5

Observability and Maintenance

In the previous chapter we learned more about how the Pods are configured to securely access and consume the data within the containers. In this chapter, we’ll discuss container health probes—more specifically, startup, readiness, and liveness probes that ensures the reliability and availability of applications running inside the Pod container.

You’ll learn about the different health verification methods and how to define them for the proper use cases. Moreover, the exam topics includes monitoring and debugging of Kubernetes cluster, nodes and Pods that might be affecting application performance of complete outage. You want to make yourself familiarize with various tools and strategies for troubleshooting a misbehaving or misconfigured Kubernetes object. At the end of the chapter, we will discuss about Admission Controllers and API deprecations.

At a high level, this chapter covers the following concepts:

* Startup probe
* Liveness probe
* Readiness probe
* Logging and Utilizing POD Container logs
* Kubernetes monitoring
* Cluster Observability
* Troubleshooting and debugging in Kubernetes
* Admission Controllers and API deprecations

**Understanding Health Probing**

Kubernetes provides a health checkingmechanism to verify if a container inside a Pod is working or not. Even with the best automated tools and test cases that performs application testing, it’s quite impossible to identify all bugs and issues before deploying software to a production environment. It also happens that the issue may occurs in Production environment only after using the software for some extended period of time or when the usage exceeds the resource limits. Knowing application health in production environment is important and crucial for business perspective since the software might be used by an external clients or users. Using health checks, it becomes easy to let the system know whether the application containers are health and servicing the traffic to make sure the apps are working. For whatever reasons if the application container(s) gets stuck or stop working then other microservice component will not try to connect to that container but instead the discovery service will find another Pod container serving similar traffic using proper labeling and discovery service. That’s especially possible for application failure situations that only occur in production after operating the software after the deployment that may not happen in lower environment but only affects production environment due to memory leaks, infinite loops, deadlocks etc. Once the application is used by end users that generates loads. Appropriate monitoring can help identify and troubleshoot those issues; however, operations team still have to handle the situation and mitigate and overcome all challenges and fix those issues. Without proper mechanism it is nearly impossible to isolate the real issues in production environment as there could be many moving parts when applications run in production environments. In order to isolate issues and many layers of microservices and components application-level debugging and smart health endpoints are key to nail down applications related issues.

When applications run as container load in Kubernetes environments it even becomes difficult to gather application-level metrics and exceptions as it requires extra coding and afford by an engineering team to incorporate those type of mechanisms inside applications code. For example: having an application health endpoint like JSON API makes it easier to visualize and troubleshoot any issues that may occur in live environment when applications may not start properly due to an external connection to database, messaging systems or configuration buckets like Amazon S3. If applications will have some sort of checking mechanism and provide health endpoints, then the operations team can easily find out which application is having issues and what could be the issue quickly as possible and look out for further issues.

Kubernetes provides similar concept and mechanism called “*health probing”* which helps operations team to identify, detect and correct such type of issues. It also helps to automate such operations built in the application definition files to detect and correct issues if they want to avoid any further issues in live production environment. You should be able to configure a Pod’s container to execute a job that can periodically check certain conditions like application availability, endpoints, resource limits and usage.

The Kubernetes framework is a great self-healing system that can bring the application container to a healthy state if it encounters any issues. By default, Kubernetes will start sending traffic to all the containers inside the pod upon its startup and it will restart them when and if they crash. These features make Kubernetes a industries leading containers management system that supports self-healing using custom creation of various health checking probes. So, let’s look at all health probes and how can we configure Pods that implements the health checks that makes application deployment reliable and robust. Following topics covers those areas of writing applicant’s Pod definition files and demonstrate how to apply those mechanisms that provides better health checks within applications pod life cycle.

Kubernetes provides following types of health checking probes that are performed by the Kubernetes nodes agent,the kubelet.

The probes are:

1. Liveness Probe -> App is healthy, Container is running.
2. Readiness Probe -> App is ready to serve traffic, Container is ready to serve traffic
3. Startup Probe -> App is properly started, Container is Started. Disable both Liveness and Readiness probes until Startup probe finish successfully.

**Liveness probe**

Kubernetes provides “Liveness probes” mechanisms to detect and remediate irresponsive and unhealthy applications by checking the status of the container. It is highly possible that longer running applications may crash due to many reasons over period of time, meaning the containers running those applications are no longer available for access. To avoid such a chaotic situation for customer facing applications it is a good idea to configure application deployment with liveness probe which automatically will check the health endpoint of container and restart the Pod by destroying currently running Pod and create new one based on the restart policy defined in the manifests file. The Liveness probes comes in play once the Pod completely starts up and application starts running by making sure the Pod is still working and passes all health checks.

Let’s look at the real-world scenario in which an application that is running in production environment crashes due to an issue with the code bug that causes an application to get stuck and hangs. This issue can lead to a complete downtime for any customer facing applications since the process was still running as an application container and so Kubernetes will still send traffic to the broken Pod. We can make a use of Kubernetes feature “Liveness probes” that detect and identify that the Pod is broken due to a crashed application and no longer available to service the traffic request, Kubernetes will stop the request, restart the Pod and wait for the Pod to come online before putting it back to rotation.

To further understand the scenario and for CKAD exam perspective it is important that you know the “Liveness probe” concept very all and implement it during the exam. Using following configuration example, we will demonstrate and implement “Liveness probe” inside the Pod’s container. We will use any custom Linux command to implement and verify the health status of a container. The custom command can be added during the application containerization or any command-line tool that is part of the base image that can help achieve this task. For the demo purpose our application will create a file called: /tmp/healthz.txt with string called: “OK” inside it is using the Linux command called “touch” every ten seconds. The probe will periodically check for modified timestamp of the file “/tmp/healthz.txt” and string “OK” to make sure the application is running as expected and don’t interfere with Pod restart. For whatever reason if the file healthz.txt don’t get updated every ten seconds then Kubernetes will assume that application isn’t functioning properly, and it will restart the container inside the Pod.

As shown in the Figure 5-2 the Kubelet will periodically verify the container’s health and restart if it fails.

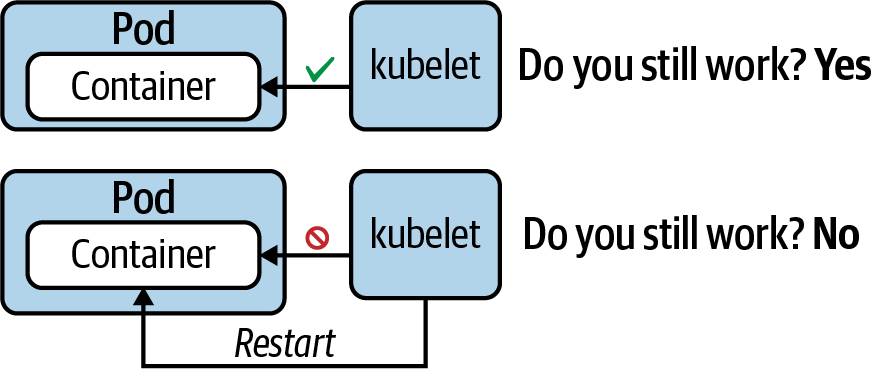


Figure 5-2. A liveness probe checks if the application is still considered healthy, if so, the service will send traffic to the Pod’s container.

**Example 5-2. A liveness probe that uses a custom command**

apiVersion: v1

kind: Pod

metadata:

name: liveness-probe-pod

spec:

containers:

- image: busybox --- image that has built-in Linux commands

name: app

args:

- /bin/sh

- -c

- 'while true; do touch /tmp/healthz.txt; sleep 10; done;' # *Linux command touch is updating timestamp of a file*

livenessProbe:

exec:

command:

- test `find /tmp/healthz.txt -mmin -1`

initialDelaySeconds: 5

periodSeconds: 30

As you can see the above Pod definition that the touch command is updating the timestamp of a file called /tmp/healthz.txt every 10 seconds by running “sleep 10” command. “sleep” and another command available in Linux operating system that allows a process to wait for certain time before executing next run

**HTTP Liveness probes (httpGet):** Another way to provide Liveness check is httpGet which in which we can create a lightweight HTTP server within the application that will provide Liveness probe status. Kubernetes will send a ping signal to a path (for example, **/healthz**) at a given port (for example 8080). If it gets an HTTP response in the range of 200 (202) or 300 (301) , it will assume the application is healthy and available to service the traffic to the Pod’s container.

Following is the Pod definition file that can be used to configure HTTP GET method to provide health check.

apiVersion: v1

kind: Pod

metadata:

name: liveness-probe-http

spec:

containers:

- name: liveness-probe-http

image: k8s.gcr.io/liveness

args:

- /server

livenessProbe:

httpGet:

path: /healthz

port: 8080

initialDelaySeconds: 5

periodSeconds: 10

failureThreshold: 1

Once you create the yaml definition file use kubectl create command followed by describe command to see the output of the Pod created by the manifests file.

**$ kubectl create -f liveness-probe-http-demo.yaml**

pod/liveness-pod created

**$ kubectl get pod liveness-probe-http**

NAME READY STATUS RESTARTS AGE

pod/liveness-probe-http 1/1 Running 0 32s

**$ kubectl describe pod liveness-probe-http**

...

Containers:

app:

...

Restart Count: 0

Liveness: exec [test `find /tmp/healthz.txt -mmin -1`] delay=5s \

timeout=1s period=30s #success=1 #failure=3

...

**Startup probe**

The main purpose of a startup probe is to figure out when an application is fully started. Defining the probe is especially useful for an application that takes a long time to start up. The kubelet puts the readiness and liveness probes on hold while the startup probe is running. A startup probe finishes its operation under one of the following conditions:

1. If it could verify that the application has been started.
2. If the application doesn’t respond within the timeout period.

To demonstrate the functionality of the startup probe, Example 5-3 defines a Pod that runs the Apache HTTP server in a container. By default, the image exposes the container port 80, and that’s what we’re probing for using a TCP socket connection.

**Example 5-3. A startup probe that uses a TCP socket connection**

apiVersion: v1

kind: Pod

metadata:

name: startup-pod

spec:

containers:

- image: httpd:2.4.46

name: http-server

startupProbe:

tcpSocket:

port: 80

initialDelaySeconds: 3

periodSeconds: 15

As you can see in the following terminal output, the describe command can retrieve the configuration of a startup probe as well:

**$ kubectl create -f startup-probe.yaml**

pod/startup-pod created

**$ kubectl get pod startup-pod**

NAME READY STATUS RESTARTS AGE

pod/startup-pod 1/1 Running 0 31s

**$ kubectl describe pod startup-pod**

...

Containers:

http-server:

...

Startup: tcp-socket :80 delay=3s timeout=1s period=15s \

#success=1 #failure=3

...

Sometimes, the legacy applications take longer than expected during startup due to many dependencies. For legacy applications first initialization depends on many variables like thick database connections, loading multiple libraries and objects, tightly coupled components etc., in such scenarios, it can be tricky to configure liveness probe spec parameters without distressing the quick response to deadlocks that caused such a probe. The idea is that the startup probe can also be set using the same command, TCP, or HTTP, with a “failureThreshold \* periodSeconds” parameters with long enough threshold to overcome the worst-case startup time scenario. This probe can be configured to wait for a predefined time value before a liveness probe is kicking off. Startup probe helps prevent application overload with frequent and unnecessary probing requests.

overwhelming the application process with probing requests. Startup probes kill the container if the application couldn’t start within the set time frame. Figure 5-3 illustrates the behavior of a startup probe.

To understand this further, let us look at few real-world scenarios: In the first scenario we will take an example application that requires some time to download some data from external sources like GitHub or third-party data providers using secure API connection. In ideal case scenario the application shouldn’t receive traffic until it’s fully up and ready. ------ By default, Kubernetes will start sending traffic as soon as the process inside the container starts. Using the readiness probe, Kubernetes will wait until the app has fully started before it allows the service to send traffic to the new copy.

Startup probes provide a way to defer the execution of liveness and readiness probes until a container indicates it’s able to handle them. Kubernetes won’t direct the other probe types to a container if it has a startup probe that hasn’t yet succeeded.

Whenever we are dealing with **physical/Legacy apps** those may require extra startup time at **first initialization**. In this case, **we have a tendency to** **established** a startup probe with the **constant** command, **protocol,** or **TCP** check, with a failure threshold period seconds long enough **to hide** **the more severe** case startup time.

livenessProbe:

httpGet:

path: /healthz

port: liveness-port

failureThreshold: 1

periodSeconds: 10

startupProbe:

httpGet:

path: /healthz

port: liveness-port

failureThreshold: 30

periodSeconds: 10

So, the previous example would become:

Thanks to the startup probe, the application will have a maximum of 5 minutes (30 \* 10 = 300s) to finish its startup. Once the startup probe has succeeded once, the liveness probe takes over to provide a fast response to container deadlocks. If the startup probe never succeeds, the container is killed after 300s and subject to the pod's restartPolicy.

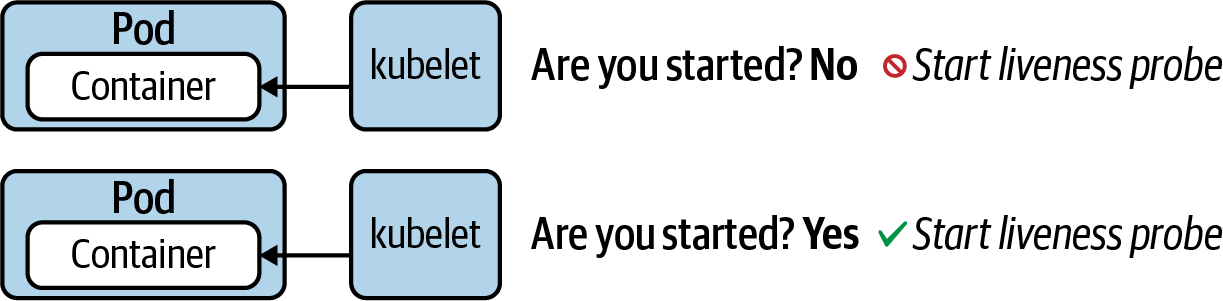


Figure 5-3. A startup probe checks if the application is started

Each probe offers three distinct methods to verify the health of a container. You can define one or many of the health verification methods for a container. Table 5-1 describes the available health verification methods, their corresponding YAML attribute, and their runtime behavior.

**Readiness probe**

Even after an application has been started up, it may still need to execute configuration procedures—for example, connecting to a database and preparing data. This probe checks if the application is ready to serve incoming requests. Figure 5-1 shows the readiness probe.

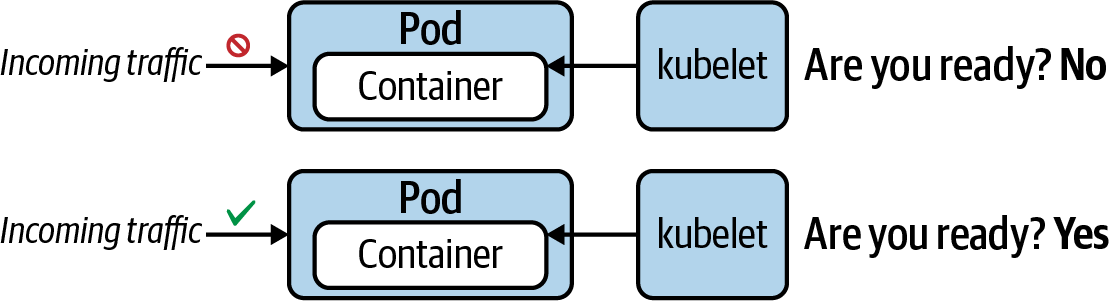


Figure 5-1. A readiness probe checks if the application is ready to accept traffic

**Liveness probe**

Once the application is running, we’ll want to make sure that it still works as expected without issues. This probe periodically checks for the application’s responsiveness. Kubernetes restarts the Pod automatically if the probe considers the application be in an unhealthy state, as shown in Figure 5-2.

