Best practices

Considerations for large clusters

Running in multiple zones

Validate node setup

Enforcing Pod Security Standards

PKI certificates and requirements

Enforcing Pod Security Standards

This page provides an overview of best practices when it comes to enforcing Pod Security Standards.

Using the built-in Pod Security Admission Controller

FEATURE STATE: `Kubernetes v1.25 [stable]`

The Pod Security Admission Controller intends to replace the deprecated PodSecurityPolicies.

Configure all cluster namespaces

Namespaces that lack any configuration at all should be considered significant gaps in your cluster security model. We recommend taking the time to analyze the types of workloads occurring in each namespace, and by referencing the Pod Security Standards, decide on an appropriate level for each of them. Unlabeled namespaces should only indicate that they've yet to be evaluated.

In the scenario that all workloads in all namespaces have the same security requirements, we provide an example that illustrates how the PodSecurity labels can be applied in bulk.

Embrace the principle of least privilege

In an ideal world, every pod in every namespace would meet the requirements of the `restricted` policy. However, this is not possible nor practical, as some workloads will require elevated privileges for legitimate reasons.

- Namespaces allowing `privileged` workloads should establish and enforce appropriate access controls.
- For workloads running in those permissive namespaces, maintain documentation about their unique security requirements. If at all possible, consider how those requirements could be further constrained.

Adopt a multi-mode strategy

The `audit` and `warn` modes of the Pod Security Standards admission controller make it easy to collect important security insights about your pods without breaking existing workloads.

It is good practice to enable these modes for all namespaces, setting them to the *desired* level and version you would eventually like to `enforce`. The warnings and audit annotations generated in this phase can guide you toward that state. If you expect workload authors to make changes to fit within the desired level, enable the `warn` mode. If you expect to use audit logs to monitor/drive changes to fit within the desired level, enable the `audit` mode.

When you have the `enforce` mode set to your desired value, these modes can still be useful in a few different ways:

- By setting `warn` to the same level as `enforce`, clients will receive warnings when attempting to create Pods (or resources that have Pod templates) that do not pass validation. This will help them update those resources to become compliant.
- In Namespaces that pin `enforce` to a specific non-latest version, setting the `audit` and `warn` modes to the same level as `enforce`, but to the `latest` version, gives visibility into settings that were allowed by previous versions but are not allowed per current best practices.

Third-party alternatives

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

Other alternatives for enforcing security profiles are being developed in the Kubernetes ecosystem:

- Kubewarden.
- Kyverno.
- OPA Gatekeeper.

The decision to go with a *built-in* solution (e.g. PodSecurity admission controller) versus a third-party tool is entirely dependent on your own situation. When evaluating any solution, trust of your supply chain is crucial. Ultimately, using *any* of the aforementioned approaches will be better than doing nothing.

Running in multiple zones

This page describes running Kubernetes across multiple zones.

Background

Kubernetes is designed so that a single Kubernetes cluster can run across multiple failure zones, typically where these zones fit within a logical grouping called a *region*. Major cloud providers define a region as a set of failure zones (also called *availability zones*) that provide a consistent set of features: within a region, each zone offers the same APIs and services.

Typical cloud architectures aim to minimize the chance that a failure in one zone also impairs services in another zone.

Control plane behavior

All control plane components support running as a pool of interchangeable resources, replicated per component.

When you deploy a cluster control plane, place replicas of control plane components across multiple failure zones. If availability is an important concern, select at least three failure zones and replicate each individual control plane component (API server, scheduler, etcd, cluster controller manager) across at least three failure zones. If you are running a cloud controller manager then you should also replicate this across all the failure zones you selected.

Note:

Kubernetes does not provide cross-zone resilience for the API server endpoints. You can use various techniques to improve availability for the cluster API server, including DNS round-robin, SRV records, or a third-party load balancing solution with health checking.

Node behavior

Kubernetes automatically spreads the Pods for workload resources (such as Deployment or StatefulSet) across different nodes in a cluster. This spreading helps reduce the impact of failures.

When nodes start up, the kubelet on each node automatically adds labels to the Node object that represents that specific kubelet in the Kubernetes API. These labels can include zone information.

