Kubernetes production best practices

A curated checklist of best practices designed to help you release to production

This checklist provides actionable best practices for deploying secure, scalable, and resilient services on Kubernetes.

The content is open source and available in this repository.
If you think there are missing best practices or they are not right, consider submitting an issue.

Check things off to keep track as you go.

YOUR PROGRESS

0% complete

Categories

1. Application development

2. Governance

3. Cluster configuration

1. Application development

Health checks

Containers have Readiness probes

> Please note that there's no default value for readiness and liveness.

If you don't set the readiness probe, the kubelet assumes that the app is ready to receive traffic as soon as the container starts.

If the container takes 2 minutes to start, all the requests to it will fail for those 2 minutes.

Containers crash when there's a fatal error

If the application reaches an unrecoverable error, you should let it crash.

Examples of such unrecoverable errors are:

- an uncaught exception
- a typo in the code (for dynamic languages)
- unable to load a header or dependency

Please note that you should not signal a failing Liveness probe.

Instead, you should immediately exit the process and let the kubelet restart the container.

Configure a passive Liveness probe

The Liveness probe is designed to restart your container when it's stuck.

Consider the following scenario: if your application is processing an infinite loop, there's no way to exit or ask for help.

When the process is consuming 100% CPU, it won't have time to reply to the (other) Readiness probe checks, and it will be eventually removed from the Service.

However, the Pod is still registered as an active replica for the current Deployment.

If you don't have a Liveness probe, it stays *Running* but detached from the Service.

In other words, not only is the process not serving any requests, but it is also consuming resources.

What should you do?

1. Expose an endpoint from your app
2. The endpoint always replies with a success response
3. Consume the endpoint from the Liveness probe

Please note that you should not use the Liveness probe to handle fatal errors in your app and request Kubernetes to restart the app.

Instead, you should let the app crash.

The Liveness probe should be used as a recovery mechanism only in case the process is not responsive.

Liveness probes values aren't the same as the Readiness

When Liveness and Readiness probes are pointing to the same endpoint, the effects of the probes are combined.

When the app signals that it's not ready or live, the kubelet detaches the container from the Service and delete it **at the same time**.

You might notice dropping connections because the container does not have enough time to drain the current connections or process the incoming ones.

You can dig deeper in the following article that discussed graceful shutdown.

Apps are independent

The Readiness probes are independent

The readiness probe doesn't include dependencies to services such as:

- databases
- database migrations
- APIs
- third party services

You can explore what happens when there're dependencies in the readiness probes in this essay.

The app retries connecting to dependent services

When the app starts, it shouldn't crash because a dependency such as a database isn't ready.

Instead, the app should keep retrying to connect to the database until it succeeds.

Kubernetes expects that application components can be started in any order.

When you make sure that your app can reconnect to a dependency such as a database you know you can deliver a more robust and resilient service.

Graceful shutdown

The app doesn't shut down on SIGTERM, but it gracefully terminates connections

It might take some time before a component such as kube-proxy or the Ingress controller is notified of the endpoint changes.

Hence, traffic might still flow to the Pod despite it being marked as terminated.

The app should stop accepting new requests on all remaining connections, and close these once the outgoing queue is drained.

If you need a refresher on how endpoints are propagated in your cluster, read this article on how to handle client requests properly.

The app still processes incoming requests in the grace period

You might want to consider using the container lifecycle events such as the preStop handler to customize what happens before a Pod is deleted.

The CMD in the `Dockerfile` forwards the SIGTERM to the process

You can be notified when the Pod is about to be terminated by capturing the SIGTERM signal in your app.

You should also pay attention to forwarding the signal to the right process in your container.

Close all idle keep-alive sockets

If the calling app is not closing the TCP connection (e.g. using TCP keep-alive or a connection pool) it will connect to one Pod and not use the other Pods in that Service.

But what happens when a Pod is deleted?

Ideally, the request should go to another Pod.

However, the calling app has a long-lived connection open with the Pod that is about to be terminated, and it will keep using it.

On the other hand, you shouldn't abruptly terminate long-lived connections.

Instead, you should terminate them before shutting down the app.

Fault tolerance

Run more than one replica for your Deployment

Never run a single Pod individually.

Instead consider deploying your Pod as part of a Deployment, DaemonSet, ReplicaSet or StatefulSet.

