Resource metrics pipeline

For Kubernetes, the *Metrics API* offers a basic set of metrics to support automatic scaling and similar use cases. This API makes information available about resource usage for node and pod, including metrics for CPU and memory. If you deploy the Metrics API into your cluster, clients of the Kubernetes API can then query for this information, and you can use Kubernetes' access control mechanisms to manage permissions to do so.

The HorizontalPodAutoscaler (HPA) and VerticalPodAutoscaler (VPA) use data from the metrics API to adjust workload replicas and resources to meet customer demand.

You can also view the resource metrics using the `kubectl top` command.

Note:

The Metrics API, and the metrics pipeline that it enables, only offers the minimum CPU and memory metrics to enable automatic scaling using HPA and / or VPA. If you would like to provide a more complete set of metrics, you can complement the simpler Metrics API by deploying a second metrics pipeline that uses the *Custom Metrics API*.

Figure 1 illustrates the architecture of the resource metrics pipeline.

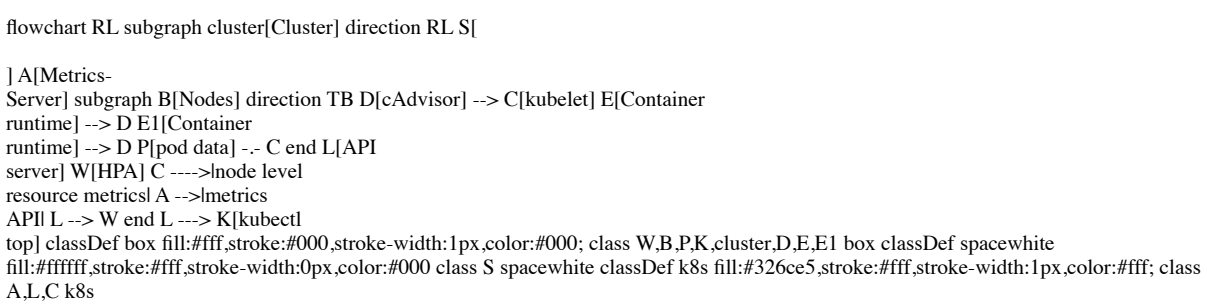

Figure 1. Resource Metrics Pipeline

The architecture components, from right to left in the figure, consist of the following:

- cAdvisor: Daemon for collecting, aggregating and exposing container metrics included in Kubelet.
- kubelet: Node agent for managing container resources. Resource metrics are accessible using the `/metrics/resource` and `/stats` kubelet API endpoints.
- node level resource metrics: API provided by the kubelet for discovering and retrieving per-node summarized stats available through the `/metrics/resource` endpoint.
- metrics-server: Cluster addon component that collects and aggregates resource metrics pulled from each kubelet. The API server serves Metrics API for use by HPA, VPA, and by the `kubectl top` command. Metrics Server is a reference implementation of the Metrics API.
- Metrics API: Kubernetes API supporting access to CPU and memory used for workload autoscaling. To make this work in your cluster, you need an API extension server that provides the Metrics API.

  Note:

  cAdvisor supports reading metrics from cgroups, which works with typical container runtimes on Linux. If you use a container runtime that uses another resource isolation mechanism, for example virtualization, then that container runtime must support CRI Container Metrics in order for metrics to be available to the kubelet.

Metrics API

FEATURE STATE: `Kubernetes 1.8 [beta]`

The metrics-server implements the Metrics API. This API allows you to access CPU and memory usage for the nodes and pods in your cluster. Its primary role is to feed resource usage metrics to K8s autoscaler components.

Here is an example of the Metrics API request for a `minikube` node piped through `jq` for easier reading:

```
kubectl get --raw "/apis/metrics.k8s.io/v1beta1/nodes/minikube" | jq '.'
```

Here is the same API call using `curl`:

```
curl http://localhost:8080/apis/metrics.k8s.io/v1beta1/nodes/minikube
```

Sample response:


```
{
  "kind": "NodeMetrics",
  "apiVersion": "metrics.k8s.io/v1beta1",
  "metadata": {
    "name": "minikube",
    "selfLink": "/apis/metrics.k8s.io/v1beta1/nodes/minikube",
    "creationTimestamp": "2022-01-27T18:48:43Z"
```

```
  },
  "timestamp": "2022-01-27T18:48:33Z",
  "window": "30s",
  "usage": {
    "cpu": "487558164n",
    "memory": "732212Ki"
  }
}
```


Here is an example of the Metrics API request for a `kube-scheduler-minikube` pod contained in the `kube-system` namespace and piped through `jq` for easier reading:

```
kubectl get --raw "/apis/metrics.k8s.io/v1beta1/namespaces/kube-system/pods/kube-scheduler-minikube" | jq '.'
```

Here is the same API call using `curl`:

```
curl http://localhost:8080/apis/metrics.k8s.io/v1beta1/namespaces/kube-system/pods/kube-scheduler-minikube
```

Sample response:


```
{
  "kind": "PodMetrics",
  "apiVersion": "metrics.k8s.io/v1beta1",
  "metadata": {
    "name": "kube-scheduler-minikube",
    "namespace": "kube-system",
    "selfLink": "/apis/metrics.k8s.io/v1beta1/namespaces/kube-system/pods/kube-scheduler-minikube",
    "creationTimestamp": "2022-01-27T19:25:00Z"
  },
  "timestamp": "2022-01-27T19:24:31Z",
  "window": "30s",
  "containers": [
    {
      "name": "kube-scheduler",
      "usage": {
        "cpu": "9559630n",
        "memory": "22244Ki"
      }
    }
  ]
}
```


The Metrics API is defined in the k8s.io/metrics repository. You must enable the API aggregation layer and register an APIService for the `metrics.k8s.io` API.

To learn more about the Metrics API, see resource metrics API design, the metrics-server repository and the resource metrics API.

Note:

You must deploy the metrics-server or alternative adapter that serves the Metrics API to be able to access it.

Measuring resource usage

CPU

CPU is reported as the average core usage measured in cpu units. One cpu, in Kubernetes, is equivalent to 1 vCPU/Core for cloud providers, and 1 hyper-thread on bare-metal Intel processors.

This value is derived by taking a rate over a cumulative CPU counter provided by the kernel (in both Linux and Windows kernels). The time window used to calculate CPU is shown under window field in Metrics API.

To learn more about how Kubernetes allocates and measures CPU resources, see meaning of CPU.

Memory

Memory is reported as the working set, measured in bytes, at the instant the metric was collected.

In an ideal world, the "working set" is the amount of memory in-use that cannot be freed under memory pressure. However, calculation of the working set varies by host OS, and generally makes heavy use of heuristics to produce an estimate.

The Kubernetes model for a container's working set expects that the container runtime counts anonymous memory associated with the container in question. The working set metric typically also includes some cached (file-backed) memory, because the host OS cannot always reclaim pages.

To learn more about how Kubernetes allocates and measures memory resources, see meaning of memory.

Metrics Server

The metrics-server fetches resource metrics from the kubelets and exposes them in the Kubernetes API server through the Metrics API for use by the HPA and VPA. You can also view these metrics using the `kubectl top` command.

The metrics-server uses the Kubernetes API to track nodes and pods in your cluster. The metrics-server queries each node over HTTP to fetch metrics. The metrics-server also builds an internal view of pod metadata, and keeps a cache of pod health. That cached pod health information is available via the extension API that the metrics-server makes available.

For example with an HPA query, the metrics-server needs to identify which pods fulfill the label selectors in the deployment.