If your cluster spans multiple zones or regions, you can use node labels in conjunction with Pod topology spread constraints to control how Pods are spread across your cluster among fault domains: regions, zones, and even specific nodes. These hints enable the scheduler to place Pods for better expected availability, reducing the risk that a correlated failure affects your whole workload.

For example, you can set a constraint to make sure that the 3 replicas of a StatefulSet are all running in different zones to each other, whenever that is feasible. You can define this declaratively without explicitly defining which availability zones are in use for each workload.

Distributing nodes across zones

Kubernetes' core does not create nodes for you; you need to do that yourself, or use a tool such as the Cluster API to manage nodes on your behalf.

Using tools such as the Cluster API you can define sets of machines to run as worker nodes for your cluster across multiple failure domains, and rules to automatically heal the cluster in case of whole-zone service disruption.

Manual zone assignment for Pods

You can apply node selector constraints to Pods that you create, as well as to Pod templates in workload resources such as Deployment, StatefulSet, or Job.

Storage access for zones

When persistent volumes are created, Kubernetes automatically adds zone labels to any PersistentVolumes that are linked to a specific zone. The scheduler then ensures, through its `NoVolumeZoneConflict` predicate, that pods which claim a given PersistentVolume are only placed into the same zone as that volume.

Please note that the method of adding zone labels can depend on your cloud provider and the storage provisioner you're using. Always refer to the specific documentation for your environment to ensure correct configuration.

You can specify a StorageClass for PersistentVolumeClaims that specifies the failure domains (zones) that the storage in that class may use. To learn about configuring a StorageClass that is aware of failure domains or zones, see Allowed topologies.

Networking

By itself, Kubernetes does not include zone-aware networking. You can use a network plugin to configure cluster networking, and that network solution might have zone-specific elements. For example, if your cloud provider supports Services with `type=LoadBalancer`, the load balancer might only send traffic to Pods running in the same zone as the load balancer element processing a given connection. Check your cloud provider's documentation for details.

For custom or on-premises deployments, similar considerations apply. Service and Ingress behavior, including handling of different failure zones, does vary depending on exactly how your cluster is set up.

Fault recovery

When you set up your cluster, you might also need to consider whether and how your setup can restore service if all the failure zones in a region go off-line at the same time. For example, do you rely on there being at least one node able to run Pods in a zone?
Make sure that any cluster-critical repair work does not rely on there being at least one healthy node in your cluster. For example: if all nodes are unhealthy, you might need to run a repair Job with a special toleration so that the repair can complete enough to bring at least one node into service.

Kubernetes doesn't come with an answer for this challenge; however, it's something to consider.

What's next

To learn how the scheduler places Pods in a cluster, honoring the configured constraints, visit Scheduling and Eviction.

PKI certificates and requirements

Kubernetes requires PKI certificates for authentication over TLS. If you install Kubernetes with kubeadm, the certificates that your cluster requires are automatically generated. You can also generate your own certificates -- for example, to keep your private keys more secure by not storing them on the API server. This page explains the certificates that your cluster requires.

How certificates are used by your cluster

Kubernetes requires PKI for the following operations:

Server certificates

- Server certificate for the API server endpoint
- Server certificate for the etcd server
- Server certificates for each kubelet (every node runs a kubelet)
- Optional server certificate for the front-proxy

Client certificates

- Client certificates for each kubelet, used to authenticate to the API server as a client of the Kubernetes API
- Client certificate for each API server, used to authenticate to etcd
- Client certificate for the controller manager to securely communicate with the API server
- Client certificate for the scheduler to securely communicate with the API server
- Client certificates, one for each node, for kube-proxy to authenticate to the API server
- Optional client certificates for administrators of the cluster to authenticate to the API server
- Optional client certificate for the front-proxy

Kubelet's server and client certificates

To establish a secure connection and authenticate itself to the kubelet, the API Server requires a client certificate and key pair.

In this scenario, there are two approaches for certificate usage:

- Shared Certificates: The kube-apiserver can utilize the same certificate and key pair it uses to authenticate its clients. This means that the existing certificates, such as `apiserver.crt` and `apiserver.key`, can be used for communicating with the kubelet servers.

- Separate Certificates: Alternatively, the kube-apiserver can generate a new client certificate and key pair to authenticate its communication with the kubelet servers. In this case, a distinct certificate named `kubelet-client.crt` and its corresponding private key, `kubelet-client.key` are created.