Running more than one instance of your Pods guarantees that deleting a single Pod won't cause downtime.

Avoid Pods being placed into a single node

Even if you run several copies of your Pods, there are no guarantees that losing a node won't take down your service.

Consider the following scenario: you have 11 replicas on a single cluster node.

If the node is made unavailable, the 11 replicas are lost, and you have downtime.

You should apply anti-affinity rules to your Deployments so that Pods are spread in all the nodes of your cluster.

The inter-pod affinity and anti-affinity documentation describe how you can you could change your Pod to be located (or not) in the same node.

Set Pod disruption budgets

When a node is drained, all the Pods on that node are deleted and rescheduled.

But what if you are under heavy load and you can't lose more than 50% of your Pods?

The drain event could affect your availability.

To protect the Deployments from unexpected events that could take down several Pods at the same time, you can define Pod Disruption Budget.

Imagine saying: *"Kubernetes, please make sure that there are always at least 5 Pods running for my app".*

Kubernetes will prevent the drain event if the final state results in less than 5 Pods for that Deployment.

The official documentation is an excellent place to start to understand Pod Disruption Budgets.

Resources utilisation

Set memory limits and requests for all containers

Resource limits are used to constrain how much CPU and memory your containers can utilise and are set using the resources property of a `containerSpec` .

The scheduler uses those as one of metrics to decide which node is best suited for the current Pod.

A container without a memory limit has memory utilisation of zero — according to the scheduler.

An unlimited number of Pods if schedulable on any nodes leading to resource overcommitment and potential node (and kubelet) crashes.

The same applies to CPU limits.

But should you always set limits and requests for memory and CPU?

Yes and no.

If your process goes over the memory limit, the process is terminated.

Since CPU is a compressible resource, if your container goes over the limit, the process is throttled.

Even if it could have used some of the CPU that was available at that moment.

CPU limits are hard.

If you wish to dig deeper into CPU and memory limits you should check out the following articles:

- Understanding resource limits in kubernetes: memory
- Understanding resource limits in kubernetes: cpu time

> Please note that if you are not sure what should be the *right* CPU or memory limit, you can use the Vertical Pod Autoscaler in Kubernetes with the recommendation mode turned on. The autoscaler profiles your app and recommends limits for it.

Set CPU request to 1 CPU or below

Unless you have computational intensive jobs, it is recommended to set the request to 1 CPU or below.

Disable CPU limits — unless you have a good use case

CPU is measured as CPU timeunits per timeunit.

`cpu: 1` means 1 CPU second per second.

If you have 1 thread, you can't consume more than 1 CPU second per second.

If you have 2 threads, you can consume 1 CPU second in 0.5 seconds.

8 threads can consume 1 CPU second in 0.125 seconds.

After that, your process is throttled.

If you're not sure about what's the best settings for your app, it's better not to set the CPU limits.

If you wish to learn more, this article digs deeper in CPU requests and limits.

The namespace has a LimitRange

If you think you might forget to set memory and CPU limits, you should consider using a LimitRange object to define the standard size for a container deployed in the current namespace.

The official documentation about LimitRange is an excellent place to start.

Set an appropriate Quality of Service (QoS) for Pods

When a node goes into an overcommitted state (i.e. using too many resources) Kubernetes tries to evict some of the Pod in that Node.

Kubernetes ranks and evicts the Pods according to a well-defined logic.

You can find more about configuring the quality of service for your Pods on the official documentation.

Tagging resources

Resources have technical labels defined

You could tag your Pods with:

- `name` , the name of the application such "User API"
- `instance` , a unique name identifying the instance of an application (you could use the container image tag)
- `version` , the current version of the appl (an incremental counter)
- `component` , the component within the architecture such as "API" or "database"
- `part-of` , the name of a higher-level application this one is part of such as "payment gateway"
- `managed-by` , the tool being used to manage the operation of an application such as "kubectl" or "Helm"

