form with the **Blank** or **Create from** options.

Disk properties are convenient to review in the web interface, in **Virtualization** → **Disks**:

- The list shows the disk name, status, size, storage class in the **Class** column, the virtual machine that uses the disk in the **Used by** column, and the resource age.
- On the disk page, the **Configuration** tab shows the data source, size, storage class, and the list of VMs in the **Used in** row.
- The **Diagnostics** tab shows the PVC name, its phase, size, storage class, PV name, and age, as well as the duration of the disk creation stages in the **Diagnostics summary** block.

Additional resources

- Module `virtualization` custom resources
- Module `virtualization` parameters

> **i**
>
> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Resizing and migrating virtual machine disks

An existing disk can be expanded or moved to another storage without deleting it and without recreating the machine.

Changing the disk size

You can grow a disk even while it's attached to a running virtual machine (VM). You can't shrink a disk.

- Using the CLI
- Using the web interface

1. Check the current disk size:

```
d8 k get vd linux-vm-root
```

Example output:

```
NAME            PHASE   CAPACITY   VIRTUALMACHINE   AGE
linux-vm-root   Ready   10Gi       linux-vm         10m
```

2. Set the new size in the `.spec.persistentVolumeClaim.size` parameter:

```
d8 k patch vd linux-vm-root --type merge -p '{"spec":{"persistentVolumeClaim":
{"size":"11Gi"}}}'

# You can achieve the same result by editing the resource.
d8 k edit vd linux-vm-root
```

3. Verify that the size has changed:

```
d8 k get vd linux-vm-root
```

   Example output:

```
NAME            PHASE   CAPACITY   VIRTUALMACHINE   AGE
linux-vm-root   Ready   11Gi       linux-vm         12m
```

You can change the size from the virtual machine page:

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Virtual machines**.
3. Select the VM the disk is attached to from the list and click its name.
4. On the **Configuration** tab, in the **Disks** section, click the pencil icon next to the disk size.
5. In the window that opens, specify a larger size.
6. Click **Apply**.

Or from the disk page itself:

1. Go to **Virtualization** → **Disks**.
2. Select the disk you need and click its name.
3. On the **Configuration** tab, specify a larger size in the **Size** field.
4. Click the **Save** button that appears.
5. Check the disk status on its page.

Migrating disks to other storage

In commercial Deckhouse Platform (DP) editions, you can move a disk to another storage by changing its storage class. The move works both for disks defined in the VM specification and for disks attached as a separate resource.

> ⚠ The virtual machine must be in the `Running` phase, and the source and target storage must be of the same type. You can’t move a disk from a file system volume to a block device or the other way around.

To move a disk, specify the new storage class in the `.spec.persistentVolumeClaim.storageClassName` parameter:

```
d8 k patch vd disk --type=merge --patch '{"spec":{"persistentVolumeClaim":
{"storageClassName":"new-storage-class-name"}}}'

# You can achieve the same result by editing the resource.
d8 k edit vd disk
```

After that, a live migration of the VM starts, during which the disk moves to the new storage.

If you need to move several disks of the same machine, change the storage class one disk at a time:

```
d8 k patch vd disk1 --type=merge --patch '{"spec":{"persistentVolumeClaim":
{"storageClassName":"new-storage-class-name"}}}'
d8 k patch vd disk2 --type=merge --patch '{"spec":{"persistentVolumeClaim":
{"storageClassName":"new-storage-class-name"}}}'
```

DP retries a failed migration with a growing delay. The first attempt runs immediately, the next ones after 5 and 10 seconds, then the delay doubles and from the seventh attempt stays at 300 seconds. To cancel the migration, restore the previous storage class in the specification.

Additional resources

- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Exporting virtual machine disk data

Export writes the contents of a virtual machine (VM) disk or its snapshot to a file so that you can move the data outside the cluster. For this feature to work, the `storage-volume-data-manager` module has to be enabled, and the command line requires the `d8` utility version 0.20.7 or later.

> ⚠ The disk must not be in use at the moment of export. If the disk is attached to a virtual machine, stop the VM first.

To import a downloaded disk back into the cluster, upload it as an image or as a disk.

- Using the CLI
- Using the web interface

An example of exporting a disk, with the command run on a cluster node:

```
d8 data export download -n <NAMESPACE> vd/<VD_NAME> -o file.img
```

An example of exporting a disk snapshot, with the command run on a cluster node:

```
d8 data export download -n <NAMESPACE> vds/<VD_SNAPSHOT_NAME> -o file.img
```

If you export data from somewhere other than a cluster node (for example, from your local machine), use the `--publish` flag.

1. Go to the **Projects** tab and select the project you need.
2. Stop the virtual machine the disk is attached to. While the disk is in use, the **Download** item isn't available.
3. Go to **Virtualization** → **Disks**.
4. In the row of the disk you need, click the ellipsis button and select **Download**.
5. Wait until the **Preparing...** step in the **Download** window changes to **Ready**: the file downloads automatically as soon as it's created. If that doesn't happen, click **Download** in the window.

> ⚠ While an export is active, the disk is in the `Exporting` phase and a virtual machine with this disk won't start. The export ends when its lifetime expires or when the DataExport resource created for it is deleted.

Additional resources

- Module `storage-volume-data-manager` documentation

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Creating a virtual machine

To create a virtual machine (VM), use the VirtualMachine resource. Its parameters let you configure:

- the virtual machine class;
- the resources required for the virtual machine to run (CPU, memory, disks, and images);
- the placement rules for the virtual machine on cluster nodes;
- the bootloader settings and optimal parameters for the guest OS;
- the virtual machine startup policy and the policy for applying changes;
- the initial configuration scripts (cloud-init);
- the list of block devices.

For a full description of virtual machine configuration parameters, see the configuration reference.

Creating a virtual machine

The following steps show how to start an Ubuntu 24.04 virtual machine on the disk you created earlier. The cloud-init script installs the `qemu-guest-agent` agent and the `nginx` service, and creates the `cloud` user with the `cloud` password.

- Using the CLI
- Using the web interface

1. Create a VirtualMachine resource:

```
d8 k apply -f - <<"EOF"
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachine
metadata:
  name: linux-vm
spec:
  # VM class name.
  virtualMachineClassName: generic
  # OS type: Generic for Linux and Windows for Windows. Generic by default.
  # osType: Generic
  # Bootloader type: BIOS, EFI, or EFIWithSecureBoot. BIOS by default.
  # bootloader: BIOS
  # VM initialization script.
  provisioning:
    type: UserData
    userData: |
      #cloud-config
      package_update: true
      packages:
        - nginx
        - qemu-guest-agent
      runcmd:
        - systemctl daemon-reload
        - systemctl enable --now nginx.service
        - systemctl enable --now qemu-guest-agent.service
      ssh_pwauth: True
      users:
        - name: cloud
          passwd: <PASSWORD_HASH>
          shell: /bin/bash
          sudo: ALL=(ALL) NOPASSWD:ALL
          lock_passwd: False
      final_message: "The system is finally up, after $UPTIME seconds"
  # VM resource settings.
  cpu:
    # Number of CPU cores.
    cores: 1
    # Guaranteed share of the CPU time of one core.
    coreFraction: 10%
  memory:
    # Amount of RAM.
    size: 1Gi
  # List of disks and images attached to the VM.
  blockDeviceRefs:
    # The order in this block sets the boot priority.
    - kind: VirtualDisk
      name: linux-vm-root
EOF
```

Where `<PASSWORD_HASH>` is the hash of the `cloud` user password, in quotes, generated with `mkpasswd --method=SHA-512 --rounds=4096`.

2. Verify that the machine has started:

```
d8 k get vm linux-vm
```

Example output:

```
NAME       PHASE     UPTIME   NODE           IPADDRESS     AGE
linux-vm   Running   11m      virtlab-pt-2   10.66.10.12   11m
```

The machine gets an IP address automatically from the range that the administrator sets in the

module settings.

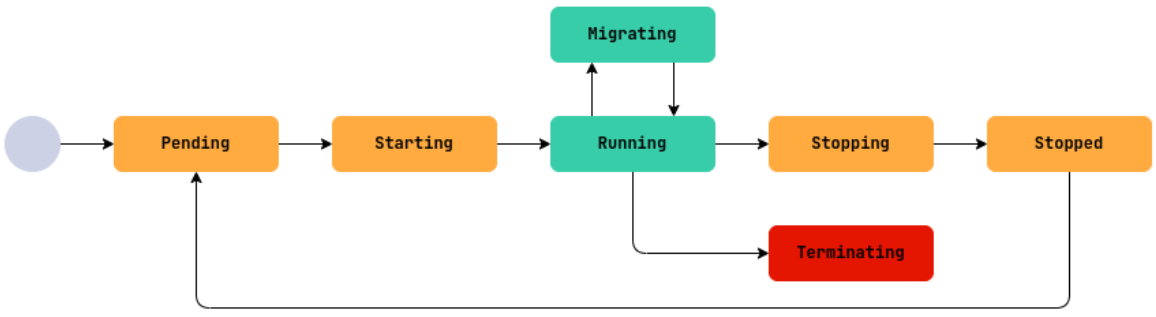
1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization → Virtual machines**.
3. Click **Create**.
4. In the form that opens, enter `linux-vm` in the **Name** field.
5. In the **Resources** section, set `1` in the **CPU cores** field, `10%` in the **Core fraction** field, and `1Gi` in the **Memory size** field.
6. In the **Disks** section, click **Add**.
7. In the **Disks / Images** window that opens, select **Existing** and pick the `linux-vm-root` disk from the list.
8. Scroll down to the **Cloud-init** toggle and enable it.
9. Paste the script into the field that appears, replacing `<PASSWORD_HASH>` with the password hash in quotes:

```
#cloud-config
package_update: true
packages:
  - nginx
  - qemu-guest-agent
runcmd:
  - systemctl daemon-reload
  - systemctl enable --now nginx.service
  - systemctl enable --now qemu-guest-agent.service
ssh_pwauth: True
users:
  - name: cloud
    passwd: <PASSWORD_HASH>
    shell: /bin/bash
    sudo: ALL=(ALL) NOPASSWD:ALL
    lock_passwd: False
final_message: "The system is finally up, after $UPTIME seconds"
```

10. Click **Create**.
11. Check the VM status on its page.

Virtual machine life cycle

From creation to deletion, a virtual machine goes through several phases. The current one is shown by the `.status.phase` field, and the details of what's happening to the machine are in the `.status.conditions` block.



The conditions in the `.status.conditions` block answer the question of why the machine is in its current phase. To view the ones that have a message, run the following command:

```
d8 k get vm <VM_NAME> -o json | jq '.status.conditions[] | select(.message != "")'
```

Where `<VM_NAME>` is the virtual machine name.

Diagnostics by phase

While a VM is in the `Pending` phase, it waits for its dependent resources to become ready, that is, disks, images, the VM class, and the secret with the initial configuration script. A delay in this phase means that one of the resources isn't ready or that the namespace or project quotas are exhausted. The conditions ending in `Ready` show what exactly blocks the startup:

```
d8 k get vm <VM_NAME> -o json | jq '.status.conditions[] | select(.type | test(".*Ready"))'
```

In the `Starting` phase, the dependent resources are ready and Deckhouse Platform (DP) starts the VM on one of the nodes. If the startup drags on, there's no suitable node, or the suitable nodes lack CPU or memory. The `Running` condition reports the reason:

```
d8 k get vm <VM_NAME> -o json | jq '.status.conditions[] | select(.type=="Running")'
```

In the `Migrating` phase, the machine moves to another node by live migration. The migration doesn't start or gets interrupted if the CPU instruction sets on the nodes are incompatible, the kernel versions differ, no node matches the placement rules, or the suitable nodes lack resources. The `Migrating` condition together with the `.status.migrationState` block shows the migration progress:

```
d8 k get vm <VM_NAME> -o json | jq '.status | {condition: .conditions[] |
select(.type=="Migrating"), migrationState}'
```

The `Terminating` phase is irreversible, all resources associated with the VM are released, but the resources themselves aren't deleted.

Conditions of a running VM

For a running machine, a few conditions are worth watching:

- `AgentReady` with the `True` status means that `qemu-guest-agent` is running in the guest system, and then the `.status.guestOSInfo` block contains information about the guest OS.
- `FirmwareUpToDate` with the `False` status means that it's time to update the VM firmware.
- `ConfigurationApplied` with the `False` status means that the specified configuration hasn't been applied to the running machine yet.
- `AwaitingRestartToApplyConfiguration` with the `True` status means that some of the changes apply only after a restart, and you have to perform it manually.
- `SizingPolicyMatched` with the `False` status means that the machine resources don't match the sizing policy of its class. Until you bring the parameters in line with the policy, you can't save configuration changes.
- `Migratable` shows whether the machine can be moved by live migration. The condition is computed only for a running VM, and a powered-off one doesn't have it. The `False` status with the `VirtualMachineNoMigrationTarget` reason means that the machine itself is suitable for migration, but there's no suitable node in the cluster. The `True` status with the `VirtualMachineWaitingForMigrationTarget` reason means that suitable nodes exist, but none of them can accept the machine right now, and this state resolves on its own.

Eviction from a node

The `EvictionRequired` condition appears when the node with your VM is put into maintenance mode. If the node was only made unschedulable with the `d8 k cordon` command but maintenance hasn't started, the condition doesn't appear.

The condition message tells you what will happen to the machine, namely a live migration without stopping the guest OS, a restart by DP with the cluster administrator's permission, or waiting until the machine is restarted. A machine that can be moved by live migration isn't restarted just to free the node. Until the eviction starts, the condition is only a warning, because maintenance can be canceled.

After a restart, the machine starts on another suitable node. If there's no such node, it stays in the `Pending` phase, and the `Running` condition shows the reason from the scheduler. A node in maintenance mode doesn't accept new machines, so a VM pinned to it by placement rules or using a device passed through from it starts only after the node returns to service.

Viewing the state in the web interface

The web interface shows the phase of a machine, its resources, and current problems on the machine page.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization → Virtual machines**.
3. Select the VM you need from the list and click its name.

The page header shows the current VM phase, its IP address, class, resource configuration, the number of attached disks and images, the node it’s placed on, whether the guest OS agent is running, and the time since startup. The page itself is split into the **Configuration**, **Monitoring**, **Operations**, **Events**, **VNC**, **TTY**, **Network policies**, **Snapshots**, **Diagnostics**, **Meta**, and **YAML** tabs.

On the **Configuration** tab of a running VM, usage charts appear next to the **CPU cores** and **Memory size** fields. In the **Disks** block, each device shows its boot order number, name, size, status, attachment method, storage class, and current disk load. In the **Networks** block, the network name, its status, and the IP and MAC addresses are shown, and the main cluster network is labeled **Main**.

Additional resources

- Module `virtualization` custom resources
- Module `virtualization` parameters

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Virtual machine CPU and memory

The CPU resources of a machine are defined by the number of cores and the core fraction, and the allowed combinations are limited by the virtual machine (VM) class.

Configuring CPU and coreFraction

Two parameters set the CPU resources of a machine. The `.spec.cpu.cores` parameter sets the number of virtual cores, and `.spec.cpu.coreFraction` sets the guaranteed share of the power of each of them.