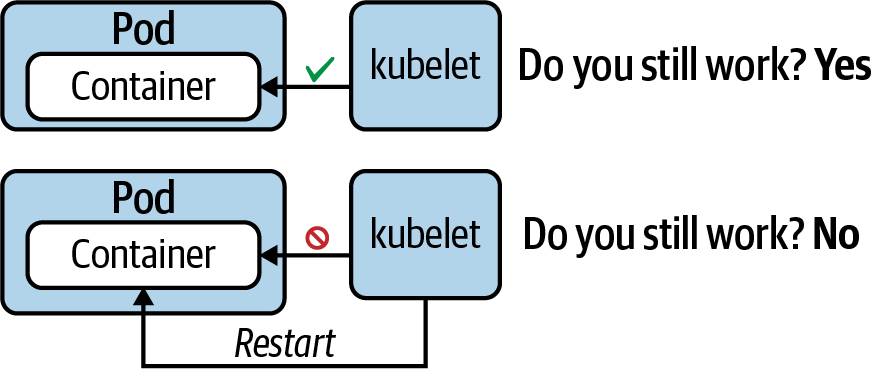


Figure 5-2. A liveness probe checks if the application is still considered healthy

| **Method** | **Option** | **Description** |
| --- | --- | --- |
| Custom command | exec.command | Executes a command inside of the container (e.g., a cat command) and checks its exit code. Kubernetes considers a zero exit code to be successful. A non-zero exit code indicates an error. |
| HTTP GET request | httpGet | Sends an HTTP GET request to an endpoint exposed by the application. An HTTP response code in the range of 200 and 399 indicates success. Any other response code is regarded as an error. |
| TCP socket connection | tcpSocket | Tries to open a TCP socket connection to a port. If the connection could be established, the probing attempt was successful. The inability to connect is accounted for as an error. |
| Table 5-1. Available health verification methods | | |

Every probe offers a set of attributes that can further configure the runtime behavior, as shown in Table 5-2. For more information, see the API of the Probe object.

| **Attribute** | **Default value** | **Description** |
| --- | --- | --- |
| initialDelaySeconds | 0 | Delay in seconds until first check is executed. |
| periodSeconds | 10 | Interval for executing a check (e.g., every 20 seconds). |
| timeoutSeconds | 1 | Maximum number of seconds until check operation times out. |
| successThreshold | 1 | Number of successful check attempts until probe is considered successful after a failure. |
| failureThreshold | 3 | Number of failures for check attempts before probe is marked failed and takes action. |
| Table 5-2. Attributes for fine-tuning the health check runtime behavior | | |

The following sections will demonstrate the usage of those verification methods for different probe types. Remember that you can combine any probe with any health check method. From an operational perspective, the most important probe to implement is the readiness probe. Without defining liveness and startup probes, the Kubernetes control plane components will handle most of the default behavior.

Readiness Probe

In this scenario, we’ll want to define a readiness probe for a Node.js application. The Node.js application exposes an HTTP endpoint on the root context path and runs on port 3000. Dealing with a web-based application makes an HTTP GET request a perfect fit for probing its readiness. You can find the source code of the application in the book’s GitHub repository.

In the YAML manifest shown in Example 5-1, the readiness probe executes its first check after two seconds and repeats checking every eight seconds thereafter. All other attributes use the default values. A readiness probe will continue to periodically check, even after the application has been successfully started.

**Example 5-1. A readiness probe that uses an HTTP GET request**

apiVersion: v1

kind: Pod

metadata:

name: readiness-pod

spec:

containers:

- image: bmuschko/nodejs-hello-world:1.0.0

name: hello-world

ports:

- name: nodejs-port

containerPort: 3000

readinessProbe:

httpGet:

path: /

port: nodejs-port

initialDelaySeconds: 2

periodSeconds: 8

Create a Pod by pointing the create command to the YAML manifest. During the Pod’s startup process, it’s very possible that the status shows Running but the container isn’t ready to accept incoming requests yet, as indicated by 0/1 under the READY column:

**$ kubectl create -f readiness-probe.yaml**

pod/readiness-pod created

**$ kubectl get pod readiness-pod**

NAME READY STATUS RESTARTS AGE

pod/readiness-pod 0/1 Running 0 6s

**$ kubectl get pod readiness-pod**

NAME READY STATUS RESTARTS AGE

pod/readiness-pod 1/1 Running 0 68s

**$ kubectl describe pod readiness-pod**

...

Containers:

hello-world:

...

Readiness: http-get http://:nodejs-port/ delay=2s timeout=1s \

period=8s #success=1 #failure=3

...

**Probe Parameters**

Probes have a number of fields that can be used to more precisely control the behavior of liveness and readiness checks.

* initialDelaySeconds: Number of seconds after the container has started before probes are initiated. (default: 0, minimum: 0)
* periodSeconds: How often to perform the probe (i.e. frequency). (default: 10, minimum: 1)
* timeoutSeconds: Number of seconds after which the probe times out. (default: 1, minimum: 1)
* successThreshold: Minimum consecutive successes for the probe to be considered successful after failure. (default: 1, minimum: 1)
* failureThreshold: How many failed results were received to transition from a healthy to a failure state. (default: 3, minimum: 1)

Following diagram explains various probes in Kuberntes in an easy to understand flow.

Diagram

Description automatically generated

**Debugging in Kubernetes**

When operating an application in a production Kubernetes cluster, it’s almost inevitable that you’ll come across failure situations. You can’t completely leave this job up to the Kubernetes administrator—it’s your responsibility as an application developer to be able to troubleshoot issues for the Kubernetes objects you designed and deployed.

In this section, we’re going to have a look at debugging strategies that can help with identifying the root cause of an issue so that you can act and correct the failure appropriately. The strategies discussed here start with the high-level perspective of a Kubernetes object and then drill into more detail as needed.

**Troubleshooting Pods**

In most cases, creating a Pod is no issue. You simply emit the run, create, or apply commands to instantiate the Pod. If the YAML manifest is formed properly, Kubernetes accepts your request, so the assumption is that everything works as expected. To verify the correct behavior, the first thing you’ll want to do is to check the high-level runtime information of the Pod. The operation could involve other Kubernetes objects like a Deployment responsible for rolling out multiple replicas of a Pod.

**Retrieving high-level information**

To retrieve the information, run either the kubectl get pods command for just the Pods running in the namespace or the kubectl get all command to retrieve the most prominent object types in the namespace (which includes Deployments). You will want to have a look at the columns READY, STATUS, and RESTARTS. In the optimal case, the number of ready containers matches the number of containers expected to be created by the Pod. For a single-container Pod, the READY column would say 1/1. The status should say Running to indicate that the Pod entered the proper lifecycle state. Be aware that it’s totally possible that a Pod renders a Running state, but the application isn’t working properly. If the number of restarts is greater than 0, then you might want to check the logic of the liveness probe (if defined) and identify the reason why a restart was necessary.

The following Pod observes the status ErrImagePull and makes 0/1 containers available to incoming traffic. In short, this Pod has a problem:

**$ kubectl get pods**

NAME READY STATUS RESTARTS AGE

pod/misbehaving-pod 0/1 ErrImagePull 0 2s

After working with Kubernetes for a while, you’ll automatically recognize common error conditions. Table 5-3 lists some of those error statuses and explains how to fix them.

| **Status** | **Root cause** | **Potential fix** |
| --- | --- | --- |
| ImagePullBackOff or ErrImagePull | Image could not be pulled from registry. | Check correct image name, check that image name exists in registry, verify network access from node to registry, ensure proper authentication. |
| CrashLoopBackOff | Application or command run in container crashes. | Check command executed in container, ensure that image can properly execute (e.g., by creating a container with Docker). |
| CreateContainerConfigError | ConfigMap or Secret referenced by container cannot be found. | Check correct name of the configuration object, verify the existence of the configuration object in the namespace. |
| Table 5-3. Common Pod error statuses | | |

**Inspecting events**

It’s totally possible that you’ll not encounter any of those error statuses. But there’s still a chance of the Pod having a configuration issue. You can retrieve detailed information about the Pod and its events using the kubectl describe pod command to inspect its events. The following output belongs to a Pod that tries to mount a Secret that doesn’t exist. Instead of rendering a specific error message, the Pod gets stuck with the status ContainerCreating:

**$ kubectl get pods**

NAME READY STATUS RESTARTS AGE

secret-pod 0/1 ContainerCreating 0 4m57s

**$ kubectl describe pod secret-pod**

...

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal Scheduled <unknown> default-scheduler Successfully assigned \

default/secret-pod to \

minikube

Warning FailedMount 3m15s kubelet, minikube Unable to attach or \

mount volumes: \

unmounted \

volumes=[mysecret], \

unattached volumes= \

[default-token-bf8rh \

mysecret]: timed out \

waiting for the \

condition

Warning FailedMount 68s (x10 over 5m18s) kubelet, minikube MountVolume.SetUp \

failed for volume \

"mysecret" : secret \

"mysecret" not found

Warning FailedMount 61s kubelet, minikube Unable to attach or \

mount volumes: \

unmounted volumes= \

[mysecret], unattached \

volumes=[mysecret \

default-token-bf8rh]: \

timed out waiting for \

the condition

Another helpful command is kubectl get events. The output of the command lists the events across all Pods for a given namespace. You can use additional command-line options to further filter and sort events:

**$ kubectl get events**

LAST SEEN TYPE REASON OBJECT MESSAGE

3m14s Warning BackOff pod/custom-cmd Back-off restarting \

failed container

2s Warning FailedNeedsStart cronjob/google-ping Cannot determine if \

job needs to be \

started: too many \

missed start time \

(> 100). Set or \

decrease .spec. \

startingDeadline \

Seconds or check \

clock skew

**Inspecting logs**

When debugging a Pod, the next level of details can be retrieved by downloading and inspecting its logs. You may or may not find additional information that points to the root cause of a misbehaving Pod. It’s worth a look. The YAML manifest shown in Example 5-4 defines a Pod running a shell command.

**Example 5-4. A Pod running a failing shell command**

apiVersion: v1

kind: Pod

metadata:

name: incorrect-cmd-pod

spec:

containers:

- name: test-container

image: busybox

command: ["/bin/sh", "-c", "unknown"]

After creating the object, the Pod fails with the status CrashLoopBackOff. Running the logs command reveals that the command run in the container has an issue:

**$ kubectl create -f crash-loop-backoff.yaml**

pod/incorrect-cmd-pod created

**$ kubectl get pods incorrect-cmd-pod**

NAME READY STATUS RESTARTS AGE

incorrect-cmd-pod 0/1 CrashLoopBackOff 5 3m20s

**$ kubectl logs incorrect-cmd-pod**

/bin/sh: unknown: not found

The logs command provides two helpful options I’d like to mention here. The option -f streams the logs, meaning you’ll see new log entries as they’re being produced in real time. The option --previous gets the logs from the previous instantiation of a container, which is helpful if the container has been restarted.

**Opening an Interactive Shell**

If any of the previous commands don’t point you to the root cause of the failing Pod, it’s time to open an interactive shell to a container. As an application developer, you’ll probably know best what behavior to expect from the application at runtime. Ensure that the correct configuration has been created and inspect the running processes by using the Unix or Windows utility tools, depending on the image run in the container.

Say you encounter a situation where a Pod seems to work properly on the surface, as shown in Example 5-5.

**Example 5-5. A Pod periodically writing the current date to a file**

apiVersion: v1

kind: Pod

metadata:

name: failing-pod

spec:

containers:

- args:

- /bin/sh

- -c

- while true; do echo $(date) >> ~/tmp/curr-date.txt; sleep \

5; done;

image: busybox

name: failing-pod

After creating the Pod, you check the status. It says Running; however, when making a request to the application, the endpoint reports an error. Next, you check the logs. The log output renders an error message that points to a nonexistent directory. Apparently, the directory hasn’t been set up correctly but is needed by the application:

**$ kubectl create -f failing-pod.yaml**

pod/failing-pod created

**$ kubectl get pods failing-pod**

NAME READY STATUS RESTARTS AGE

failing-pod 1/1 Running 0 5s

**$ kubectl logs failing-pod**

/bin/sh: can't create /root/tmp/curr-date.txt: nonexistent directory

The exec command opens an interactive shell to further investigate the issue. Below, we’re using the Unix tools mkdir, cd, and ls inside of the running container to fix the problem. Obviously, the better mitigation strategy is to create the directory from the application or provide an instruction in the Dockerfile:

**$ kubectl exec failing-pod -it -- /bin/sh**

# mkdir -p ~/tmp

# cd ~/tmp

# ls -l

total 4

-rw-r--r-- 1 root root 112 May 9 23:52 curr-date.txt

**Using an Ephemeral Container**

Some images run in a container are designed to be very minimalistic. For example, the Google distroless images don’t have any Unix utility tools preinstalled. You can’t even open a shell to a container, as it doesn’t even come with a shell. That’s the case for the image k8s.gcr.io/pause:3.1, a minimal, distroless image that keeps the container running, used in Example 5-6.

**Example 5-6. Running a distroless image**

apiVersion: v1

kind: Pod

metadata:

name: minimal-pod

spec:

containers:

- image: k8s.gcr.io/pause:3.1

name: pause

As you can see in the following exec command, the image doesn’t provide a shell:

**$ kubectl create -f minimal-pod.yaml**

pod/minimal-pod created

**$ kubectl get pods minimal-pod**

NAME READY STATUS RESTARTS AGE

minimal-pod 1/1 Running 0 8s

**$ kubectl alpha debug -it minimal-pod --image=busybox**

Defaulting debug container name to debugger-jf98g.

If you don't see a command prompt, try pressing enter.

/ # pwd

/

/ # exit

Session ended, resume using 'kubectl alpha attach minimal-pod -c \

debugger-jf98g -i -t' command when the pod is running

**Troubleshooting Services**

A Service provides a unified network interface for Pods. Here, we want to point out troubleshooting techniques for this primitive. In case you can’t reach the Pods that should map to the Service, start by ensuring that the label selector matches with the assigned labels of the Pods. You can query the information by describing the Service and then render the labels of the available Pods with the option --show-labels. The following example does not have matching labels and therefore wouldn’t apply to any of the Pods running in the namespace:

**$ kubectl describe service example-service**

...

Selector: app=myapp

...

**$ kubectl get pods --show-labels**

NAME READY STATUS RESTARTS AGE LABELS

myapp-68bf896d89-qfhlv 1/1 Running 0 7m39s app=hello

myapp-68bf896d89-tzt55 1/1 Running 0 7m37s app=world

Alternatively, you can also query the endpoints of the Service instance. Say you expected three Pods to be selected by a matching label but only two have been exposed by the Service. You’ll want to look at the label selection criteria:

**$ kubectl get endpoints example-service**

NAME ENDPOINTS AGE

example-service 172.17.0.5:80,172.17.0.6:80 9m31s

A common source of confusion is the type of a Service. By default, the Service type is ClusterIP, which means that a Pod can only be reached through the Service if queried from the same node inside of the cluster. First, check the Service type. If you think that ClusterIP is the proper type you wanted to assign, open an interactive shell from a temporary Pod inside the cluster and run a curl or wget command:

**$ kubectl get services**

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

example-service ClusterIP 10.99.155.165 <none> 80/TCP 15m

**$ kubectl run tmp --image=busybox -it --rm -- wget -O- 10.99.155.165:80**

....