Here's an example on how you could use such labels in a Deployment:

deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment
  labels:
    app.kubernetes.io/name: user-api
    app.kubernetes.io/instance: user-api-5fa65d2
    app.kubernetes.io/version: "42"
    app.kubernetes.io/component: api
    app.kubernetes.io/part-of: payment-gateway
    app.kubernetes.io/managed-by: kubectl
spec:
  replicas: 3
  selector:
    matchLabels:
      application: my-app
  template:
    metadata:
      labels:
        app.kubernetes.io/name: user-api
        app.kubernetes.io/instance: user-api-5fa65d2
        app.kubernetes.io/version: "42"
        app.kubernetes.io/component: api
        app.kubernetes.io/part-of: payment-gateway
    spec:
      containers:
      - name: app
        image: myapp
```

Those labels are recommended by the official documentation.

> Please note that you're recommended to tag **all resources**.

Resources have business labels defined

You could tag your Pods with:

- `owner`, used to identify who is responsible for the resource
- `project`, used to determine the project that the resource belongs to
- `business-unit`, used to identify the cost centre or business unit associated with a resource; typically for cost allocation and tracking

Here's an example on how you could use such labels in a Deployment:

deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment
  labels:
    owner: payment-team
    project: fraud-detection
    business-unit: "80432"
spec:
  replicas: 3
  selector:
    matchLabels:
      application: my-app
  template:
    metadata:
      labels:
        owner: payment-team
        project: fraud-detection
        business-unit: "80432"
    spec:
      containers:
      - name: app
        image: myapp
```

You can explore labels and tagging for resources on the AWS tagging strategy page.

The article isn't specific to Kubernetes but explores some of the most common strategies for tagging resources.

> Please not that you're recommended to tag **all resources**.

Resources have security labels defined

You could tag your Pods with:

- `confidentiality` , an identifier for the specific data-confidentiality level a resource supports
- `compliance` , an identifier for workloads designed to adhere to specific compliance requirements

Here's an example on how you could use such labels in a Deployment:

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment
  labels:
    confidentiality: official
    compliance: pci
spec:
  replicas: 3
  selector:
    matchLabels:
      application: my-app
  template:
    metadata:
      labels:
        confidentiality: official
        compliance: pci
    spec:
      containers:
      - name: app
        image: myapp
```

You can explore label and tagging for resources on the AWS tagging strategy page.

The article isn't specific to Kubernetes but explores some of the most common strategies for tagging resources.

> Please not that you're recommended to tag **all resources**.

Logging

The application logs to `stdout` and `stderr`

There are two logging strategies: *passive* and *active*.

Apps that use passive logging are unaware of the logging infrastructure and log messages to standard outputs.

This best practice is part of the twelve-factor app.

In active logging, the app makes network connections to intermediate aggregators, sends data to third-party logging services, or writes directly to a database or index.

Active logging is considered an antipattern, and it should be avoided.

Avoid sidecars for logging (if you can)

If you wish to apply log transformations to an application with a non-standard log event model, you may want to use a sidecar container.

With a sidecar container, you can normalise the log entries before they are shipped elsewhere.

For example, you may want to transform Apache logs into Logstash JSON format before shipping it to the logging infrastructure.

However, if you have control over the application, you could output the right format, to begin with.

You could save on running an extra container for each Pod in your cluster.

Scaling

Containers do not store any state in their local filesystem

Containers have a local filesystem and you might be tempted to use it for persisting data.

However, storing persistent data in a container's local filesystem prevents the encompassing Pod from being scaled horizontally (that is, by adding or removing replicas of the Pod).

This is because, by using the local filesystem, each container maintains its own "state", which means that the states of Pod replicas may diverge over time. This results in inconsistent behaviour from the user's point of view (for example, a specific piece of user information is available when the request hits one Pod, but not when the request hits another Pod).

Instead, any persistent information should be saved at a central place outside the Pods. For example, in a PersistentVolume in the cluster, or even better in some storage service outside the cluster.

Use the Horizontal Pod Autoscaler for apps with variable usage patterns

The Horizontal Pod Autoscaler (HPA) is a built-in Kubernetes feature that monitors your application and automatically adds or removes Pod replicas based on the current usage.

Configuring the HPA allows your app to stay available and responsive under any traffic conditions, including unexpected spikes.

To configure the HPA to autoscale your app, you have to create a HorizontalPodAutoscaler resource, which defines what metric to monitor for your app.

The HPA can monitor either built-in resource metric (CPU and memory usage of your Pods) or custom metrics. In the case of custom metrics, you are also responsible for collecting and exposing these metrics, which you can do, for example, with Prometheus and the Prometheus Adapter.

Don't use the Vertical Pod Autoscaler while it's still in beta

Analogous to the Horizontal Pod Autoscaler (HPA), there exists the Vertical Pod Autoscaler (VPA).

The VPA can automatically adapt the resource requests and limits of your Pods so that when a Pod needs more resources, it can get them (increasing/decreasing the resources of a single Pod is called *vertical scaling*, as opposed to *horizontal scaling*, which means increasing/decreasing the number of replicas of a Pod).

This can be useful for scaling applications that can't be scaled horizontally.

However, the VPA is curently in beta and it has some known limitations (for example, scaling a Pod by changing its resource requirements, requires the Pod to be killed and restarted).

Given these limitations, and the fact that most applications on Kubernetes can be scaled horizontally anyway, it is recommended to not use the VPA in production (at least until there is a stable version).

Use the Cluster Autoscaler if you have highly varying workloads

The Cluster Autoscaler is another type of "autoscaler" (besides the Horizontal Pod Autoscaler and Vertical Pod Autoscaler).

The Cluster Autoscaler can automatically scale the size of your cluster by adding or removing worker nodes.

A scale-up operation happens when a Pod fails to be scheduled because of insufficient resources on the existing worker nodes. In this case, the Cluster Autoscaler creates a new worker node, so that the Pod can be scheduled. Similarly, when the utilisation of the existing worker nodes is low, the Cluster Autoscaler can scale down by evicting all the workloads from one of the worker nodes and removing it.

Using the Cluster Autoscaler makes sense for highly variable workloads, for example, when the number of Pods may multiply in a short time, and then go back to the previous value. In such scenarios, the Cluster Autoscaler allows you to meet the demand spikes without wasting resources by overprovisioning worker nodes.

However, if your workloads do not vary so much, it may not be worth to set up the Cluster Autoscaler, as it may never be triggered. If your workloads grow slowly and monotonically, it may be enough to monitor the utilisations of your existing worker nodes and add an additional worker node manually when they reach a critical value.

Configuration and secrets

Externalise all configuration

Configuration should be maintained outside the application code.

This has several benefits. First, changing the configuration does not require recompiling the application. Second, the configuration can be updated when the application is running. Third, the same code can be used in different environments.

In Kubernetes, the configuration can be saved in ConfigMaps, which can then be mounted into containers as volumes are passed in as environment variables.

Save only non-sensitive configuration in ConfigMaps. For sensitive information (such as credentials), use the Secret resource.

Mount Secrets as volumes, not enviroment variables

The content of Secret resources should be mounted into containers as volumes rather than passed in as environment variables.

This is to prevent that the secret values appear in the command that was used to start the container, which may be inspected by individuals that shouldn't have access to the secret values.

2. Governance