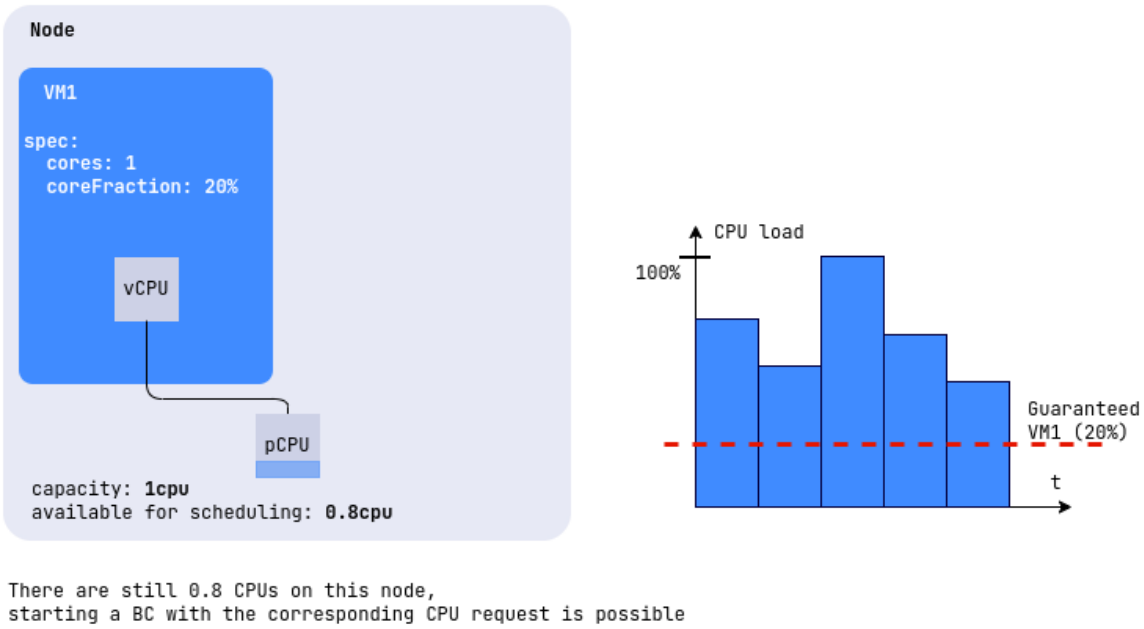
```
spec:
  cpu:
    cores: 2
    coreFraction: 20%
```

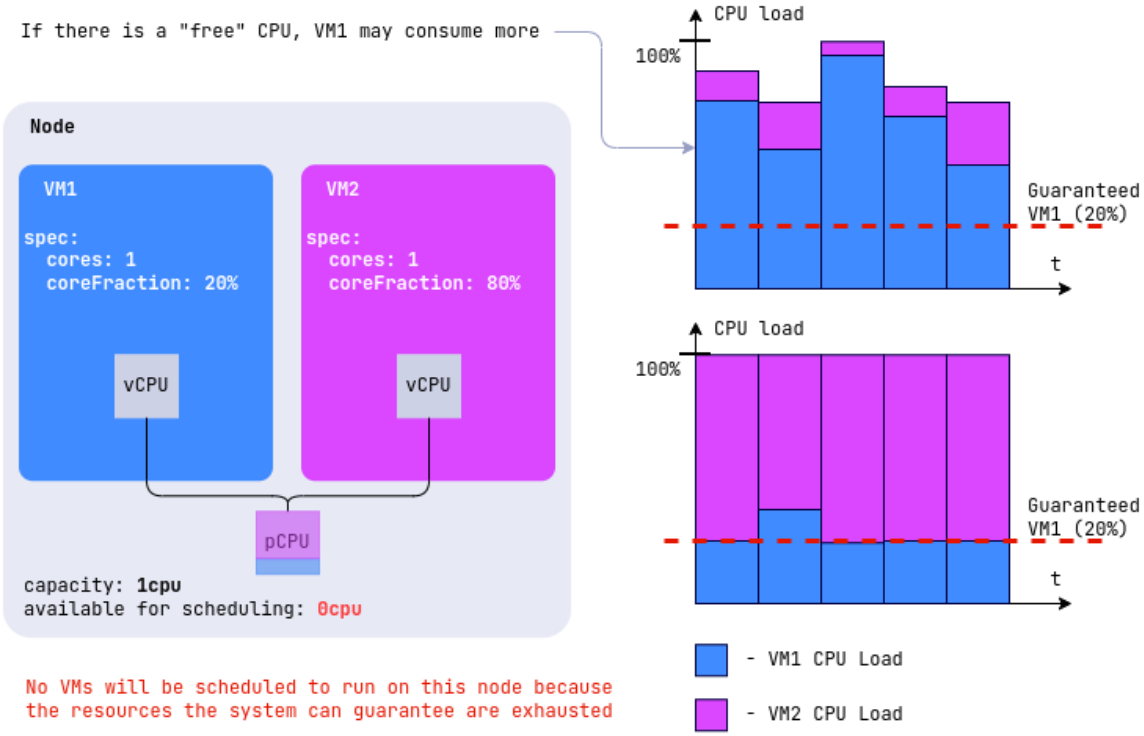
In this example, the machine gets two virtual cores and a guaranteed 20% of the power of each, that is, 0.4 cores in total, regardless of the node load. When the node has free resources, the machine can take both cores in full. Such a reserve lets you keep more machines on a node than it has physical cores, without losing stability under load.

If `coreFraction` isn’t set, each virtual core gets 100% of a physical one.

> An administrator can restrict the set of allowed `coreFraction` values in the sizing policy of the VM class, and then you have to choose from them.

The guaranteed share is taken into account when selecting a node, so a machine doesn’t start where the node can’t provide the guarantees to all machines placed on it. The figure shows two machines with one core each, the first with `coreFraction: 20%` and the second with `coreFraction: 80%`.





Automatic coreFraction (Auto)

Deckhouse Platform (DP) can pick the CPU time share on its own, following how much the machine consumes.

Instead of a fixed percentage, you can set `coreFraction: Auto`. Then DP picks the fraction, raising it when the machine lacks CPU and lowering it when the machine is idle. The number of cores and the amount of memory stay unchanged, and the new fraction applies without a restart.

```
spec:
  cpu:
    cores: 4
    coreFraction: Auto
```

The selection works as follows:

- A machine created with the `Auto` value starts at 10% or the nearest value allowed by the sizing policy, because a machine with no consumption history counts as idle.
- A machine moved to `Auto` from a fixed percentage keeps its current share and waits there for the first recommendation, so the switch doesn’t reduce the guaranteed share. If the current share isn’t one of the steps, the nearest step above it applies, and only the 100% share is replaced with the nearest step below, because 100% is never selected automatically.
- The fraction changes in steps. If the class sizing policy defines a `coreFractions` list, its values become the steps, otherwise 5%, 10%, 15%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 99% are used.
- The 100% value is never selected automatically, because at that value the CPU requests equal the limits, and such a machine can’t be changed without a restart. If 100% is listed in `coreFractions`, it simply isn’t used, and the ceiling becomes the next value down, while without a sizing policy the ceiling is 99%.
- For the same reason, the sizing policy has to leave at least two values to choose from. A policy that allows only 50%, or 50% and 100%, would pin the machine at 50% forever, so such a combination is rejected and you have to set the fraction explicitly.
- The recommended value is published in the `.status.recommendedResources.cpu.coreFraction` field, and the applied one in `.status.resources.cpu.coreFraction`. Every change is reported by the `CoreFractionScaling` event.
- If the node doesn’t have room for the new requests, the machine moves to another node and keeps running.

The `CoreFractionAutoscaling` condition shows whether the selection is running. While it works, the condition has the `True` status, and for the first few minutes, until statistics accumulate, the reason is `WaitingForRecommendation`, and then `CoreFractionAutoscalingEnabled`.

If the selection becomes unavailable, the condition switches to `False`, and the reason explains why:

- `CoreFractionAutoscalingDisabled`: Vertical autoscaling is disabled.
- `InPlaceResizeDisabled`: In-place resource changes are disabled.
- `SizingPolicyHasNoSteps`: The sizing policy is narrowed down to a single value.

The machine keeps running with its current fraction, but stops following the load.

To opt out of automatic selection, set an explicit percentage. Switching between `100%` and `Auto` in either direction requires a machine restart, because it changes the QoS class, and the other transitions apply on the fly.

Sizing policy

An administrator can restrict the resource combinations available to machines of a certain class by setting a sizing policy in the `.spec.sizingPolicies` parameter of the VirtualMachineClass resource. If there’s no policy, resources are set freely.

The policy splits the number of cores into ranges and sets the allowed amount of memory and the allowed `coreFraction` values for each of them:

```
spec:
  sizingPolicies:
    - cores:
        min: 1
        max: 4
      memory:
        min: 1Gi
        max: 8Gi
      coreFractions: [5, 10, 20, 50, 100]
    - cores:
        min: 5
        max: 8
      memory:
        min: 5Gi
        max: 16Gi
      coreFractions: [20, 50, 100]
```

With such a policy, a machine with two cores falls into the first range, gets from 1 to 8 GiB of memory and one of the values 5%, 10%, 20%, 50%, or 100%. A machine with six cores falls into the second range, where 5 to 16 GiB of memory is available and the core fraction is 20%, 50%, or 100%.

The policy also limits oversubscription. For example, the minimum value `coreFraction: 20%` guarantees each machine one fifth of a core, which means oversubscription doesn't exceed 5 to 1.

If the machine configuration doesn't match the policy, the `SizingPolicyMatched` condition with the `False` status appears in the status. Such a machine keeps running, but you can't save changes to its configuration until the resources are brought in line with the policy. The same happens when an administrator changes the policy of a class whose machines are already running.

Besides the bounds, a range can set a grid step in the `cores.step` and `memory.step` parameters, and memory bounds per single core in the `memory.perCore` block.

A request that violates the policy is rejected with a message that names the parameter and the allowed values. Every message ends with the hint `check the sizing policy of the VirtualMachineClass or contact the administrator for more information`, which is omitted below.

For the `supercpu` class with the policy above, the messages look like this:

- the number of cores is outside all ranges, `cores: 10: does not match any sizing policy of VirtualMachineClass "supercpu": its 10 CPU core(s) fall outside the allowed ranges (1-4, 5-8); set the number of cores (spec.cpu.cores) accordingly;`
- an unsupported core fraction, `cores: 2` and `coreFraction: 30%: the CPU core fraction "30%" is not allowed; set the core fraction (spec.cpu.coreFraction) to one of: 5%, 10%, 20%, 50%, 100%;`
- memory outside the range, `cores: 2` and `size: 16Gi: the memory size (16Gi) is out of the range allowed by the sizing policy; set the memory size (spec.memory.size) between 1Gi and 8Gi`.

If a range defines a step or per-core memory bounds, four more messages become possible:

- cores off the step grid: `the number of CPU cores (7) does not match the sizing policy step; set the number of cores (spec.cpu.cores) to 6 or 8;`
- memory off the step grid: `the memory size (1536Mi) does not match the sizing policy step; set the memory size (spec.memory.size) to 1Gi or 2Gi;`
- memory per core outside the range: `the memory size (18Gi) is not allowed for 6 CPU core(s); set the memory size (spec.memory.size) between 6Gi and 12Gi, or change the number of cores (spec.cpu.cores) (the sizing policy allows between 1Gi and 2Gi of memory per core);`
- memory per core off the step grid: `the memory size (2560Mi) does not match the per-core sizing policy step for 2 CPU core(s); set the memory size (spec.memory.size) to 2Gi or 4Gi, or change the number of cores (spec.cpu.cores)`.

When there are several violations, all the reasons are listed in one message under the `does not match the sizing policy of VirtualMachineClass "supercpu" for several reasons:` heading.

CPU topologies

The topology determines how the CPU cores of a machine are distributed across sockets, and compatibility with applications sensitive to the CPU configuration depends on it. You set only the total number of cores in the `.spec.cpu.cores` parameter, and DP calculates the number of sockets itself:

```
spec:
  cpu:
    cores: 20
```

The more cores there are, the more sockets they're split across, and the larger the step with which you can change their number. The total number of cores has to be a multiple of the number of sockets, otherwise the request is rejected.

Number of cores	Sockets	Multiple of	Cores per socket
`1 ≤ cores ≤ 16`	1	1	1 to 16
`16 < cores ≤ 32`	2	2	9 to 16
`32 < cores ≤ 64`	4	4	9 to 16
`64 < cores ≤ 248`	8	8	9 to 31

For example, 20 cores give two sockets of 10 cores, and 80 cores give eight sockets of 10. The maximum for one machine is 248 cores and 1024 GB of memory.

DP publishes the calculated topology in the status:

```
status:
  resources:
    cpu:
      topology:
        coresPerSocket: 10
        sockets: 2
```

The memory overhead depends on the actually active cores and amounts to 8 MiB per logical core, that is, per the product of the number of sockets, cores per socket, and threads per core.

Additional resources

- Module `vertical-pod-autoscaler` documentation
- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Guest OS and bootloader of a virtual machine

The `osType` parameter defines the operating system type and applies the optimal set of virtual devices and parameters for a virtual machine (VM) to work correctly.

Supported values:

- `Generic` (default): For Linux and other operating systems. The standard virtual device configuration is used.
- `Windows`: For Microsoft Windows operating systems. Automatically enables Hyper-V features, a TPM device, and other settings optimized for Windows.
- `Legacy`: For operating systems without built-in AHCI and virtio drivers: Windows XP, Windows 2000, Windows Server 2003, DOS-era systems, and Linux with a kernel older than 2.6.19. Such a VM gets the i440fx chipset, and with `enableParavirtualization: false` it also gets the IDE bus for disks and CD-ROM and the RTL8139 network adapter, whose drivers these operating systems have.

> ⚠ A virtual machine gets an emulated TPM whose state is kept in memory and isn’t persisted. When the VM restarts or migrates, the TPM state is reset. Keep this limitation in mind when using Windows security features that depend on TPM.

The set of virtual devices the guest OS sees:

Device	Generic	Windows	Legacy
Chipset	q35	q35	i440fx
Bootloader	`BIOS`, `EFI`, `EFIWithSecureBoot`	`BIOS`, `EFI`, `EFIWithSecureBoot`	`BIOS` only
Disk bus	virtio-scsi, SATA with `enableParavirtualization: false`	virtio-scsi, SATA with `enableParavirtualization: false`	virtio-blk, IDE with `enableParavirtualization: false`
CD-ROM bus	virtio-scsi, SATA with `enableParavirtualization: false`	virtio-scsi, SATA with `enableParavirtualization: false`	IDE
Block devices in `.spec.blockDeviceRefs`	up to 16	up to 16	up to 16, up to 4 with `enableParavirtualization: false`
Network adapter	virtio-net, e1000 with `enableParavirtualization: false`	virtio-net, e1000 with `enableParavirtualization: false`	virtio-net, RTL8139 with `enableParavirtualization: false`
USB controller	xHCI (USB 3.0)	xHCI (USB 3.0)	UHCI (USB 1.1)
TPM	no	TPM 2.0	no
Random number generator	virtio-rng	no	no
Hyper-V features	no	yes	no
Attaching disks on the fly	yes	yes	only with `enableParavirtualization: true` and only if the guest OS has the virtio-scsi driver
Changing CPU and memory on the fly	yes	yes	no

USB device passthrough works for `Legacy`, but the UHCI controller is limited to the USB 1.1 speed of 12 Mbps, so fast storage in such a VM hits the bus limit.

Choose `Legacy` when the guest operating system can't work with an AHCI controller, not just because it's old. For Linux with kernel 2.6.19 and newer, `osType: Generic` with `enableParavirtualization: false` is a fit, because there's no four-device limit there and attaching disks on the fly through VirtualMachineBlockDeviceAttachment remains available.

For `Legacy`, the following isn't available:

- Changing the number of CPU cores and the amount of memory on a running VM, because these guest operating systems don't bring them into service. The change is accepted, the VM shows it in `.status.restartAwaitingChanges` along with the `AwaitingRestartToApplyConfiguration` condition, and it applies after a restart.
- Changing the contents of `.spec.blockDeviceRefs` on a running VM, because these disks stay static in both paravirtualization modes, so add a block device before the VM starts.
- The `EFI` and `EFIWithSecureBoot` bootloaders, as well as initialization (`cloud-init` and Sysprep), because these guest operating systems don't support them.
- Guest OS information in the VM status and file system information.

You can install the QEMU guest agent in such an OS from an archived virtio-win release, and the VM does show `AgentReady`, but its version is too old for Deckhouse Platform (DP). The VM gets the `AgentVersionNotSupported` condition, guest OS information isn't collected, and there's nothing to update the agent to.

A snapshot with `requiredConsistency: true` also doesn't complete successfully, but for a different reason. DP requests a file system freeze, and the agent replies that the command is disabled in its build with the `guest-fsfreeze-status has been disabled for this instance` message. On Windows, the freeze goes through the VSS provider, which isn't in this build. The snapshot waits in the `InProgress` phase for about ten minutes and moves to `Failed`, so set `requiredConsistency: false` for such VMs.

With `enableParavirtualization: false`, one more limitation applies. There can be no more than four block devices in total, because the IDE bus provides two channels with two devices each.

Example configuration for a Windows XP virtual machine:

```
spec:
  osType: Legacy
  bootloader: BIOS
  enableParavirtualization: false
  # other parameters...
```

The `bootloader` parameter defines the bootloader type of the virtual machine:

- `BIOS` (default): Uses the legacy BIOS.
- `EFI`: Uses the Unified Extensible Firmware Interface (UEFI/EFI).
  - `EFIWithSecureBoot`: Uses UEFI/EFI with Secure Boot support.

Example configuration for a Windows virtual machine:

```
spec:
  osType: Windows
  bootloader: EFI
  # other parameters...
```

Example configuration for a Linux virtual machine (you can omit the default values):

```
spec:
  osType: Generic
  bootloader: BIOS
  # other parameters...
```

> **i** For modern Linux distributions, choose `bootloader: EFI`, and for Windows, choose `bootloader: EFI` or `bootloader: EFIWithSecureBoot`.

⚠ `EFIWithSecureBoot` needs a persistent volume for the Secure Boot state, and creating it needs a default StorageClass in the cluster. If there's none, the virtual machine doesn't start and stays in the `Pending` state, and its status says that the default StorageClass isn't found. As soon as a default StorageClass appears, the machine starts automatically.

The `enableParavirtualization` parameter controls the use of the `virtio` bus for attaching the virtual devices of the VM. A change to this parameter takes effect only after the VM restarts.

- `true` (default): The `virtio` bus is used for disks, network interfaces, and other devices, which gives better performance. You can change the contents of `.spec.blockDeviceRefs` on a running VM without a restart, adding and removing devices, if the disk is available on the node where the VM runs.
- `false`: Standard device emulation is used (SATA for disks, e1000 for network interfaces; IDE and RTL8139 for the `Legacy` OS type), which may be required for compatibility with older operating systems that lack `VirtIO` drivers. Changes to `.spec.blockDeviceRefs` on a running VM (adding and removing disks and images, including ISO) take effect after the VM restarts. To attach and detach disks without a restart, use the VirtualMachineBlockDeviceAttachment (`vmbda`) resource, without changing the list in the VM specification.

i To use the paravirtualization mode (`virtio`), some operating systems require the matching drivers to be installed. If the drivers aren't installed, the VM may fail to boot or the devices may work incorrectly.

For the `Legacy` OS type, the default value `true` isn't a fit, because these operating systems have no built-in virtio drivers, so for a VM you're going to install from the original media, set `enableParavirtualization: false`, otherwise the installer reports that it found no hard drives. A warning is issued when creating and modifying such a VM.

Keep `enableParavirtualization: true` for a VM with the `Legacy` OS type only when the virtio drivers are already installed in the guest OS. Then the VM keeps the i440fx chipset and the `BIOS` bootloader, but gets disks on virtio-blk, the virtio-net adapter, and loses the four-device limit. The CD-ROM stays on the IDE bus, because virtio-blk has no drive.

To switch a system that's already installed:

1. Install the virtio drivers in the guest OS. For Windows XP, 2000, and Server 2003, take an archived virtio-win release, because the current releases no longer contain drivers for these systems, and install `viostor`, the virtio-blk driver. The package has no virtio-scsi driver for these operating systems.
2. Power off the VM.
3. Set `enableParavirtualization: true`.
4. Start the VM.