Finally, check if the port mapping from the target port of the Service to the container port of the Pod is configured correctly. Both ports need to match:

**$ kubectl get service myapp -o yaml | grep targetPort:**

targetPort: 80

**$ kubectl get pods myapp-68bf896d89-qfhlv -o yaml | grep containerPort:**

- containerPort: 80

Monitoring

Deploying software to a Kubernetes cluster is only the start of operating an application long-term. Developers need to understand resource consumption patterns and behaviors of their applications with the goal of providing a scalable and reliable service.

In the Kubernetes world, monitoring tools like Prometheus and Datadog help with collecting, processing, and visualizing the information over time. The CKAD exam does not expect you to be familiar with commercial monitoring, logging, tracing, and aggregation tools; however, it is helpful to gain a rough understanding of the underlying Kubernetes infrastructure responsible for collecting usage metrics, such as a container’s CPU and memory usage.

This responsibility falls into the hands of the metrics server, a cluster-wide aggregator of resource usage data. Refer to the documentation for more information on its installation process. If you’re using Minikube as your practice environment, enabling the metrics-server add-on is straightforward using the following command:

**$ minikube addons enable metrics-server**

The 'metrics-server' addon is enabled

You can now query for metrics of cluster nodes and Pods with the top command:

**$ kubectl top nodes**

NAME CPU(cores) CPU% MEMORY(bytes) MEMORY%

minikube 283m 14% 1262Mi 32%

**$ kubectl top pod frontend**

NAME CPU(cores) MEMORY(bytes)

frontend 0m 2Mi

Summary

Diagnosing the root cause of runtime issues for an application operated in a Kubernetes cluster can be difficult and tedious. Observability covers health probing, logging, monitoring, and debugging of cloud native services.

One of the important takeaways is the fact that Kubernetes can act upon certain failure conditions automatically without the need for manual intervention from a system administrator or application developer. In this chapter, we looked at all available health probe types you can define for a Pod. A health probe is a periodically running mini process that asks the application running in a container for its status. You can think of it as feeling the pulse of your system.

The readiness probe ensures that incoming traffic is only accepted by the container if the application runs properly. The liveness probe makes sure that the application is functioning as expected and will restart the container if necessary. The startup probe pauses readiness and liveness probes until application startup has been completed. In practice, you’ll often find that a container defines all three probes.

We also discussed strategies for approaching failed or misbehaving Pods. The main goal is to diagnose the root cause of a failure and then fix it by taking the right action. Troubleshooting Pods doesn’t have to be hard. With the right kubectl commands in your tool belt, you can rule out root causes one by one to get a clearer picture.

The Kubernetes ecosystem provides a lot of options for collecting and processing metrics of your cluster over time. Among those options are commercial monitoring tools like Prometheus and Datadog. Many of those tools use the metrics server as the source of truth for those metrics. We briefly touched on the installation process and the kubectl top command for retrieving metrics from the command line.

Exam Essentials

*Understand all health probes*

In preparation for the exam, put the majority of effort in this section into understanding and using health probes. You should understand the purpose of startup, readiness, and liveness probes and practice how to configure them. In your Kubernetes cluster, try to emulate success and failure conditions to see the effects of a probe and the actions they take.

*Know how to debug Kubernetes objects*

In this chapter, we mainly focused on troubleshooting problematic Pods and Services. Practice all relevant kubectl commands that can help with diagnosing issues. Refer to the Kubernetes documentation to learn more about debugging other Kubernetes resource types.

*Have a basic understanding of monitoring*

Monitoring a Kubernetes cluster is an important aspect of successfully operating in a real-world environment. This topic mostly consists of reading up on commercial monitoring products and the metrics that can be gathered.

Sample Exercises

1. Define a new Pod named webserver with the image nginx in a YAML manifest. Expose the container port 80. Do not create the Pod yet.
2. For the container, declare a startup probe of type httpGet. Verify that the root context endpoint can be called. Use the default configuration for the probe.
3. For the container, declare a readiness probe of type httpGet. Verify that the root context endpoint can be called. Wait five seconds before checking for the first time.
4. For the container, declare a liveness probe of type httpGet. Verify that the root context endpoint can be called. Wait 10 seconds before checking for the first time. The probe should run the check every 30 seconds.
5. Create the Pod and follow the lifecycle phases of the Pod during the process.
6. Inspect the runtime details of the probes of the Pod.
7. Retrieve the metrics of the Pod (e.g., CPU and memory) from the metrics server.
8. Create a Pod named custom-cmd with the image busybox. The container should run the command top-analyzer with the command-line flag --all.
9. Inspect the status. How would you further troubleshoot the Pod to identify the root cause of the failure?

**Logging Architecture**

Application logs can help you understand what is happening inside your application. The logs are particularly useful for debugging problems and monitoring cluster activity. Most modern applications have some kind of logging mechanism. Likewise, container engines are designed to support logging. The easiest and most adopted logging method for containerized applications is writing to standard output and standard error streams.

However, the native functionality provided by a container engine or runtime is usually not enough for a complete logging solution.

For example, you may want to access your application's logs if a container crashes, a pod gets evicted, or a node dies.

In a cluster, logs should have a separate storage and lifecycle independent of nodes, pods, or containers. This concept is called [cluster-level logging](https://kubernetes.io/docs/concepts/cluster-administration/logging/" \l "cluster-level-logging-architectures).

Cluster-level logging architectures require a separate backend to store, analyze, and query logs. Kubernetes does not provide a native storage solution for log data. Instead, there are many logging solutions that integrate with Kubernetes. The following sections describe how to handle and store logs on nodes.

Pod and container logs

Kubernetes captures logs from each container in a running Pod.

This example uses a manifest for a Pod with a container that writes text to the standard output stream, once per second.

[debug/counter-pod.yaml](https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/debug/counter-pod.yaml)

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: counter

**spec**:

**containers**:

- **name**: count

**image**: busybox:1.28

**args**: [/bin/sh, -c,

'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done']

To run this pod, use the following command:

kubectl apply -f https://k8s.io/examples/debug/counter-pod.yaml

The output is:

pod/counter created

To fetch the logs, use the kubectl logs command, as follows:

kubectl logs counter

The output is similar to:

0: Fri Apr 1 11:42:23 UTC 2022

1: Fri Apr 1 11:42:24 UTC 2022

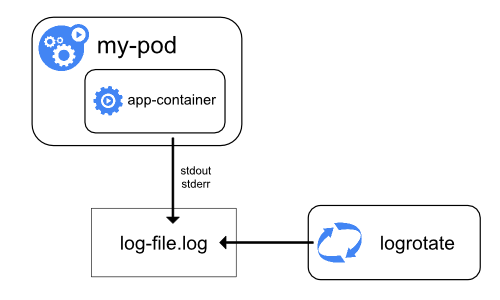
2: Fri Apr 1 11:42:25 UTC 2022

You can use kubectl logs --previous to retrieve logs from a previous instantiation of a container. If your pod has multiple containers, specify which container's logs you want to access by appending a container name to the command, with a -c flag, like so:

kubectl logs counter -c count

See the [kubectl logs documentation](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "logs) for more details.

How nodes handle container logs



A container runtime handles and redirects any output generated to a containerized application's stdout and stderr streams. Different container runtimes implement this in different ways; however, the integration with the kubelet is standardized as the *CRI logging format*.

By default, if a container restarts, the kubelet keeps one terminated container with its logs. If a pod is evicted from the node, all corresponding containers are also evicted, along with their logs.

The kubelet makes logs available to clients via a special feature of the Kubernetes API. The usual way to access this is by running kubectl logs.

Log rotation

**FEATURE STATE:** Kubernetes v1.21 [stable]

You can configure the kubelet to rotate logs automatically.

If you configure rotation, the kubelet is responsible for rotating container logs and managing the logging directory structure. The kubelet sends this information to the container runtime (using CRI), and the runtime writes the container logs to the given location.

You can configure two kubelet [configuration settings](https://kubernetes.io/docs/reference/config-api/kubelet-config.v1beta1/" \l "kubelet-config-k8s-io-v1beta1-KubeletConfiguration), containerLogMaxSize and containerLogMaxFiles, using the [kubelet configuration file](https://kubernetes.io/docs/tasks/administer-cluster/kubelet-config-file/). These settings let you configure the maximum size for each log file and the maximum number of files allowed for each container respectively.

When you run [kubectl logs](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "logs) as in the basic logging example, the kubelet on the node handles the request and reads directly from the log file. The kubelet returns the content of the log file.

**Note:**

Only the contents of the latest log file are available through kubectl logs.

For example, if a Pod writes 40 MiB of logs and the kubelet rotates logs after 10 MiB, running kubectl logs returns at most 10MiB of data.

System component logs

There are two types of system components: those that typically run in a container, and those components directly involved in running containers. For example:

* The kubelet and container runtime do not run in containers. The kubelet runs your containers (grouped together in [pods](https://kubernetes.io/docs/concepts/workloads/pods/" \t "_blank))
* The Kubernetes scheduler, controller manager, and API server run within pods (usually [static Pods](https://kubernetes.io/docs/tasks/configure-pod-container/static-pod/" \t "_blank)). The etcd component runs in the control plane, and most commonly also as a static pod. If your cluster uses kube-proxy, you typically run this as a DaemonSet.

Log locations

The way that the kubelet and container runtime write logs depends on the operating system that the node uses:

* [Linux](https://kubernetes.io/docs/concepts/cluster-administration/logging/" \l "log-location-node-tabs-0)
* [Windows](https://kubernetes.io/docs/concepts/cluster-administration/logging/" \l "log-location-node-tabs-1)

On Linux nodes that use systemd, the kubelet and container runtime write to journald by default. You use journalctl to read the systemd journal; for example: journalctl -u kubelet.

If systemd is not present, the kubelet and container runtime write to .log files in the /var/log directory. If you want to have logs written elsewhere, you can indirectly run the kubelet via a helper tool, kube-log-runner, and use that tool to redirect kubelet logs to a directory that you choose.

You can also set a logging directory using the deprecated kubelet command line argument --log-dir. However, the kubelet always directs your container runtime to write logs into directories within /var/log/pods.

For more information on kube-log-runner, read [System Logs](https://kubernetes.io/docs/concepts/cluster-administration/system-logs/" \l "klog).

For Kubernetes cluster components that run in pods, these write to files inside the /var/log directory, bypassing the default logging mechanism (the components do not write to the systemd journal). You can use Kubernetes' storage mechanisms to map persistent storage into the container that runs the component.

For details about etcd and its logs, view the [etcd documentation](https://etcd.io/docs/). Again, you can use Kubernetes' storage mechanisms to map persistent storage into the container that runs the component.

**Note:**

If you deploy Kubernetes cluster components (such as the scheduler) to log to a volume shared from the parent node, you need to consider and ensure that those logs are rotated. **Kubernetes does not manage that log rotation**.

Your operating system may automatically implement some log rotation - for example, if you share the directory /var/log into a static Pod for a component, node-level log rotation treats a file in that directory the same as a file written by any component outside Kubernetes.

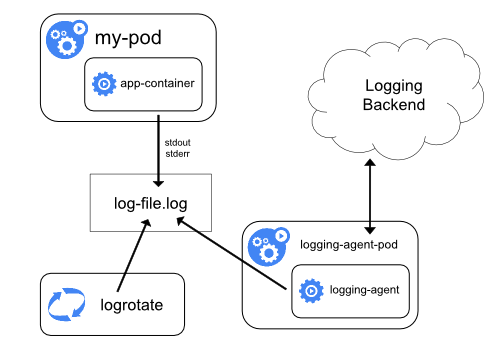
Some deploy tools account for that log rotation and automate it; others leave this as your responsibility.

Cluster-level logging architectures

While Kubernetes does not provide a native solution for cluster-level logging, there are several common approaches you can consider. Here are some options:

* Use a node-level logging agent that runs on every node.
* Include a dedicated sidecar container for logging in an application pod.
* Push logs directly to a backend from within an application.

Using a node logging agent



You can implement cluster-level logging by including a *node-level logging agent* on each node. The logging agent is a dedicated tool that exposes logs or pushes logs to a backend. Commonly, the logging agent is a container that has access to a directory with log files from all of the application containers on that node.

Because the logging agent must run on every node, it is recommended to run the agent as a DaemonSet.

Node-level logging creates only one agent per node and doesn't require any changes to the applications running on the node.

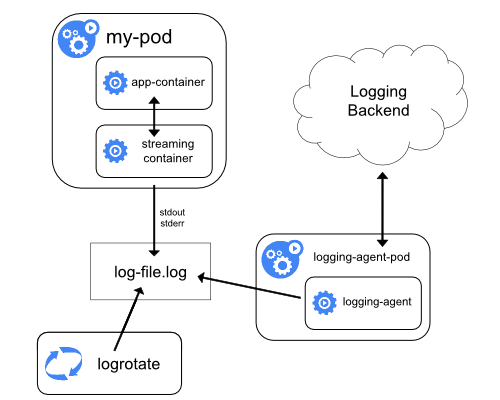
Containers write to stdout and stderr, but with no agreed format. A node-level agent collects these logs and forwards them for aggregation.

Using a sidecar container with the logging agent

You can use a sidecar container in one of the following ways:

* The sidecar container streams application logs to its own stdout.
* The sidecar container runs a logging agent, which is configured to pick up logs from an application container.

Streaming sidecar container



By having your sidecar containers write to their own stdout and stderr streams, you can take advantage of the kubelet and the logging agent that already run on each node. The sidecar containers read logs from a file, a socket, or journald. Each sidecar container prints a log to its own stdout or stderr stream.

This approach allows you to separate several log streams from different parts of your application, some of which can lack support for writing to stdout or stderr. The logic behind redirecting logs is minimal, so it's not a significant overhead. Additionally, because stdout and stderr are handled by the kubelet, you can use built-in tools like kubectl logs.

For example, a pod runs a single container, and the container writes to two different log files using two different formats. Here's a manifest for the Pod:

[admin/logging/two-files-counter-pod.yaml](https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/admin/logging/two-files-counter-pod.yaml)

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: counter

**spec**:

**containers**:

- **name**: count

**image**: busybox:1.28

**args**:

- /bin/sh

- -c

- >

*i=0;*

*while true;*

*do*

*echo "$i: $(date)" >> /var/log/1.log;*

*echo "$(date) INFO $i" >> /var/log/2.log;*

*i=$((i+1));*

*sleep 1;*

*done*

**volumeMounts**:

- **name**: varlog

**mountPath**: /var/log

**volumes**:

- **name**: varlog

**emptyDir**: {}

It is not recommended to write log entries with different formats to the same log stream, even if you managed to redirect both components to the stdout stream of the container. Instead, you can create two sidecar containers. Each sidecar container could tail a particular log file from a shared volume and then redirect the logs to its own stdout stream.