Note:

`front-proxy` certificates are required only if you run kube-proxy to support an extension API server.

etcd also implements mutual TLS to authenticate clients and peers.

Where certificates are stored

If you install Kubernetes with kubeadm, most certificates are stored in `/etc/kubernetes/pki`. All paths in this documentation are relative to that directory, with the exception of user account certificates which kubeadm places in `/etc/kubernetes`.

Configure certificates manually

If you don't want kubeadm to generate the required certificates, you can create them using a single root CA or by providing all certificates. See Certificates for details on creating your own certificate authority. See Certificate Management with kubeadm for more on managing certificates.

Single root CA

You can create a single root CA, controlled by an administrator. This root CA can then create multiple intermediate CAs, and delegate all further creation to Kubernetes itself.

Required CAs:

Path	Default CN	Description
ca.crt,key	kubernetes-ca	Kubernetes general CA
etcd/ca.crt,key	etcd-ca	For all etcd-related functions
front-proxy-ca.crt,key	kubernetes-front-proxy-ca	For the front-end proxy

On top of the above CAs, it is also necessary to get a public/private key pair for service account management, `sa.key` and `sa.pub`. The following example illustrates the CA key and certificate files shown in the previous table:

```
/etc/kubernetes/pki/ca.crt
/etc/kubernetes/pki/ca.key
/etc/kubernetes/pki/etcd/ca.crt
/etc/kubernetes/pki/etcd/ca.key
/etc/kubernetes/pki/front-proxy-ca.crt
/etc/kubernetes/pki/front-proxy-ca.key
```

All certificates

If you don't wish to copy the CA private keys to your cluster, you can generate all certificates yourself.

Required certificates:

Default CN	Parent CA	O (in Subject)	kind	hosts (SAN)
kube-etcd	etcd-ca		server, client	`<hostname>`, `<Host_IP>`, `localhost`, `127.0.0.1`
kube-etcd-peer	etcd-ca		server, client	`<hostname>`, `<Host_IP>`, `localhost`, `127.0.0.1`
kube-etcd-healthcheck-client	etcd-ca		client	
kube-apiserver-etcd-client	etcd-ca		client	
kube-apiserver	kubernetes-ca		server	`<hostname>`, `<Host_IP>`, `<advertise_IP>`[1]
kube-apiserver-kubelet-client	kubernetes-ca	system:masters	client	
front-proxy-client	kubernetes-front-proxy-ca		client	

Note:

Instead of using the super-user group `system:masters` for `kube-apiserver-kubelet-client` a less privileged group can be used. kubeadm uses the `kubeadm:cluster-admins` group for that purpose.

where `kind` maps to one or more of the x509 key usage, which is also documented in the `.spec.usages` of a CertificateSigningRequest type:

kind	Key usage
server	digital signature, key encipherment, server auth
client	digital signature, key encipherment, client auth

Note:

Hosts/SAN listed above are the recommended ones for getting a working cluster; if required by a specific setup, it is possible to add additional SANs on all the server certificates.

Note:

For kubeadm users only:

- The scenario where you are copying to your cluster CA certificates without private keys is referred as external CA in the kubeadm documentation.
- If you are comparing the above list with a kubeadm generated PKI, please be aware that `kube-etcd`, `kube-etcd-peer` and `kube-etcd-healthcheck-client` certificates are not generated in case of external etcd.

Certificate paths

Certificates should be placed in a recommended path (as used by kubeadm). Paths should be specified using the given argument regardless of location.