The order of the steps matters. The disk controller driver has to appear in the guest OS before the switch, not after, because the VM boots from that very controller. If the VM doesn't boot after the switch, set `enableParavirtualization: false` back and restart it. The disks return to the IDE bus and the guest OS boots as before.

Example configuration with paravirtualization disabled:

```
spec:
  enableParavirtualization: false
  # other parameters...
```

Additional resources

- Module `virtualization` custom resources

i **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**

The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Initial provisioning and the guest OS agent

On the first boot of a virtual machine (VM), the guest system is configured by an initialization script, and after that Deckhouse Platform (DP) communicates with it through the guest OS agent.

VM initialization scripts

Initialization scripts are designed for the initial configuration of a virtual machine when it starts.

The following initialization scripts are supported:

- Cloud-Init.
- Sysprep.

Cloud-Init

Cloud-Init is a tool for automatically configuring virtual machines at first boot. It performs a wide range of configuration tasks without manual intervention.

> **i** The Cloud-Init configuration is written in YAML and has to start with the `#cloud-config` header at the beginning of the configuration block. For other possible headers and their purpose, see the official cloud-init documentation.

Key Cloud-Init capabilities:

- creating users, setting passwords, adding SSH keys for access;
- automatically installing the required software at first boot;
- running arbitrary commands and scripts to configure the system;
- automatically starting and enabling system services (for example, `qemu-guest-agent`).

Here are the typical scenarios.

1. Adding an SSH key for a preinstalled user that may already be present in a cloud image (for example, the `ubuntu` user in official Ubuntu images). The name of such a user depends on the image. Check it in the documentation for your distribution.

   ```
   #cloud-config
   ssh_authorized_keys:
     - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD... your-public-key ...
   ```

2. Creating a user with a password and an SSH key:

   ```
   #cloud-config
   users:
     - name: cloud
       passwd: <PASSWORD_HASH>
       lock_passwd: false
       sudo: ALL=(ALL) NOPASSWD:ALL
       shell: /bin/bash
       ssh-authorized-keys:
         - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD... your-public-key ...
   ssh_pwauth: True
   ```

   Where `<PASSWORD HASH>` is the password hash in quotes, generated with the `mkpasswd --method=SHA-512 --rounds=4096` command.

3. Installing packages and services:

```
#cloud-config
package_update: true
packages:
  - nginx
  - qemu-guest-agent
runcmd:
  - systemctl daemon-reload
  - systemctl enable --now nginx.service
  - systemctl enable --now qemu-guest-agent.service
```

The following example shows how to pass the script to a virtual machine:

- Using the CLI
- Using the web interface

You can embed a Cloud-Init script directly into the virtual machine specification, but such a script is limited to 2048 bytes:

```
spec:
  provisioning:
    type: UserData
    userData: |
      #cloud-config
      package_update: true
      ...
```

For longer scripts or scripts with private data, create the virtual machine initialization script in a Secret resource. Here is an example of a Secret resource with a Cloud-Init script:

```
apiVersion: v1
kind: Secret
metadata:
  name: cloud-init-example
data:
  userData: <base64 data>
type: provisioning.virtualization.deckhouse.io/cloud-init
```

A fragment of the virtual machine configuration that uses a Cloud-Init initialization script stored in a Secret resource:

```
spec:
  provisioning:
    type: UserDataRef
    userDataRef:
      kind: Secret
      name: cloud-init-example
```

> **i** The value of the `.data.userData` field has to be Base64-encoded. To encode it, use the `base64 -w 0` or `echo -n "content" | base64` command.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization → Virtual machines**.
3. Create a virtual machine or select an existing one and click its name.
4. On the **Configuration** tab, scroll down to the **Cloud-init** toggle and enable it.
5. Select the input mode:
   - **Basic setup**: Fill in the **Username**, **Password**, and **Public SSH key** fields, and enable the **Unrestricted sudo access** toggle if required. DP builds the cloud-init configuration itself.
   - **Editing**: Enter the cloud-init configuration manually in the **Parameters** field. The used volume is shown below the field (no more than 2048 bytes). In the **Linked secret** field, you can select an existing initialization script, and its contents load into the field. If no secret is linked, the configuration is stored in the VM specification.
6. Click the **Save** button that appears (or **Create** when creating the VM).

You can store a script as a separate resource and reuse it for several VMs. To create such a resource:

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Initialization scripts**.
3. Click **Create**.
4. In the **Name** field, enter the script name, and in the **Type** field, select `cloud-init` or `sysprep`.
5. In the **Files** block, set the key in the **File name** field (`userData` by default), and enter the contents manually, drag a file to the field, or click it to upload a file.
6. Click **Create**.

The **Initialization scripts** section shows secrets of the `provisioning.virtualization.deckhouse.io/*` type, each with its name, type (`cloud-init` or `sysprep`), the list of keys, and the resource age. To make a virtual machine use such a script, reference it in the `.spec.provisioning.userDataRef` parameter.

Sysprep

To configure virtual machines running Windows with Sysprep, only the Secret resource option is supported.

Here is an example of a Secret resource with a Sysprep script:

```
apiVersion: v1
kind: Secret
metadata:
  name: sysprep-example
data:
  unattend.xml: <base64 data>
type: provisioning.virtualization.deckhouse.io/sysprep
```

> **i** The value of the `.data.unattend.xml` field has to be Base64-encoded. To encode it, use the `base64 -w 0` or `echo -n "content" | base64` command.

A fragment of the virtual machine configuration that uses a Sysprep initialization script in a Secret resource:

```
spec:
  provisioning:
    type: SysprepRef
    sysprepRef:
      kind: Secret
      name: sysprep-example
```

Guest OS agent

Install QEMU Guest Agent in the guest system so that DP can interact with the operating system inside the VM. The agent is needed for three things:

- it makes consistent disk and VM snapshots possible;
- it reports information about the running system, and that information lands in the `.status.guestOSInfo` block;
- it shows that the operating system has actually booted, rather than just the virtual machine having started.

DP works with `qemu-guest-agent` version 5.2.0 and later. To check the installed version, run the following command:

```
qemu-guest-agent --version
```

Guest system information looks like this:

```
status:
  guestOSInfo:
    id: fedora
    kernelRelease: 6.11.4-301.fc41.x86_64
    kernelVersion: "#1 SMP PREEMPT_DYNAMIC Sun Oct 20 15:02:33 UTC 2024"
    machine: x86_64
    name: Fedora Linux
    prettyName: Fedora Linux 41 (Cloud Edition)
    version: 41 (Cloud Edition)
    versionId: "41"
```

The `AGENT` column shows whether the agent is running:

```
d8 k get vm -o wide
```

Example output:

```
NAME     PHASE     UPTIME   CORES   COREFRACTION   MEMORY   NEED RESTART   AGENT
MIGRATABLE   NODE           IPADDRESS    AGE
fedora   Running   5d21h    6       5%             8000Mi   False          True
True         virtlab-pt-1   10.66.10.1   5d21h
```

Install the agent with the command for your distribution and start the service:

```
# Debian and derivatives.
sudo apt install qemu-guest-agent

# CentOS and derivatives.
sudo yum install qemu-guest-agent

sudo systemctl enable --now qemu-guest-agent
```

For Linux, it’s convenient to automate the installation with an initialization script:

```
#cloud-config
package_update: true
packages:
  - qemu-guest-agent
runcmd:
  - systemctl enable --now qemu-guest-agent.service
```

The agent doesn’t need any configuration after installation. If your snapshots need application data consistency, put the preparation scripts in the `/etc/qemu-ga/hooks.d/` directory on Debian and Ubuntu, or `/etc/qemu/fsfreeze-hook.d/` on RHEL, CentOS, and Fedora. The scripts have to be executable, and the agent runs them before the file system freeze and after the thaw, so you don’t have to stop the application services.

Additional resources

- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Connecting to a virtual machine

You can connect to a running machine from the command line and from the web interface, while the

run policy controls whether the machine is running at all.

Connecting to a virtual machine

You can connect to a virtual machine (VM) in four ways. The first is a remote management protocol such as SSH, which you configure in the guest OS yourself. The second is the serial console. The third is VNC. The fourth is SPICE, if it is enabled for the machine.

- Using the CLI
- Using the web interface

Serial console:

```
d8 v console linux-vm
```

Example output:

```
Successfully connected to linux-vm console. The escape sequence is ^]
linux-vm login: cloud
Password: cloud
```

To exit the console, press `Ctrl+]`.

Connecting over VNC:

```
d8 v vnc linux-vm
```

Connecting over SPICE:

```
d8 v spice linux-vm
```

Connecting over SSH:

```
d8 v ssh cloud@linux-vm
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization → Virtual machines**.
3. Select the VM you need from the list and click its name.
4. Go to the **TTY** tab to work with the serial console, or to the **VNC** tab to connect over VNC.

The serial console and VNC are exclusive, only one user works in them, and a new connection disconnects whoever is already working with the machine.

Before connecting, `d8 v console` and `d8 v vnc` report who took the stream and since when, and offer a choice:

```
The serial console of linux-vm is in use:
  user       serviceaccount default/alice
  connected  12 minutes ago (14:32), from the d8 v command line

Connect and disconnect them? [y] yes  [N] no  [w] wait until free:
```

The `w` answer means waiting until the other user disconnects and connecting automatically. Pressing Enter cancels the connection, because the safe option is selected by default. The `--force` flag connects without asking, and it’s also what you need for a non-interactive run in a script.

> **i** The serial console doesn’t resize the terminal automatically. If the command output wraps incorrectly, set the size manually with the `stty rows <ROWS> cols <COLUMNS>` command, for example `stty rows 50 cols 200`. When the `xterm` package is installed in the system, the `resize` command does the same job.

SPICE

SPICE is a second remote display protocol that, unlike VNC, brings the sound of the guest system, redirects USB devices from your computer into it, and shares a clipboard with it. It works alongside

VNC, so open VNC sessions and the web interface keep working.

SPICE is disabled by default. To turn it on, set the `.spec.spice.enabled` parameter and restart the machine:

```
spec:
  spice:
    enabled: true
```

Together with SPICE, the machine gets a virtio-gpu video adapter. If the guest system has no driver for it, as in Windows 7 and Windows XP, set another adapter model with the `virtualization.deckhouse.io/video` annotation, using the `vga`, `bochs`, or `ramfb` value.

A shared clipboard, a resize to the client window, and a local cursor are added by the SPICE guest agent. Install it in the guest system, on Linux it's the `spice-vdagent` package, on Windows it's `spice-guest-tools`.

Connecting requires the `remote-viewer` client from the `virt-viewer` package, and the `d8 v spice` command opens it for you. If you don't have such a client, run the proxy alone and connect with your own client to the port the command prints:

```
d8 v spice linux-vm --proxy-only
```

⚠ The SPICE display is exclusive the same way the serial console and VNC are, and `d8 v spice` warns you before disconnecting whoever is already connected.

SPICE reserves memory whether a client is connected or not. This memory is part of the machine overhead, so on the node the machine takes more memory than its specification asks for.

Startup policy and VM state management

The startup policy determines how Deckhouse Platform (DP) maintains the machine state. It's set by the `.spec.runPolicy` parameter:

- `AlwaysOnUnlessStoppedManually`: The default option. The machine always runs, and you can stop it only manually.
- `AlwaysOn`: The machine always runs, and even after a shutdown from the guest OS DP starts it again.
- `Manual`: You manage the machine state yourself.
- `AlwaysOff`: The machine is always off, and you can't start it.

You can manage the machine state in two ways, by creating a VirtualMachineOperation resource or by using the `d8` utility. The resource describes the action declaratively, and the utility creates the same resource for you.

d8 command	Operation type	Action
`d8 v stop`	`Stop`	Stop the VM
`d8 v start`	`Start`	Start the VM
`d8 v restart`	`Restart`	Restart the VM
`d8 v evict`	`Evict`	Evict the VM to another node
`d8 v migrate`	`Migrate`	Migrate the VM to another node

The following example shows how to perform an operation on a virtual machine:

- Using the CLI
- Using the web interface

The easiest way to restart a machine is with the `d8` utility:

```
d8 v restart linux-vm
```

The same operation with a resource:

```
d8 k create -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineOperation
metadata:
  generateName: restart-linux-vm-
spec:
  virtualMachineName: linux-vm
  # Type of the operation to perform.
  type: Restart
EOF
```

The list of operations shows the result:

```
d8 k get virtualmachineoperation

# Short form of the command.
d8 k get vmop
```

The startup policy is set on the machine page:

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Virtual machines**.
3. Select the VM you need from the list and click its name.
4. On the **Configuration** tab, scroll down to the **Life cycle** section.
5. Select the policy you need from the **Startup policy** list.

Operations are available from the machine list:

1. Go to **Virtualization** → **Virtual machines**.
2. Select the virtual machine you need from the list and click the ellipsis button.
3. In the menu that opens, select the operation.

Only one operation runs at a time for a single machine. A new operation either supersedes the active one or fails, and which of the two happens depends on the pair of types:

Active operation	What can supersede it
`Start`	`Stop`
`Stop` or `Restart` without `force`	`Stop` or `Restart` with `force: true`
`Migrate`, `Evict`	`Stop`, `Restart`
`Stop` or `Restart` with `force: true`	nothing

A superseded operation moves to the `Superseded` phase. An operation that can’t supersede the active one moves to the `Failed` phase, so you have to create it again after the active one finishes. Restore and clone operations don’t supersede other operations.

Additional resources

- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Changing the virtual machine configuration

You can change the configuration of a virtual machine (VM) at any time after creation. On a powered-off machine, the changes apply right away, and on a running one Deckhouse Platform (DP) applies them differently depending on what exactly you changed.

Configuration block	How it applies on a running VM
`.metadata.labels`	Right away, and it propagates to the VM pod
`.metadata.annotations`	Right away, and it propagates to the VM pod
`.spec.liveMigrationPolicy`	Right away
`.spec.runPolicy`	Right away
`.spec.disruptions.restartApprovalMode`	Right away
`.spec.affinity`	Right away in commercial DP editions, a restart is required in DP Open
`.spec.nodeSelector`	Right away in commercial DP editions, a restart is required in DP Open
`.spec.cpu.cores`	Without a restart if changing the number of cores without a restart is enabled in commercial DP editions, otherwise a restart is required
`.spec.networks`	Adding and removing networks applies on a running VM if the guest OS supports attaching interfaces on the fly
Other `.spec` fields	A restart is required

The following example shows how to change the configuration of a virtual machine:

- Using the CLI
- Using the web interface

The following example changes the number of cores.

1. Check how many cores the guest OS sees now:

   ```
   d8 v ssh cloud@linux-vm --command "nproc"
   ```

   Example output:

   ```
   1
   ```

2. Set the new number of cores:

   ```
   d8 k patch vm linux-vm --type merge -p '{"spec":{"cpu":{"cores":2}}}'

   # You can achieve the same result by editing the resource.
   d8 k edit vm linux-vm
   ```

3. Verify that the change is accepted but not applied yet. The guest OS still sees one core, and the list of pending changes isn’t empty:

   ```
   d8 k get vm linux-vm -o jsonpath="{.status.restartAwaitingChanges}" | jq .
   ```

   Example output:

   ```
   [
     {
       "currentValue": 1,
       "desiredValue": 2,
       "operation": "replace",
       "path": "cpu.cores"
     }
   ]
   ```

   The `NEED RESTART` column shows the same:

```
d8 k get vm linux-vm -o wide
```

Example output:

```
NAME       PHASE     UPTIME   CORES   COREFRACTION   MEMORY   NEED RESTART
AGENT   MIGRATABLE   NODE           IPADDRESS     AGE
linux-vm   Running   5m16s    2       100%           1Gi      True
True    True         virtlab-pt-1   10.66.10.13   5m16s
```

4. Restart the machine:

```
d8 v restart linux-vm
```

5. Check the result. After the restart, the `.status.restartAwaitingChanges` block is empty and the guest OS sees two cores:

```
d8 v ssh cloud@linux-vm --command "nproc"
```

Example output:

```
2
```

By default, you confirm the restart. To make DP apply the changes itself, set the `.spec.disruptions.restartApprovalMode` parameter to `Automatic`:

```
spec:
  disruptions:
    restartApprovalMode: Automatic
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Virtual machines**.
3. Select the VM you need from the list and click its name.
4. Make the changes on the **Configuration** tab. If the machine has to be restarted, DP shows a warning and the list of pending changes.
5. To make the changes apply without your confirmation, scroll down to the **Life cycle** section, enable the **Auto-apply changes** toggle, and click **Save**.

Changing the number of cores without a restart

You can change the number of cores of a running machine without rebooting it, if the change is applicable through live migration. Within the current CPU topology, you can both add and remove cores.

The feature is disabled by default. To enable it, an administrator adds `HotplugCPUWithLiveMigration` to the `.spec.settings.featureGates` parameter:

```
kind: ModuleConfig
metadata:
  name: virtualization
spec:
  settings:
    featureGates:
      - HotplugCPUWithLiveMigration
```

In the web interface, the same toggle is called **Change CPU without reboot** and is located in the **Experimental features** block on the **System** tab, in **Deckhouse** → **Modules** → `virtualization` → **Configuration**. Only a DP administrator has the rights for this.

When the feature is enabled and the new `.spec.cpu.cores` value stays within the current topology, DP applies the change by live migration. If the change requires a different topology, the machine has to be rebooted. The topology calculation rules are described in CPU topologies.