Namespace limits

Namespaces have LimitRange

Containers without limits can lead to resource contention with other containers and unoptimized consumption of computing resources.

Kubernetes has two features for constraining resource utilisation: ResourceQuota and LimitRange.

With the LimitRange object, you can define default values for resource requests and limits for individual containers inside namespaces.

Any container created inside that namespace, without request and limit values explicitly specified, is assigned the default values.

You should check out the official documentation if you need a refresher on resource quotas.

Namespaces have ResourceQuotas

With ResourceQuotas, you can limit the total resource consumption of all containers inside a Namespace.

Defining a resource quota for a namespace limits the total amount of CPU, memory or storage resources that can be consumed by all containers belonging to that namespace.

You can also set quotas for other Kubernetes objects such as the number of Pods in the current namespace.

If you're thinking that someone could exploit your cluster and create 20000 ConfigMaps, using the LimitRange is how you can prevent that.

Pod security policies

Enable Pod Security Policies

For example, you can use Kubernetes Pod security policies for restricting:

- Access the host process or network namespace
- Running privileged containers
- The user that the container is running as
- Access the host filesystem
- Linux capabilities, Seccomp or SELinux profiles

Choosing the right policy depends on the nature of your cluster.

The following article explains some of the Kubernetes Pod Security Policy best practices

Disable privileged containers

In a Pod, containers can run in "privileged" mode and have almost unrestricted access to resources on the host system.

While there are specific use cases where this level of access is necessary, in general, it's a security risk to let your containers do this.

Valid uses cases for privileged Pods include using hardware on the node such as GPUs.

You can learn more about security contexts and privileges containers from this article.

Use a read-only filesystem in containers

Running a read-only file system in your containers forces your containers to be immutable.

Not only does this mitigate some old (and risky) practices such as hot patching, but also helps you prevent the risks of malicious processes storing or manipulating data inside a container.

Running containers with a read-only file system might sound straightforward, but it might come with some complexity.

What if you need to write logs or store files in a temporary folder?

You can learn about the trade-offs in this article on running containers securely in production.

Prevent containers from running as root

A process running in a container is no different from any other process on the host, except it has a small piece of metadata that declares that it's in a container.

Hence, root in a container is the same root (uid 0) as on the host machine.

If a user manages to break out of an application running as root in a container, they may be able to gain access to the host with the same root user.

Configuring containers to use unprivileged users, is the best way to prevent privilege escalation attacks.

If you wish to learn more, the follow article offers some detailed explanation examples of what happens when you run your containers as root.

Limit capabilities

Linux capabilities give processes the ability to do some of the many privileged operations only the root user can do by default.

For example, `CAP_CHOWN` allows a process to "make arbitrary changes to file UIDs and GIDs".

Even if your process doesn't run as `root` , there's a chance that a process could use those root-like features by escalating privileges.

In other words, you should enable only the capabilities that you need if you don't want to be compromised.

But what capabilities should be enabled and why?

The following two articles dive into the theory and practical best-practices about capabilities in the Linux Kernel:

- Linux Capabilities: Why They Exist and How They Work
- Linux Capabilities In Practice

Prevent privilege escalation

You should run your container with privilege escalation turned off to prevent escalating privileges using `setuid` or `setgid` binaries.

Network policies

Enable network policies

Kubernetes network policies specify the access permissions for groups of pods, much like security groups in the cloud are used to control access to VM instances.

In other words, it creates firewalls between pods running on a Kubernetes cluster.

If you are not familiar with Network Policies, you can read Securing Kubernetes Cluster Networking.

There's a conservative NetworkPolicy in every namespace

This repository contains various use cases of Kubernetes Network Policies and samples YAML files to leverage in your setup. If you ever wondered how to drop/restrict traffic to applications running on Kubernetes, read on.

Role-Based Access Control (RBAC) policies

Disable auto-mounting of the default ServiceAccount

Please note that the default ServiceAccount is automatically mounted into the file system of all Pods.

You might want to disable that and provide more granular policies.

RBAC policies are set to the least amount of privileges necessary

It's challenging to find good advice on how to set up your RBAC rules. In 3 realistic approaches to Kubernetes RBAC, you can find three practical scenarios and practical advice on how to get started.

RBAC policies are granular and not shared

Zalando has a concise policy to define roles and ServiceAccounts.

First, they describe their requirements:

- Users should be able to deploy, but they shouldn't be allowed to read Secrets for example
- Admins should get full access to all resources
- Applications should not gain write access to the Kubernetes API by default
- It should be possible to write to the Kubernetes API for some uses.

The four requirements translate into five separate Roles:

- ReadOnly
- PowerUser
- Operator
- Controller
- Admin

You can read about their decision in this link.

Custom policies

Allow deploying containers only from known registries

One of the most common custom policies that you might want to consider is to restrict the images that can be deployed in your cluster.

The following tutorial explains how you can use the Open Policy Agent to restrict not approved images.

Enforce uniqueness in Ingress hostnames

When a user creates an Ingress manifest, they can use any hostname in it.

ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
    - host: first.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: service
                port:
                  number: 80
```