DefaultCN	recommendedkeypath	recommendedcertpath	command	keyargument	certargument
etcd-ca	etcd/ca.key	etcd/ca.crt	kube-apiserver		--etcd-cafile
kube-apiserver-etcd-client	apiserver-etcd-client.key	apiserver-etcd-client.crt	kube-apiserver	--etcd-keyfile	--etcd-certfile
kubernetes-ca	ca.key	ca.crt	kube-apiserver		--client-ca-file
kubernetes-ca	ca.key	ca.crt	kube-controller-manager	--cluster-signing-key-file	--client-ca-file,--root-ca-file,--cluster-signing-cert-file
kube-apiserver	apiserver.key	apiserver.crt	kube-apiserver	--tls-private-key-file	--tls-cert-file
kube-apiserver-kubelet-client	apiserver-kubelet-client.key	apiserver-kubelet-client.crt	kube-apiserver	--kubelet-client-key	--kubelet-client-certificate
front-proxy-ca	front-proxy-ca.key	front-proxy-ca.crt	kube-apiserver		--requestheader-client-ca-file
front-proxy-ca	front-proxy-ca.key	front-proxy-ca.crt	kube-controller-manager		--requestheader-client-ca-file
front-proxy-client	front-proxy-client.key	front-proxy-client.crt	kube-apiserver	--proxy-client-key-file	--proxy-client-cert-file
etcd-ca	etcd/ca.key	etcd/ca.crt	etcd		--trusted-ca-file,--peer-trusted-ca-file

DefaultCN	recommendedkeypath	recommendedcertpath	command	keyargument	certargument
kube-etcd	etcd/server.key	etcd/server.crt	etcd	--key-file	--cert-file
kube-etcd-peer	etcd/peer.key	etcd/peer.crt	etcd	--peer-key-file	--peer-cert-file
etcd-ca		etcd/ca.crt	etcdctl		--cacert
kube-etcd-healthcheck-client	etcd/healthcheck-client.key	etcd/healthcheck-client.crt	etcdctl	--key	--cert

Same considerations apply for the service account key pair:

private key path	public key path	command	argument
sa.key		kube-controller-manager	--service-account-private-key-file
	sa.pub	kube-apiserver	--service-account-key-file

The following example illustrates the file paths from the previous tables you need to provide if you are generating all of your own keys and certificates:

```
/etc/kubernetes/pki/etcd/ca.key
/etc/kubernetes/pki/etcd/ca.crt
/etc/kubernetes/pki/apiserver-etcd-client.key
/etc/kubernetes/pki/apiserver-etcd-client.crt
/etc/kubernetes/pki/ca.key
/etc/kubernetes/pki/ca.crt
/etc/kubernetes/pki/apiserver.key
/etc/kubernetes/pki/apiserver.crt
/etc/kubernetes/pki/apiserver-kubelet-client.key
/etc/kubernetes/pki/apiserver-kubelet-client.crt
/etc/kubernetes/pki/front-proxy-ca.key
/etc/kubernetes/pki/front-proxy-ca.crt
/etc/kubernetes/pki/front-proxy-client.key
/etc/kubernetes/pki/front-proxy-client.crt
/etc/kubernetes/pki/etcd/server.key
/etc/kubernetes/pki/etcd/server.crt
/etc/kubernetes/pki/etcd/peer.key
/etc/kubernetes/pki/etcd/peer.crt
/etc/kubernetes/pki/etcd/healthcheck-client.key
/etc/kubernetes/pki/etcd/healthcheck-client.crt
/etc/kubernetes/pki/sa.key
/etc/kubernetes/pki/sa.pub
```

Configure certificates for user accounts

You must manually configure these administrator accounts and service accounts:

Filename	Credential name	Default CN	O (in Subject)
admin.conf	default-admin	kubernetes-admin	<admin-group>
super-admin.conf	default-super-admin	kubernetes-super-admin	system:masters
kubelet.conf	default-auth	system:node:<nodeName> (see note)	system:nodes
controller-manager.conf	default-controller-manager	system:kube-controller-manager	
scheduler.conf	default-scheduler	system:kube-scheduler	

Note:

The value of <nodeName> for kubelet.conf **must** match precisely the value of the node name provided by the kubelet as it registers with the apiserver. For further details, read the Node Authorization.

Note:

In the above example <admin-group> is implementation specific. Some tools sign the certificate in the default admin.conf to be part of the system:masters group. system:masters is a break-glass, super user group can bypass the authorization layer of Kubernetes, such as RBAC. Also some tools do not generate a separate super-admin.conf with a certificate bound to this super user group.

kubeadm generates two separate administrator certificates in kubeconfig files. One is in admin.conf and has Subject: O = kubeadm:cluster-admins, CN = kubernetes-admin. kubeadm:cluster-admins is a custom group bound to the cluster-admin ClusterRole. This file is generated on all kubeadm managed control plane machines.