The metrics-server calls the kubelet API to collect metrics from each node. Depending on the metrics-server version it uses:

- Metrics resource endpoint `/metrics/resource` in version v0.6.0+ or
- Summary API endpoint `/stats/summary` in older versions

What's next

To learn more about the metrics-server, see the metrics-server repository.

You can also check out the following:

- metrics-server design
- metrics-server FAQ
- metrics-server known issues
- metrics-server releases
- Horizontal Pod Autoscaling

To learn about how the kubelet serves node metrics, and how you can access those via the Kubernetes API, read Node Metrics Data.

Debug a StatefulSet

This task shows you how to debug a StatefulSet.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster.
- You should have a StatefulSet running that you want to investigate.

Debugging a StatefulSet

In order to list all the pods which belong to a StatefulSet, which have a label `app.kubernetes.io/name=MyApp` set on them, you can use the following:

```
kubectl get pods -l app.kubernetes.io/name=MyApp
```

If you find that any Pods listed are in `Unknown` or `Terminating` state for an extended period of time, refer to the Deleting StatefulSet Pods task for instructions on how to deal with them. You can debug individual Pods in a StatefulSet using the Debugging Pods guide.

What's next

Learn more about debugging an init-container.

Logging in Kubernetes

Logging architecture and system logs.

This page provides resources that describe logging in Kubernetes. You can learn how to collect, access, and analyze logs using built-in tools and popular logging stacks:

- Logging Architecture
- System Logs
- A Practical Guide to Kubernetes Logging

Get a Shell to a Running Container

This page shows how to use `kubectl exec` to get a shell to a running container.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Getting a shell to a container

In this exercise, you create a Pod that has one container. The container runs the nginx image. Here is the configuration file for the Pod:

application/shell-demo.yaml Copy application/shell-demo.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: shell-demospec:  volumes:  - name: shared-data    emptyDir: {}  containers:  - name: nginx    image: ngi
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/application/shell-demo.yaml
```

Verify that the container is running:

```
kubectl get pod shell-demo
```

Get a shell to the running container:

```
kubectl exec --stdin --tty shell-demo -- /bin/bash
```

Note:

The double dash (`--`) separates the arguments you want to pass to the command from the kubectl arguments.

In your shell, list the root directory:

```
# Run this inside the container
ls /
```

In your shell, experiment with other commands. Here are some examples:

```
# You can run these example commands inside the container
ls /
cat /proc/mounts
cat /proc/1/maps
apt-get update
apt-get install -y tcpdump
tcpdump
apt-get install -y lsof
lsof
apt-get install -y procps
ps aux
ps aux | grep nginx
```

Writing the root page for nginx

Look again at the configuration file for your Pod. The Pod has an `emptyDir` volume, and the container mounts the volume at `/usr/share/nginx/html`.

In your shell, create an `index.html` file in the `/usr/share/nginx/html` directory:

```
# Run this inside the container
echo 'Hello shell demo' > /usr/share/nginx/html/index.html
```

In your shell, send a GET request to the nginx server:

```
# Run this in the shell inside your container
apt-get update
apt-get install curl
curl http://localhost/
```

The output shows the text that you wrote to the `index.html` file:

```
Hello shell demo
```

When you are finished with your shell, enter `exit`.

```
exit # To quit the shell in the container
```

Running individual commands in a container

In an ordinary command window, not your shell, list the environment variables in the running container:

```
kubectl exec shell-demo -- env
```

Experiment with running other commands. Here are some examples:

```
kubectl exec shell-demo -- ps aux
kubectl exec shell-demo -- ls /
kubectl exec shell-demo -- cat /proc/1/mounts
```

Opening a shell when a Pod has more than one container

If a Pod has more than one container, use `--container` or `-c` to specify a container in the `kubectl exec` command. For example, suppose you have a Pod named my-pod, and the Pod has two containers named *main-app* and *helper-app*. The following command would open a shell to the *main-app* container.

```
kubectl exec -i -t my-pod --container main-app -- /bin/bash
```

Note:

The short options `-i` and `-t` are the same as the long options `--stdin` and `--tty`

What's next

- Read about kubectl exec

Debug Init Containers

This page shows how to investigate problems related to the execution of Init Containers. The example command lines below refer to the Pod as `<pod-name>` and the Init Containers as `<init-container-1>` and `<init-container-2>`.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

To check the version, enter `kubectl version`.

- You should be familiar with the basics of Init Containers.
- You should have Configured an Init Container.

Checking the status of Init Containers

Display the status of your pod:

```
kubectl get pod <pod-name>
```

For example, a status of `Init:1/2` indicates that one of two Init Containers has completed successfully:

```
NAME          READY     STATUS     RESTARTS   AGE
<pod-name>    0/1       Init:1/2   0          7s
```

See Understanding Pod status for more examples of status values and their meanings.

Getting details about Init Containers

View more detailed information about Init Container execution:

```
kubectl describe pod <pod-name>
```

For example, a Pod with two Init Containers might show the following:

```
Init Containers:
  <init-container-1>:
    Container ID:    ...
    ...
    State:           Terminated
      Reason:        Completed
      Exit Code:     0
      Started:       ...
      Finished:      ...
    Ready:           True
    Restart Count:   0
    ...
  <init-container-2>:
    Container ID:    ...
    ...
    State:           Waiting
      Reason:        CrashLoopBackOff
    Last State:      Terminated
      Reason:        Error
      Exit Code:     1
      Started:       ...
      Finished:      ...
    Ready:           False
    Restart Count:   3
    ...
```

You can also access the Init Container statuses programmatically by reading the `status.initContainerStatuses` field on the Pod Spec:

```
kubectl get pod <pod-name> --template '{{.status.initContainerStatuses}}'
```

This command will return the same information as above, formatted using a Go template.

Accessing logs from Init Containers

Pass the Init Container name along with the Pod name to access its logs.

```
kubectl logs <pod-name> -c <init-container-2>
```

Init Containers that run a shell script print commands as they're executed. For example, you can do this in Bash by running `set -x` at the beginning of the script.

Understanding Pod status

A Pod status beginning with `Init:` summarizes the status of Init Container execution. The table below describes some example status values that you might see while debugging Init Containers.

Status	Meaning
`Init:N/M`	The Pod has `M` Init Containers, and `N` have completed so far.
`Init:Error`	An Init Container has failed to execute.
`Init:CrashLoopBackOff`	An Init Container has failed repeatedly.
`Pending`	The Pod has not yet begun executing Init Containers.
`PodInitializing` or `Running`	The Pod has already finished executing Init Containers.

Troubleshooting kubectl

This documentation is about investigating and diagnosing kubectl related issues. If you encounter issues accessing `kubectl` or connecting to your cluster, this document outlines various common scenarios and potential solutions to help identify and address the likely cause.

Before you begin

- You need to have a Kubernetes cluster.
- You also need to have `kubectl` installed - see install tools

Verify kubectl setup

Make sure you have installed and configured `kubectl` correctly on your local machine. Check the `kubectl` version to ensure it is up-to-date and compatible with your cluster.

Check kubectl version:

```
kubectl version
```

You'll see a similar output:

```
Client Version: version.Info{Major:"1", Minor:"27", GitVersion:"v1.27.4",GitCommit:"fa3d7990104d7c1f16943a67f11b154b71f6a132", Git
Kustomize Version: v5.0.1
Server Version: version.Info{Major:"1", Minor:"27", GitVersion:"v1.27.3",GitCommit:"25b4e43193bcda6c7328a6d147b1fb73a33f1598", Git
```

If you see `Unable to connect to the server: dial tcp <server-ip>:8443: i/o timeout`, instead of `Server Version`, you need to troubleshoot kubectl connectivity with your cluster.

Make sure you have installed the kubectl by following the official documentation for installing kubectl, and you have properly configured the `$PATH` environment variable.

Check kubeconfig

The `kubectl` requires a `kubeconfig` file to connect to a Kubernetes cluster. The `kubeconfig` file is usually located under the `~/.kube/config` directory. Make sure that you have a valid `kubeconfig` file. If you don't have a `kubeconfig` file, you can obtain it from your Kubernetes administrator, or you can copy it from your Kubernetes control plane's `/etc/kubernetes/admin.conf` directory. If you have deployed your Kubernetes cluster on a cloud platform and lost your `kubeconfig` file, you can re-generate it using your cloud provider's tools. Refer the cloud provider's documentation for re-generating a `kubeconfig` file.

Check if the `$KUBECONFIG` environment variable is configured correctly. You can set `$KUBECONFIG`environment variable or use the `--kubeconfig` parameter with the kubectl to specify the directory of a `kubeconfig` file.

Check VPN connectivity

If you are using a Virtual Private Network (VPN) to access your Kubernetes cluster, make sure that your VPN connection is active and stable. Sometimes, VPN disconnections can lead to connection issues with the cluster. Reconnect to the VPN and try accessing the cluster again.

Authentication and authorization

If you are using the token based authentication and the kubectl is returning an error regarding the authentication token or authentication server address, validate the Kubernetes authentication token and the authentication server address are configured properly.

If kubectl is returning an error regarding the authorization, make sure that you are using the valid user credentials. And you have the permission to access the resource that you have requested.

Verify contexts

Kubernetes supports multiple clusters and contexts. Ensure that you are using the correct context to interact with your cluster.

List available contexts:

```
kubectl config get-contexts
```

Switch to the appropriate context:

```
kubectl config use-context <context-name>
```

API server and load balancer

The kube-apiserver server is the central component of a Kubernetes cluster. If the API server or the load balancer that runs in front of your API servers is not reachable or not responding, you won't be able to interact with the cluster.

Check the if the API server's host is reachable by using `ping` command. Check cluster's network connectivity and firewall. If your are using a cloud provider for deploying the cluster, check your cloud provider's health check status for the cluster's API server.

Verify the status of the load balancer (if used) to ensure it is healthy and forwarding traffic to the API server.

TLS problems

- Additional tools required - `base64` and `openssl` version 3.0 or above.

The Kubernetes API server only serves HTTPS requests by default. In that case TLS problems may occur due to various reasons, such as certificate expiry or chain of trust validity.

You can find the TLS certificate in the kubeconfig file, located in the `~/.kube/config` directory. The `certificate-authority` attribute contains the CA certificate and the `client-certificate` attribute contains the client certificate.

Verify the expiry of these certificates:

```
kubectl config view --flatten --output 'jsonpath={.clusters[0].cluster.certificate-authority-data}' | base64 -d | openssl x509 -no
```

output:

```
notBefore=Feb 13 05:57:47 2024 GMT
notAfter=Feb 10 06:02:47 2034 GMT
```

```
kubectl config view --flatten --output 'jsonpath={.users[0].user.client-certificate-data}'| base64 -d | openssl x509 -noout -dates
```

output:

```
notBefore=Feb 13 05:57:47 2024 GMT
notAfter=Feb 12 06:02:50 2025 GMT
```

Verify kubectl helpers

Some kubectl authentication helpers provide easy access to Kubernetes clusters. If you have used such helpers and are facing connectivity issues, ensure that the necessary configurations are still present.

Check kubectl configuration for authentication details:

```
kubectl config view
```

If you previously used a helper tool (for example, `kubectl-oidc_login`), ensure that it is still installed and configured correctly.

Developing and debugging services locally using telepresence

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

Kubernetes applications usually consist of multiple, separate services, each running in its own container. Developing and debugging these services on a remote Kubernetes cluster can be cumbersome, requiring you to get a shell on a running container in order to run debugging tools.

`telepresence` is a tool to ease the process of developing and debugging services locally while proxying the service to a remote Kubernetes cluster. Using `telepresence` allows you to use custom tools, such as a debugger and IDE, for a local service and provides the service full access to ConfigMap, secrets, and the services running on the remote cluster.

This document describes using `telepresence` to develop and debug services running on a remote cluster locally.

Before you begin

- Kubernetes cluster is installed
- `kubectl` is configured to communicate with the cluster
- Telepresence is installed

Connecting your local machine to a remote Kubernetes cluster

After installing `telepresence`, run `telepresence connect` to launch its Daemon and connect your local workstation to the cluster.