- Using the CLI
- Using the web interface

Set the new number of cores:

```
d8 k patch vm linux-vm --type merge -p '{"spec":{"cpu":{"cores":4}}}'
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Virtual machines**.
3. Select the VM you need from the list and click its name.
4. On the **Configuration** tab, in the **Resources** section, set the new value in the **CPU cores** field.
5. Click the **Save** button that appears.

The guest OS doesn’t always bring new cores into service on its own, especially after a live migration. In Linux, a core is brought online through sysfs:

```
echo 1 > /sys/devices/system/cpu/cpu1/online
```

To make this happen automatically, add a `udev` rule:

```
cat <<'EOF' > /etc/udev/rules.d/99-hotplug-cpu.rules
SUBSYSTEM=="cpu",ACTION=="add",RUN+="/bin/sh -c '[ ! -e /sys$devpath/online ] ||
echo 1 > /sys$devpath/online'"
EOF
```

The cores brought into service appear in the output of `nproc`, `cat /proc/cpuinfo`, and `top`.

When you reduce the number of cores within the current topology, the distribution of cores across sockets is preserved.

Changing the amount of memory without a restart

You can increase the amount of memory of a running machine without rebooting it. Reducing it requires a restart.

The feature is disabled by default. To enable it, an administrator adds `HotplugMemoryWithLiveMigration` to the `.spec.settings.featureGates` parameter:

```
kind: ModuleConfig
metadata:
  name: virtualization
spec:
  settings:
    featureGates:
      - HotplugMemoryWithLiveMigration
```

In the web interface, the toggle is called **Change memory without reboot** and is located in the same place, in the **Experimental features** block of the module settings.

When the feature is enabled, the new `.spec.memory.size` value is greater than the current one, and the machine allows migration, DP applies the change by live migration. A restart is required if the memory is reduced, if the original size is less than 1 GiB, or if the machine can’t be migrated. Without a restart, memory grows up to 256 GiB, the ceiling built into the machine configuration at first start.

- Using the CLI
- Using the web interface

Set the new memory size:

```
d8 k patch vm linux-vm --type merge -p '{"spec":{"memory":{"size":"4Gi"}}}'
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Virtual machines**.
3. Select the VM you need from the list and click its name.
4. On the **Configuration** tab, in the **Resources** section, set the new value in the **Memory size** field.
5. Click the **Save** button that appears.

As with cores, the guest OS may not bring new memory blocks into service on its own. In Linux, a block is brought online through sysfs, and the device name is visible in the `lsmem` output or in the

`/sys/bus/memory/devices/` directory:

```
echo 1 > /sys/bus/memory/devices/memoryXXX/online
```

To make this happen automatically, add a `udev` rule:

```
cat <<'EOF' > /etc/udev/rules.d/99-hotplug-memory.rules
SUBSYSTEM=="memory",ACTION=="add",DEVPATH=="/devices/system/memory/memory[0-9]*",
TEST=="state", ATTR{state}!="online", ATTR{state}="online"
EOF
```

Additional resources

- Module `virtualization` custom resources
- Module `virtualization` parameters

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Placing virtual machines on nodes

Four mechanisms control where exactly a virtual machine (VM) starts:

- `.spec.nodeSelector`: The simplest way, it selects nodes with the required labels.
- `.spec.affinity.nodeAffinity`: Sets the preferred nodes for placement.
- `.spec.affinity.virtualMachineAndPodAffinity`: Places the machine next to other machines and workloads.
- `.spec.affinity.virtualMachineAndPodAntiAffinity`: On the contrary, spreads them across different nodes.

Conditions can be hard or soft. A hard `requiredDuringSchedulingIgnoredDuringExecution` condition is mandatory, and the machine doesn't start if there's no suitable node. A soft `preferredDuringSchedulingIgnoredDuringExecution` condition is taken into account by the scheduler where possible.

All rules, including `.spec.nodeSelector` from the VM class, apply together. If at least one hard condition can't be met, the machine stays in the `Pending` phase. So set consistent rules, prefer combinations of labels over single hard restrictions, and keep spare nodes for critical workloads. Also consider the startup order: if one machine has to end up next to another, the second one has to start first. If the nodes you need have `taints`, add the matching `tolerations` to the machine.

> When you change the placement rules of a running machine and its current node no longer meets the new requirements, in commercial editions Deckhouse Platform (DP) moves the machine by live migration, and in DP Open the changes apply only after a reboot. A machine that already meets the new requirements stays where it is.

To set the placement rules in the web interface:

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization → Virtual machines**.
3. Select the VM you need from the list and click its name.
4. On the **Configuration** tab, scroll down to the **VM placement** toggle.
5. Select the input mode. In the **Basic setup** mode, the rules are set with the **Co-location** and **Separate placement** toggles, and in the **Editing** mode, the placement block is set manually as YAML.
6. Enable the toggle you need and fill in the fields. The **Select rule mode** field offers **Required** (`requiredDuringSchedulingIgnoredDuringExecution`) and **Preferred** (`preferredDuringSchedulingIgnoredDuringExecution`), and the **Placement rule** field offers placement relative to other VMs or relative to node labels.
7. Click the **Save** button that appears.

Tolerance to node restrictions

`Tolerations` let a VM start on nodes with restrictions (`taints`) that otherwise block scheduling. This is useful when you need to run VMs on special nodes (for example, test nodes) or nodes with certain characteristics.

Here is an example of using `tolerations` to allow a start on nodes with the `node.deckhouse.io/group=:NoSchedule` taint:

```
spec:
  tolerations:
    - key: "node.deckhouse.io/group"
      operator: "Exists"
      effect: "NoSchedule"
```

Each element of the `tolerations` list has to match a `taint` on the node for the VM to be placed on that node.

> **i** To view information about cluster nodes (including `taints`), you need a user role with access to cluster-level resources.

To view the `taints` on cluster nodes, run the following command:

```
d8 k get nodes -o custom-columns=NAME:.metadata.name,TAINTS:.spec.taints
```

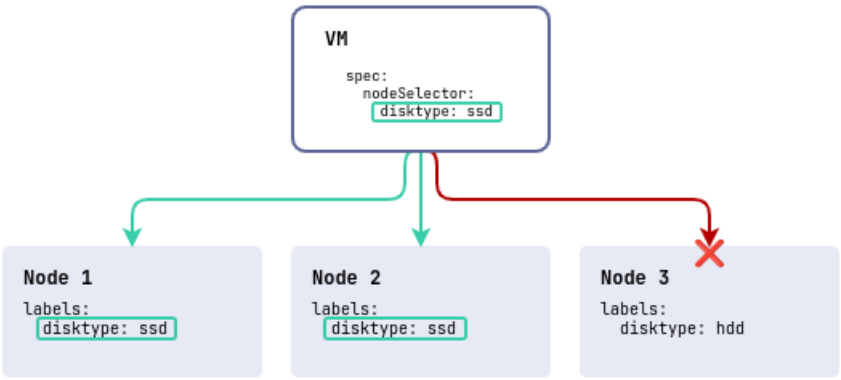
To view detailed information about a node, run the following command:

```
d8 k describe node <NODE_NAME>
```

Simple label binding (nodeSelector)

`nodeSelector` is the simplest way to control the placement of virtual machines using a set of labels. It lets you specify which nodes virtual machines can start on by selecting nodes with the required labels.

```
spec:
  nodeSelector:
    disktype: ssd
```



In this example, the cluster has three nodes, two of them with fast disks (`disktype=ssd`) and one with slow ones (`disktype=hdd`). The virtual machine is placed only on nodes that have the `disktype` label with the `ssd` value.

To perform the operation in the web interface in the placement section:

1. Enable the **Co-location** toggle.
2. In the **Select rule mode** field, select **Required**.
3. In the **Placement rule** field, select **On selected nodes**.
4. In the **How to identify the node group** field, select **By labels** and specify the node labels (for example, `disktype: ssd`); the **By name** option lets you select specific nodes.
5. Click the **Save** button that appears.

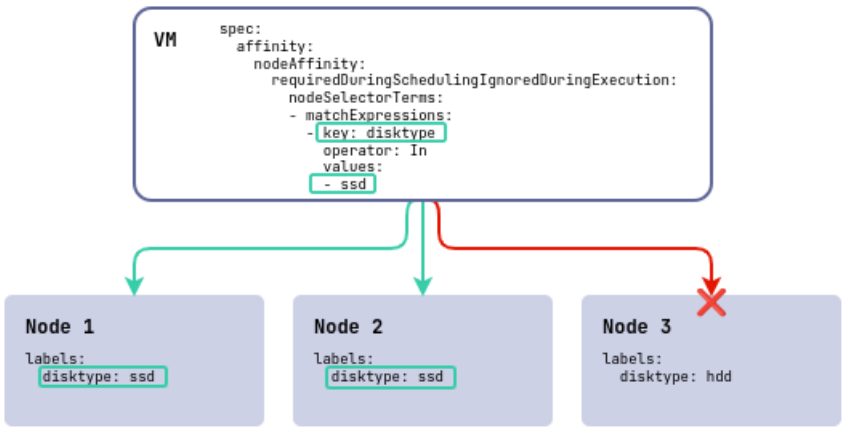
Preferred binding (Affinity)

`Affinity` provides more flexible and powerful tools compared to `nodeSelector`. It lets you set “preferences” and “requirements” for the placement of virtual machines. `Affinity` supports two kinds: `nodeAffinity` and `virtualMachineAndPodAffinity`.

`nodeAffinity` defines the nodes to run a VM on using label selector expressions.

Here is an example of using `nodeAffinity` with a hard rule:

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: disktype
                operator: In
                values:
                  - ssd
```



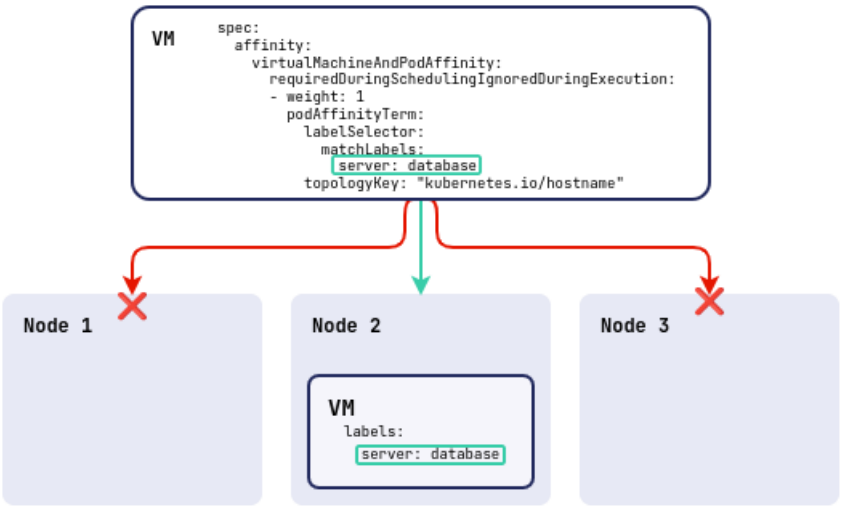
In this example, the cluster has three nodes, two of them with fast disks (`disktype=ssd`) and one with slow ones (`disktype=hdd`). The virtual machine is placed only on nodes that have the `disktype` label with the `ssd` value.

If you use a soft requirement (`preferredDuringSchedulingIgnoredDuringExecution`), then when there are no resources to run the VM on nodes with `disktype=ssd` disks, it’s scheduled on a node with `disktype=hdd` disks.

`virtualMachineAndPodAffinity` controls the placement of virtual machines relative to other virtual machines. It lets you set a preference for placing virtual machines on the same nodes where certain virtual machines are already running.

Here is an example of a soft rule:

```
spec:
  affinity:
    virtualMachineAndPodAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          podAffinityTerm:
            labelSelector:
              matchLabels:
                server: database
            topologyKey: "kubernetes.io/hostname"
```



In this example, the virtual machine is placed only on nodes that already run a virtual machine with the `server: database` label. The rule is soft (`preferred`), so if there are no such nodes, the machine starts on any suitable one.

To place VMs across availability zones (instead of pinning them to specific nodes), set `topologyKey: topology.kubernetes.io/zone` (Placing VMs across availability zones).

To set “preferences” and “requirements” for the placement of virtual machines in the web interface, in the placement section:

1. Enable the **Co-location** toggle, which corresponds to the `spec.affinity.virtualMachineAndPodAffinity` settings.
2. In the **Select rule mode** field, select **Required** or **Preferred**.
3. In the **Placement rule** field, select **On nodes with selected VMs**.
4. In the **Select labels** field, select the labels of the VMs you need from the list or enter your own in the `key: value` format.
5. Click the **Save** button that appears.

Avoiding co-location (AntiAffinity)

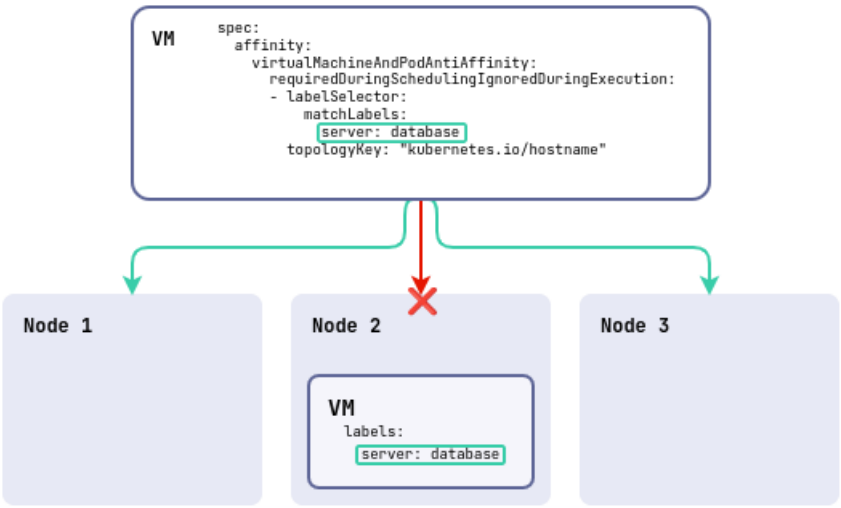
`AntiAffinity` is used to prevent VMs from being placed together on nodes. It’s useful for fault tolerance or load balancing.

> ⚠ Be careful with hard requirements in small clusters that have few nodes to run virtual machines on. If the `virtualMachineAndPodAntiAffinity` parameter with the `requiredDuringSchedulingIgnoredDuringExecution` type is used for virtual machines, it means that each VM copy has to be placed on a separate node. With a limited number of nodes in the cluster, this can lead to a situation where some VMs can’t start because of a lack of available nodes.

The terms `Affinity` and `AntiAffinity` describe the relationships between virtual machines. There’s no such antonym for nodes, but you can achieve the same result through `nodeAffinity` with the `NotIn` operator, excluding the nodes you need.

Here is an example of using `virtualMachineAndPodAntiAffinity`:

```
spec:
  affinity:
    virtualMachineAndPodAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchLabels:
              server: database
          topologyKey: "kubernetes.io/hostname"
```



In this example, the virtual machine being created isn’t placed on the same node as a virtual machine with the `server: database` label.

To place VMs across availability zones (instead of pinning them to specific nodes), set `topologyKey: topology.kubernetes.io/zone` (Placing VMs across availability zones).

To configure the prevention of co-locating VMs on nodes in the web interface, in the placement section:

1. Enable the **Separate placement** toggle, which corresponds to the `spec.affinity.virtualMachineAndPodAntiAffinity` settings.
2. In the **Select rule mode** field, select **Required** or **Preferred**.
3. In the **Placement rule** field, select **On nodes with selected VMs**.
4. In the **Select labels** field, select the labels of the VMs you don’t want the machine placed next to, or enter your own label in the `key: value` format.
5. Click the **Save** button that appears.

Placing VMs across availability zones

Placement rules work not only at the node level, but also at the availability zone level.

> **i** Availability zones have to be configured on the cluster nodes in advance. To do this, the nodes have to have the `topology.kubernetes.io/zone` label with the availability zone specified.

The examples above use `topologyKey: "kubernetes.io/hostname"`, which places VMs on the same node. To place VMs across availability zones instead of nodes, use `topologyKey: "topology.kubernetes.io/zone"`.

With `Affinity` and `topologyKey: "topology.kubernetes.io/zone"`, VMs are placed in the same availability zone where a virtual machine with the specified labels is present.

With `AntiAffinity` and `topologyKey: "topology.kubernetes.io/zone"`, VMs aren’t placed in the same availability zone as a virtual machine with the specified labels. This is useful for fault tolerance when distributing VMs across different availability zones.

To view the availability zones on cluster nodes (if those zones are set), run the following command:

```
d8 k get nodes -o custom-
columns=NAME:.metadata.name,ZONE:.metadata.labels.topology\.kubernetes\.io/zone
```

Additional resources

- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Attaching disks and images to a virtual machine

You can attach disks and images to a virtual machine (VM). They're described as block devices (BlockDevices).

Block device types and access modes:

Block device type	Comment
VirtualImage	Attached in read-only mode, or as a CD-ROM for ISO images.
ClusterVirtualImage	Attached in read-only mode, or as a CD-ROM for ISO images.
VirtualDisk	Attached in read-write mode.

There are two ways to attach devices:

- Through the VM specification (`.spec.blockDeviceRefs`): The disks are listed in the VirtualMachine configuration, and the boot order is set for them (by position in the list or through the `bootOrder` field). Recommended when configuring a VM manually, and when you need control over the boot order (for example, an ISO for OS installation).
- Through VirtualMachineBlockDeviceAttachment (`vmbda`): The disk is attached as a separate resource and doesn't take part in the boot order. Disks are attached through the `virtio-scsi` bus, regardless of the `enableParavirtualization` value. Recommended for automation and when you don't have the rights to edit the VM.

With `enableParavirtualization: true`, both ways let you attach and detach disks on a running VM without a reboot, if the disk is available on the node where it runs. With `enableParavirtualization: false`, the contents of `.spec.blockDeviceRefs` on a running VM change only after a reboot; to attach and detach disks without a reboot, use VirtualMachineBlockDeviceAttachment (`vmbda`).

> ⚠ When paravirtualization is disabled (`enableParavirtualization: false`), the devices from `.spec.blockDeviceRefs` work on the SATA bus, and on the IDE bus for the `Legacy` OS type. On a running VM, changes to this list, including attaching and detaching an ISO image, take effect only after a reboot.
>
> Disks attached through VirtualMachineBlockDeviceAttachment use the `virtio-scsi` bus and are attached without a reboot, if the guest OS has the driver for this bus. For the `Legacy` OS type with paravirtualization disabled, such an attachment is rejected, because the IDE bus doesn't support attaching on the fly: add the device to `.spec.blockDeviceRefs` and restart the VM.

You can attach a disk to a running VM only when the storage is available on the cluster node where the virtual machine runs. When creating and updating a VM, and when creating a VirtualMachineBlockDeviceAttachment, the placement rules (`nodeSelector`, `affinity`, `tolerations`) of the volume, the virtual machine, and the VM class are taken into account, and they have to share at least one valid placement. If the VM is already running on a specific node, the new disk has to be available on that node.

While a live migration of the machine is preparing the target node, a disk can be neither attached nor detached. A new VirtualMachineBlockDeviceAttachment resource stays in the `Pending` phase with the `BlockedByMigration` reason in the `Attached` condition, and a deleted one stays in the `Terminating` phase, and both finish once the migration completes. While the migration is still queued and the target node isn't being prepared yet, attaching and detaching work as usual.

Attaching through the VM specification

The devices listed in the machine specification are attached at startup and stay in place for the whole run. Their list is set in the `.spec.blockDeviceRefs` field of the VirtualMachine resource.

By default, the boot order matches the order of the devices in the list, and the optional `bootOrder` field

lets you set it explicitly (a lower value means a higher priority). If `bootOrder` is specified for at least one device, only the devices with a set `bootOrder` get into the boot chain. Integers from 1 and up are allowed, unique within the list. When a device is removed from the list, the boot order is recalculated for the remaining devices.

A change to the order of devices in the list or to the `bootOrder` values takes effect after the VM reboots. For example, you can attach an ISO image for OS installation with the boot priority you need, and remove it from the list after the installation. If the VM has paravirtualization disabled (`enableParavirtualization: false`), edits to `.spec.blockDeviceRefs` on a running VM, including ones with an ISO image, apply after the VM reboots.

- Using the CLI
- Using the web interface

A fragment of the virtual machine configuration with block devices and an explicit boot order:

```
spec:
  blockDeviceRefs:
    - kind: VirtualDisk
      name: <VD_NAME>
      bootOrder: 1
    - kind: VirtualImage
      name: <VI_NAME>
      bootOrder: 2
```

To attach a disk to a running virtual machine, add it to the `.spec.blockDeviceRefs` list:

```
spec:
  blockDeviceRefs:
    - kind: VirtualDisk
      name: <VD_NAME>
    - kind: VirtualImage
      name: <VI_NAME>
    - kind: VirtualDisk
      name: <ADDITIONAL_DISK_NAME>
```

To detach a disk, remove it from the list. With `enableParavirtualization: false`, a change to the list on a running VM takes effect after the VM reboots.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization → Virtual machines**.
3. Select the VM you need from the list and click its name.
4. On the **Configuration** tab, scroll down to the **Disks** section.
5. The following actions are available in the disk list:
   - **Add**: Attach a new disk or image to the VM.
   - **Eject**: Detach the device from the VM (the image or disk stays in the project, and you can attach it again to this or another VM).
   - **Delete**: Delete the image or disk resource itself from the cluster (after deletion, you can’t reuse it).
   - Change the disk size, with the pencil icon next to the current size.
   - Change the boot order, by changing the position of the disk in the list.

Attaching through VirtualMachineBlockDeviceAttachment

The VirtualMachineBlockDeviceAttachment resource attaches and detaches a block device on a VM without touching its specification. It suits automation and the cases where the user doesn’t have the rights to edit the machine.

The device is attached when the resource moves to the `Attached` phase. The other phases are described in the `.status.phase` field, and the `.status.conditions` block shows the reason for a delay.

- Using the CLI
- Using the web interface

Create a resource that attaches the empty `blank-disk` disk to the `linux-vm` virtual machine:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineBlockDeviceAttachment
metadata:
  name: attach-blank-disk
spec:
  blockDeviceRef:
    kind: VirtualDisk
    name: blank-disk
  virtualMachineName: linux-vm
EOF
```

Check the state of your resource:

```
d8 k get vmbda attach-blank-disk
```

Example output:

```
NAME                PHASE      VIRTUALMACHINE   AGE
attach-blank-disk   Attached   linux-vm         3m7s
```

Connect to the virtual machine and make sure the disk is attached:

```
d8 v ssh cloud@linux-vm --command "lsblk"
```

Example output:

```
NAME    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
sda       8:0    0   10G  0 disk <--- statically attached linux-vm-root disk
|-sda1    8:1    0  9.9G  0 part /
|-sda14   8:14   0    4M  0 part
`-sda15   8:15   0  106M  0 part /boot/efi
sdb       8:16   0    1M  0 disk <--- cloudinit
sdc       8:32   0 95.9M  0 disk <--- dynamically attached blank-disk disk
```

To detach the disk from the virtual machine, delete the resource you created earlier:

```
d8 k delete vmbda attach-blank-disk
```

Images are attached the same way, only the `kind` field takes the VirtualImage or ClusterVirtualImage value.

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineBlockDeviceAttachment
metadata:
  name: attach-ubuntu-iso
spec:
  blockDeviceRef:
    kind: VirtualImage # or ClusterVirtualImage
    name: ubuntu-iso
  virtualMachineName: linux-vm
EOF
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Virtual machines**.
3. Select the VM you need from the list and click its name.
4. On the **Configuration** tab, scroll down to the **Disks** section.
5. The following actions are available in the disk list:
   - **Add**: Attach a new disk or image to the VM; to attach the device as an additional one (through VirtualMachineBlockDeviceAttachment, without rebooting the VM), select the

Additional checkbox in the **Disks / Images** window.
- **Eject**: Detach the device from the VM (the image or disk stays in the project, and you can attach it again to this or another VM).
- **Delete**: Delete the image or disk resource itself from the cluster (after deletion, you can't reuse it).
- Change the disk size, with the pencil icon next to the current size.

Disk naming in the guest OS

Disk names in the guest system aren't stable, so relying on them in configuration is risky.

⚠ Block device names (`/dev/sda`, `/dev/sdb`, `/dev/sdc`, and so on) are assigned by the Linux kernel in the order the devices are discovered at boot. This order can change between reboots, so device names can change even when the SCSI addresses stay the same.

If you use `/dev/sdX` in configuration files (for example, `/etc/fstab`) or in scripts, after a VM reboot you can mount the wrong disk or end up with a malfunctioning system.

Example:

After the first VM boot:

```
$ lsscsi
[0:0:0:1]  disk    QEMU     QEMU HARDDISK   /dev/sda
[0:0:0:2]  disk    QEMU     QEMU HARDDISK   /dev/sdb
```

After a VM reboot:

```
$ lsscsi
[0:0:0:1]  disk    QEMU     QEMU HARDDISK   /dev/sdb
[0:0:0:2]  disk    QEMU     QEMU HARDDISK   /dev/sda
```

The SCSI addresses (`0:0:0:1`, `0:0:0:2`) stay the same, but the device names (`/dev/sda`, `/dev/sdb`) swap places.

Use stable identifiers instead of `/dev/sdX`:

- `/dev/disk/by-uuid/`: By partition UUID (preferable for `/etc/fstab`).
- `/dev/disk/by-path/`: By the SCSI connection path.
- `/dev/disk/by-id/`: By the SCSI device ID.

In configuration files and scripts, use partition UUIDs or symbolic links from `/dev/disk/by-*` instead of `/dev/sdX` names.

Network interface naming in the guest OS

In systems without predictable network interface naming, network interface names (`eth0`, `eth1`, `eth2`, and so on) are assigned by the Linux kernel in the order the devices are discovered at boot. When you add new network interfaces or change the order of networks in `.spec.networks`, the interface order can change, and IP addresses can end up assigned to the wrong interfaces.

Using `ethX` in configuration files (for example, `/etc/network/interfaces`, `netplan`, `systemd-networkd`) or in scripts when adding new interfaces or changing the order of networks can lead to network failures or a connection to the wrong network.

Modern distributions with systemd (Ubuntu 16.04+, Debian 9+, CentOS 7+, RHEL 7+) use predictable interface names by default (`enpXsY`, `ensX`, `enoX`), which are based on the physical characteristics of the device (PCI coordinates) and stay stable between reboots and when new interfaces are added.

But even with predictable names, bind the network configuration to the MAC addresses of the interfaces, especially if the order of networks in `.spec.networks` changes or new interfaces are added.

An example for systems without predictable naming:

Initially, the VM has two interfaces:

```
$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
```

After adding a new interface at the beginning of the `.spec.networks` list and rebooting the VM:

```
$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500  # New interface
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500  # Former eth0
4: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500  # Former eth1
```

The MAC addresses stay the same, but the interface names (`eth0`, `eth1`) shift, which can lead to IP addresses being assigned to the wrong interfaces.

Use stable identifiers instead of `ethX`:

- `enpXsY`: Predictable names based on the physical location (systemd networkd naming scheme, enabled by default in modern systems).
- Binding by MAC address: In the `netplan`, `systemd-networkd`, or `/etc/network/interfaces` configuration (preferable for guaranteed stability).

In configuration files and scripts, use stable interface names (`enpXsY`) or binding by MAC address instead of `ethX` names.

> **i** The predictable interface order holds only in guest operating systems with systemd (for example, Ubuntu, Debian). In Alpine and other distributions without systemd, the order may differ.

To open a machine application to other machines or to users outside the cluster, configure a service or Ingress as described in Accessing applications on a virtual machine.

Additional resources

- Module `virtualization` custom resources

> **i** **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Live migration of virtual machines

Live migration of virtual machines (VMs) is the process of moving a running VM from one physical node to another without shutting it down. This feature plays a key role in managing virtualized infrastructure, keeping applications running during maintenance, load balancing, or updates.

How live migration works

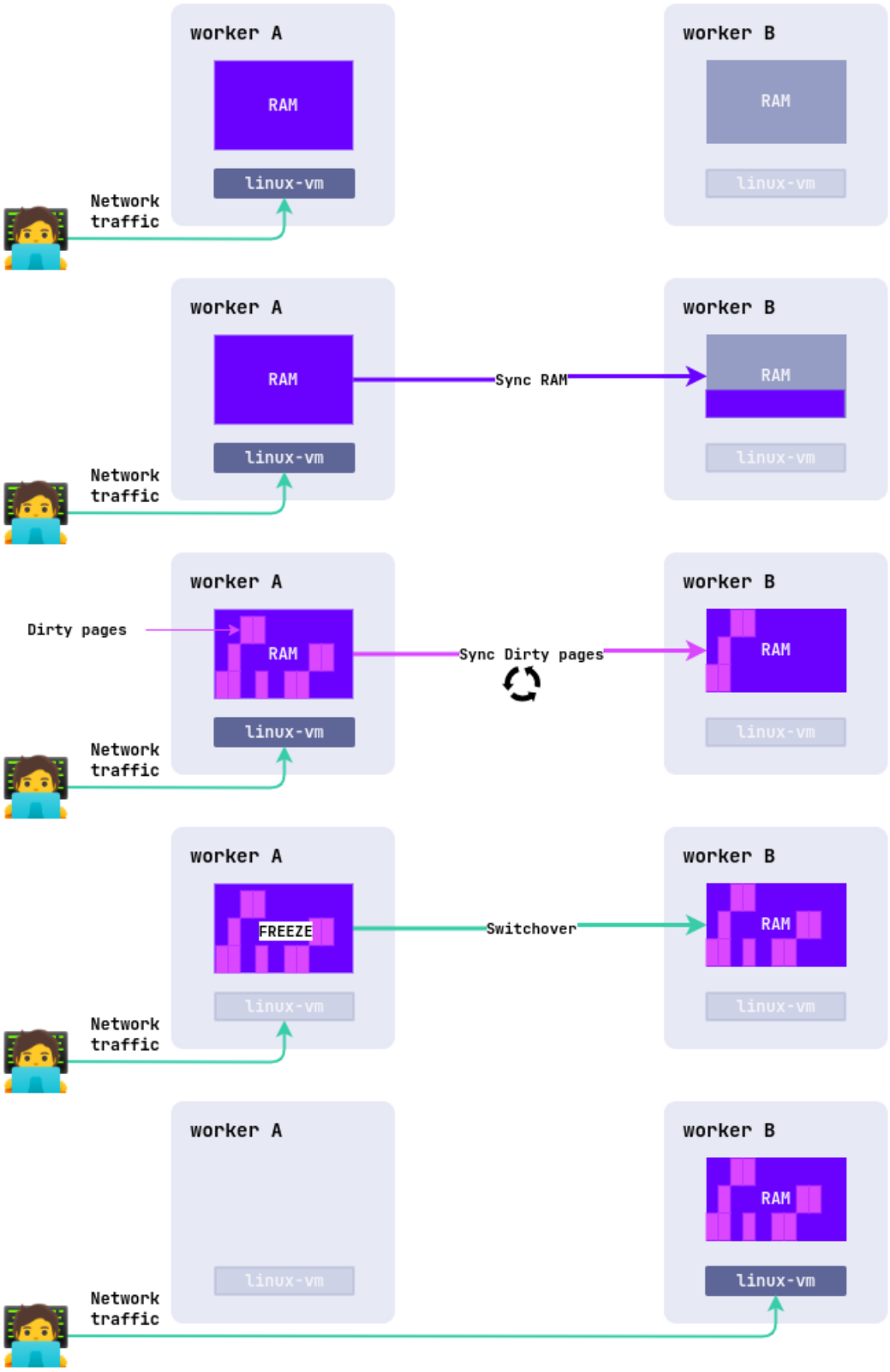
The live migration process consists of several stages:

1. A new VM is created on the target node in a paused state. Its configuration (CPU, disks, network) is copied from the source node.
2. All the RAM of the VM is copied to the target node over the network. This is called the initial transfer.
3. While the memory is being transferred, the VM keeps running on the source node and can modify some memory pages. Such pages are called dirty pages, and the hypervisor marks them.
4. After the initial transfer, only the modified pages are sent again. This process repeats in several cycles:
    - The higher the load on the VM, the more dirty pages appear, and the longer the migration

takes.
   - With good network bandwidth, the amount of unsynchronized data gradually decreases.

5. When the number of dirty pages becomes minimal, the VM on the source node is paused (usually for 100 milliseconds):
   - The remaining memory changes are transferred to the target node.
   - The state of the CPU, devices, and open connections is synchronized.
   - The VM starts on the new node, and the original copy is deleted.

Until the VM switches to the new node (step 5), the VM on the source node keeps running as usual and serving users.



Requirements and limitations

A live migration doesn’t always succeed. The following is what has to match on the source and target nodes.

Disk availability. All disks attached to the VM have to be available on the target node. With network storage such as NFS or Ceph, this requirement is met on its own, because the disks are visible from all cluster nodes. Local storage needs to be able to create a new local volume on the target node, and if such storage exists only on the source node, the migration doesn’t run.

Attaching and detaching disks. While a migration is preparing the target node, disks can be neither attached to the machine with a VirtualMachineBlockDeviceAttachment resource nor detached by deleting one. An attachment stays in the `Pending` phase with the `BlockedByMigration` reason in the `Attached` condition, and a deleted one stays in the `Terminating` phase, until the migration completes. While the migration is still queued and the target node isn’t being prepared yet, for example when it waits for the project quota to free up, attaching and detaching work as usual. The reverse is also true, a migration waits for an attach or detach request that has already been sent, and all that time the VirtualMachineOperation resource stays in the `Pending` phase with the `WaitingForBlockDeviceAttachment` reason. If the request doesn’t complete within 5 minutes, the operation fails.

Network bandwidth. The slower the network, the more memory synchronization iterations the migration goes through and the longer the VM downtime at the final stage, and in the worst case the migration doesn’t fit into the timeout. The migration policy controls how the migration runs, and the AutoConverge mechanism helps with a slow network.

Kernel versions. All cluster nodes have to run the same Linux kernel version. Differences in versions lead to incompatible interfaces, system calls, and resource handling, which breaks the migration.

CPU compatibility. The CPU type in the virtual machine class sets the CPU requirements. The `Host` type allows migration only between nodes with similar CPUs, so it works neither between Intel and AMD nor between different CPU generations with different instruction sets. The `HostPassthrough` type requires exactly the same CPU on the target node as on the source one. To let a machine migrate between nodes with different CPUs, set the `Discovery`, `Model`, or `Features` type in the class.

Duration. A migration has a completion timeout of 800 seconds per gibibyte of VM memory, plus 800 seconds per gibibyte of disk when the disks move along with it. For example, a machine with 4 GiB of memory and a 20 GiB disk gets `800 × (4 + 20) = 19200` seconds, or about 5.3 hours. A migration that doesn’t fit into this time is considered failed and is canceled, which happens with a slow network or a high load on the VM.

Checking whether a VM is ready for migration

The `type: Migratable` condition in the VM status shows whether the VM can be moved by live migration. It takes into account both the VM itself (disks, passed-through devices, CPU type) and the state of the cluster (whether there’s a node to move it to). The `True` value answers the question of whether the move is possible, not whether the VM will move at this very moment, so look at the reason along with the value.