Another is in super-admin.conf that has Subject: O = system:masters, CN = kubernetes-super-admin. This file is generated only on the node where kubeadm init was called.

1. For each configuration, generate an x509 certificate/key pair with the given Common Name (CN) and Organization (O).

2. Run kubectl as follows for each configuration:

```
KUBECONFIG=<filename> kubectl config set-cluster default-cluster --server=https://<host ip>:6443 --certificate-authority <pat
KUBECONFIG=<filename> kubectl config set-credentials <credential-name> --client-key <path-to-key>.pem --client-certificate <p
KUBECONFIG=<filename> kubectl config set-context default-system --cluster default-cluster --user <credential-name>
KUBECONFIG=<filename> kubectl config use-context default-system
```

These files are used as follows:

Filename	Command	Comment
admin.conf	kubectl	Configures administrator user for the cluster
super-admin.conf	kubectl	Configures super administrator user for the cluster

Filename	Command	Comment
kubelet.conf	kubelet	One required for each node in the cluster.
controller-manager.conf	kube-controller-manager	Must be added to manifest in `manifests/kube-controller-manager.yaml`
scheduler.conf	kube-scheduler	Must be added to manifest in `manifests/kube-scheduler.yaml`

The following files illustrate full paths to the files listed in the previous table:

```
/etc/kubernetes/admin.conf
/etc/kubernetes/super-admin.conf
/etc/kubernetes/kubelet.conf
/etc/kubernetes/controller-manager.conf
/etc/kubernetes/scheduler.conf
```

---

1. any other IP or DNS name you contact your cluster on (as used by kubeadm the load balancer stable IP and/or DNS name, `kubernetes`, `kubernetes.default`, `kubernetes.default.svc`, `kubernetes.default.svc.cluster`, `kubernetes.default.svc.cluster.local`) ↩

Validate node setup

Node Conformance Test

Node conformance test is a containerized test framework that provides a system verification and functionality test for a node. The test validates whether the node meets the minimum requirements for Kubernetes; a node that passes the test is qualified to join a Kubernetes cluster.

Node Prerequisite

To run node conformance test, a node must satisfy the same prerequisites as a standard Kubernetes node. At a minimum, the node should have the following daemons installed:

- CRI-compatible container runtimes such as Docker, containerd and CRI-O
- kubelet

Running Node Conformance Test

To run the node conformance test, perform the following steps:

1. Work out the value of the `--kubeconfig` option for the kubelet; for example: `--kubeconfig=/var/lib/kubelet/config.yaml`. Because the test framework starts a local control plane to test the kubelet, use `http://localhost:8080` as the URL of the API server. There are some other kubelet command line parameters you may want to use:

    - `--cloud-provider`: If you are using `--cloud-provider=gce`, you should remove the flag to run the test.

2. Run the node conformance test with command:

```
# $CONFIG_DIR is the pod manifest path of your kubelet.
# $LOG_DIR is the test output path.
sudo docker run -it --rm --privileged --net=host \
  -v /:/rootfs -v $CONFIG_DIR:$CONFIG_DIR -v $LOG_DIR:/var/result \  registry.k8s.io/node-test:0.2
```

Running Node Conformance Test for Other Architectures

Kubernetes also provides node conformance test docker images for other architectures:

Arch	Image
amd64	node-test-amd64
arm	node-test-arm
arm64	node-test-arm64

Running Selected Test

To run specific tests, overwrite the environment variable `FOCUS` with the regular expression of tests you want to run.

```
sudo docker run -it --rm --privileged --net=host \
  -v /:/rootfs:ro -v $CONFIG_DIR:$CONFIG_DIR -v $LOG_DIR:/var/result \  -e FOCUS=MirrorPod \ # Only run MirrorPod test  registry.k
```

To skip specific tests, overwrite the environment variable `SKIP` with the regular expression of tests you want to skip.

```
sudo docker run -it --rm --privileged --net=host \
  -v /:/rootfs:ro -v $CONFIG_DIR:$CONFIG_DIR -v $LOG_DIR:/var/result \  -e SKIP=MirrorPod \ # Run all conformance tests but skip M
```

Node conformance test is a containerized version of node e2e test. By default, it runs all conformance tests.