```
$ telepresence connect

Launching Telepresence Daemon
...
Connected to context default (https://<cluster public IP>)
```

You can curl services using the Kubernetes syntax e.g. `curl -ik https://kubernetes.default`

Developing or debugging an existing service

When developing an application on Kubernetes, you typically program or debug a single service. The service might require access to other services for testing and debugging. One option is to use the continuous deployment pipeline, but even the fastest deployment pipeline introduces a delay in the program or debug cycle.

Use the `telepresence intercept $SERVICE_NAME --port $LOCAL_PORT:$REMOTE_PORT` command to create an "intercept" for rerouting remote service traffic.

Where:

- `$SERVICE_NAME` is the name of your local service
- `$LOCAL_PORT` is the port that your service is running on your local workstation
- And `$REMOTE_PORT` is the port your service listens to in the cluster

Running this command tells Telepresence to send remote traffic to your local service instead of the service in the remote Kubernetes cluster. Make edits to your service source code locally, save, and see the corresponding changes when accessing your remote application take effect immediately. You can also run your local service using a debugger or any other local development tool.

How does Telepresence work?

Telepresence installs a traffic-agent sidecar next to your existing application's container running in the remote cluster. It then captures all traffic requests going into the Pod, and instead of forwarding this to the application in the remote cluster, it routes all traffic (when you create a global intercept or a subset of the traffic (when you create a personal intercept) to your local development environment.

What's next

If you're interested in a hands-on tutorial, check out this tutorial that walks through locally developing the Guestbook application on Google Kubernetes Engine.

For further reading, visit the Telepresence website.

Monitor Node Health

Node Problem Detector is a daemon for monitoring and reporting about a node's health. You can run Node Problem Detector as a `DaemonSet` or as a standalone daemon. Node Problem Detector collects information about node problems from various daemons and reports these conditions to the API server as Node Conditions or as Events.

To learn how to install and use Node Problem Detector, see Node Problem Detector project documentation.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Limitations

- Node Problem Detector uses the kernel log format for reporting kernel issues. To learn how to extend the kernel log format, see Add support for another log format.

Enabling Node Problem Detector

Some cloud providers enable Node Problem Detector as an Addon. You can also enable Node Problem Detector with `kubectl` or by creating an Addon DaemonSet.

Using kubectl to enable Node Problem Detector

`kubectl` provides the most flexible management of Node Problem Detector. You can overwrite the default configuration to fit it into your environment or to detect customized node problems. For example:

1. Create a Node Problem Detector configuration similar to `node-problem-detector.yaml`:

   debug/node-problem-detector.yaml  Copy debug/node-problem-detector.yaml to clipboard

   ```
   apiVersion: apps/v1
   kind: DaemonSetmetadata:  name: node-problem-detector-v0.1  namespace: kube-system  labels:    k8s-app: node-problem-detector
   ```

   Note:

   You should verify that the system log directory is right for your operating system distribution.

2. Start node problem detector with `kubectl`:

   ```
   kubectl apply -f https://k8s.io/examples/debug/node-problem-detector.yaml
   ```

Using an Addon pod to enable Node Problem Detector

If you are using a custom cluster bootstrap solution and don't need to overwrite the default configuration, you can leverage the Addon pod to further automate the deployment.

Create `node-problem-detector.yaml`, and save the configuration in the Addon pod's directory `/etc/kubernetes/addons/node-problem-detector` on a control plane node.

Overwrite the configuration

The default configuration is embedded when building the Docker image of Node Problem Detector.

However, you can use a `ConfigMap` to overwrite the configuration:

1. Change the configuration files in `config/`

2. Create the `ConfigMap` `node-problem-detector-config`:

   ```
   kubectl create configmap node-problem-detector-config --from-file=config/
   ```

3. Change the `node-problem-detector.yaml` to use the `ConfigMap`:

   debug/node-problem-detector-configmap.yaml  Copy debug/node-problem-detector-configmap.yaml to clipboard

   ```
   apiVersion: apps/v1
   kind: DaemonSetmetadata:  name: node-problem-detector-v0.1  namespace: kube-system  labels:    k8s-app: node-problem-detector
   ```

4. Recreate the Node Problem Detector with the new configuration file:

   ```
   # If you have a node-problem-detector running, delete before recreating
   kubectl delete -f https://k8s.io/examples/debug/node-problem-detector.yaml
   kubectl apply -f https://k8s.io/examples/debug/node-problem-detector-configmap.yaml
   ```

Note:

This approach only applies to a Node Problem Detector started with `kubectl`.

Overwriting a configuration is not supported if a Node Problem Detector runs as a cluster Addon. The Addon manager does not support `ConfigMap`.

Problem Daemons

A problem daemon is a sub-daemon of the Node Problem Detector. It monitors specific kinds of node problems and reports them to the Node Problem Detector. There are several types of supported problem daemons.

- A `SystemLogMonitor` type of daemon monitors the system logs and reports problems and metrics according to predefined rules. You can customize the configurations for different log sources such as filelog, kmsg, kernel, abrt, and systemd.
- A `SystemStatsMonitor` type of daemon collects various health-related system stats as metrics. You can customize its behavior by updating its configuration file.
- A `CustomPluginMonitor` type of daemon invokes and checks various node problems by running user-defined scripts. You can use different custom plugin monitors to monitor different problems and customize the daemon behavior by updating the configuration file.
- A `HealthChecker` type of daemon checks the health of the kubelet and container runtime on a node.

Adding support for other log format

The system log monitor currently supports file-based logs, journald, and kmsg. Additional sources can be added by implementing a new log watcher.

Adding custom plugin monitors

You can extend the Node Problem Detector to execute any monitor scripts written in any language by developing a custom plugin. The monitor scripts must conform to the plugin protocol in exit code and standard output. For more information, please refer to the plugin interface proposal.

Exporter

An exporter reports the node problems and/or metrics to certain backends. The following exporters are supported:

- **Kubernetes exporter**: this exporter reports node problems to the Kubernetes API server. Temporary problems are reported as Events and permanent problems are reported as Node Conditions.
- **Prometheus exporter**: this exporter reports node problems and metrics locally as Prometheus (or OpenMetrics) metrics. You can specify the IP address and port for the exporter using command line arguments.
- **Stackdriver exporter**: this exporter reports node problems and metrics to the Stackdriver Monitoring API. The exporting behavior can be customized using a configuration file.

Recommendations and restrictions

It is recommended to run the Node Problem Detector in your cluster to monitor node health. When running the Node Problem Detector, you can expect extra resource overhead on each node. Usually this is fine, because:

- The kernel log grows relatively slowly.
- A resource limit is set for the Node Problem Detector.
- Even under high load, the resource usage is acceptable. For more information, see the Node Problem Detector benchmark result.

Auditing

Kubernetes *auditing* provides a security-relevant, chronological set of records documenting the sequence of actions in a cluster. The cluster audits the activities generated by users, by applications that use the Kubernetes API, and by the control plane itself.

Auditing allows cluster administrators to answer the following questions:

- what happened?
- when did it happen?
- who initiated it?
- on what did it happen?
- where was it observed?
- from where was it initiated?
- to where was it going?

Audit records begin their lifecycle inside the kube-apiserver component. Each request on each stage of its execution generates an audit event, which is then pre-processed according to a certain policy and written to a backend. The policy determines what's recorded and the backends persist the records. The current backend implementations include logs files and webhooks.

Each request can be recorded with an associated *stage*. The defined stages are:

- `RequestReceived` - The stage for events generated as soon as the audit handler receives the request, and before it is delegated down the handler chain.
- `ResponseStarted` - Once the response headers are sent, but before the response body is sent. This stage is only generated for long-running requests (e.g. watch).
- `ResponseComplete` - The response body has been completed and no more bytes will be sent.
- `Panic` - Events generated when a panic occurred.

Note:

The configuration of an Audit Event configuration is different from the Event API object.

The audit logging feature increases the memory consumption of the API server because some context required for auditing is stored for each request. Memory consumption depends on the audit logging configuration.

Audit policy

Audit policy defines rules about what events should be recorded and what data they should include. The audit policy object structure is defined in the `audit.k8s.io` API group. When an event is processed, it's compared against the list of rules in order. The first matching rule sets the *audit level* of the event. The defined audit levels are:

- `None` - don't log events that match this rule.
- `Metadata` - log events with metadata (requesting user, timestamp, resource, verb, etc.) but not request or response body.
- `Request` - log events with request metadata and body but not response body. This does not apply for non-resource requests.
- `RequestResponse` - log events with request metadata, request body and response body. This does not apply for non-resource requests.

You can pass a file with the policy to `kube-apiserver` using the `--audit-policy-file` flag. If the flag is omitted, no events are logged. Note that the `rules` field **must** be provided in the audit policy file. A policy with no (0) rules is treated as illegal.

Below is an example audit policy file:

audit/audit-policy.yaml Copy audit/audit-policy.yaml to clipboard

```
apiVersion: audit.k8s.io/v1 # This is required.
kind: Policy# Don't generate audit events for all requests in RequestReceived stage.omitStages:  - "RequestReceived"rules:  # Log
```

You can use a minimal audit policy file to log all requests at the `Metadata` level:

```
# Log all requests at the Metadata level.
apiVersion: audit.k8s.io/v1kind: Policyrules:- level: Metadata
```

If you're crafting your own audit profile, you can use the audit profile for Google Container-Optimized OS as a starting point. You can check the configure-helper.sh script, which generates an audit policy file. You can see most of the audit policy file by looking directly at the script.

You can also refer to the `Policy` configuration reference for details about the fields defined.

Audit backends

Audit backends persist audit events to an external storage. Out of the box, the kube-apiserver provides two backends:

- Log backend, which writes events into the filesystem
- Webhook backend, which sends events to an external HTTP API

In all cases, audit events follow a structure defined by the Kubernetes API in the `audit.k8s.io` API group.

Note:

In case of patches, request body is a JSON array with patch operations, not a JSON object with an appropriate Kubernetes API object. For example, the following request body is a valid patch request to `/apis/batch/v1/namespaces/some-namespace/jobs/some-job-name`:

```
[
  {
    "op": "replace",
    "path": "/spec/parallelism",
    "value": 0
  },
  {
    "op": "remove",
    "path": "/spec/template/spec/containers/0/terminationMessagePolicy"
  }
]
```

Log backend

The log backend writes audit events to a file in JSONlines format. You can configure the log audit backend using the following `kube-apiserver` flags:

- `--audit-log-path` specifies the log file path that log backend uses to write audit events. Not specifying this flag disables log backend. `-` means standard out
- `--audit-log-maxage` defined the maximum number of days to retain old audit log files
- `--audit-log-maxbackup` defines the maximum number of audit log files to retain
- `--audit-log-maxsize` defines the maximum size in megabytes of the audit log file before it gets rotated

If your cluster's control plane runs the kube-apiserver as a Pod, remember to mount the `hostPath` to the location of the policy file and log file, so that audit records are persisted. For example:

```
  - --audit-policy-file=/etc/kubernetes/audit-policy.yaml
  - --audit-log-path=/var/log/kubernetes/audit/audit.log
```

then mount the volumes:

```
...
volumeMounts:  - mountPath: /etc/kubernetes/audit-policy.yaml    name: audit    readOnly: true  - mountPath: /var/log/kubernetes/a
```

and finally configure the `hostPath`:

```
...
volumes:- name: audit  hostPath:    path: /etc/kubernetes/audit-policy.yaml    type: File- name: audit-log  hostPath:    path: /va
```

Webhook backend

The webhook audit backend sends audit events to a remote web API, which is assumed to be a form of the Kubernetes API, including means of authentication. You can configure a webhook audit backend using the following kube-apiserver flags:

- `--audit-webhook-config-file` specifies the path to a file with a webhook configuration. The webhook configuration is effectively a specialized kubeconfig.
- `--audit-webhook-initial-backoff` specifies the amount of time to wait after the first failed request before retrying. Subsequent requests are retried with exponential backoff.

The webhook config file uses the kubeconfig format to specify the remote address of the service and credentials used to connect to it.

Event batching

Both `log` and `webhook` backends support batching. Below is a list of available flags specific to each backend. By default, batching and throttling are **enabled** for the `webhook` backend and **disabled** for the `log` backend.

- webhook
- log

- `--audit-webhook-mode` defines the buffering strategy. One of the following:
  - `batch` - buffer events and asynchronously process them in batches. This is the default mode for the `webhook` backend.
  - `blocking` - block API server responses on processing each individual event.
  - `blocking-strict` - Same as blocking, but when there is a failure during audit logging at the RequestReceived stage, the whole request to the kube-apiserver fails.

The following flags are used only in the `batch` mode:

- `--audit-webhook-batch-buffer-size` defines the number of events to buffer before batching. If the rate of incoming events overflows the buffer, events are dropped. The default value is 10000.
- `--audit-webhook-batch-max-size` defines the maximum number of events in one batch. The default value is 400.
- `--audit-webhook-batch-max-wait` defines the maximum amount of time to wait before unconditionally batching events in the queue. The default value is 30 seconds.
- `--audit-webhook-batch-throttle-enable` defines whether batching throttling is enabled. Throttling is enabled by default.
- `--audit-webhook-batch-throttle-qps` defines the maximum average number of batches generated per second. The default value is 10.
- `--audit-webhook-batch-throttle-burst` defines the maximum number of batches generated at the same moment if the allowed QPS was underutilized previously. The default value is 15.

- `--audit-log-mode` defines the buffering strategy. One of the following:
  - `batch` - buffer events and asynchronously process them in batches. Batching is not recommended for the `log` backend.
  - `blocking` - block API server responses on processing each individual event. This is the default mode for the `log` backend.
  - `blocking-strict` - Same as blocking, but when there is a failure during audit logging at the RequestReceived stage, the whole request to the kube-apiserver fails.

The following flags are used only in the `batch` mode (batching is **disabled** by default for the `log` backend, and when batching is disabled, all batching-related flags are ignored):

- `--audit-log-batch-buffer-size` defines the number of events to buffer before batching. If the rate of incoming events overflows the buffer, events are dropped.
- `--audit-log-batch-max-size` defines the maximum number of events in one batch.
- `--audit-log-batch-max-wait` defines the maximum amount of time to wait before unconditionally batching events in the queue.
- `--audit-log-batch-throttle-enable` defines whether batching throttling is enabled.
- `--audit-log-batch-throttle-qps` defines the maximum average number of batches generated per second.
- `--audit-log-batch-throttle-burst` defines the maximum number of batches generated at the same moment if the allowed QPS was underutilized previously.

Parameter tuning

Parameters should be set to accommodate the load on the API server.

For example, if kube-apiserver receives 100 requests each second, and each request is audited only on `ResponseStarted` and `ResponseComplete` stages, you should account for ≅200 audit events being generated each second. Assuming that there are up to 100 events in a batch, you should set throttling level at least 2 queries per second. Assuming that the backend can take up to 5 seconds to write events, you should set the buffer size to hold up to 5 seconds of events; that is: 10 batches, or 1000 events.

In most cases however, the default parameters should be sufficient and you don't have to worry about setting them manually. You can look at the following Prometheus metrics exposed by kube-apiserver and in the logs to monitor the state of the auditing subsystem.

- `apiserver_audit_event_total` metric contains the total number of audit events exported.
- `apiserver_audit_error_total` metric contains the total number of events dropped due to an error during exporting.

Log entry truncation

Both log and webhook backends support limiting the size of events that are logged. As an example, the following is the list of flags available for the log backend:

- `audit-log-truncate-enabled` whether event and batch truncating is enabled.
- `audit-log-truncate-max-batch-size` maximum size in bytes of the batch sent to the underlying backend.
- `audit-log-truncate-max-event-size` maximum size in bytes of the audit event sent to the underlying backend.

By default truncate is disabled in both `webhook` and `log`, a cluster administrator should set `audit-log-truncate-enabled` or `audit-webhook-truncate-enabled` to enable the feature.

What's next

- Learn about Mutating webhook auditing annotations.
- Learn more about `Event` and the `Policy` resource types by reading the Audit configuration reference.

Debug Running Pods

This page explains how to debug Pods running (or crashing) on a Node.

Before you begin

- Your Pod should already be scheduled and running. If your Pod is not yet running, start with Debugging Pods.
- For some of the advanced debugging steps you need to know on which Node the Pod is running and have shell access to run commands on that Node. You don't need that access to run the standard debug steps that use `kubectl`.

Using `kubectl describe pod` to fetch details about pods

For this example we'll use a Deployment to create two pods, similar to the earlier example.

application/nginx-with-request.yaml Copy application/nginx-with-request.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deploymentspec:  selector:    matchLabels:      app: nginx  replicas: 2  template:    metad
```

Create deployment by running following command:

```
kubectl apply -f https://k8s.io/examples/application/nginx-with-request.yaml
```

```
deployment.apps/nginx-deployment created
```

Check pod status by following command:

```
kubectl get pods
```

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-67d4bdd6f5-cx2nz   1/1     Running   0          13s
nginx-deployment-67d4bdd6f5-w6kd7   1/1     Running   0          13s
```

We can retrieve a lot more information about each of these pods using `kubectl describe pod`. For example:

```
kubectl describe pod nginx-deployment-67d4bdd6f5-w6kd7
```

```
Name:         nginx-deployment-67d4bdd6f5-w6kd7
Namespace:    default
Priority:     0
Node:         kube-worker-1/192.168.0.113
Start Time:   Thu, 17 Feb 2022 16:51:01 -0500
Labels:       app=nginx
              pod-template-hash=67d4bdd6f5
Annotations:  <none>
Status:       Running
IP:           10.88.0.3
IPs:
  IP:           10.88.0.3
  IP:           2001:db8::1
Controlled By:  ReplicaSet/nginx-deployment-67d4bdd6f5
Containers:
  nginx:
    Container ID:   containerd://5403af59a2b46ee5a23fb0ae4b1e077f7ca5c5fb7af16e1ab21c00e0e616462a
    Image:          nginx
    Image ID:       docker.io/library/nginx@sha256:2834dc507516af02784808c5f48b7cbe38b8ed5d0f4837f16e78d00deb7e7767
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Thu, 17 Feb 2022 16:51:05 -0500
    Ready:          True
    Restart Count:  0
    Limits:
      cpu:     500m
      memory:  128Mi
    Requests:
      cpu:        500m
      memory:     128Mi
    Environment:  <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-bgsgp (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-bgsgp:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   Guaranteed
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  34s   default-scheduler  Successfully assigned default/nginx-deployment-67d4bdd6f5-w6kd7 to kube-worker-1
  Normal  Pulling    31s   kubelet            Pulling image "nginx"
  Normal  Pulled     30s   kubelet            Successfully pulled image "nginx" in 1.146417389s
  Normal  Created    30s   kubelet            Created container nginx
  Normal  Started    30s   kubelet            Started container nginx
```

Here you can see configuration information about the container(s) and Pod (labels, resource requirements, etc.), as well as status information about the container(s) and Pod (state, readiness, restart count, events, etc.).

The container state is one of Waiting, Running, or Terminated. Depending on the state, additional information will be provided - here you can see that for a container in Running state, the system tells you when the container started.

Ready tells you whether the container passed its last readiness probe. (In this case, the container does not have a readiness probe configured; the container is assumed to be ready if no readiness probe is configured.)

Restart Count tells you how many times the container has been restarted; this information can be useful for detecting crash loops in containers that are configured with a restart policy of `Always`.

Currently the only Condition associated with a Pod is the binary Ready condition, which indicates that the pod is able to service requests and should be added to the load balancing pools of all matching services.

Lastly, you see a log of recent events related to your Pod. "From" indicates the component that is logging the event. "Reason" and "Message" tell you what happened.

Example: debugging Pending Pods

A common scenario that you can detect using events is when you've created a Pod that won't fit on any node. For example, the Pod might request more resources than are free on any node, or it might specify a label selector that doesn't match any nodes. Let's say we created the previous Deployment with 5 replicas (instead of 2) and requesting 600 millicores instead of 500, on a four-node cluster where each (virtual) machine has 1 CPU. In that case one of the Pods will not be able to schedule. (Note that because of the cluster addon pods such as fluentd, skydns, etc., that run on each node, if we requested 1000 millicores then none of the Pods would be able to schedule.)

```
kubectl get pods
```

```
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-1006230814-6winp   1/1       Running   0          7m
nginx-deployment-1006230814-fmgu3   1/1       Running   0          7m
nginx-deployment-1370807587-6ekbw   1/1       Running   0          1m
nginx-deployment-1370807587-fg172   0/1       Pending   0          1m
nginx-deployment-1370807587-fz9sd   0/1       Pending   0          1m
```

To find out why the nginx-deployment-1370807587-fz9sd pod is not running, we can use `kubectl describe pod` on the pending Pod and look at its events:

```
kubectl describe pod nginx-deployment-1370807587-fz9sd
```

```
  Name:         nginx-deployment-1370807587-fz9sd
  Namespace:    default
  Node:         /
  Labels:               app=nginx,pod-template-hash=1370807587
  Status:               Pending
  IP:
  Controllers:  ReplicaSet/nginx-deployment-1370807587
  Containers:
    nginx:
      Image:    nginx
      Port:     80/TCP
      QoS Tier:
        memory: Guaranteed
        cpu:    Guaranteed
      Limits:
        cpu:    1
        memory: 128Mi
      Requests:
        cpu:    1
        memory: 128Mi
      Environment Variables:
  Volumes:
    default-token-4bcbi:
      Type:     Secret (a volume populated by a Secret)
      SecretName:       default-token-4bcbi
  Events:
    FirstSeen   LastSeen        Count   From                            SubobjectPath   Type            Reason
    ---------   --------        -----   ----                            -------------   --------        ------
    1m              48s             7       {default-scheduler }                            Warning         FailedScheduling
  fit failure on node (kubernetes-node-6ta5): Node didn't have enough resource: CPU, requested: 1000, used: 1420, capacity: 2000
  fit failure on node (kubernetes-node-wul5): Node didn't have enough resource: CPU, requested: 1000, used: 1100, capacity: 2000
```

Here you can see the event generated by the scheduler saying that the Pod failed to schedule for reason `FailedScheduling` (and possibly others). The message tells us that there were not enough resources for the Pod on any of the nodes.

To correct this situation, you can use `kubectl scale` to update your Deployment to specify four or fewer replicas. (Or you could leave the one Pod pending, which is harmless.)

Events such as the ones you saw at the end of `kubectl describe pod` are persisted in etcd and provide high-level information on what is happening in the cluster. To list all events you can use

```
kubectl get events
```

but you have to remember that events are namespaced. This means that if you're interested in events for some namespaced object (e.g. what happened with Pods in namespace `my-namespace`) you need to explicitly provide a namespace to the command:

```
kubectl get events --namespace=my-namespace
```

To see events from all namespaces, you can use the `--all-namespaces` argument.

In addition to `kubectl describe pod`, another way to get extra information about a pod (beyond what is provided by `kubectl get pod`) is to pass the `-o yaml` output format flag to `kubectl get pod`. This will give you, in YAML format, even more information than `kubectl describe pod` - essentially all of the information the system has about the Pod. Here you will see things like annotations (which are key-value metadata without the label restrictions, that is used internally by Kubernetes system components), restart policy, ports, and volumes.

```
kubectl get pod nginx-deployment-1006230814-6winp -o yaml
```

```
apiVersion: v1
kind: Podmetadata:  creationTimestamp: "2022-02-17T21:51:01Z"  generateName: nginx-deployment-67d4bdd6f5-  labels:    app: nginx
```

Examining pod logs

First, look at the logs of the affected container:

```
kubectl logs ${POD_NAME} -c ${CONTAINER_NAME}
```

If your container has previously crashed, you can access the previous container's crash log with:

```
kubectl logs ${POD_NAME} -c ${CONTAINER_NAME} --previous
```

Debugging with container exec

If the container image includes debugging utilities, as is the case with images built from Linux and Windows OS base images, you can run commands inside a specific container with `kubectl exec`:

```
kubectl exec ${POD_NAME} -c ${CONTAINER_NAME} -- ${CMD} ${ARG1} ${ARG2} ... ${ARGN}
```

Note:

`-c ${CONTAINER_NAME}` is optional. You can omit it for Pods that only contain a single container.

As an example, to look at the logs from a running Cassandra pod, you might run

```
kubectl exec cassandra -- cat /var/log/cassandra/system.log
```

You can run a shell that's connected to your terminal using the `-i` and `-t` arguments to `kubectl exec`, for example:

```
kubectl exec -it cassandra -- sh
```

For more details, see Get a Shell to a Running Container.

Debugging with an ephemeral debug container

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

Ephemeral containers are useful for interactive troubleshooting when `kubectl exec` is insufficient because a container has crashed or a container image doesn't include debugging utilities, such as with distroless images.

Example debugging using ephemeral containers

You can use the `kubectl debug` command to add ephemeral containers to a running Pod. First, create a pod for the example:

```
kubectl run ephemeral-demo --image=registry.k8s.io/pause:3.1 --restart=Never
```

The examples in this section use the `pause` container image because it does not contain debugging utilities, but this method works with all container images.

If you attempt to use `kubectl exec` to create a shell you will see an error because there is no shell in this container image.