Here's a manifest for a pod that has two sidecar containers:

[admin/logging/two-files-counter-pod-streaming-sidecar.yaml](https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/admin/logging/two-files-counter-pod-streaming-sidecar.yaml)

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: counter

**spec**:

**containers**:

- **name**: count

**image**: busybox:1.28

**args**:

- /bin/sh

- -c

- >

*i=0;*

*while true;*

*do*

*echo "$i: $(date)" >> /var/log/1.log;*

*echo "$(date) INFO $i" >> /var/log/2.log;*

*i=$((i+1));*

*sleep 1;*

*done*

**volumeMounts**:

- **name**: varlog

**mountPath**: /var/log

- **name**: count-log-1

**image**: busybox:1.28

**args**: [/bin/sh, -c, 'tail -n+1 -F /var/log/1.log']

**volumeMounts**:

- **name**: varlog

**mountPath**: /var/log

- **name**: count-log-2

**image**: busybox:1.28

**args**: [/bin/sh, -c, 'tail -n+1 -F /var/log/2.log']

**volumeMounts**:

- **name**: varlog

**mountPath**: /var/log

**volumes**:

- **name**: varlog

**emptyDir**: {}

Now when you run this pod, you can access each log stream separately by running the following commands:

kubectl logs counter count-log-1

The output is similar to:

0: Fri Apr 1 11:42:26 UTC 2022

1: Fri Apr 1 11:42:27 UTC 2022

2: Fri Apr 1 11:42:28 UTC 2022

...

kubectl logs counter count-log-2

The output is similar to:

Fri Apr 1 11:42:29 UTC 2022 INFO 0

Fri Apr 1 11:42:30 UTC 2022 INFO 0

Fri Apr 1 11:42:31 UTC 2022 INFO 0

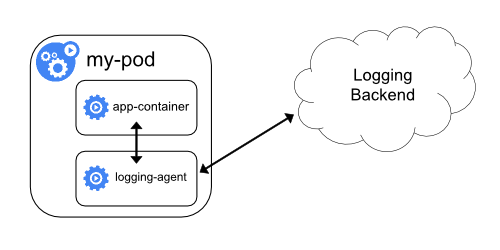
...

If you installed a node-level agent in your cluster, that agent picks up those log streams automatically without any further configuration. If you like, you can configure the agent to parse log lines depending on the source container.

Even for Pods that only have low CPU and memory usage (order of a couple of millicores for cpu and order of several megabytes for memory), writing logs to a file and then streaming them to stdout can double how much storage you need on the node. If you have an application that writes to a single file, it's recommended to set /dev/stdout as the destination rather than implement the streaming sidecar container approach.

Sidecar containers can also be used to rotate log files that cannot be rotated by the application itself. An example of this approach is a small container running logrotate periodically. However, it's more straightforward to use stdout and stderr directly, and leave rotation and retention policies to the kubelet.

Sidecar container with a logging agent



If the node-level logging agent is not flexible enough for your situation, you can create a sidecar container with a separate logging agent that you have configured specifically to run with your application.

**Note:** Using a logging agent in a sidecar container can lead to significant resource consumption. Moreover, you won't be able to access those logs using kubectl logs because they are not controlled by the kubelet.

Here are two example manifests that you can use to implement a sidecar container with a logging agent. The first manifest contains a [ConfigMap](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/) to configure fluentd.

[admin/logging/fluentd-sidecar-config.yaml](https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/admin/logging/fluentd-sidecar-config.yaml)

**apiVersion**: v1

**kind**: ConfigMap

**metadata**:

**name**: fluentd-config

**data**:

**fluentd.conf**: |

*<source>*

*type tail*

*format none*

*path /var/log/1.log*

*pos\_file /var/log/1.log.pos*

*tag count.format1*

*</source>*

*<source>*

*type tail*

*format none*

*path /var/log/2.log*

*pos\_file /var/log/2.log.pos*

*tag count.format2*

*</source>*

*<match \*\*>*

*type google\_cloud*

*</match>*

**Note:** In the sample configurations, you can replace fluentd with any logging agent, reading from any source inside an application container.

The second manifest describes a pod that has a sidecar container running fluentd. The pod mounts a volume where fluentd can pick up its configuration data.

[admin/logging/two-files-counter-pod-agent-sidecar.yaml](https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/admin/logging/two-files-counter-pod-agent-sidecar.yaml)

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: counter

**spec**:

**containers**:

- **name**: count

**image**: busybox:1.28

**args**:

- /bin/sh

- -c

- >

*i=0;*

*while true;*

*do*

*echo "$i: $(date)" >> /var/log/1.log;*

*echo "$(date) INFO $i" >> /var/log/2.log;*

*i=$((i+1));*

*sleep 1;*

*done*

**volumeMounts**:

- **name**: varlog

**mountPath**: /var/log

- **name**: count-agent

**image**: registry.k8s.io/fluentd-gcp:1.30

**env**:

- **name**: FLUENTD\_ARGS

**value**: -c /etc/fluentd-config/fluentd.conf

**volumeMounts**:

- **name**: varlog

**mountPath**: /var/log

- **name**: config-volume

**mountPath**: /etc/fluentd-config

**volumes**:

- **name**: varlog

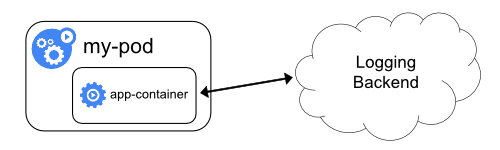
**emptyDir**: {}

- **name**: config-volume

**configMap**:

**name**: fluentd-config

Exposing logs directly from the application



Cluster-logging that exposes or pushes logs directly from every application is outside the scope of Kubernetes.

To scale an application and provide a reliable service, you need to understand how the application behaves when it is deployed. You can examine application performance in a Kubernetes cluster by examining the containers, [pods](https://kubernetes.io/docs/concepts/workloads/pods/), [services](https://kubernetes.io/docs/concepts/services-networking/service/), and the characteristics of the overall cluster. Kubernetes provides detailed information about an application's resource usage at each of these levels. This information allows you to evaluate your application's performance and where bottlenecks can be removed to improve overall performance.

In Kubernetes, application monitoring does not depend on a single monitoring solution. On new clusters, you can use [resource metrics](https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-usage-monitoring/" \l "resource-metrics-pipeline) or [full metrics](https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-usage-monitoring/" \l "full-metrics-pipeline) pipelines to collect monitoring statistics.

Resource metrics pipeline

The resource metrics pipeline provides a limited set of metrics related to cluster components such as the [Horizontal Pod Autoscaler](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/) controller, as well as the kubectl top utility. These metrics are collected by the lightweight, short-term, in-memory [metrics-server](https://github.com/kubernetes-sigs/metrics-server) and are exposed via the metrics.k8s.io API.

metrics-server discovers all nodes on the cluster and queries each node's [kubelet](https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/) for CPU and memory usage. The kubelet acts as a bridge between the Kubernetes master and the nodes, managing the pods and containers running on a machine. The kubelet translates each pod into its constituent containers and fetches individual container usage statistics from the container runtime through the container runtime interface. If you use a container runtime that uses Linux cgroups and namespaces to implement containers, and the container runtime does not publish usage statistics, then the kubelet can look up those statistics directly (using code from [cAdvisor](https://github.com/google/cadvisor)). No matter how those statistics arrive, the kubelet then exposes the aggregated pod resource usage statistics through the metrics-server Resource Metrics API. This API is served at /metrics/resource/v1beta1 on the kubelet's authenticated and read-only ports.

Full metrics pipeline

A full metrics pipeline gives you access to richer metrics. Kubernetes can respond to these metrics by automatically scaling or adapting the cluster based on its current state, using mechanisms such as the Horizontal Pod Autoscaler. The monitoring pipeline fetches metrics from the kubelet and then exposes them to Kubernetes via an adapter by implementing either the custom.metrics.k8s.io or external.metrics.k8s.io API.

[Prometheus](https://prometheus.io/), a CNCF project, can natively monitor Kubernetes, nodes, and Prometheus itself. Full metrics pipeline projects that are not part of the CNCF are outside the scope of Kubernetes documentation.

7

Maintenance and Monitoring

In the previous chapter we learned more about Pod design and application deployment configured to securely access and consume the data within the containers. In this chapter we will be learning more about the Pod startup and readiness that ensure the availability and reliability of applications running inside the Pod’s container.

In this chapter, we’ll discuss container health probes—more specifically, startup, readiness, and liveness probes. You’ll learn about the different health verification methods and how to define them for the proper use cases. Moreover, the exam expects you to be familiar with strategies for debugging a misconfigured or misbehaving Kubernetes object. We’ll round out the chapter by talking about monitoring cluster nodes and Pods.

At a high level, this chapter covers the following concepts:

* Kubernetes monitoring
* Cluster logs
* Pods and Nodes monitoring
* Cluster Observability
* How to take a Kubernetes Cluster node out for Maintenance
  + Step 1: Check all the Pods
  + Step 2: Verify all the Nodes
  + Step 3: Drain out the Node
  + Step 4: Verify all the apps
  + Step 5: Schedule the Node
  + Step 6: Verify the Cluster
  + Step 7: Verify Pods Status

# OS Upgrades

From time to time, a node or a group of nodes may be needed to be taken down for maintenance. For example, the base software may be needed to be upgraded, there is an urgent security patch, or just a regular operating system upgrade.

Consider a scenario where a node A needs to be taken down for maintenance.

* In situation, a green pod only runs on Node A, it doesn’t even belong in a replica set for argument’s sake.
* Red pods belong in a replica set

What will Kubernetes do if Node A is taken down?

Chart, bubble chart

Description automatically generated

Initial pods running on two nodes. Node A has to be taken down for maintenance

First, we have to understand pod eviction timeout.

**Pod eviction timeout:**

* The time the cluster waits for a pod to come online.
* It waits for 5 mins before it considers the green pod dead.

By default, it is set to 5 minutes which can be set with:

kube-controller-manager --pod-eviction-timeout=5m0s

Now that Node A is offline:

* The red pod is assigned to Node B because it is part of a replica set
* Kubernetes will wait for 5 minutes before considering the green pod dead.

Chart, bubble chart

Description automatically generated

Killing in the name of Node A

A quick hacky way to perform an upgrade is by trying to do this maintenance within the pod eviction timeout. For example, if you can tolerate some downtime and ensure that a node comes back online in 5mins, then the above strategy is an option.

However, there is a safer way of doing this by draining node.

## Draining a node

An alternative to drain a node using the command:

kubectl drain <node>

This will ensure that all workloads move away from a node that has been drained.

“Moving away” means:

* green and red pods are gracefully terminated and
* recreated in Node B

Chart, bubble chart

Description automatically generated

Draining Node A is safer. After workloads have been migrated away to another Node, Node A has been marked as cordoned

Node A is now marked as cordoned which means that no pods can be scheduled on this node.

Once the node has been upgraded and ready to be used, it can be uncordoned with the command:

kubectl uncordon <node>

This means that new workloads are ready to be scheduled on to the newly maintained node.

If you don’t want a node to be drained but just be marked as unschedulable, you can use the command:

kubectl cordon <node>

Chart, bubble chart

Description automatically generated

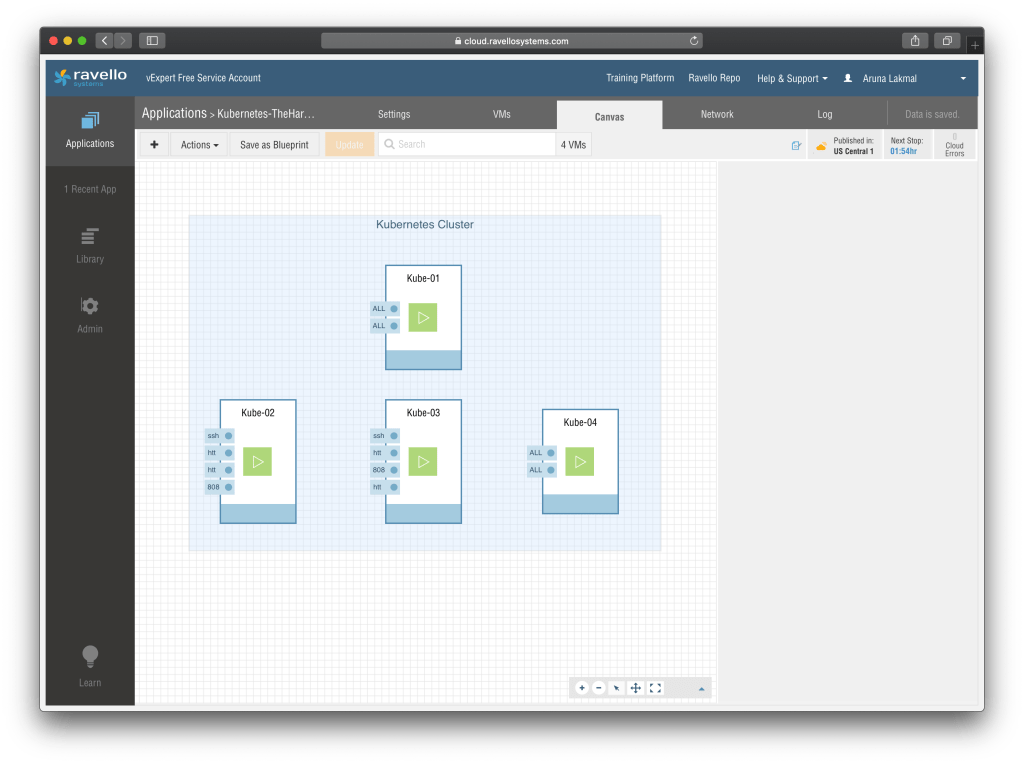
Node A is not accepting new pods, it is cordoned but not drained.

The strategy of cordoning a node maybe useful if you have some pods that can’t afford a downtime. Depending on your application, this strategy may give you more time to move it over to a different node.

Cordoning a node is also useful if a node has pods that have not been defined in a replica set. If you’ve actually tried the earlier drain step and try to train a node that has pods not in a replica set, the drain command will fail. In this case, a cordon command will be useful to manage these pods that do not belong in a replica set.

In terms of node maintenance, we all need to accept that the compute resources such as physical servers or cloud instances need to have a downtime for maintenance works such as hardware changes, Operating system upgrade or any other maintenance work. Same theory applies to the Kubernetes Cluster Nodes and lets see how we can perform the Kubernetes Node Management, Maintenance and editing PODs without interrupting to it’s services.

I have four node Kubernetes cluster on Oracle Ravello Cloud, one Master node and three worker nodes with running [Nginx](https://www.nginx.com/" \t "_blank) POD with three replicas.



Lets see how we can perform the Cluster Maintenance work without interrupting the running Services

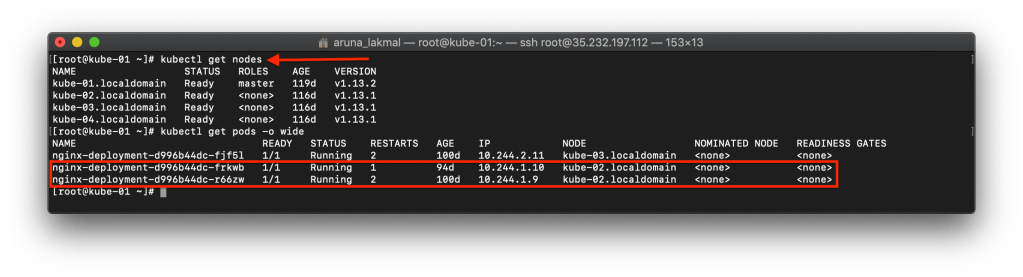
### **Evict PODs and Safely Perform A Node Maintenance**

To view the Kubernetes node status execute the below commands

kubectl get nodes

To check the running POD status and the respective nodes execute below commands

kubectl get pods -o wide

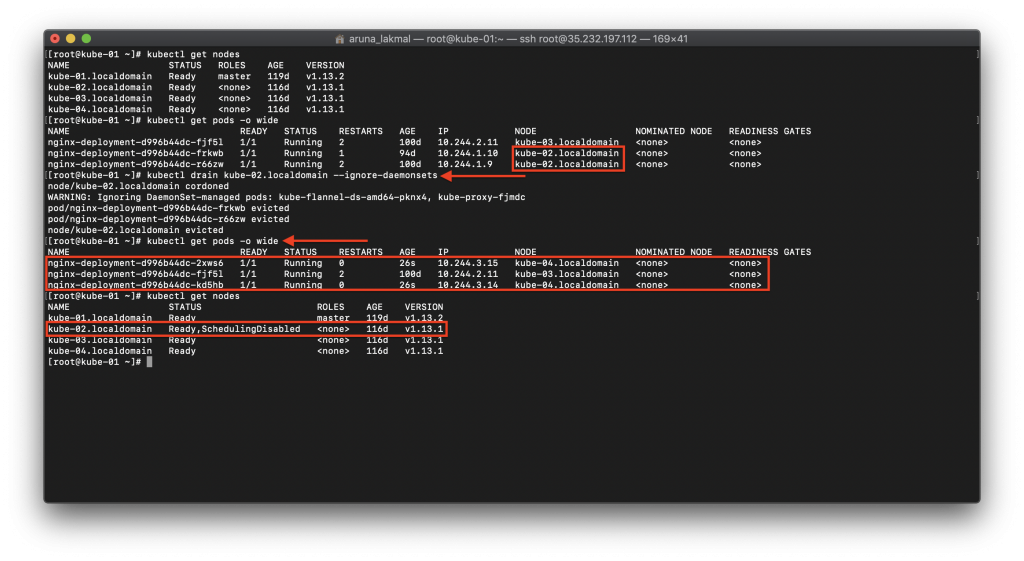


Executing below the command you can safely take a node out from the cluster and running PODs will be evicted from the node

kubectl drain [NODE\_HOSTNAME] --ignore-daemonsets

In the below screen capture two Nginx replicas are running in “kube-02.localdomain” node and after executing the command PODs are running in “kube-04.localdomain” node.