The overall picture for all VMs:

```
d8 k get vm -o wide
```

The value in the `MIGRATABLE` column shows the result, and the condition describes the reason:

```
d8 k get vm <VM_NAME> -o json | jq '.status.conditions[] | select(.type=="Migratable")'
```

The most common reasons:

Reason	What it means	What to do
`VirtualMachineMigratable`	The VM can be moved by live migration	—
`VirtualMachineNoMigrationTarget`	The VM is capable of migrating, but no other cluster node can host it	Check the `spec.nodeSelector`, `spec.affinity`, and `spec.tolerations` of the VM and the same parameters of its VirtualMachineClass

Reason	What it means	What to do
`VirtualMachineWaitingForMigrationTarget`	The VM is capable of migrating and suitable nodes exist in the cluster, but none of them can accept it right now, because the nodes are unschedulable, not ready, or don’t run virtualization	If maintenance is in progress, migration becomes possible as soon as such a node returns. In other cases, check why the nodes are unschedulable and whether virtualization runs on them
`VirtualMachineDisksNotMigratable`	The VM disks are in storage available from only one node	Move the disks to storage with the `ReadWriteMany` access mode
`VirtualMachineHostDevicesNotMigratable`	The VM has a device attached that can’t be moved to another node	Detach the device and restart the VM
`VirtualMachineNonMigratable`	The VM can’t be moved by live migration, and the reason is in the `message` field of the condition.	Read the `message` of the condition. If it’s about the CPU, use the `Discovery`, `Model`, or `Features` types in the VM class
`VirtualMachineDisksShouldBeMigrating`	The VM can be moved, and its local disks are moved along with it	—

Moving disks along with a VM is available in commercial Deckhouse Platform (DP) editions, so the `VirtualMachineDisksShouldBeMigrating` reason appears only there. In DP Open, a VM with disks in storage available from one node gets the `VirtualMachineDisksNotMigratable` reason.

Specifics of the Migratable condition

A few specifics of the condition that matter when planning maintenance and reading the status:

- The `Migratable` condition describes not only the VM but the cluster itself. If you remove the required label from the only suitable node without changing the VM parameters, the condition still becomes `False`. When a suitable node appears, the condition returns to `True`.
- When a node is made unschedulable, the VM stays capable of migrating. Cordoning, rebooting, and node maintenance happen on their own, so the condition stays `True` and only the reason changes to `VirtualMachineWaitingForMigrationTarget`. Otherwise, planned maintenance of a neighboring node would turn a CPU or memory change into a VM restart. In DP Open, such changes require a reboot in any case.
- The `True` value means that the VM is capable of migrating, not that it’s ready to migrate right now. Before a migration, look at the reason. The `VirtualMachineMigratable` reason means there’s a node to migrate to, and `VirtualMachineWaitingForMigrationTarget` means there’s no suitable node at this moment. The `d8_virtualization_virtualmachine_migratable` metric carries the same answer in the `reason` label, so a dashboard that filters VMs only by value counts waiting VMs together with those ready to migrate.
- A migration started with the `VirtualMachineWaitingForMigrationTarget` reason doesn’t wait for a node indefinitely. If the target pod can’t be scheduled within five minutes, the operation fails, and the `Completed` condition of the VirtualMachineOperation resource gets the `TargetUnschedulable` reason. If maintenance drags on, restart the migration.
- A stopped VM has no such condition, because the capability to migrate is computed only for a running VM. During the downtime, the disks could have moved to different storage and a device could have been detached.
- In DP Open, placement changes are taken into account after the VM restarts. While the VM runs, it uses the parameters it was started with, and the condition describes exactly those. New `nodeSelector`, `affinity`, or VM class values get into the calculation only after a restart. In commercial DP editions, such changes apply without a restart and get into the condition calculation right away.
- Local disks don’t prevent migration, but a node is still needed. In commercial DP editions, a VM with local disks moves along with them, so the condition stays `True` with the `VirtualMachineDisksShouldBeMigrating` reason. But if no cluster node matches its placement rules, the condition is `False`, because there’s nowhere to move the disks along with the VM.

Starting a live migration

A migration is started by the `Migrate` operation of the VirtualMachineOperation resource, which you create manually or with a `d8` command. You can interrupt the migration while it's in the `Pending` or `InProgress` phase by deleting that resource.

> ⚠ To keep the virtual machine schedulable, the node selector must not conflict with other placement rules, such as the virtual machine affinity, node selectors, and the node selector rules of the virtual machine class.

- Using the CLI
- Using the web interface

Before starting the migration, check the current status of the virtual machine:

```
d8 k get vm
```

Example output:

```
NAME       PHASE     UPTIME   NODE           IPADDRESS     AGE
linux-vm   Running   79m      virtlab-pt-1   10.66.10.14   79m
```

At this moment it runs on the `virtlab-pt-1` node.

To migrate a virtual machine from one node to another, taking its placement requirements into account, use the following command:

```
d8 v migrate -n <NAMESPACE> <VM_NAME> [--force] [--target-node-name string]
```

Running this command creates a VirtualMachineOperations resource.

The `--force` flag activates the AutoConverge mechanism when migrating a virtual machine. This mechanism automatically reduces the load on the virtual machine CPU (throttles it) if the migration has to be sped up to complete successfully, even when the VM memory transfer is too slow. Use this flag if a standard migration can't complete because of high VM activity.

To place the virtual machine on a specific target node, specify the name of that node in the `--target-node-name` option. For example, if the virtual machine has to be placed on the `production-1` node:

```
d8 v migrate -n project-1 linux-vm --target-node-name production-1
```

Under the hood, a virtual machine operation is created with the specific node selector `kubernetes.io/hostname: production-1`, where `production-1` is the node name.

You can also start a migration by manually creating a VirtualMachineOperation (`vmop`) resource of the `Migrate` type:

```
d8 k create -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineOperation
metadata:
  generateName: migrate-linux-vm-
  namespace: project-1
spec:
  # Virtual machine name.
  virtualMachineName: linux-vm
  # Operation for migration.
  type: Migrate
  # Defines the virtual machine migration operation.
  migrate:
    nodeSelector:
      # You can also set any suitable node selector.
      kubernetes.io/hostname: production-1
  # Allow CPU throttling by the AutoConverge mechanism to guarantee that the
migration completes.
  force: true
EOF
```

If you don’t need to specify target node parameters, you can omit the `migrate` field or evict the virtual machine to another suitable node using the `d8 v evict` command or by creating a VirtualMachineOperation resource of the `Evict` type.

To track the virtual machine migration right after the VirtualMachineOperation resource is created, run the following command:

```
d8 k get vm -w
```

Example output:

```
NAME       PHASE       UPTIME   NODE           IPADDRESS     AGE
linux-vm   Running     79m      virtlab-pt-1   10.66.10.14   79m
linux-vm   Migrating   79m      virtlab-pt-1   10.66.10.14   79m
linux-vm   Migrating   79m      virtlab-pt-1   10.66.10.14   79m
linux-vm   Running     79m      virtlab-pt-2   10.66.10.14   79m
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Virtual machines**.
3. Select the virtual machine you need from the list and click the ellipsis button.
4. In the menu that opens, select **Migrate**.
5. In the **Virtual machine migration** window that opens, select the mode:
   - **Migrate to an arbitrary node**: The scheduler picks the target node.
   - **Migrate to a selected node**: You pick the node manually in the **Nodes available for migration** field (the list contains only nodes that match the placement parameters of the VM and its class).
6. Enable additional options if required:
   - **Migrate disks**: Move the disks along with the VM (used when changing storage).
   - **Force (throttle guest CPU)**: Apply AutoConverge so that the migration completes even when network bandwidth is insufficient.
7. The window shows the current migration policy of the VM, for example “VM migration policy: PreferSafe”.
8. Click **Migrate** or cancel the operation with **Cancel**.

Configuring the migration policy

The migration policy determines when to use the AutoConverge mechanism (CPU throttling) to guarantee that a migration completes.

The AutoConverge mechanism helps a migration complete even with low network bandwidth, which makes a successful operation highly likely. However, it throttles the virtual machine CPU, which can affect the performance of applications running on the virtual machine.

The AutoConverge mechanism works in two stages:

1. **Throttling the virtual machine CPU**

   The hypervisor gradually lowers the CPU frequency of the source virtual machine. This reduces the rate at which new dirty pages appear. The higher the load on the virtual machine, the stronger the throttling.

2. **Automatic migration completion**

   As soon as the data transfer rate exceeds the memory change rate, the final synchronization starts and the virtual machine switches to the new node.

To configure the migration policy, use the `.spec.liveMigrationPolicy` parameter in the virtual machine configuration. Allowed values:

- `AlwaysSafe`: The migration always runs without CPU throttling (AutoConverge isn't used). Suitable when maximum virtual machine performance matters, but it requires high network bandwidth.
- `PreferSafe` (used as the default policy): The migration runs without CPU throttling (AutoConverge isn't used). However, you can start a migration with CPU throttling using a VirtualMachineOperation resource with the `type=Migrate` and `force=true` parameters.
- `AlwaysForced`: The migration always uses AutoConverge, that is, the CPU is throttled when required. This guarantees that the migration completes even on a poor network, but it can reduce virtual machine performance.
- `PreferForced`: The migration uses AutoConverge, that is, the CPU is throttled when required. However, you can start a migration without CPU throttling using a VirtualMachineOperation resource with the `type=Migrate` and `force=false` parameters.

Migrations with insufficient network bandwidth

During a live migration of a virtual machine, the network bandwidth may not be enough to transfer data faster than it changes in the virtual machine memory. In that case, the number of dirty pages keeps growing and the migration may not complete within the timeout.

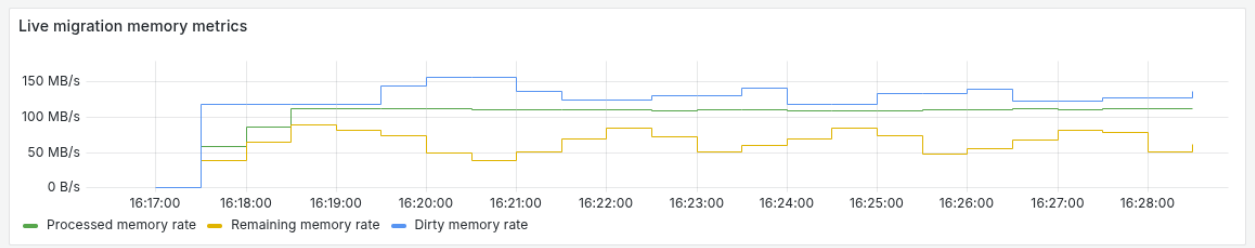
The AutoConverge mechanism, configured through the migration policy, solves this problem.

To tell that the network bandwidth isn't enough for a live migration of a virtual machine, check the charts in **Namespace / Virtual Machine → VM Status details → Live migration memory metrics**:

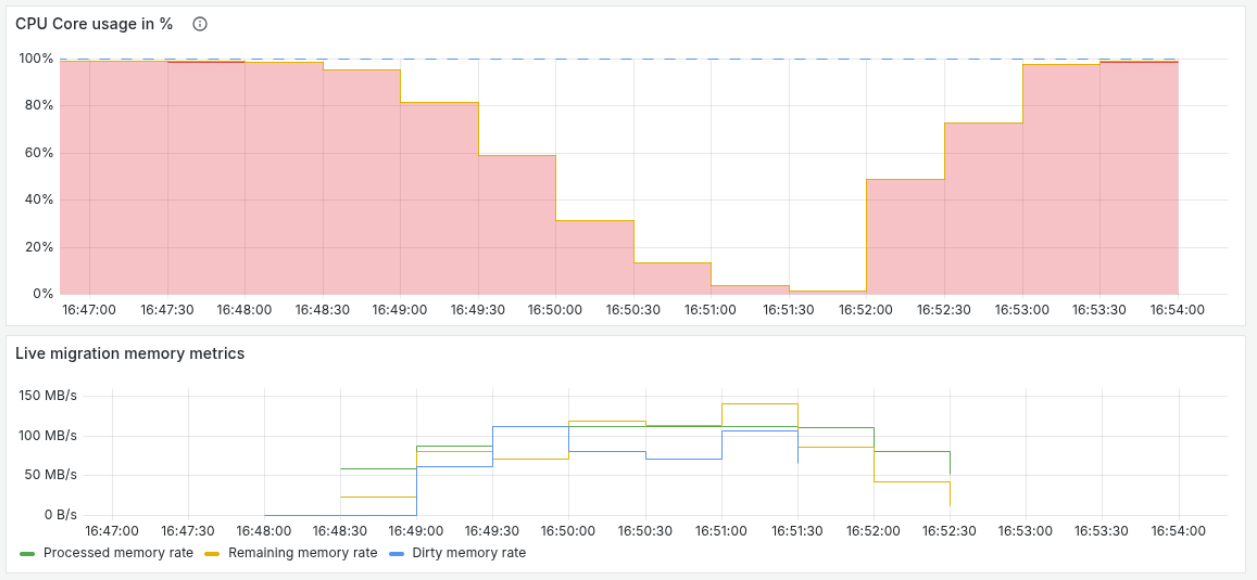
- **Processed memory rate** is lower than **Dirty memory rate**;
- **Remaining memory rate** doesn't decrease for a long time.

This means the network has become the bottleneck for the migration.

Here is an example of a situation where the migration can't complete because of insufficient network bandwidth. Inside the virtual machine, memory changes continuously with stress-ng.



Here is an example of migrating the same virtual machine with the `--force` flag of the `d8 v migrate` command (which enables the AutoConverge mechanism). You can clearly see that the CPU frequency is lowered in stages to reduce the rate of memory content changes.



If the network limits the migration speed, you can do the following:

1. Wait until the operation fails because of a timeout.
2. Cancel the current migration operation by deleting the VirtualMachineOperation resource, where `<VMOP_NAME>` is the name of that resource:

   ```
   d8 k delete vmop <VMOP_NAME>
   ```

3. Restart the migration with the `--force` flag to enable the AutoConverge mechanism. Using the `--force` flag has to match the current virtual machine migration policy.

Migrations started by the system

DP starts some migrations itself, by creating a VirtualMachineOperation resource of the `Evict` type. The prefix of the resource name shows what caused such a migration:

What caused the migration	Resource name prefix
A virtual machine firmware update	`firmware-update-`
Load redistribution in the cluster	`evacuation-`
Putting a node into maintenance mode (node drain)	`evacuation-`
A change in the VM placement parameters	`nodeplacement-update-`
A change of the core count or memory size without a restart	`hotplug-resources-`
Moving disks to another storage	`volume-migration-`

The migration has completed successfully when the resource moves to the `Completed` phase. The other phases are described in the `.status.phase` field, and to cancel a migration you delete the resource.

The following example shows how to view the list of such operations:

- Using the CLI
- Using the web interface

To view the active operations, run the following command:

```
d8 k get vmop
```

Example output:

```
NAME                     PHASE       PROGRESS   TYPE    VIRTUALMACHINE   AGE
firmware-update-fnbk2    Completed   100%       Evict   linux-vm         1m
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization → Virtual machines**.
3. Select the VM you need from the list and click its name.
4. Go to the **Operations** tab.

Live VM migration on a placement parameter change

When the placement rules of a running machine change, DP moves it to a suitable node with a live migration.

The following example shows the migration mechanism in a cluster with two node groups, `green` and `blue`. Suppose a virtual machine initially runs on a node of the `green` group, and its configuration has no placement restrictions.

First, add a requirement to be placed in the `green` group to the VM specification:

```
spec:
  nodeSelector:
    node.deckhouse.io/group: green
```

After you save the changes, the VM keeps running on the current node, because the `nodeSelector` condition is already met.

Now change the requirement to the `blue` group:

```
spec:
  nodeSelector:
    node.deckhouse.io/group: blue
```

The current node from the `green` group no longer meets the new conditions. DP creates a VirtualMachineOperation resource of the `Evict` type and starts a live migration of the VM to an available node of the `blue` group.

Example output:

```
NAME                          PHASE       PROGRESS   TYPE    VIRTUALMACHINE   AGE
nodeplacement-update-dabk4    Completed   100%       Evict   linux-vm         1m
```

Additional resources

- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Collecting debug information about a virtual machine

If a machine doesn’t behave as expected, collect its state and the state of the related resources into a single archive.

> The `collect-debug-info` command requires `d8` v0.27.0 or later.

The archive contains the following information:

- the virtual machine configuration;
- operations on the virtual machine;
- migration information;
- block devices;
- related PVCs and PVs;
- pods related to the VM, including their logs (the last 10000 lines);

- events for all related resources;
- the XML configuration of the VM domain.

The collected data is stored in the archive in YAML format (for resources) and in text files (for logs). You can send such an archive to technical support to analyze a problem.

The following example shows how to collect such an archive:

- Using the CLI
- Using the web interface

The `collect-debug-info` command collects diagnostic data about a virtual machine and all related resources into a compressed archive (tar.gz) that goes to stdout, so the output has to be redirected to a file.

Usage example:

```
# Collect debug information for the 'linux-vm' virtual machine
d8 v collect-debug-info linux-vm > debug-info.tar.gz

# Collect debug information for a VM with the namespace specified
d8 v collect-debug-info linux-vm -n mynamespace > debug-info.tar.gz

# Collect debug information for a VM with the full name specified (name.namespace)
d8 v collect-debug-info linux-vm.mynamespace > debug-info.tar.gz
```

Without the output redirected to a file, the command fails, and on success you get the `debug-info.tar.gz` archive.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Virtual machines**.
3. Select the VM you need from the list and click its name.
4. Go to the **Diagnostics** tab.
5. The **VM pods** block shows the pods of the virtual machine, their phase, and the node they’re placed on.
6. In the **VM pod logs** block, you can view the pod logs, filter the lines with a regular expression, and set the number of last lines.
7. To download the archive with diagnostic data, click **Download diagnostic data**.