Theoretically, you can run any node e2e test if you configure the container and mount required volumes properly. But **it is strongly recommended to only run conformance test**, because it requires much more complex configuration to run non-conformance test.

Caveats

- The test leaves some docker images on the node, including the node conformance test image and images of containers used in the functionality test.
- The test leaves dead containers on the node. These containers are created during the functionality test.

Considerations for large clusters

A cluster is a set of nodes (physical or virtual machines) running Kubernetes agents, managed by the control plane. Kubernetes v1.34 supports clusters with up to 5,000 nodes. More specifically, Kubernetes is designed to accommodate configurations that meet *all* of the following criteria:

- No more than 110 pods per node
- No more than 5,000 nodes
- No more than 150,000 total pods
- No more than 300,000 total containers

You can scale your cluster by adding or removing nodes. The way you do this depends on how your cluster is deployed.

Cloud provider resource quotas

To avoid running into cloud provider quota issues, when creating a cluster with many nodes, consider:

- Requesting a quota increase for cloud resources such as:
  - Computer instances
  - CPUs
  - Storage volumes
  - In-use IP addresses
  - Packet filtering rule sets
  - Number of load balancers
  - Network subnets
  - Log streams
- Gating the cluster scaling actions to bring up new nodes in batches, with a pause between batches, because some cloud providers rate limit the creation of new instances.

Control plane components

For a large cluster, you need a control plane with sufficient compute and other resources.

Typically you would run one or two control plane instances per failure zone, scaling those instances vertically first and then scaling horizontally after reaching the point of falling returns to (vertical) scale.

You should run at least one instance per failure zone to provide fault-tolerance. Kubernetes nodes do not automatically steer traffic towards control-plane endpoints that are in the same failure zone; however, your cloud provider might have its own mechanisms to do this.

For example, using a managed load balancer, you configure the load balancer to send traffic that originates from the kubelet and Pods in failure zone *A*, and direct that traffic only to the control plane hosts that are also in zone *A*. If a single control-plane host or endpoint failure zone *A* goes offline, that means that all the control-plane traffic for nodes in zone *A* is now being sent between zones. Running multiple control plane hosts in each zone makes that outcome less likely.

etcd storage

To improve performance of large clusters, you can store Event objects in a separate dedicated etcd instance.

When creating a cluster, you can (using custom tooling):

- start and configure additional etcd instance
- configure the API server to use it for storing events

See Operating etcd clusters for Kubernetes and Set up a High Availability etcd cluster with kubeadm for details on configuring and managing etcd for a large cluster.

Addon resources

Kubernetes resource limits help to minimize the impact of memory leaks and other ways that pods and containers can impact on other components. These resource limits apply to addon resources just as they apply to application workloads.

For example, you can set CPU and memory limits for a logging component:

```
...
containers:
- name: fluentd-cloud-logging
  image: fluent/fluentd-kubernetes-daemonset:v1
  resources:
    limits:
      cpu: 100m
      memory: 200Mi
```

Addons' default limits are typically based on data collected from experience running each addon on small or medium Kubernetes clusters. When running on large clusters, addons often consume more of some resources than their default limits. If a large cluster is deployed without adjusting these values, the addon(s) may continuously get killed because they keep hitting the memory limit. Alternatively, the addon may run but with poor performance due to CPU time slice restrictions.

To avoid running into cluster addon resource issues, when creating a cluster with many nodes, consider the following:

- Some addons scale vertically - there is one replica of the addon for the cluster or serving a whole failure zone. For these addons, increase requests and limits as you scale out your cluster.
- Many addons scale horizontally - you add capacity by running more pods - but with a very large cluster you may also need to raise CPU or memory limits slightly. The Vertical Pod Autoscaler can run in *recommender* mode to provide suggested figures for requests and limits.
- Some addons run as one copy per node, controlled by a DaemonSet: for example, a node-level log aggregator. Similar to the case with horizontally-scaled addons, you may also need to raise CPU or memory limits slightly.

What's next

- `VerticalPodAutoscaler` is a custom resource that you can deploy into your cluster to help you manage resource requests and limits for pods. Learn more about Vertical Pod Autoscaler and how you can use it to scale cluster components, including cluster-critical addons.
- Read about Node autoscaling
- The addon resizer helps you in resizing the addons automatically as your cluster's scale changes.