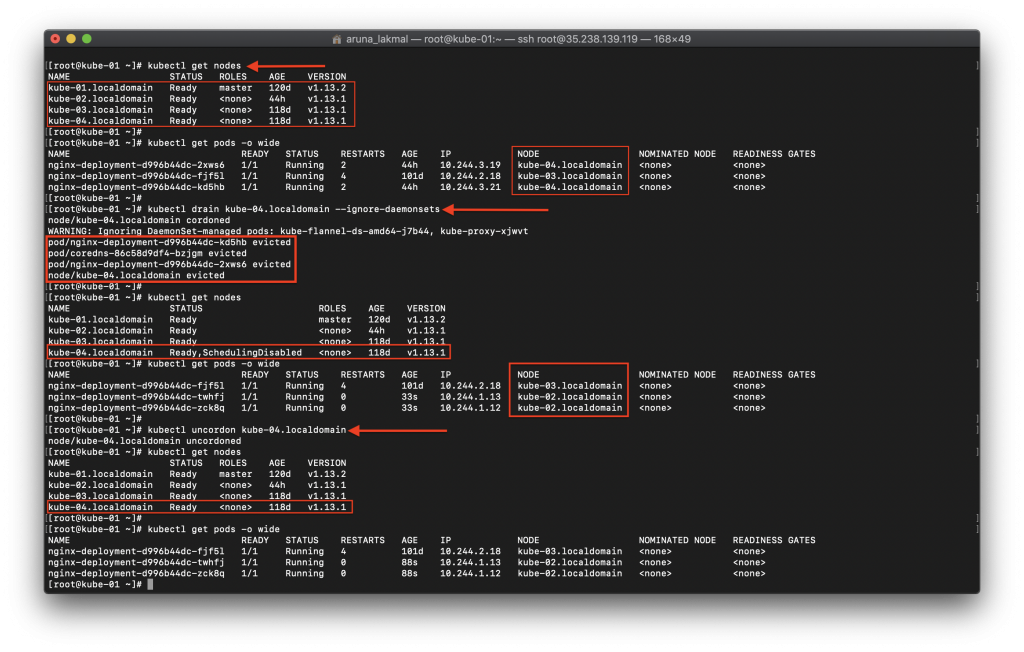
Subjected node which is “Kubelet-02.localdomain” is disabled for POD scheduling after performing this command



After verifying the POD status you can bring down the node and perform the maintenance operation now.

Once, the operation completed we can “**uncordon**” the node and schooling will be enabled again

Complete Operation is explained in the below screen capture

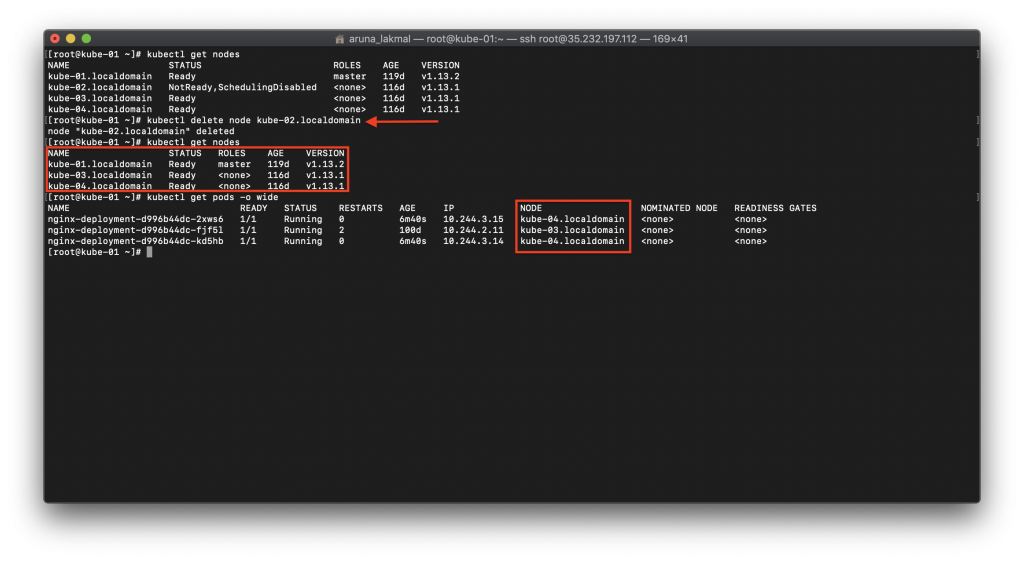


### **Deleting A Node From The Cluster**

If we are planning to delete a node from the Kubernetes cluster, we have to drain, evict the running PODs and disable the POD scheduling first.

To delete the POD below command is used

kubectl delete node [NODE\_NAME]



### **Generate A New Token And Add A New Worker Node**

Lets assume that after deleting a node we are planning to add a new node to our Kubernetes Cluster. We might don’t have the “**kubeadm join**” command and the token with us. We can follow the below procedure to generate a new token and add the worker node to the cluster

To view the token list execute the below command

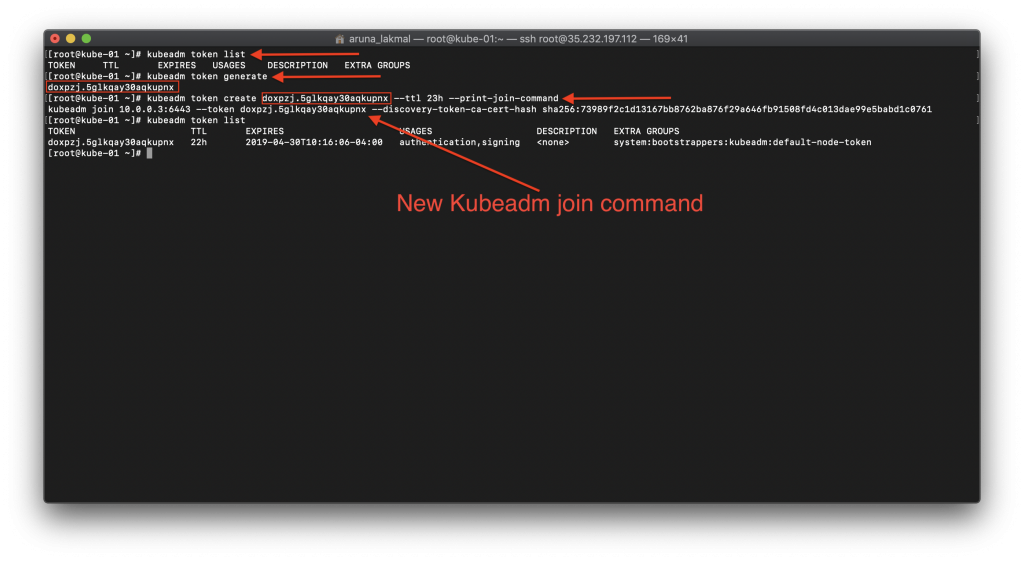
kubeadm token list

Generate a token with below command, if you don’t have a listed token

kubeadm token generate

Take the generated token and create a command to join the worker node with below command

kubeadm token create [GENERATED\_TOKEN] --ttl 23h --print-join-command



Issue this command in the worker node and it will be joining to the cluster. If you need to see how to create a Kubernetes cluster from the scratch read my [previous article](https://www.techcrumble.net/2019/01/deploy-kubernetes-cluster-on-oracle-ravello-with-centos-7/).

In this article, we will understand how to take a Kubernetes Cluster node out for Maintenance. Performing node maintenance is a standard task which needs to be done periodically. Sometimes taking the node out from a Cluster and joining it back to the Cluster can become tricky and cumbersome. Fortunately it is very easy to do in a Kubernetes [Cluster](https://www.cyberithub.com/best-kubectl-command-examples/). You just have to follow the process well. Here I am going to explain you the entire process with the help of an example node so that it will be easy for you to understand and perform in a production kind of environment. But before that let's quickly go through our lab setup.

## **Lab Setup**

We are having one worker node called "node-1" and one master called "master" in our [Kubernetes Cluster](https://www.cyberithub.com/how-to-deploy-kubernetes-cluster-using-vagrant-in-7-simple-steps/). We are having three test pod currently running on node-1 and no pod running on master. Our task is to take the node "node-1" out of the [Kubernetes Cluster](https://www.cyberithub.com/how-to-connect-replicaset-to-a-pod-in-kubernetes-cluster/) for maintenance and then joined it back once the maintenance is completed. It is important to ensure that when we take the node out, all its load should be transferred to master without any hiccups.

## **How to take a Kubernetes Cluster node out for Maintenance**

**Also Read:** **[How to Deploy Metrics Server on a Kubernetes Cluster [Step by Step]](https://www.cyberithub.com/how-to-deploy-metrics-server-on-a-kubernetes-cluster-step-by-step/" \t "_blank)**

### **Step 1: Check all the Pods**

First let's verify all the pods are running fine by using [kubectl get pods -o wide command](https://www.cyberithub.com/15-kubectl-kubeadm-commands-example-kubernetes/). This command will show detailed output which includes the pod status, node on which it is running, number of restart it had, Age, IP etc.

root@master:~# kubectl get pods -o wide

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

test-746c87566d-6fzjx 1/1 Running 0 11m 10.253.1.2 node-1 <none> <none>

test-746c87566d-j49rx 1/1 Running 0 11m 10.253.1.4 node-1 <none> <none>

test-746c87566d-xbnj5 1/1 Running 0 11m 10.253.1.3 node-1 <none> <none>

### **Step 2: Verify all the Nodes**

Then the next step is to verify the health of the nodes by using kubectl get nodes. This should show node status as Ready. If any of the nodes say otherwise, then first you need to fix the node status and then proceed with the next step.

root@master:~# kubectl get nodes

NAME STATUS ROLES AGE VERSION

master Ready master 20m v1.20.0

node-1 Ready <none> 19m v1.20.0

### **Step 3: Drain out the Node**

Now you can move ahead with the maintenance and drain out first node-1 using kubectl drain node-1 command as shown below. The idea of drain out is to make sure all the loads of node-1 gets transferred to other available node in the Cluster. In our case, since we have only two nodes i.e master and node-1 so in a successful drain out all the loads of node-1 should get transferred to master. But hey what's showing below. It looks like there is some error and it is unable to drain node "node-1". If you are also facing this kind of error then here you need to add one more option i.e --ignore-daemonsets to get through this.

If there are daemon set-managed pods, drain will not proceed without --ignore-daemonsets, and regardless it will not delete any daemon set-managed pods, because those pods would be immediately replaced by the daemon set controller, which ignores unschedulable markings. Hence we need to use this option. More on Kubernetes [official](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "drain" \t "_blank) documentation.

Advertisements

root@master:~# kubectl drain node-1

node/node-1 cordoned

error: unable to drain node "node-1", aborting command...

There are pending nodes to be drained:

node-1

cannot delete Pods not managed by ReplicationController, ReplicaSet, Job, DaemonSet or StatefulSet (use --force to override): default/simple-webapp-1

cannot delete DaemonSet-managed Pods (use --ignore-daemonsets to ignore): kube-system/kube-flannel-ds-xngfr, kube-system/kube-proxy-hsq9m

So now you have to run kubectl drain node-1 --ignore-daemonsets command as shown below. This will mark the node unschedulable to prevent new pods from arriving. You have to wait till all the pods get evicted.

root@master:~# kubectl drain node-1 --ignore-daemonsets

node/node-1 cordoned

WARNING: ignoring DaemonSet-managed Pods: kube-system/kube-flannel-ds-g2v5g, kube-system/kube-proxy-fsfjt

evicting pod default/test-746c87566d-xbnj5

evicting pod default/test-746c87566d-6fzjx

evicting pod default/test-746c87566d-j49rx

pod/test-746c87566d-j49rx evicted

pod/test-746c87566d-6fzjx evicted

pod/test-746c87566d-xbnj5 evicted

node/node-1 evicted

### **Step 4: Verify all the apps**

Now you can verify if all the pods got transferred to master by using kubectl get pods -o wide command. Then you can see that indeed it is transferred and running fine. So this confirms that node-1 has been successfully unscheduled from running any apps. Now you can proceed with the maintenance of node "node-1".

root@master:~# kubectl get pods -o wide

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

test-746c87566d-8h8gz 1/1 Running 0 80s 10.253.0.5 master <none> <none>

test-746c87566d-k5mxz 1/1 Running 0 80s 10.253.0.6 master <none> <none>

test-746c87566d-zslkf 1/1 Running 0 80s 10.253.0.4 master <none> <none>

### **Step 5: Schedule the Node**

After maintenance is completed, we need to make the node schedulable again by bringing it back to the cluster using kubectl uncordon node-1 command as shown below.

root@master:~# kubectl uncordon node-1

node/node-1 uncordoned

### **Step 6: Verify the Cluster**

To verify if the node is back in cluster, you can run kubectl get nodes command. If you see the status as Ready then it confirms that node-1 successfully scheduled for arriving pods.

root@master:~# kubectl get nodes

NAME STATUS ROLES AGE VERSION

master Ready master 27m v1.20.0

node-1 Ready <none> 27m v1.20.0

### **Step 7: Verify Pods Status**

But here is another interesting output you can notice when you run kubectl get pods -o wide command again. You can notice that currently there are no pods running on node-1 even after the node has been successfully joined back. Well, it is simply because we had removed this node for maintenance and now it will only schedule when new pod arrives.

root@master:~# kubectl get pods -o wide

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

test-746c87566d-8h8gz 1/1 Running 0 9m30s 10.253.0.5 master <none> <none>

test-746c87566d-k5mxz 1/1 Running 0 9m30s 10.253.0.6 master <none> <none>

test-746c87566d-zslkf 1/1 Running 0 9m30s 10.253.0.4 master <none> <none>

Another interesting point you might think why all the pods were placed on master when node-1 was taken out for maintenance. Well, it is simply because master node did not had any taint. This you can verify by grepping the taint keyword from kubectl describe nodes master command as shown below.

root@master:~# kubectl describe nodes master | grep -i taint

Taints: <none>

**Quick Summary:-** As applications become more dynamic, the need to improve their performance also increases. Kubernetes Observability offers a complete picture of your infrastructure -- its strengths, gaps, and areas of improvement to help you build a resilient system. This article talks about every significant aspect of implementing Kubernetes Observability for organizations planning to adopt it.