Current and completed operations on the VM are shown on the **Operations** tab: for each operation, the date, the VirtualMachineOperation resource name, the type (**Start**, **Stop**, and others), the status, the progress, and the message are shown; you can limit the list to the **Day**, **Week**, or **Month** period. Events are shown on the **Events** tab, and resource consumption charts on the **Monitoring** tab.

Additional resources

- Module `virtualization` custom resources

> **i** **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Snapshots of disks and virtual machines

Snapshots let you capture the current state of a resource for later recovery or cloning. A disk snapshot saves only the data of the selected disk, while a virtual machine (VM) snapshot includes the VM parameters and the state of all its disks.

Consistent snapshots

Snapshots can be consistent or inconsistent. The `requiredConsistency` parameter is responsible for this, and its default value is `true`, which means that a consistent snapshot is required.

A consistent snapshot captures a coherent and integral state of the disk data. You can create such a snapshot when one of the following conditions is met:

- the disk isn’t attached to any virtual machine, and then the snapshot is always consistent;

- the virtual machine is powered off;
- `qemu-guest-agent` is installed and running in the guest OS. When the snapshot is created, it temporarily pauses (“freezes”) the file system to keep the data coherent.

An inconsistent snapshot may not reflect a coherent state of the virtual machine disks and its components. Such a snapshot is created if the VM is running and `qemu-guest-agent` isn’t installed or isn’t running in the guest OS. If the snapshot manifest explicitly specifies `requiredConsistency: false` but `qemu-guest-agent` is running, an attempt to freeze the file system is still made so that the snapshot comes out consistent.

QEMU Guest Agent supports hook scripts that prepare applications for a snapshot without stopping services, keeping the state coherent at the application level. Configuring hook scripts is described in Guest OS agent.

⚠ When recovering from such a snapshot, file system integrity problems are possible, because the data state may be incoherent.

Creating disk snapshots

A disk snapshot saves the disk data at the moment of creation and serves as a source for new disks, for example to clone or recover information. A snapshot is described by the VirtualDiskSnapshot resource.

Data integrity is guaranteed when a snapshot is created in one of the following cases:

- The disk isn’t attached to any virtual machine.
- The VM is powered off.
- The VM is running, qemu-guest-agent is installed in the guest OS, and the file system was successfully frozen (the fsfreeze operation).

If data consistency isn’t required (for example, for test scenarios), set the `.spec.requiredConsistency` parameter of the snapshot to `false`. The snapshot is then created on a running VM without freezing the file system, including when the disk is attached to it.

A disk snapshot goes through the following phases:

- `Pending`: Waiting for all dependent resources required to create the snapshot to become ready.
- `InProgress`: The virtual disk snapshot is being created.
- `Ready`: The snapshot was created successfully and the virtual disk snapshot is available for use.
- `Failed`: An error occurred while creating the virtual disk snapshot.
- `Terminating`: The resource is being deleted.

The `.status.conditions` block shows the reason for a problem with the resource.

- Using the CLI
- Using the web interface

Here is an example manifest for creating a disk snapshot:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualDiskSnapshot
metadata:
  name: linux-vm-root-snapshot
spec:
  requiredConsistency: true
  virtualDiskName: linux-vm-root
EOF
```

To view the list of disk snapshots, run the following command:

```
d8 k get vdsnapshot
```

Example output:

```
NAME                    PHASE     CONSISTENT   AGE
linux-vm-root-snapshot Ready     true         3m2s
```

The `true` value in the `CONSISTENT` column means that the snapshot is consistent. It’s determined

automatically from the snapshot creation conditions and isn't changed manually.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Disk snapshots**.
3. Click **Create**.
4. In the **Create resource** window that opens, enter the snapshot name in the **Name** field.
5. On the **Configuration** tab, select the disk to take the snapshot from in the **Virtual disk name** field.
6. Enable the **Required consistency** toggle.
7. Click **Apply**.
8. The snapshot status is shown in the **Status** column.

Recovering disks from snapshots

A new disk is created from a snapshot, and the original disk stays untouched.

- Using the CLI
- Using the web interface

To recover a disk from a previously created disk snapshot, specify the corresponding object as the `dataSource`:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualDisk
metadata:
  name: linux-vm-root
spec:
  # Disk storage parameters.
  persistentVolumeClaim:
    # Specify a size no smaller than the size of the source disk.
    size: 10Gi
    # Specify the name of your StorageClass.
    storageClassName: rv-thin-r2
  # The source the disk is created from.
  dataSource:
    type: ObjectRef
    objectRef:
      kind: VirtualDiskSnapshot
      name: linux-vm-root-snapshot
EOF
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Disks**.
3. Click **Create**.
4. In the form that opens, enter the disk name in the **Disk name** field.
5. In the **Source** field, select the disk snapshot you want to recover from in the drop-down list.
6. In the **Size** field, set a size equal to or larger than the size of the original disk.
7. In the **Storage class** field, select the StorageClass of the original disk.
8. Click **Create**.
9. The disk status is shown on its page.

Creating VM snapshots

A virtual machine snapshot is the saved state of a virtual machine at a certain point in time. To create virtual machine snapshots, use the VirtualMachineSnapshot resource.

> ⚠ Detach all images (VirtualImage/ClusterVirtualImage) from a virtual machine before taking its snapshot. Disk images aren't saved along with the VM snapshot, and their absence in the cluster during recovery can leave the virtual machine unable to start, in the Pending state, waiting for the image to become available.

Creating a virtual machine snapshot fails if at least one of the following conditions is met:

- not all dependent devices of the virtual machine are ready;
- one of the dependent devices is a disk that is being resized.

> ⚠ If the virtual machine has changes pending a restart at the moment the snapshot is taken, the updated configuration goes into the snapshot.

When a snapshot is created, the dynamic IP address of the VM is automatically converted to a static one and saved for recovery. If you don’t need the conversion and the reuse of the old address, set the `.spec.keepIPAddress` parameter to `Never`, and the address type stays as it is (`Auto` or `Static`).

The following example shows how to create a virtual machine snapshot:

- Using the CLI
- Using the web interface

Here is an example manifest for creating a virtual machine snapshot:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineSnapshot
metadata:
  name: linux-vm-snapshot
spec:
  virtualMachineName: linux-vm
  requiredConsistency: true
  keepIPAddress: Never
EOF
```

After the snapshot is created successfully, its status reflects the list of saved resources.

Example output:

```
status:
  ...
  resources:
  - apiVersion: virtualization.deckhouse.io/v1alpha2
    kind: VirtualMachine
    name: linux-vm
  - apiVersion: v1
    kind: Secret
    name: cloud-init
  - apiVersion: virtualization.deckhouse.io/v1alpha2
    kind: VirtualDisk
    name: linux-vm-root
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Virtual machines**.
3. Select the VM you need from the list and click its name.
4. Go to the **Snapshots** tab.
5. Click **Add**.
6. In the form that opens, enter `linux-vm-snapshot` in the **Snapshot name** field.
7. Enable the **Integrity guarantee** toggle.
8. Click **Create**.
9. The snapshot status is shown on its page.
10. The created snapshots are listed on the **Snapshots** tab of the virtual machine, with the **Name**, **Status**, **Creation date**, and **Consistent** columns.

Recovering a VM

Recovery returns a machine and its disks to the state saved in a snapshot. It’s started by a VirtualMachineOperation resource of the `Restore` type, which has three modes:

- `DryRun`: A dry run of the recovery operation, needed to check for possible conflicts, which are shown in the resource status (`status.resources`).
- `Strict`: The strict recovery mode, when the VM has to be recovered exactly as in the snapshot; missing external dependencies can leave the VM in `Pending` after recovery.
- `BestEffort`: Missing external dependencies (ClusterVirtualImage, VirtualImage) are ignored and removed from the VM configuration.

Recovering a virtual machine from a snapshot is possible only when all of the following conditions are met:

- The VM being recovered is present in the cluster (the VirtualMachine resource exists and its `.metadata.uid` matches the identifier used when the snapshot was created).
- The disks being recovered (identified by name) either aren't attached to other VMs or are absent from the cluster.
- The IP address being recovered either isn't taken by another VM or is absent from the cluster.
- The MAC addresses being recovered either aren't used by other VMs or are absent from the cluster.

The disks of the machine are recovered along with it, so the disk specification contains the `dataSource` parameter with a reference to the disk snapshot needed.

⚠ If some resources the VM depends on (for example, VirtualMachineClass, VirtualImage, ClusterVirtualImage) are absent from the cluster but existed at the moment the snapshot was created, the VM stays in the `Pending` state after recovery. In that case, edit the VM configuration manually and update or remove the missing dependencies.

⚠ Don't cancel a recovery operation from a snapshot, that is, don't delete the VirtualMachineOperation resource in the `InProgress` phase, because this can leave the virtual machine being recovered in an inconsistent state.

- Using the CLI
- Using the web interface

Recovery is started by a VirtualMachineOperation resource of the `Restore` type:

```
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineOperation
metadata:
  name: <VMOP_NAME>
spec:
  type: Restore
  virtualMachineName: <VM_NAME>
  restore:
    mode: DryRun | Strict | BestEffort
    virtualMachineSnapshotName: <VM_SNAPSHOT_NAME>
```

To view information about conflicts when recovering a VM from a snapshot, check the resource status:

```
d8 k get vmop <VMOP_NAME> -o json | jq '.status.resources'
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Virtual machines**.
3. Select the virtual machine you need from the list and click the ellipsis button.
4. In the menu that opens, select **Restore**.
5. In the **Machine recovery** window that opens, select the snapshot in the **Virtual machine snapshot name** field.
6. In the **Recovery mode** field, select `Strict` or `BestEffort`.
7. Click **Restore**.

Additional resources

- Module `virtualization` custom resources

ℹ **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**

The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Cloning virtual machines

A virtual machine (VM) clone is created either from an existing VM or from a previously created snapshot of that machine.

⚠ The cloned VM gets a new IP address for the cluster network and new MAC addresses for the additional network interfaces (if there are any), so after cloning you have to reconfigure the network parameters of the guest OS.

i Labels aren’t copied from the source VM to the clone. This prevents Service traffic (Services select VMs by labels) from being routed to the clone. If the clone has to be part of a Service, add the labels you need after cloning. For example:

```
d8 k label vm <VM_NAME> label-name=label-value
```

Cloning creates a copy of a VM, so the resources of the new VM have to have unique names. The `nameReplacements` and `customization` parameters are used for this:

- `nameReplacements`: Lets you replace the names of existing resources with new ones to avoid conflicts.
- `customization`: Sets a prefix or a suffix for the names of all cloned VM resources (disks, IP addresses, and so on).

Here is an example of renaming specific resources:

```
nameReplacements:
  - from:
      kind: VirtualMachine
      name: <OLD_VM_NAME>
    to:
      name: <NEW_VM_NAME>
  - from:
      kind: VirtualDisk
      name: <OLD_DISK_NAME>
    to:
      name: <NEW_DISK_NAME>
  ...
```

As a result, a VM named `<NEW_VM_NAME>` is created, and the specified resources are renamed according to the replacement rules.

Here is an example of adding a prefix or a suffix to all resources:

```
customization:
  namePrefix: <PREFIX>
  nameSuffix: <SUFFIX>
```

As a result, a VM named `<PREFIX><ORIGINAL_VM_NAME><SUFFIX>` is created, and all resources (disks, IP addresses, and so on) get the prefix and the suffix.

You can use one of three modes for the cloning operation:

- `DryRun`: A test run to check for possible conflicts. The results are shown in the `status.resources` field of the corresponding operation resource.
- `Strict`: The strict mode, which requires all resources with new names and their dependencies (for example, images) to be present in the VM being cloned.
- `BestEffort`: The mode in which missing external dependencies (for example, ClusterVirtualImage, VirtualImage) are automatically removed from the configuration of the VM being cloned.

To view information about the conflicts that arose during cloning, check the status of the operation resource:

```
# For cloning from an existing VM.
d8 k get vmop <VMOP_NAME> -o json | jq '.status.resources'

# For cloning from a VM snapshot.
d8 k get vmsop <VMSOP_NAME> -o json | jq '.status.resources'
```

Creating a clone of an existing VM

A clone is assembled from temporary snapshots of a machine, so you don’t have to stop it.

A VM is cloned using the VirtualMachineOperation resource with the `Clone` operation type, both for powered-off and for running virtual machines. When a running VM is cloned, a consistent snapshot is created automatically, and the clone is then built from it.

During cloning, temporary snapshots are created automatically for the virtual machine and all its disks, and the new VM is assembled from them. After cloning finishes, the temporary snapshots are deleted, so you won’t see them in the resource list. The specification of the cloned disks still keeps a reference (`dataSource`) to the corresponding snapshot, even though the snapshot itself no longer exists. This is expected behavior and doesn’t indicate a problem, because by the time the clone starts, all the necessary data has already been transferred to the new disks.

⚠ Set the `.spec.runPolicy: AlwaysOff` parameter in the configuration of the VM being cloned to prevent the clone from starting automatically: the clone inherits the behavior of the parent VM.

Before cloning, prepare the guest OS to avoid conflicts of unique identifiers and network settings.

Linux:

- clear `machine-id` with the `sudo truncate -s 0 /etc/machine-id` command (for systemd) or delete the `/var/lib/dbus/machine-id` file;
- delete the SSH host keys: `sudo rm -f /etc/ssh/ssh_host_*`;
- clear the network interface configurations (if static settings are used);
- clear the Cloud-Init cache (if it’s used): `sudo cloud-init clean`.

Windows:

- run generalization with `sysprep` using the `/generalize` parameter, or use tools to clear unique identifiers (SID, hostname, and so on).

- Using the CLI
- Using the web interface

To create a VM clone, use the following resource:

```
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineOperation
metadata:
  name: <VMOP_NAME>
spec:
  type: Clone
  virtualMachineName: <name of the VM to be cloned>
  clone:
    mode: DryRun | Strict | BestEffort
    nameReplacements: []
    customization: {}
```

The `nameReplacements` and `customization` parameters are set in the `.spec.clone` block.

The following example shows cloning a VM named `database` and the `database-root` disk attached to it.

An example with renaming specific resources:

```
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineOperation
metadata:
  name: clone-database
spec:
  type: Clone
  virtualMachineName: database
  clone:
    mode: Strict
    nameReplacements:
      - from:
          kind: VirtualMachine
          name: database
        to:
          name: database-clone
      - from:
          kind: VirtualDisk
          name: database-root
        to:
          name: database-clone-root
```

As a result, a VM named `database-clone` and a disk named `database-clone-root` are created.

An example with a prefix for all resources:

```
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineOperation
metadata:
  name: clone-database
spec:
  type: Clone
  virtualMachineName: database
  clone:
    mode: Strict
    customization:
      namePrefix: clone-
      nameSuffix: -prod
```

As a result, a VM named `clone-database-prod` and a disk named `clone-database-root-prod` are created.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **Virtual machines**.
3. Select the virtual machine you need from the list and click the ellipsis button.
4. In the menu that opens, select **Clone**.
5. In the **Machine cloning** window that opens, select the snapshot to create the clone from in the **Virtual machine snapshot name** field. The clone is created from a snapshot, so prepare the snapshot in advance.
6. In the **Cloning mode** field, select `Strict` or `BestEffort`.
7. If required, set new names for the clone resources in the **Customization** → **Resource renaming** block, specifying the resource type, the original name, and the new name.
8. Click **Clone**.

Creating a clone from a VM snapshot

A VM is cloned from a snapshot using the VirtualMachineSnapshotOperation resource with the `CreateVirtualMachine` operation type.

To create a VM clone from a snapshot, use the following resource:

```
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineSnapshotOperation
metadata:
  name: <VMSOP_NAME>
spec:
  type: CreateVirtualMachine
  virtualMachineSnapshotName: <name of the VM snapshot from which to clone>
  createVirtualMachine:
    mode: DryRun | Strict | BestEffort
    nameReplacements: []
    customization: {}
```

The `nameReplacements` and `customization` parameters are configured in the `.spec.createVirtualMachine` block (general description above).

To view the list of resources saved in a snapshot, run the following command:

```
d8 k get vmsnapshot <SNAPSHOT_NAME> -o jsonpath='{.status.resources}' | jq
```

> **i** When a VM is cloned from a snapshot, the disks related to it are also created from the corresponding snapshots, so the disk specification contains the `dataSource` parameter with a reference to the disk snapshot needed.

The following example shows cloning from a VM snapshot named `database-snapshot`, which contains the `database` VM and the `database-root` disk.

An example with renaming specific resources:

```
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineSnapshotOperation
metadata:
  name: clone-database-from-snapshot
spec:
  type: CreateVirtualMachine
  virtualMachineSnapshotName: database-snapshot
  createVirtualMachine:
    mode: Strict
    nameReplacements:
      - from:
          kind: VirtualMachine
          name: database
        to:
          name: database-clone
      - from:
          kind: VirtualDisk
          name: database-root
        to:
          name: database-clone-root
```

As a result, a VM named `database-clone` and a disk named `database-clone-root` are created.

An example with a prefix for all resources:

```
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachineSnapshotOperation
metadata:
  name: clone-database-from-snapshot
spec:
  type: CreateVirtualMachine
  virtualMachineSnapshotName: database-snapshot
  createVirtualMachine:
    mode: Strict
    customization:
      namePrefix: clone-
      nameSuffix: -prod
```

As a result, a VM named `clone-database-prod` and a disk named `clone-database-root-prod` are created.

Additional resources

- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

Virtual machine pools

The VirtualMachinePool resource maintains a given number of identical virtual machines (VMs) and lets you scale them through the `scale` subresource, HorizontalPodAutoscaler (HPA), or KEDA. The `virtualMachineTemplate.spec` field matches the regular `VirtualMachineSpec`, so a replica is no different from a manually created virtual machine.