However, you might want to prevent users using **the same hostname multiple times** and overriding each other.

The official documentation for the Open Policy Agent has a tutorial on how to check Ingress resources as part of the validation webhook.

Only use approved domain names in the Ingress hostnames

When a user creates an Ingress manifest, they can use any hostname in it. a

ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
    - host: first.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: service
                port:
                  number: 80
```

However, you might want to prevent users using **invalid hostnames**.

The official documentation for the Open Policy Agent has a tutorial on how to check Ingress resources as part of the validation webhook.

3. Cluster configuration

This section is a work in progress. Do you have an opinion on what you should be included?
File an issue.

Approved Kubernetes configuration

The cluster passes the CIS benchmark

The Center for Internet Security provides several guidelines and benchmark tests for best practices in securing your code.

They also maintain a benchmark for Kubernetes which you can download from the official website.

While you can read the lengthy guide and manually check if your cluster is compliant, an easier way is to download and execute `kube-bench` .

`kube-bench` is a tool designed to automate the CIS Kubernetes benchmark and report on misconfigurations in your cluster.

Example output:

```bash
[INFO] 1 Master Node Security Configuration
[INFO] 1.1 API Server
[WARN] 1.1.1 Ensure that the --anonymous-auth argument is set to false
[PASS] 1.1.2 Ensure that the --basic-auth-file argument is not set (Sco
[PASS] 1.1.3 Ensure that the --insecure-allow-any-token argument is not
[PASS] 1.1.4 Ensure that the --kubelet-https argument is set to true (S
[PASS] 1.1.5 Ensure that the --insecure-bind-address argument is not se
[PASS] 1.1.6 Ensure that the --insecure-port argument is set to 0 (Scor
[PASS] 1.1.7 Ensure that the --secure-port argument is not set to 0 (Sc
[FAIL] 1.1.8 Ensure that the --profiling argument is set to false (Scor
```

> Please note that it is not possible to inspect the master nodes of managed clusters such as GKE, EKS and AKS, using `kube-bench`. The master nodes are controlled and managed by the cloud provider.

Disable metadata cloud providers metadata API

Cloud platforms (AWS, Azure, GCE, etc.) often expose metadata services locally to instances.

By default, these APIs are accessible by pods running on an instance and can contain cloud credentials for that node, or provisioning data such as kubelet credentials.

These credentials can be used to escalate within the cluster or to other cloud services under the same account.

Restrict access to alpha or beta features

Alpha and beta Kubernetes features are in active development and may have limitations or bugs that result in security vulnerabilities.

Always assess the value an alpha or beta feature may provide against the possible risk to your security posture.

When in doubt, disable features you do not use.

Authentication

Use OpenID (OIDC) tokens as a user authentication strategy

Kubernetes supports various authentication methods, including OpenID Connect (OIDC).

OpenID Connect allows single sign-on (SSO) such as your Google Identity to connect to a Kubernetes cluster and other development tools.

You don't need to remember or manage credentials separately.

You could have several clusters connect to the same OpenID provider.

You can learn more about the OpenID connect in Kubernetes in this article.

Role-Based Access Control (RBAC)

ServiceAccount tokens are for applications and controllers **only**

Service Account Tokens should not be used for end-users trying to interact with Kubernetes clusters, but they are the preferred authentication strategy for applications and workloads running on Kubernetes.

Logging setup

There's a retention and archival strategy for logs

You should retain 30-45 days of historical logs.

Logs are collected from Nodes, Control Plane, Auditing

What to collect logs from:

- Nodes (kubelet, container runtime)
- Control plane (API server, scheduler, controller manager)
- Kubernetes auditing (all requests to the API server)

What you should collect:

- Application name. Retrieved from metadata labels.
- Application instance. Retrieved from metadata labels.
- Application version. Retrieved from metadata labels.
- Cluster ID. Retrieved from Kubernetes cluster.
- Container name. Retrieved from Kubernetes API.
- Cluster node running this container. Retrieved from Kubernetes cluster.
- Pod name running the container. Retrieved from Kubernetes cluster.
- The namespace. Retrieved from Kubernetes cluster.

Prefer a daemon on each node to collect the logs instead of sidecars

Applications should log to stdout rather than to files.

A daemon on each node can collect the logs from the container runtime (if logging to files, a sidecar container for each pod might be necessary).

Provision a log aggregation tool

Use a log aggregation tool such as EFK stack (Elasticsearch, Fluentd, Kibana), DataDog, Sumo Logic, Sysdig, GCP Stackdriver, Azure Monitor, AWS CloudWatch.

What is Learnk8s?

In-depth Kubernetes training that is practical and easy to understand.

⎈ Instructor-led workshops ›

Deep dive into containers and Kubernetes with the help of our instructors and become an expert in deploying applications at scale.

⎈ Online courses ›

Learn Kubernetes online with hands-on, self-paced courses. No need to leave the comfort of your home.

⎈ Corporate training ›

Train your team in containers and Kubernetes with a customised learning path — remotely or on-site.

PARTNERS, NEWS & COMMUNITY

- Kube Careers
- Kube Events
- Kubernetes Architect
- Kubesploit
- K3s Daily
- See all