Kubernetes offers a seamless, distributed solution for managing application workloads hosted across cluster nodes. According to the Cloud Native Computing Foundation’s most recent data, Kubernetes is the leading container orchestration system, with [69% of respondents](https://www.containiq.com/post/kubernetes-statistics) using Kubernetes in production.

Kubernetes offers a seamless, distributed solution for managing application workloads hosted across cluster nodes. According to the Cloud Native Computing Foundation’s most recent data, Kubernetes is the leading container orchestration system, with [69% of respondents](https://www.containiq.com/post/kubernetes-statistics) using Kubernetes in production.

Kubernetes offers many benefits, including improved productivity for engineering teams, efficient autoscaling, and cost savings.

And because a [Kubernetes ecosystem](https://loft.sh/blog/100-kubernetes-tutorials-to-get-you-from-zero-to-hero/?utm_medium=reader&utm_source=other&utm_campaign=blog_kubernetes-monitoring-best-practices) comprises multiple machines (nodes) working together within the same subsystem (cluster), making all of these elements observable is a key aspect of maintaining operational efficiency.

In this post, we’ll explore the topic of Kubernetes monitoring, including a brief analysis of some challenges, an overview of the most important metrics to monitor, and some best practices implemented by leading engineering organizations. In addition, we highlight several open-source and commercial tools that allow administrators to monitor component health within a cluster, as well as application performance for the end user.

## [#](https://loft.sh/blog/kubernetes-monitoring-best-practices" \l "the-challenge-of-monitoring-in-distributed-containerized-systems)The Challenge of Monitoring in Distributed, Containerized Systems

Kubernetes is incredibly useful in that it abstracts away a lot of complexity in order to speed up deployment. But because of this, engineers are often left in the dark about how resources are being used — it’s hard to tell what is actually happening.

Containers help package responsive, highly available applications as executable units that allow OS-level abstraction. By design, this enables a distributed framework of containerized applications and services that makes monitoring more intricate. Microservices architectures and stateless application design have many benefits; however, these modern software practices tend to lead to dynamic application architectures,  which hinder observability. This limited observability makes troubleshooting issues with Kubernetes more complicated.

Kubernetes clusters make use of a number of components and processes that further complicate things: traditional machine monitoring mechanisms are ineffective for checking performance parameters of a dynamically transient services-oriented architecture.

And while the Kubernetes Dashboard allows users to access information on resource utilization, it does not include some of the refined mechanisms needed for monitoring ephemeral application workloads. This is why various mechanisms have been developed to improve monitoring and event logging in Kubernetes clusters.

## [#](https://loft.sh/blog/kubernetes-monitoring-best-practices" \l "what-metrics-are-monitored-in-kubernetes)What Metrics Are Monitored in Kubernetes

The [best monitoring strategies](https://www.containiq.com/post/kubernetes-monitoring-methods-and-tools) closely track several metric types, depending on the components being observed. These are:

* POD/Container Metrics: These metrics help administrators check for resources allocated to every POD in a cluster. This enables them to see whether containers are over-or under-provisioned. Some common container metrics include CPU usage, memory usage, disk usage, and network traffic bandwidth, while common POD metrics include deployment progress, POD health, network data, and the number of running instances.
* Node Metrics: Every node running in a Kubernetes cluster has definite storage, memory, and CPU capacity to be used by PODs and cluster resources. Observing these metrics includes node-level consumption of storage and network bandwidth.
* Cluster Metrics: These metrics encompass data on the performance of a cluster’s control plane components. The performance data of master node components like the API Server, Scheduler, Controllers, and etcd server help to ensure cluster components run in the desired state.
* Application Metrics: These metrics are developed within an application’s code and are written to meet the requirements of the business case. A database application, for instance, can provide performance information on query latency and table storage. Such metrics are usually exposed to Kubernetes monitoring tools through an application programming interface (API).

## [#](https://loft.sh/blog/kubernetes-monitoring-best-practices" \l "10-best-practices-for-monitoring-and-observability-in-kubernetes)10 Best Practices for Monitoring and Observability in Kubernetes

To make the most of a Kubernetes monitoring solution, it’s important to follow practices that provide valuable insights. Some of these include:

1. Enforce deep visibility to make sure every cluster component is observable.
2. Pick an appropriate instrumentation strategy using libraries and sidecar agents so that all critical metrics are collected without fail.
3. Include event logs and historical performance data so that team members can easily trace the root causes of performance problems.
4. Constantly monitor Kubernetes control plane elements to validate the performance of cluster services.
5. Find a toolset that makes for easy and quick correlations between data, including metrics to events, events to logs, and traces to logs.
6. Pick a toolset that provides anomaly detection or intelligent alerting, to avoid alert fatigue. And set alerts only on truly meaningful changes or events.
7. Use a managed service if open-source, for example, a managed Prometheus offering from AWS or GCP, in order to avoid spending time monitoring your monitoring.
8. Choose a SaaS monitoring solution that can be deployed as code for easier maintenance and management.
9. Understand the costs early. If you are using a SaaS solution, take time to estimate what it will cost, including all charges like metric volume, log ingest, and per-seat pricing.
10. Use a toolset that allows for user permissions and role-based access controls. For example, some tools allow users to designate permissions at the cluster and namespace levels.

## [#](https://loft.sh/blog/kubernetes-monitoring-best-practices" \l "methods-of-collecting-data)Methods of Collecting Data

Kubernetes offers two main approaches for collecting data on running clusters: DaemonSets and the Metrics Server.

* A DaemonSet is a Kubernetes object that ensures a number of nodes in the cluster run a copy of a POD. These PODs are used to monitor nodes and collect logs on POD events. This data can then be passed to advanced monitoring solutions for further analysis.
* The Metrics Server is a POD that collects resource consumption data from the Kubelet service on each node and exposes this data to the ‘apiserver’ using the Metrics API. This data can then be used to autoscale an application and adjust resources to be used by containers.

In today’s era of microservices, containers, and containerized applications, software architecture is more complex. Kubernetes is king in this environment, orchestrating an army of Docker containers in more distributed environments. As that architecture meets demand for high availability, reliability, security, and auto-scaling during peak time, the scale of these operations requires robust management of the three pillars of observability—logs, metrics, and traces—to debug and maintain systems. Thus, Kubernetes observability is now a top priority for DevOps teams.

The scale requires the execution of many operations to ensure that all services are always accessible for end users. The good news is that, when there are issues, these operational activities provide information to help developers and IT staff find the root causes of problems and fix them before their customers notice.

## **The Three Pillars in Kubernetes Observability**

The “three pillars of observability” are logs, metrics, and traces. When combined, they provide sufficient insights to monitor software at any scale. Let’s look at how to take advantage of these three pillars to make systems more observable, especially when working in the Kubernetes environment.

### **Logs**

Logs represent discrete events which, in most cases, describe what’s happening with your service. The payload is an actual message emitted from your system along with a metadata section containing a timestamp, labels, and tracking identifiers. The following is a sample log message:

*time=”2019-12-23T01:27:38-04:00″ level=debug msg=”Application starting” environment=dev*

Log messages represent data in a structured textual form. Most programming languages and frameworks support logging out of the box. However, creating log messages often requires a large volume to store logs, so you should carefully consider your storage strategy as your amount of data grows.

In Kubernetes, every container writes its logs to standard output which can be aggregated by a centralized logging solution for future analysis.

### **Metrics**

Metrics are numeric representations of data measured over intervals of time. They are essential for understanding system health and for gaining insights using telemetry signals. The following is an example of a metric:

*http\_requests\_total=100*

Metrics are usually stored in time series data stores like Prometheus. Metrics are optimized for storage, compression, and processing. It’s easier to retrieve and query them than it is to retrieve and query logs. This makes metrics a perfect fit for both alerting and building dynamic dashboards which show the system’s real-time or historical data.

The example below uses Prometheus query language to read the total number of HTTP requests for service *hello* running in *dev* environment:

*http\_requests\_total{service=”hello”, environment=”dev”}*

Kubernetes generates events out of the box when, for example, a PersistentVolume can’t be created or there aren’t enough resources for your application. Any other application can use the Kubernetes API to generate custom events that indicate changes in the application’s state.

### **Traces**

Modern systems are implemented as interconnected complex and distributed microservices. Understanding request flows and system communications is a huge technical challenge, mostly because all systems in a chain need to be modified to propagate tracing information. Distributed tracing is a new approach to increasing observability and understanding performance bottlenecks. A trace is a representation of consecutive events which reflect an end-to-end request path in a distributed system.

When you have a good understanding of an entire request flow, it’s possible to troubleshoot most performance bottlenecks or analyze dependencies between different services.

Unfortunately, Kubernetes does not provide native tracing capability. However, there are multiple technologies out there that perform this function. See the CNCF’s [Cloud-Native Tracing Landscape](https://landscape.cncf.io/card-mode?category=tracing&grouping=category" \t "_blank) trail map for more details.

## **5 Apps for Implementing Kubernetes Observability**

Observability in Kubernetes can be implemented by using the five different open-source technologies described below.

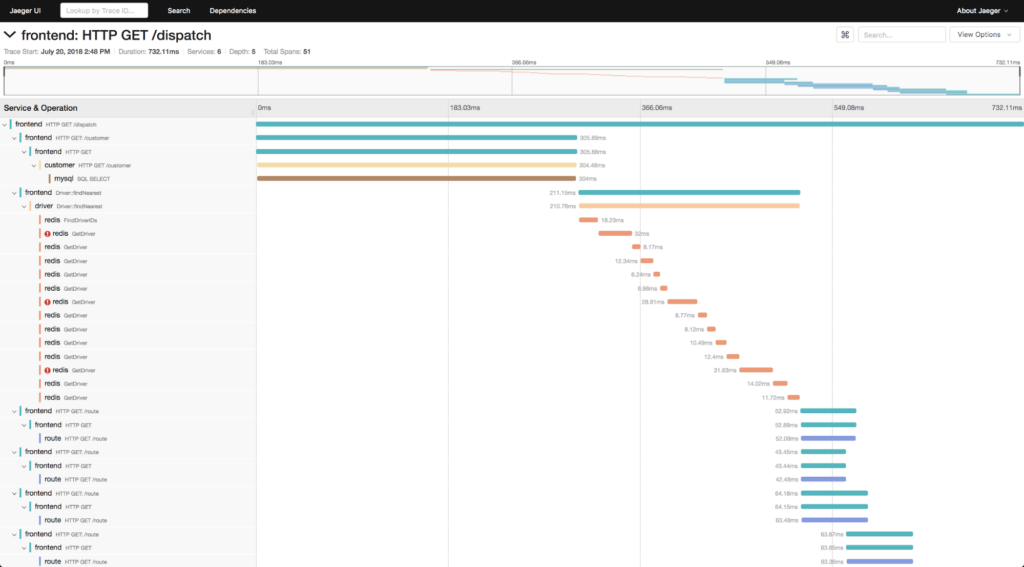
### **1. Jaeger—Distributed Systems Tracing with Kubernetes**

[Jaeger](https://www.jaegertracing.io/" \t "_blank) is an open-source distributed tracing system developed by Uber. According to [Jaeger’s documentation](https://www.jaegertracing.io/docs/1.16/" \t "_blank), the tool is designed to monitor and troubleshoot distributed microservices, mostly focusing on:

* Distributed context propagation
* Distributed transaction monitoring
* Root cause analysis
* Service dependency analysis
* Performance/latency optimization

Your application needs to be instrumented before it can send tracing information. Jaeger provides [client libraries](https://www.jaegertracing.io/docs/1.16/client-libraries" \t "_blank) for most programming languages, including Java, Go, Node.js, and Python.

The screenshot below shows a detailed request timeline view:

Figure 1: Jaeger request timeline view (Source: Jaeger)

Jaeger can be deployed in Kubernetes using [Jaeger Operator](https://www.jaegertracing.io/docs/1.16/operator/" \t "_blank), which helps to deploy and manage Jaeger instances. If you are not familiar with operators, please take a look at the [Kubernetes Operator documentation](https://coreos.com/operators/" \t "_blank).

### **2. Fluentd—Collecting Kubernetes Logs**

Fluentd is an open-source data collector for unified logging layers. It works nicely with Kubernetes running as [DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/" \t "_blank). This combination ensures that all nodes run one copy of a pod.

Let’s take a look at an example of logging to ElasticSearch:

*apiVersion: extensions/v1beta1*

*kind: DaemonSet*

*metadata:*

*name: fluentd*

*namespace: kube-system*

*…*

*spec:*

*…*

*spec:*

*containers:*

*– name: fluentd*

*image: quay.io/fluent/fluentd-kubernetes-daemonset*

*env:*

*– name:  FLUENT\_ELASTICSEARCH\_HOST*

*value: “elasticsearch-logging”*

*– name:  FLUENT\_ELASTICSEARCH\_PORT*

*value: “9200”*

*– name:  FLUENT\_ELASTICSEARCH\_SSL\_VERIFY*

*value: “true”*

*– name:  FLUENT\_ELASTICSEARCH\_SSL\_VERSION*

*value: “TLSv1\_2”*

*…*

The example above shows a DaemonSet manifest which deploys a Fluentd agent using the container image *fluentd-kubernetes-daemonset* on every Kubernetes node with a configured export to an Elasticsearch instance via TLS 1.2.

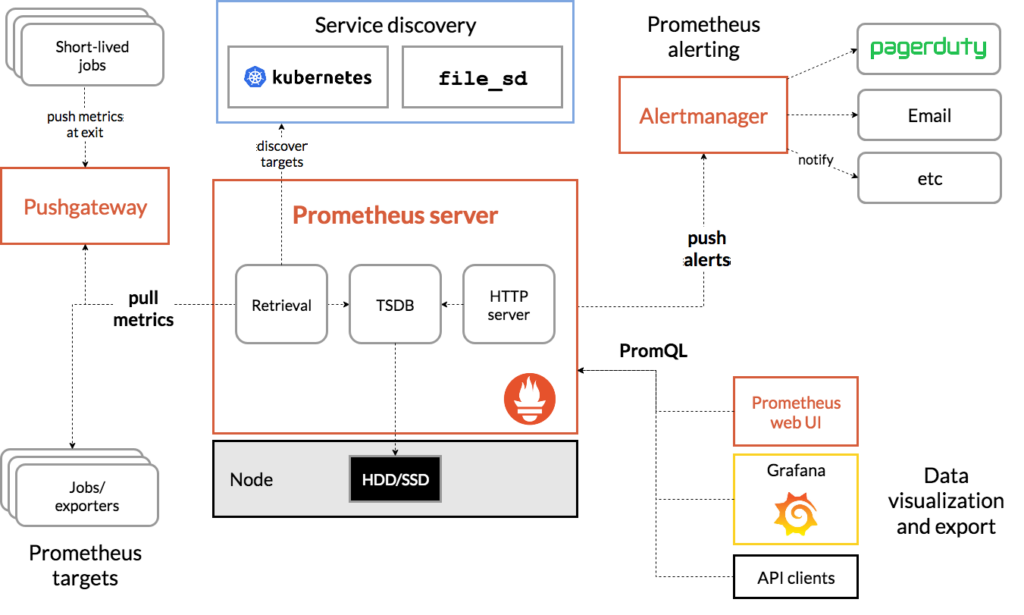
Fluentd supports multiple [data output plugins](https://www.fluentd.org/dataoutputs" \t "_blank) for exporting logs to third party applications like ElasticSearch or Stackdriver. If your application running in Kubernetes streams logs to standard output (stdout), the Fluentd agent will make sure everything is properly distributed to a centralized logging solution. Later in this article, we’ll look at analyzing this data with ELK.

If you are interested in a hands-on example of Kubernetes logging with Fluentd, take a look at [this article](https://logz.io/blog/kubernetes-logging-fluentd/" \t "_blank).