> ⚠ The `Legacy` OS type isn't supported in a pool, because replicas are differentiated by initialization, which these operating systems don't have, so every replica would be a byte-for-byte copy of one disk, and for Windows guest operating systems that also means the same SID on the network. A pool template with `osType: Legacy` is rejected. Create such virtual machines individually.

The following example shows how to create a virtual machine pool:

- Using the CLI
- Using the web interface

Create a pool with the number of replicas you need and a virtual machine template. Pool disks are described in two blocks:

- `virtualDiskTemplates` describes each replica disk once, setting the `reclaim` policy, the size, and the data source.
- The `blockDeviceRefs` of the template references these disks by name with `kind: VirtualDisk` and sets the device order, that is, the boot order, exactly as in a regular VirtualMachine.

Every `virtualDiskTemplates` entry has to appear in `blockDeviceRefs` exactly once, otherwise DP rejects the pool. Disk template names are unique.

Besides replica disks, `blockDeviceRefs` can list shared VirtualImage and ClusterVirtualImage images, for example a single ISO or CD-ROM for all replicas. Such images are attached read-only, there's one of them for the whole pool, and they don't need an entry in `virtualDiskTemplates`.

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachinePool
metadata:
  name: runners
```

```
  namespace: ci
spec:
  replicas: 3
  scaleDownPolicy: NewestFirst
  virtualMachineTemplate:
    spec:
      runPolicy: AlwaysOn
      virtualMachineClassName: generic
      cpu:
        cores: 2
      memory:
        size: 4Gi
      # Cloud-init: each replica configures itself at first boot (identically for
all).
      provisioning:
        type: UserData
        userData: |
          #cloud-config
          users:
            - name: cloud
              sudo: ALL=(ALL) NOPASSWD:ALL
              ssh_authorized_keys:
                - <SSH_PUBLIC_KEY>
      # Devices and boot order (the first one is bootable). VirtualDisk entries
      # reference virtualDiskTemplates by name (per-replica, resolved by the
controller);
      # VirtualImage/ClusterVirtualImage is a shared read-only image for all
replicas.
      blockDeviceRefs:
        - kind: VirtualDisk
          name: root          # boot disk
        - kind: VirtualDisk
          name: cache
        - kind: ClusterVirtualImage
          name: tools-iso      # shared CD-ROM, attached to every replica
  # Per-replica disk parameters (reclaim/size/source). Each of them has to be listed
above.
  virtualDiskTemplates:
    # Writable root disk: one per replica, cloned from an image, deleted along with
the replica.
    - name: root
      reclaim:
        onScaleDown: Delete
      spec:
        persistentVolumeClaim:
          size: 30Gi
        dataSource:
          type: ObjectRef
          objectRef:
            kind: VirtualImage
            name: ubuntu
    # Reusable cache: survives a scale-down and is reattached on a scale-up.
    - name: cache
      reclaim:
        onScaleDown: Retain
        keep: 5
        ttl: 30m
      spec:
        persistentVolumeClaim:
          size: 50Gi
EOF
```

Replicas are named `<POOL>-<RANDOM>`. Disks follow the same scheme, and a per-replica disk (`Delete`) is named `<REPLICA>-<TEMPLATE>` (for example, `runners-1b2e84-root`), while a reusable one (`Retain`) gets the `<POOL>-<TEMPLATE>-<RANDOM>` name. To view the replicas, run `d8 k get vm -l vmpool.virtualization.deckhouse.io/pool=runners`.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **VM pools**.
3. Click **Create**.
4. In the **Create resource** window that opens, enter the pool name in the **Name** field.
5. On the **Configuration** tab, set the number of replicas in the **Replicas** field and the replica deletion policy in the **Scale Down Policy** field.
6. In the **Virtual Disk Templates** block, describe the replica disks, and in the **Virtual Machine Template** block, describe the virtual machine template.
7. Click **Apply**.

> **i** The pool form is built from the VirtualMachinePool resource specification, so the field names match the resource parameters. You can paste a ready specification on the **YAML** tab.

Attaching a shared CD-ROM (or any shared image) to all replicas

Besides per-replica disks, `blockDeviceRefs` can reference read-only images, ClusterVirtualImage or VirtualImage. Such an image is shared, and all replicas attach the same file, for example an ISO with tools or drivers. Images aren’t listed in `virtualDiskTemplates` (they have no per-replica state) and aren’t part of the bijection.

Add the image to `blockDeviceRefs` at the position you need in the boot order. For an installation ISO, put it before the disk, and for a CD-ROM with tools, after it:

```
spec:
  virtualMachineTemplate:
    spec:
      blockDeviceRefs:
        - kind: VirtualDisk         # Writable per-replica root disk, boots first.
          name: root
        - kind: ClusterVirtualImage   # Shared read-only CD-ROM, attached to every
replica.
          name: tools-iso
  virtualDiskTemplates:
    - name: root
      spec:
        persistentVolumeClaim:
          size: 30Gi
        dataSource:
          type: ObjectRef
          objectRef:
            kind: ClusterVirtualImage
            name: ubuntu
```

An image is attached to existing replicas the same way as any other device, and a change to `blockDeviceRefs` applies to a live replica the next time it’s recreated (rotation or scale-up).

Scaling the pool

The number of replicas in a pool changes either manually or automatically, by an autoscaler. A pool supports the standard `scale` subresource and publishes `status.selector`, so HPA reads CPU and memory metrics straight from the replicas without extra plumbing.

Besides CPU and memory, the pool works with custom metrics (`Pods`/`External` through `custom.metrics.k8s.io`/`external.metrics.k8s.io`) and with KEDA, for example to scale by the length of an external queue.

The `spec.scaleDownPolicy` field determines which replica is deleted on an unaddressed scale-down:

- `NewestFirst`: The youngest replicas are deleted first.
- `OldestFirst`: The oldest replicas are deleted first.
- `Explicit`: An unaddressed scale-down is forbidden; replicas can be removed only by name. Use it when only the caller knows which replica can be safely removed (for example, an idle one).

With `scaleDownPolicy: Explicit`, an autoscaler can only increase the number of replicas, and an unaddressed scale-down through the `scale` subresource is rejected.

- Using the CLI
- Using the web interface

To change the number of replicas manually, run the following command:

```
d8 k scale virtualmachinepool/runners -n ci --replicas=8
```

Here is an example of HPA configuration for a pool:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: runners
  namespace: ci
spec:
  scaleTargetRef:
    apiVersion: virtualization.deckhouse.io/v1alpha2
    kind: VirtualMachinePool
    name: runners
  minReplicas: 3
  maxReplicas: 20
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
```

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization** → **VM pools**.
3. Select the pool you need from the list and click its name.
4. On the **Configuration** tab, set the new value in the **Replicas** field.
5. Click **Apply**.
6. The scaling progress is shown in the pool list, in the **Status** and **Ready** columns.

Deleting specific replicas

By default, when a pool scales down, the controller picks which replica to delete itself.

To remove exactly the replicas you specify (and scale the pool down by that number), use the `scaleDownWith` subresource:

```
d8 k create --raw \

/apis/subresources.virtualization.deckhouse.io/v1alpha2/namespaces/ci/virtualmachinepools/runners/scaledownw
ith \
  -f - <<'EOF'
{"targets": ["runners-1b2e84", "runners-9c0d11"]}
EOF
```

A plain `d8 k delete vm` doesn’t scale the pool down, because the controller treats it as a lost replica and creates a replacement.

Reusable disks (reclaim)

The `reclaim` policy sets what happens to a replica disk when the replica is removed from the pool.

The `reclaim.onScaleDown` parameter of a `virtualDiskTemplates` element defines this behavior. `reclaim` is optional; if it isn’t set, the disk is treated as `Delete`.

- `Delete` (default): The disk belongs to the virtual machine and is deleted along with it; nothing is left after the replica.

- `Retain`: The disk belongs to the pool, survives the replica, and is reattached to the next one on a scale-up. It suits state that's expensive to recreate and has to survive VM recreation, so that scaling back up is warm rather than cold.

`keep` and `ttl` configure the pool of free `Retain` disks (they apply only to `Retain`):

- `keep`: How many recently freed disks to always keep warm for an instant scale-up. `ttl` doesn't apply to them.
- `ttl`: How long a free disk lives beyond the warm buffer before garbage collection.

Examples:

```
# Ephemeral disk: deleted along with the replica (Delete by default).
- name: root
  spec:
    persistentVolumeClaim: { size: 30Gi }
    dataSource: { type: ObjectRef, objectRef: { kind: VirtualImage, name: ubuntu } }

# Reusable disk: keep 3 warm for a fast scale-up, collect the rest after 1h of
idling.
- name: cache
  reclaim:
    onScaleDown: Retain
    keep: 3
    ttl: 1h
  spec:
    persistentVolumeClaim: { size: 100Gi }

# Reusable disk without a limit: always reused, never deleted automatically (no
ttl).
- name: data
  reclaim:
    onScaleDown: Retain
  spec:
    persistentVolumeClaim: { size: 20Gi }
```

Invalid combinations are rejected on creation and modification. The `keep` and `ttl` parameters are allowed only with `Retain`, and `keep > 0` requires `ttl`, because without `ttl` nothing is collected and `keep` has no effect. A `Retain` disk without `ttl` keeps all freed disks indefinitely; limit it with `ttl` if that isn't what you need.

Pool limitations and specifics

Here are the limitations and non-obvious pool behaviors worth remembering during operation.

- Removing an entry from `virtualDiskTemplates` deletes its disks. For `Retain` disks, this destroys reusable data, so remove a template only when it's no longer needed.
- The pool maintains the number of replicas, not their health. An existing but unhealthy VM isn't recreated, a restart at the VM level brings it back. A `Stopped` replica is preserved rather than replaced, and only a fully deleted replica is recreated.
- `Retain` disks are shared between replicas. On a scale-up, a new replica can get a freed disk of another replica along with its data; there's no hard binding between a replica and a disk.
- A change to `virtualDiskTemplates[].spec` affects only new disks, except for `size`, which grows existing ones (shrinking isn't allowed). `dataSource`, `storageClassName`, and the rest don't apply to disks that already exist.
- Each replica has its own copy of every disk from `virtualDiskTemplates`. A shared read-only image, VirtualImage or ClusterVirtualImage, for example a single ISO, can be attached to all replicas by listing it in the `blockDeviceRefs` of the template, while a writable disk isn't shared between replicas.
- An edit to the `blockDeviceRefs` of the template (reordering, adding, or removing a shared image) applies to new replicas; live replicas keep their current devices until they're recreated (rotation or scale-up), as with other template changes that require a restart.
- Template changes that require a restart apply only after the replica restarts, according to `.spec.disruptions.restartApprovalMode` in the template.

Additional resources

- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

USB devices in a virtual machine

DP supports USB device passthrough to virtual machines (VMs) using DRA (Dynamic Resource Allocation). The physical device is connected to a cluster node, and the virtual machine works with it as if the device were plugged into the machine itself.

An administrator connects the device to a node and makes it available to your namespace. After that, a USBDevice resource appears in the namespace, and you attach it to a virtual machine. If the device you need isn’t in the list, contact the administrator.

The administrator takes care of the node and cluster version requirements, so they don’t depend on you.

Project devices (USBDevice)

USBDevice is a namespaced resource that represents a USB device available for attaching to virtual machines in a given namespace. It appears automatically after an administrator assigns the device to the namespace.

An example of viewing the USB devices in a namespace:

```
d8 k get usbdevice -n my-project
```

Example output:

```
NAME              NODE     MANUFACTURER   PRODUCT       ATTACHED   AGE
logitech-webcam   node-2   Logitech       Webcam C920   False      10m
```

The resource keeps the vendor and product identifiers, the bus, the device number, the serial number, the speed, and the rest of the device details in the `.status.attributes` block.

USBDevice conditions

Two conditions in the `.status.conditions` block describe the device state.

The `Ready` condition shows whether the device is ready for use, and takes one of the following reasons:

- `Ready`: The device is ready for use.
- `NotReady`: The device exists but isn’t ready.
- `NotFound`: The device is absent from the node.

The `Attached` condition shows whether the device is attached to a virtual machine:

- `AttachedToVirtualMachine`: The device is attached to a VM.
- `Available`: The device is free and can be attached.
- `DetachedForMigration`: The device is detached for the duration of a VM migration and is attached again on the target node.
- `NoFreeUSBIPPort`: The device is requested by a virtual machine, but the target node has no free USBIP ports left, so the condition has the `False` status.

Attaching a USB device to a VM

A device is attached to and detached from a machine without stopping it. A USB device is automatically passed through over the network (USBIP) to the node where the machine runs, so you don’t have to place the machine on the node with the device manually.

> ⚠ During a VM migration, the USB device briefly disconnects and reconnects on the new node at the moment the machine switches over. If the migration fails, the device stays on the old node.

Infrastructure requirements, USBIP port limits, and device discovery on nodes are described in USB devices in virtual machines.

- Using the CLI
- Using the web interface

Add the device to the `.spec.usbDevices` parameter of the VirtualMachine resource:

```
d8 k apply -f - <<EOF
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachine
metadata:
  name: linux-vm
spec:
  # ... other VM settings ...
  usbDevices:
    - name: logitech-webcam
EOF
```

After the VM is created or updated, the USB device is attached to the specified virtual machine.

1. Go to the **Projects** tab and select the project you need.
2. Go to **Virtualization → Virtual machines**.
3. Select the VM you need from the list and click its name.
4. On the **Configuration** tab, scroll down to the **USB devices** section and click **Add**.
5. In the **Attach USB device** window that opens, select the device in the **Select USB device** field and click **Add**.
6. Click the **Save** button that appears.

The USB devices available in the project are shown in **Virtualization → USB devices**: the resource name, status, manufacturer, product, serial number, node, bus, and device number.

Additional resources

- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

GPU devices in a virtual machine

DP attaches physical GPU devices to virtual machines using DRA (Dynamic Resource Allocation). A device is requested by a reference to a `GPUClass` in the `.spec.gpus` block of the VirtualMachine resource.

An administrator prepares the `GPUClass` resources, so ask them which classes are available in the cluster.

To request a GPU device, add the `.spec.gpus` block to the machine specification:

```
apiVersion: virtualization.deckhouse.io/v1alpha2
kind: VirtualMachine
metadata:
  name: linux-vm
spec:
  # ... other VM settings ...
  gpus:
    - gpuClassName: nvidia-h100
```

In the `gpuClassName` parameter, specify the name of an existing `GPUClass` resource. To attach several devices, add more elements to the list, their order doesn't matter. A single machine takes no more than 16 devices.

A change to the `.spec.gpus` block applies only after the virtual machine restarts.

Additional resources

- Module `virtualization` custom resources

> **Starting from version 1.76, Deckhouse Kubernetes Platform has been renamed to Deckhouse Platform.**
>
> The platform now offers more features, and editions have changed. We are already updating the documentation, but if you have any questions, please contact your account manager or technical support.

PCI devices in a virtual machine

PCI device passthrough lets you use a physical device of a node in a virtual machine (VM), for example an industrial controller, a hardware security module, a capture card, or an FPGA. The device works in the guest operating system under its own driver, so install that driver in the guest system yourself.

Devices are granted to a project by the administrator, who also handles the requirements for the nodes and the cluster versions.

Attaching a PCI device to a VM

The devices that the administrator has made available to your project appear in the namespace as PCIDevice resources. To attach such a device to a machine, follow these steps.

1. Choose a device among the available ones:

   ```
   d8 k get pcidevice -n my-project
   ```

   Example output:

   ```
   NAME                                                NODE     ADDRESS
   ATTACHED   AGE
   pci-4f2c0b1e8d9a3c5b7e1f0a2d4c6b8e0f1a3c5d7e    node-1   0000:3b:00.0   False
   10m
   ```

   If the list is empty, contact the administrator so that they assign a device to your namespace.

2. Add the device to the `.spec.pciDevices` parameter of the VirtualMachine resource:

   ```
   d8 k apply -f - <<EOF
   apiVersion: virtualization.deckhouse.io/v1alpha2
   kind: VirtualMachine
   metadata:
     name: linux-vm
   spec:
     # ... other VM settings ...
     pciDevices:
       - name: pci-4f2c0b1e8d9a3c5b7e1f0a2d4c6b8e0f1a3c5d7e
   EOF
   ```

   Devices are attached when the machine starts, so a change to this parameter requires its restart.

3. Make sure that the device is attached to the machine:

   ```
   d8 k get vm linux-vm -o jsonpath='{.status.pciDevices}'
   ```

   The `true` value in the `ready` field means that the device is available, and the `true` value in the `attached` field means that the device is attached to the running machine. Once the machine starts, the device appears in the guest system, for example in the `lspci` output.

To detach a device, remove it from the `.spec.pciDevices` parameter and restart the machine. Until then, the device stays occupied even if the machine is stopped, so you can’t attach it to another one.

If the device is removed from the node, the machine doesn’t start and stays in the `Pending` phase. In the `.status.pciDevices` parameter, such a device stops being ready, but it stays with the machine as long as its specification lists it. DP doesn’t stop a running machine in the meantime, including when the device becomes unavailable to the project.

Limitations

When attaching PCI devices, consider the following limitations:

- A machine with a PCI device starts only on the node of that device and can’t be moved by live migration, so it has to be stopped when the node is put into maintenance.
- All PCI devices of a machine have to be on the same node, otherwise the specification is rejected.
- A device is attached to one machine only. If another one already lists it, DP rejects your specification.
- A machine takes no more than eight PCI devices.
- A network card that’s passed through works around the cluster network subsystem. It doesn’t appear in the `.spec.networks` parameter, and DP neither assigns nor tracks its IP and MAC addresses.

Additional resources

- Module `virtualization` custom resources