### **3. Prometheus—Automating Collection and Storage of Observability Data**

[Prometheus](https://prometheus.io/" \t "_blank) is a cloud native time series data store with built-in rich query language for metrics. It can be run in Kubernetes using a [Kubernetes Operator](https://github.com/coreos/prometheus-operator" \t "_blank) or in a stand-alone mode.

The picture below shows the high-level architecture of Prometheus and its external components:

Figure 2: Prometheus architecture (Source: Prometheus)

Prometheus uses [exporters](https://prometheus.io/docs/instrumenting/exporters/" \t "_blank) to allow third party data to be brought into its data store. There are a number of ready-to-use exporters maintained as part of the official [Prometheus GitHub organization](https://github.com/prometheus" \t "_blank), including the [Elasticsearch stats exporter](https://github.com/justwatchcom/elasticsearch_exporter" \t "_blank) for Prometheus, the [Exporter for MySQL](https://github.com/prometheus/mysqld_exporter" \t "_blank) server metrics, and the [Kafka exporter](https://github.com/danielqsj/kafka_exporter" \t "_blank) for Prometheus.

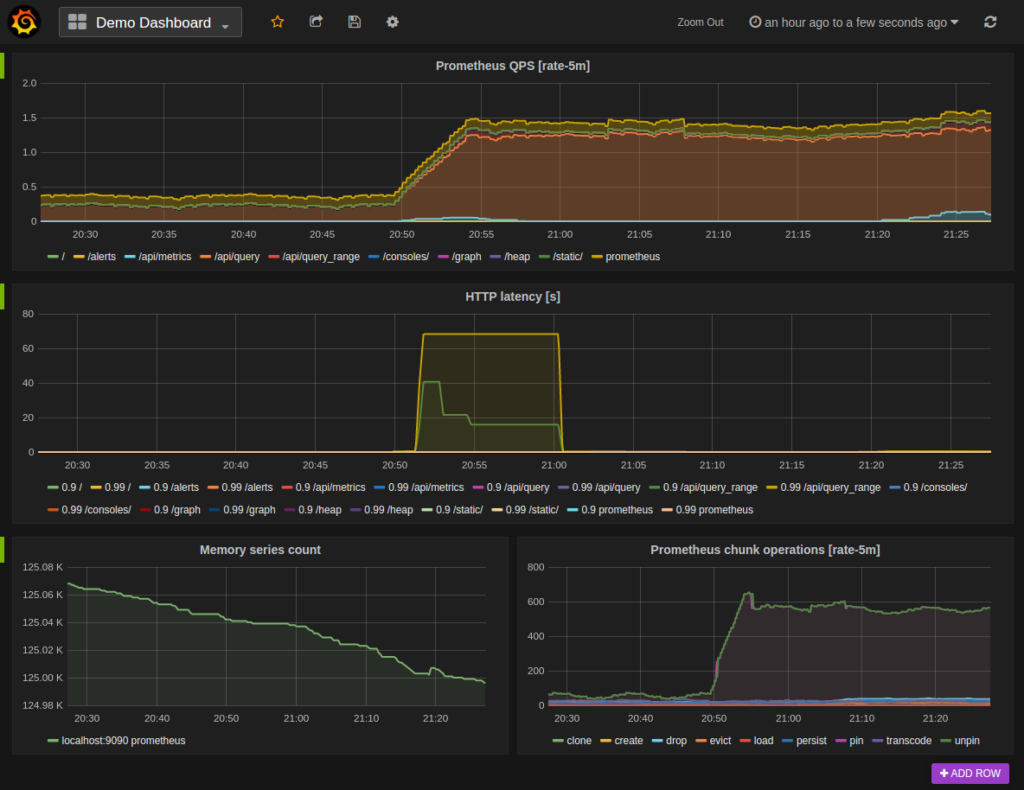
Usually, Prometheus scrapes specific metrics from an HTTP endpoint in order to read text-based metrics from it. However, in some cases, you will need to monitor components which cannot be scraped. [Pushgateway](https://prometheus.io/docs/instrumenting/pushing/" \t "_blank) allows you to push time series directly into a Prometheus data store.

Collecting data with Prometheus opens up many possibilities for increasing the observability of your infrastructure and of the containers running in Kubernetes. For instance, you can get a picture of your current capacity by measuring the number of requests handled by your service compared to the available connection pool in your web server.

### **4. Grafana—Visualizing Observability Data**

Raw metrics are not the best for the visualization of observability data, since they often consist of time series, text-based data with thousands of events. This information is not readable by human beings.

[Grafana](https://grafana.com/" \t "_blank), an observability platform which can be easily deployed in Kubernetes, helps you to overcome this problem by processing raw metrics. According to the [Grafana documentation](https://grafana.com/grafana/" \t "_blank), this tool *“*allows you to query, visualize, alert on, and understand your metrics no matter where they are stored.” It does this through [official and community-built dashboards](https://grafana.com/grafana/dashboards" \t "_blank) which can be downloaded and imported into your Grafana instance. Grafana supports the querying of Prometheus metrics. The example below shows queries for Prometheus data itself:

Figure 3: Queries for Prometheus data (Source: Prometheus)

#### **Which Kubernetes Metrics Should I Monitor?**

Even though there are a number of [ready-to-use dashboards](https://grafana.com/grafana/dashboards?search=kubernetes" \t "_blank) for Kubernetes-specific metrics, it’s not obvious which of these metrics you should actually pay attention to. In a production Kubernetes environment, the following metrics should be visible in your dashboard:

* Resource utilization saturation—the containers’ resource consumption and allocation.
* The number of failing pods and errors within a specific namespace.
* Kubernetes resource capacity—the total number of nodes, CPU cores, and memory available.

Most of the metrics above can be exported using [node\_exporter](https://github.com/prometheus/node_exporter" \t "_blank) and [cAdvisor](https://github.com/google/cadvisor" \t "_blank).

In most cases, the Cluster Monitoring for Kubernetes dashboard provides enough insights to monitor nodes and workloads.

Figure 4: Cluster Monitoring for Kubernetes dashboard (Source: Grafana)

### **5. The ELK Stack—Analyzing Kubernetes Logs**

ELK is the acronym used for the combination of three open-source projects: Elasticsearch, Logstash, and Kibana. Elasticsearch is a search and analytics engine. Logstash is an ingest pipeline, and Kibana is a flexible visualization tool. A fourth component, Beats, is a lightweight shipper for log data.

ELK works nicely in Kubernetes, but is even more streamlined by Logz.io. The guide entitled [Deploying the ELK Stack on Kubernetes with Helm](https://logz.io/blog/deploying-the-elk-stack-on-kubernetes-with-helm/" \t "_blank) provides more details about this pairing.

Compared to other tools, the Logz.io version of the ELK Stack provides more analytics capabilities that can in turn be categorized into these four areas:

* [Alerting](https://www.elastic.co/what-is/elasticsearch-alerting" \t "_blank)—Identifying changes in your data that are interesting to you.
* [Monitoring](https://www.elastic.co/what-is/elasticsearch-monitoring" \t "_blank)—Keeping a pulse on the performance of Elasticsearch, Kibana and Logstash.
* [Security](https://www.elastic.co/what-is/elastic-stack-security" \t "_blank)—Integrating with a number of industry standard identity management systems.
* [Reporting](https://www.elastic.co/what-is/kibana-reporting" \t "_blank)—Generating, scheduling and emailing reports.

**Admission Controllers and API deprecations:**

Admission controllers are a set of extensions that help define and govern operations for Kubernetes clusters. They act as gatekeepers and process Kubernetes API server requests before the object data is executed or persisted into etcd, the distributed key-value store.

Admission controllers can completely deny/accept the requests or change the request object altogether. Depending on the Kubernetes distribution, admission controllers may come compiled and shipped into the kube-apiserver binary and can be easily enabled by the cluster administrators, to run secure and reliable services.

Admission controllers have different functions but implement a typical two-phase admission control process.

The first phase is called the *change of request object* or *mutating phase*. In this phase, changes are made to the incoming request.

This is followed by the allow/disallow of requests—the second or *validating phase*—which acts on changes made in the mutating phase. If any request is disallowed in the validating phase, the entire request gets rejected, and error messages are returned as responses.

**Why do I need them?**

Many advanced features in Kubernetes require an admission controller to be enabled in order to properly support the feature. As a result, a Kubernetes API server that is not properly configured with the right set of admission controllers is an incomplete server and will not support all the features you expect.

**How do I turn on an admission controller?**

The Kubernetes API server flag enable-admission-plugins takes a comma-delimited list of admission control plugins to invoke prior to modifying objects in the cluster. For example, the following command line enables the NamespaceLifecycle and the LimitRanger admission control plugins:

**Admission Controller Types**

Admission controllers are divided into two basic classes for ease of use.

1. **Validating admission controllers:** These types of admission controllers validate API object contents and make sure they are valid. The responses are returned as binary results in the form of yes, if the object is granted permission, or no when it is not.
2. **Mutating admission controllers:** These admission controllers assess the API object and may add or modify the API object contents.

It’s also worth noting that admission controllers can be both validating and mutating controllers. A great example of that is the [LimitRanger](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/" \l "limitranger) admission controller, which can handle functioning for both of these controller methods. As a validating admission controller, it can allow or deny the deployment objects from violating the limit range constraints. Meanwhile, as a mutating admission controller, it can change or assign resource limits for pods.

There is a set of static Kubernetes admission controllers that you can find in any vanilla Kubernetes deployment, which we’ll discuss in the next section. To check the complete list of available admission controllers, visit the [Kubernetes documentation here](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/).

**[#](https://loft.sh/blog/kubernetes-admission-controllers-what-they-are-and-why-they-matter/" \l "admission-controller-benefits)Admission Controller Benefits**

We’ve covered how admission controllers enforce control on the Kubernetes cluster, but you need to see some concrete scenarios.

There are actually many ways admission controllers can benefit your business workflows and provide security compliance and validation.

**[#](https://loft.sh/blog/kubernetes-admission-controllers-what-they-are-and-why-they-matter/" \l "improved-security)Improved Security**

As mentioned earlier, admission controllers intercept API server requests before they’re executed or persisted in [etcd](https://etcd.io/). This makes them a perfect contender to apply security measures to help organizations meet policy requirements. Admission controllers can improve the security of Kubernetes workloads through a built-in PodSecurityPolicy (PSP), which mounts the root file system as read-only and prevents the containers from running as the root user.

Also, denying the pulling of images from registries that are unknown to the enterprise can mitigate many security risks. You can implement a catch-based logic, where an admission controller handles every image differently after its content is scanned during the build process.

**[#](https://loft.sh/blog/kubernetes-admission-controllers-what-they-are-and-why-they-matter/" \l "seamless-configuration-and-deployment)Seamless Configuration and Deployment**

Configuration management is an essential practice required in Kubernetes to automatically deploy components based on different parameters. An admission controller in this context can validate the configuration of objects on a cluster and manage deployment misconfiguration requesting too many resources, missing appropriate pod labels, or image tags.

Also, using an image signature ensures that a verified image is used for deployment to enforce security in a pipeline.

**[#](https://loft.sh/blog/kubernetes-admission-controllers-what-they-are-and-why-they-matter/" \l "easy-kubernetes-governance)Easy Kubernetes Governance**

To be Kubernetes production-ready, governance is critical. You have to ensure that teams deploying clusters are adhering to specific rules.

Kubernetes admission controllers allow enforcement of these rules through labels, annotations, and automated policy management. Label validation is enforced across departments, groups, or the entire organization, so an appropriate label is assigned for every project. Deployment policies are also created based on the particular namespace so that only certain users or service accounts have access to deploy containers.

**[#](https://loft.sh/blog/kubernetes-admission-controllers-what-they-are-and-why-they-matter/" \l "built-in-static-kubernetes-admission-controllers)Built-in Static Kubernetes Admission Controllers**

Kubernetes recommends enabling several built-in admission controllers by default to secure the running containers:

* **PodSecurityPolicy:** This controller implements pod admission (i.e., create and modify pods) and determines whether they should be admitted based on the security policies. Note that PSPs [will be deprecated](https://kubernetes.io/blog/2021/04/06/podsecuritypolicy-deprecation-past-present-and-future/) in a future Kubernetes release.
* **DenyEscalatingExec:** This controller blocks exec and attach commands from privileged containers so hackers cannot have access to the host even if they have access to a shell in privileged containers.
* **AlwaysPullImages:** This controller ensures that the latest builds for the container images are always downloaded. While there is a performance deficit for pulling and using new images every time for a node, it’s important to run up-to-date container images.
* **LimitRange and ResourceQuota:** This controller observes incoming requests and checks them for limits violations to prevent DoS attacks and unauthorized processes from running.
* **DefaultStorageClass:** This controller handles PersistentVolumeClaim (PVC) objects creation. It also automatically adds a default storage class to them if they are not mapped to any specific ones.
* **NamespaceLifecycle:** This controller ensures that namespaces under the process of termination are not allowed to create new objects. All the requests made to nonexisting namespaces are rejected, while modification of system namespaces is disabled.
* **Service Account:** In Kubernetes, service accounts are processes that run in pods. This controller implements the automation of service accounts.
* **DefaultTolerationSeconds:** This controller sets toleration duration for pods to tolerate the taints, which allows a particular node to repel a set of pods.
* **MutatingAdmissionWebhook and ValidatingAdmissionWebhook:** These are dynamic admission controllers that can be extended with custom logic through webhooks.

**[#](https://loft.sh/blog/kubernetes-admission-controllers-what-they-are-and-why-they-matter/" \l "dynamic-admission-controllers)Dynamic Admission Controllers**

In Kubernetes, dynamic admission controllers are user-configured controllers that can modify or reject API requests based on custom logic. As previously discussed, Kubernetes already provides a large set of admission controllers to change basic behavior. However, those are not enough for organizations that have their own specific set of policies to implement.

That’s where the *dynamic* part comes in. These controllers can extend the Kubernetes API functionality and provide customized configuration through webhooks.

A webhook is a standard interface that listens to API server incoming requests and responds with the results. Unlike other interfaces, webhooks are flexible and easy to code in any language or framework, which allows seamless integration of Kubernetes admission controllers with any third-party code.

There are various scenarios where cluster admins can use webhooks to create admission logic for Kubernetes resources. Some are:

* Limiting customization each time a resource gets created or modified in the cluster.
* Making the resources immutable through RBAC policies so unauthorized teams cannot add or delete resources.
* Injecting sidecar containers for implementing additional traffic controllers, auditing, and logging tools.

**Dynamic Admission Controller Types**

There are three specific types of dynamic admission controllers that can extend API functionality through webhooks:

* ImagePolicyWebhook: Controls admission for a particular container image.
* MutatingAdmissionWebhook: Modifies the objects received in the admission requests. It runs before the validating webhooks.
* ValidatingAdmissionWebhook: Accepts or rejects admission requests.

To create and configure a webhook in the Kubernetes cluster, MutatingAdmissionWebhook and ValidatingAdmissionWebhook must be enabled. A MutatingAdmissionWebhook controller can trigger the registered webhooks to enforce custom options, while the ValidatingAdmissionWebhook controller can trigger registered webhooks to enforce custom policies.

kube-apiserver --enable-admission-plugins=NamespaceLifecycle,LimitRanger ...

How do I turn off an admission controller?

The Kubernetes API server flag disable-admission-plugins takes a comma-delimited list of admission control plugins to be disabled, even if they are in the list of plugins enabled by default.

kube-apiserver --disable-admission-plugins=PodNodeSelector,AlwaysDeny ...

Which plugins are enabled by default?

To see which admission plugins are enabled:

kube-apiserver -h | grep enable-admission-plugins

Kubernetes is always under massive and development which means you should expect more frequent updates to your manifests and definition files that utilizes various APIs that it releases more APIs that evolves over period it either upgrades the version of supported API or remove the support for the old versions. Old APIs are getting deprecated and eventually removed from the repository. As a Kubernetes application developer, you should know each APIs supported by

Kubernetes Admission Controller is an advanced plugin for gating and governing the configuration changes and workload deployment in a cluster.

Diagram

Description automatically generated

Admission Controller enables DevOps and Security personnel to enforce deployment requirements and restrictions in the cluster upon every workload start and any configuration c hange. Think of an Admission Controller as an Advanced Resource manager with a shield. It provides a very efficient way to govern and control what can be deployed in order to limit risk and stop many security problems from occurring in the future.

An example of admission control can be a policy that prevents privileged pods from execution in a cluster. Running a pod in privileged mode means that the pod has access to the host’s resources and kernel capabilities. A hacker will be extra motivated to exploit this pod and access the host’s resources, which causes a significant amount of damage to the cluster.

## Benefits of Kubernetes Admission Controller

### Extended Security Controls

Kubernetes Admission Controllers offer an additional line of defense on top of other mechanisms. [RBAC](https://www.armosec.io/blog/a-guide-for-using-kubernetes-rbac/" \t "_blank) offers security, however, it is not sufficiently granular and does not have control over, for example, image vulnerability status, and cannot enforce network policy definition, privilege level, and other important characteristics.

### Company-Wide Guardrails

Admission controllers may enforce company-wide rules on multiple clusters without changing existing RBAC and workload deployment configuration.

### Preventive Capabilities

As opposed to audit log analysis tools and configuration scanning, admission controllers not only detect problems but also prevent them from entering into a cluster.

### Highly customizable

Unlike [built-in Kubernetes security mechanisms](https://www.armosec.io/blog/kubernetes-security-best-practices/" \t "_blank), dynamic admission controllers (described below) are adjustable to many different user-specific scenarios and environments. There are many open-source and commercial admission controller implementations that customers can choose from and enforce their special constraints.

## Static Admission Controllers

Static admission controllers are admission controllers that are built-in and provided by [Kubernetes itself](https://www.armosec.io/glossary/kubernetes/" \t "_blank). Not all of them are enabled by default. Some of them are also blocked or taken by the cloud vendors for their own use. If you own your [Kubernetes deployment](https://www.armosec.io/blog/kubernetes-deployment-and-service/" \t "_blank), you can enable and use them.

Let’s look at a few examples of static controllers and see what they do:

* **LimitRanger**: ensures that incoming requests don’t violate any of the constraints enumerated in the LimitRange object in a namespace. If you are using LimitRange objects in your Kubernetes deployment, you MUST use this admission controller to enforce those constraints. LimitRanger can also be used to apply default resource requests to pods that don’t specify any.
* **AlwaysPullImages**: changes the image pull policy for every new Pod. This is useful, for example, in multitenant clusters to ensure that only those with the credentials to fetch private images can access them. Without this admission controller, after an image has been pulled to a node, any pod from any user can use it just by knowing the image’s name without any authorization checks. This feature must be enabled in the cluster.
* **PodSecurityPolicy**: sometimes referred to as PSP, this is an essential controller that is responsible for deciding whether or not pods should be admitted to the cluster depending on the security policies, pod configuration and container posture. However, this feature was deprecated in Kubernetes v1.21 and [will be removed in v1.25](https://www.armosec.io/blog/secure-kubernetes-pods-post-psps-deprecation/" \t "_blank).

## Dynamic Admission Controllers

There will be scenarios when the native built-in options and functionality just don’t suffice. The requirements may demand the development of additional functionalities that will help us meet our business objectives. Dynamic admission control is a mechanism in which we inject our custom-built business logic into the admission control pipeline.

The purpose of the admission controller is to approve, reject or even modify the API requests before executing them or persisting them in the etcd. There are slight differences between self-managed and cloud-hosted Kubernetes.

In cloud-hosted Kubernetes, the control plane and components such as etcd and Kubernetes API server are completely handled by the provider, such as AWS. While the complete management infrastructure operates in the background across several availability zones, AWS will automatically cure any unhealthy nodes and keep the service available.

When using self-hosted Kubernetes, however, users will have full control over the management components of the cluster. This means that you are fully responsible to keep Kubernetes management nodes and services up-to-date and properly configured.

Dynamic admission controllers can be of two types – validating and mutating. They can be used separately and together.

To implement dynamic admission controllers, you must configure the API server first with a change to the –enable-admission-plugins.

--enable-admission-plugins=...,MutatingAdmissionWebhook,\

ValidatingAdmissionWebhook

Diagram

Description automatically generated

### Validating Admission Controller

Validating admission controllers are responsible for gating the API requests and enforcing custom policies prior to request execution or persistence of the new object in the etcd.

Based upon predefined policies expressing customers’ business logic, the validating admission controller will accept or reject users’ requests. In the event of a rejection, an appropriate HTTP status code is returned to the API server and stored in the Kubernetes events so that users can read the detailed reason for the failure.

The API server uses an HTTP REST interface to communicate with dynamic admission controllers. It sends a JSON-encoded object that is about to be persisted in the etcd which allows the admission controller to validate the minimal requirements compliance and policy violations and prevent this object from entering the cluster in case of a failure. For example, block all pods that request to run as root. Obviously, such a policy requires exceptions. One object can be validated against multiple admission policy controls.

More sophisticated admission controllers can assess a risk score based on many different criteria rather than a single validation failure. For example, block privileged pods from starting if they point to a [container image](https://www.armosec.io/glossary/container-image/" \t "_blank) in the repository that is not explicitly permitted.

Admission controllers usually offer a basic set of policies that users may activate and allow users to define their own custom policies.

One of the very important qualities of admission controllers is their diagnostics capabilities. Admission controller preventing pods from starting may be considered as frustrating, even if it is done for a greater good. Therefore diagnostics with a precise description of the denial reasoning and maybe even suggested remediation is the most important thing to look for in the admission controllers.

### Mutating Admission Controller

Mutating admission control is an additional step offered by Kubernetes API servers to help automate policy compliance processes. It is performed by a mutating webhook interface which is logically similar to the validating admission that we have described above, but it also allows changing the object that is being admitted instead of rejecting it.

The mutating step is performed prior to the validating call. Admission controllers that support mutation functionality may perform the mutation and validation in one step or still separate them and act as two independent capabilities. It is important to note that Kubernetes cluster supports multiple mutating and validating webhooks. Therefore, it is more reliable to separate the mutation from the validation because the validation could cover mutations that are performed by other admission controllers.

While mutating capability is very powerful and has a great potential for auto-remediation, it also has a downside. The actual objects that are going to be persisted in the cluster are not the same that the user has designed. It may cause complexities in the debugging and troubleshooting and may even break the desired application logic as the automation is not always right. Therefore, the actual usage of the mutating admission is pretty limited today.

In the contemporary DevOps environments where deployments are constructed from HELM charts and parameters and the pods are created from deployments, mutating admission may cause a conflict with HELM if it will modify the deployment object. Therefore, the most common approach is to modify the pods, but this only further increases the distance between what the user has designed and what is deployed.

Having said that, I believe that mutating admission will grow and help DevOps and security teams to improve their cluster posture.

## Conclusion

Admission controllers are a very important mechanism for maintaining an organization’s policies and compliance with security standards and regulations. Moreover, webhooks enhance the functionality of the Kubernetes API server, making it simpler to monitor, not just enforce the desired rules.

Working together with other security and deployment best practices, admission controllers can be applied not only in the production environments but also in the development stage to provide early indications of potential violations. In turn, this can help fix issues earlier in the pipeline, speeding up the deployment process and helping developers understand the rules and avoid repetitive mistakes in the future.

Kubernetes in the version you are running in your environments. For example, following API has been removed from the Kubernetes v1.27 and it will stop serving withing the object creation API: “CSIStorageCapacity”

The **storage.k8s.io/v1beta1** API version of CSIStorageCapacity will no longer be served in v1.27.

* You must migrate all manifests and API clients to use the **storage.k8s.io/v1** API version.
* All objected created previously created within the cluster will still be accessible via the API.
* Notable API changes won’t happen within minor release but rather it happens between two major releases.

If for example the definition of Deployment object changes, then it’ll be done first under a beta API Version, like batch/v1beta1. The old version and new version will coexist for some time until it becomes stable release and after public review and feedback, the changes will be released under first stable version, for example Deployment/v1.

For more details about all deprecated APIs by version please follow the URL here - <https://tinyurl.com/3pnapmf7>

As an CKAD candidate and application developer you want to keep your definition files up to date and match with the supported version of APIs with the version of Kubernetes. IT is important to understand that the version used in current environment should use the resources for your applications with correct version of API in order to successfully deployment the applications within the cluster.

Chapter summary:

· While the main purpose of the Liveness probe is to detect failures in the container application that may be resolved by terminating the Pod (i.e. restarting the container app), Readiness Probe detects failure conditions where the application may be interrupted and temporarily unavailable.

· Liveness probe succeeds when the container itself is healthy, but on top of that the readiness probe verifies that everything that the application needs in terms of servicing traffic is in place and the required backend services are available. This helps you avoid directing traffic to Pods that can only respond with error messages.

· By combining readiness and liveness probes, you can instruct Kubernetes to automatically restart pods or remove them from the pool. If your application is heavily dependent on backend services, then implementing both readiness and liveness probes is important and advisable.

· By default, the probe checking will stop if the application is not ready after three (failureThreshold=3) attempts. In case if liveness probe fails, it will restart the container. In this case, if the readiness probe fails, it will mark pods as unhealthy.

· By default, initialDelaySeconds is set to 0 meaning kubelet will not wait for the container to get ready and start sending traffic. For the best case scenario the initialDelaySeconds parameter could be set to 5 seconds to that kubelet should wait before checking for the first probe.

·  To perform the initial probe, the kubelet executes the command called “cat /tmp/healthy” in the container. It returns output “0” if the command successfully completes and marks the container healthy to take traffic.

· In general, the probing algorithm works like this:

Kubelet wait for the number is seconds set in initialDelaySeconds then start performing a readiness check and waits for a number of seconds set in the timeoutSeconds parameter. If the number of success counts is equal to one or greater than the parameter successThreshold (default=1) parameter, it will mark Pod success but for whatever reason if it encounters the parameter failureThreshold count (default=3) it will mark Pod as a failure and start new “Readiness” check after a number of seconds defined in a parameter called periodSeconds.

**Admission Controller tasks:**

**Task 1:**

Write all Admission Controller Plugins, which are enabled in the kube-apiserver manifest, into /tmp/admission-plugins

We filter for the argument:

cat /etc/kubernetes/manifests/kube-apiserver.yaml | grep admission-plugins

And then create a new file in the requested location /root/admission-plugins with content:

NodeRestriction

LimitRanger

Priority

**Task 2:**

Enable the Admission Controller Plugin MutatingAdmissionWebhook .

It can take a few minutes for the apiserver container to restart after changing the manifest. You can watch using watch crictl ps .

We need to edit the apiserver manifest:

# ALWAYS make a backup

cp /etc/kubernetes/manifests/kube-apiserver.yaml ~/kube-apiserver.yaml

vim /etc/kubernetes/manifests/kube-apiserver.yaml

And add the new plugin to the list:

--enable-admission-plugins=NodeRestriction,LimitRanger,Priority,MutatingAdmissionWebhook

Now we wait till the apiserver restarted using watch crictl ps .

Tip: It can take a few minutes for the apiserver container to restart after changing the manifest. You can watch using watch crictl ps

For quick update, we could also move the kube-apiserver.yaml manifest out of the main directory to temp folders like /tmp or /var/tmp, wait till the process ended and then move it back it, for example:

mv /etc/kubernetes/manifests/kube-apiserver.yaml /tmp

once the process dies we can move the file back to the original folder:

mv /tmp/kube-apiserver.yaml /etc/kubernetes/manifests/kube-apiserver.yaml

**Task 3:**

Delete Namespace space1 .

Delete Namespace default (throws error)

Disable the Admission Controller Plugin NamespaceLifecycle . It's **not recommended** to do this at all, we just do this for showing the effect.

It can take a few minutes for the apiserver container to restart after changing the manifest. You can watch using watch crictl ps .

Now delete Namespace default

First we delete Namespace space1 :

k delete ns space1

And same for default . But it won't work because an admission plugin prevents the deletion!

k delete ns default # error

We need to edit the apiserver manifest:

# ALWAYS make a backup

cp /etc/kubernetes/manifests/kube-apiserver.yaml ~/kube-apiserver.yaml

vim /etc/kubernetes/manifests/kube-apiserver.yaml

And add a new argument:

--disable-admission-plugins=NamespaceLifecycle

Now we wait till the apiserver restarted using watch crictl ps . Then we can delete the default one:

k delete ns default

If we wait a bit we should see that it gets automatically created again:

k get ns

**API deprecation tasks:**

**Task 1:**

Create file /tmp/k8s\_versions with three lines. Each line containing only one number from the installed K8s server version:

1. Major
2. Minor
3. Patch

We can use kubectl to show the installed K8s version:

controlplane $ kubectl version

WARNING: This version information is deprecated and will be replaced with the output from kubectl version --short. Use --output=yaml|json to get the full version.

Client Version: version.Info{Major:"1", Minor:"25", GitVersion:"v1.25.3", GitCommit:"434bfd82814af038ad94d62ebe59b133fcb50506", GitTreeState:"clean", BuildDate:"2022-10-12T10:57:26Z", GoVersion:"go1.19.2", Compiler:"gc", Platform:"linux/amd64"}

Kustomize Version: v4.5.7

Server Version: version.Info{Major:"1", Minor:"25", GitVersion:"v1.25.3", GitCommit:"434bfd82814af038ad94d62ebe59b133fcb50506", GitTreeState:"clean", BuildDate:"2022-10-12T10:49:09Z", GoVersion:"go1.19.2", Compiler:"gc", Platform:"linux/amd64"}

controlplane $ k version --short

Flag --short has been deprecated, and will be removed in the future. The --short output will become the default.

Client Version: v1.25.3

Kustomize Version: v4.5.7

Server Version: v1.25.3

Kubernetes version is written as Major.Minor.Patch .

Write each number in one line into /tmp/k8s\_versions .

Open any text editor and create a file called “/tmp/k8s\_versions” then write following version items in each line and then save the file the content shown below.

1

25

3

**Task 2:**

Write the Api Group of Deployments into /tmp/deploy\_group

We can use kubectl to list information about a resource type:

controlplane $ kubectl explain deploy

KIND: Deployment

VERSION: apps/v1

This will show VERSION: apps/v1

The version is displayed as VERSION: {group}/{version}

echo apps > /tmp/deploy\_group

**Task 3:**

There is a CronJob file at /apps/cronjob.yaml which uses a deprecated Api version.

Update the file to use the non deprecated one.

When creating a resource which uses a deprecated version, kubectl will show a warning. Also please check the [K8s Docs](https://kubernetes.io/docs/reference/using-api/deprecation-guide" \t "_blank).

Creating the resource will show a warning and a note which version to use instead:

kubeclt -f /apps/cronjob.yaml create

controlplane $ kubectl -f /apps/cronjob.yaml create

error: resource mapping not found for name: "backup" namespace: "" from "/apps/cronjob.yaml": no matches for kind "CronJob" in version "batch/v1beta1"

ensure CRDs are installed first

controlplane $ kubectl explain cronjob

KIND: CronJob

VERSION: batch/v1

The file should be changed to:

**apiVersion: batch/v1**

kind: CronJob

metadata:

name: backup

spec:

schedule: "5 5 \* \* \*"

jobTemplate:

spec:

template:

spec:

...