```
kubectl exec -it ephemeral-demo -- sh
```

```
OCI runtime exec failed: exec failed: container_linux.go:346: starting container process caused "exec: \"sh\": executable file not
```

You can instead add a debugging container using `kubectl debug`. If you specify the `-i`/`--interactive` argument, `kubectl` will automatically attach to the console of the Ephemeral Container.

```
kubectl debug -it ephemeral-demo --image=busybox:1.28 --target=ephemeral-demo
```

```
Defaulting debug container name to debugger-8xzrl.
If you don't see a command prompt, try pressing enter.
/ #
```

This command adds a new busybox container and attaches to it. The `--target` parameter targets the process namespace of another container. It's necessary here because `kubectl run` does not enable process namespace sharing in the pod it creates.

Note:

The `--target` parameter must be supported by the Container Runtime. When not supported, the Ephemeral Container may not be started, or it may be started with an isolated process namespace so that `ps` does not reveal processes in other containers.

You can view the state of the newly created ephemeral container using `kubectl describe`:

```
kubectl describe pod ephemeral-demo
```

```
...
Ephemeral Containers:
  debugger-8xzrl:
    Container ID:   docker://b888f9adfd15bd5739fefaa39e1df4dd3c617b9902082b1cfdc29c4028ffb2eb
    Image:          busybox
    Image ID:       docker-pullable://busybox@sha256:1828edd60c5efd34b2bf5dd3282ec0cc04d47b2ff9caa0b6d4f07a21d1c08084
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Wed, 12 Feb 2020 14:25:42 +0100
    Ready:          False
    Restart Count:  0
    Environment:    <none>
    Mounts:         <none>
...
```

Use `kubectl delete` to remove the Pod when you're finished:

```
kubectl delete pod ephemeral-demo
```

Debugging using a copy of the Pod

Sometimes Pod configuration options make it difficult to troubleshoot in certain situations. For example, you can't run `kubectl exec` to troubleshoot your container if your container image does not include a shell or if your application crashes on startup. In these situations you can use `kubectl debug` to create a copy of the Pod with configuration values changed to aid debugging.

Copying a Pod while adding a new container

Adding a new container can be useful when your application is running but not behaving as you expect and you'd like to add additional troubleshooting utilities to the Pod.

For example, maybe your application's container images are built on `busybox` but you need debugging utilities not included in `busybox`. You can simulate this scenario using `kubectl run`:

```
kubectl run myapp --image=busybox:1.28 --restart=Never -- sleep 1d
```

Run this command to create a copy of `myapp` named `myapp-debug` that adds a new Ubuntu container for debugging:

```
kubectl debug myapp -it --image=ubuntu --share-processes --copy-to=myapp-debug
```

```
Defaulting debug container name to debugger-w7xmf.
If you don't see a command prompt, try pressing enter.
root@myapp-debug:/#
```

Note:

- `kubectl debug` automatically generates a container name if you don't choose one using the `--container` flag.
- The `-i` flag causes `kubectl debug` to attach to the new container by default. You can prevent this by specifying `--attach=false`. If your session becomes disconnected you can reattach using `kubectl attach`.
- The `--share-processes` allows the containers in this Pod to see processes from the other containers in the Pod. For more information about how this works, see Share Process Namespace between Containers in a Pod.

Don't forget to clean up the debugging Pod when you're finished with it:

```
kubectl delete pod myapp myapp-debug
```

Copying a Pod while changing its command

Sometimes it's useful to change the command for a container, for example to add a debugging flag or because the application is crashing.

To simulate a crashing application, use `kubectl run` to create a container that immediately exits:

```
kubectl run --image=busybox:1.28 myapp -- false
```

You can see using `kubectl describe pod myapp` that this container is crashing:

```
Containers:
  myapp:
    Image:         busybox
    ...
    Args:
      false
    State:          Waiting
      Reason:       CrashLoopBackOff
    Last State:     Terminated
      Reason:       Error
      Exit Code:    1
```

You can use `kubectl debug` to create a copy of this Pod with the command changed to an interactive shell:

```
kubectl debug myapp -it --copy-to=myapp-debug --container=myapp -- sh
```

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

Now you have an interactive shell that you can use to perform tasks like checking filesystem paths or running the container command manually.

Note:

- To change the command of a specific container you must specify its name using `--container` or `kubectl debug` will instead create a new container to run the command you specified.
- The `-i` flag causes `kubectl debug` to attach to the container by default. You can prevent this by specifying `--attach=false`. If your session becomes disconnected you can reattach using `kubectl attach`.

Don't forget to clean up the debugging Pod when you're finished with it:

```
kubectl delete pod myapp myapp-debug
```

Copying a Pod while changing container images

In some situations you may want to change a misbehaving Pod from its normal production container images to an image containing a debugging build or additional utilities.

As an example, create a Pod using `kubectl run`:

```
kubectl run myapp --image=busybox:1.28 --restart=Never -- sleep 1d
```

Now use `kubectl debug` to make a copy and change its container image to `ubuntu`:

```
kubectl debug myapp --copy-to=myapp-debug --set-image=*=ubuntu
```

The syntax of `--set-image` uses the same `container_name=image` syntax as `kubectl set image`. `*=ubuntu` means change the image of all containers to `ubuntu`.

Don't forget to clean up the debugging Pod when you're finished with it:

```
kubectl delete pod myapp myapp-debug
```

Debugging via a shell on the node

If none of these approaches work, you can find the Node on which the Pod is running and create a Pod running on the Node. To create an interactive shell on a Node using `kubectl debug`, run:

```
kubectl debug node/mynode -it --image=ubuntu
```

```
Creating debugging pod node-debugger-mynode-pdx84 with container debugger on node mynode.
If you don't see a command prompt, try pressing enter.
root@ek8s:/#
```

When creating a debugging session on a node, keep in mind that:

- `kubectl debug` automatically generates the name of the new Pod based on the name of the Node.
- The root filesystem of the Node will be mounted at `/host`.
- The container runs in the host IPC, Network, and PID namespaces, although the pod isn't privileged, so reading some process information may fail, and `chroot /host` may fail.
- If you need a privileged pod, create it manually or use the `--profile=sysadmin` flag.

Don't forget to clean up the debugging Pod when you're finished with it:

```
kubectl delete pod node-debugger-mynode-pdx84
```

Debugging a Pod or Node while applying a profile

When using `kubectl debug` to debug a node via a debugging Pod, a Pod via an ephemeral container, or a copied Pod, you can apply a profile to them. By applying a profile, specific properties such as securityContext are set, allowing for adaptation to various scenarios. There are two types of profiles, static profile and custom profile.

Applying a Static Profile

A static profile is a set of predefined properties, and you can apply them using the `--profile` flag. The available profiles are as follows:

Profile	Description
legacy	A set of properties backwards compatibility with 1.22 behavior
general	A reasonable set of generic properties for each debugging journey
baseline	A set of properties compatible with PodSecurityStandard baseline policy
restricted	A set of properties compatible with PodSecurityStandard restricted policy
netadmin	A set of properties including Network Administrator privileges
sysadmin	A set of properties including System Administrator (root) privileges

Note:

If you don't specify `--profile`, the `legacy` profile is used by default, but it is planned to be deprecated in the near future. So it is recommended to use other profiles such as `general`.

Assume that you create a Pod and debug it. First, create a Pod named `myapp` as an example:

```
kubectl run myapp --image=busybox:1.28 --restart=Never -- sleep 1d
```

Then, debug the Pod using an ephemeral container. If the ephemeral container needs to have privilege, you can use the `sysadmin` profile:

```
kubectl debug -it myapp --image=busybox:1.28 --target=myapp --profile=sysadmin
```

```
Targeting container "myapp". If you don't see processes from this container it may be because the container runtime doesn't support
Defaulting debug container name to debugger-6kg4x.
If you don't see a command prompt, try pressing enter.
/ #
```

Check the capabilities of the ephemeral container process by running the following command inside the container:

```
/ # grep Cap /proc/$$/status
```

```
...
CapPrm: 000001ffffffffff
```

```
CapEff: 000001fffffffff
...
```

This means the container process is granted full capabilities as a privileged container by applying `sysadmin` profile. See more details about capabilities.

You can also check that the ephemeral container was created as a privileged container:

```
kubectl get pod myapp -o jsonpath='{.spec.ephemeralContainers[0].securityContext}'
```

```
{"privileged":true}
```

Clean up the Pod when you're finished with it:

```
kubectl delete pod myapp
```

Applying Custom Profile

FEATURE STATE: `Kubernetes v1.32 [stable]`

You can define a partial container spec for debugging as a custom profile in either YAML or JSON format, and apply it using the `--custom` flag.

Note:

Custom profile only supports the modification of the container spec, but modifications to `name`, `image`, `command`, `lifecycle` and `volumeDevices` fields of the container spec are not allowed. It does not support the modification of the Pod spec.

Create a Pod named myapp as an example:

```
kubectl run myapp --image=busybox:1.28 --restart=Never -- sleep 1d
```

Create a custom profile in YAML or JSON format. Here, create a YAML format file named `custom-profile.yaml`:

```
env:
- name: ENV_VAR_1  value: value_1- name: ENV_VAR_2  value: value_2securityContext:  capabilities:    add:    - NET_ADMIN    - SYS_
```

Run this command to debug the Pod using an ephemeral container with the custom profile:

```
kubectl debug -it myapp --image=busybox:1.28 --target=myapp --profile=general --custom=custom-profile.yaml
```

You can check that the ephemeral container has been added to the target Pod with the custom profile applied:

```
kubectl get pod myapp -o jsonpath='{.spec.ephemeralContainers[0].env}'
```

```
[{"name":"ENV_VAR_1","value":"value_1"},{"name":"ENV_VAR_2","value":"value_2"}]
```

```
kubectl get pod myapp -o jsonpath='{.spec.ephemeralContainers[0].securityContext}'
```

```
{"capabilities":{"add":["NET_ADMIN","SYS_TIME"]}}
```

Clean up the Pod when you're finished with it:

```
kubectl delete pod myapp
```

Debug Pods

This guide is to help users debug applications that are deployed into Kubernetes and not behaving correctly. This is *not* a guide for people who want to debug their cluster. For that you should check out this guide.

Diagnosing the problem

The first step in troubleshooting is triage. What is the problem? Is it your Pods, your Replication Controller or your Service?

- Debugging Pods
- Debugging Replication Controllers
- Debugging Services

Debugging Pods

The first step in debugging a Pod is taking a look at it. Check the current state of the Pod and recent events with the following command:

```
kubectl describe pods ${POD_NAME}
```

Look at the state of the containers in the pod. Are they all `Running`? Have there been recent restarts?

Continue debugging depending on the state of the pods.

My pod stays pending

If a Pod is stuck in `Pending` it means that it can not be scheduled onto a node. Generally this is because there are insufficient resources of one type or another that prevent scheduling. Look at the output of the `kubectl describe ...` command above. There should be messages from the scheduler about why it can not schedule your pod. Reasons include:

- **You don't have enough resources**: You may have exhausted the supply of CPU or Memory in your cluster, in this case you need to delete Pods, adjust resource requests, or add new nodes to your cluster. See Compute Resources document for more information.

- **You are using `hostPort`**: When you bind a Pod to a `hostPort` there are a limited number of places that pod can be scheduled. In most cases, `hostPort` is unnecessary, try using a Service object to expose your Pod. If you do require `hostPort` then you can only schedule as many Pods as there are nodes in your Kubernetes cluster.

My pod stays waiting

If a Pod is stuck in the `Waiting` state, then it has been scheduled to a worker node, but it can't run on that machine. Again, the information from `kubectl describe ...` should be informative. The most common cause of `Waiting` pods is a failure to pull the image. There are three things to check:

- Make sure that you have the name of the image correct.
- Have you pushed the image to the registry?
- Try to manually pull the image to see if the image can be pulled. For example, if you use Docker on your PC, run `docker pull <image>`.

My pod stays terminating

If a Pod is stuck in the `Terminating` state, it means that a deletion has been issued for the Pod, but the control plane is unable to delete the Pod object.

This typically happens if the Pod has a finalizer and there is an admission webhook installed in the cluster that prevents the control plane from removing the finalizer.

To identify this scenario, check if your cluster has any ValidatingWebhookConfiguration or MutatingWebhookConfiguration that target `UPDATE` operations for `pods` resources.

If the webhook is provided by a third-party:

- Make sure you are using the latest version.
- Disable the webhook for `UPDATE` operations.
- Report an issue with the corresponding provider.

If you are the author of the webhook:

- For a mutating webhook, make sure it never changes immutable fields on `UPDATE` operations. For example, changes to containers are usually not allowed.
- For a validating webhook, make sure that your validation policies only apply to new changes. In other words, you should allow Pods with existing violations to pass validation. This allows Pods that were created before the validating webhook was installed to continue running.

My pod is crashing or otherwise unhealthy

Once your pod has been scheduled, the methods described in Debug Running Pods are available for debugging.

My pod is running but not doing what I told it to do

If your pod is not behaving as you expected, it may be that there was an error in your pod description (e.g. `mypod.yaml` file on your local machine), and that the error was silently ignored when you created the pod. Often a section of the pod description is nested incorrectly, or a key name is typed incorrectly, and so the key is ignored. For example, if you misspelled `command` as `commnd` then the pod will be created but will not use the command line you intended it to use.

The first thing to do is to delete your pod and try creating it again with the `--validate` option. For example, run `kubectl apply --validate -f mypod.yaml`. If you misspelled `command` as `commnd` then will give an error like this:

```
I0805 10:43:25.129850   46757 schema.go:126] unknown field: commnd
I0805 10:43:25.129973   46757 schema.go:129] this may be a false alarm, see https://github.com/kubernetes/kubernetes/issues/6842
pods/mypod
```

The next thing to check is whether the pod on the apiserver matches the pod you meant to create (e.g. in a yaml file on your local machine). For example, run `kubectl get pods/mypod -o yaml > mypod-on-apiserver.yaml` and then manually compare the original pod description, `mypod.yaml` with the one you got back from apiserver, `mypod-on-apiserver.yaml`. There will typically be some lines on the "apiserver" version that are not on the original version. This is expected. However, if there are lines on the original that are not on the apiserver version, then this may indicate a problem with your pod spec.

Debugging Replication Controllers

Replication controllers are fairly straightforward. They can either create Pods or they can't. If they can't create pods, then please refer to the instructions above to debug your pods.

You can also use `kubectl describe rc ${CONTROLLER_NAME}` to introspect events related to the replication controller.

Debugging Services

Services provide load balancing across a set of pods. There are several common problems that can make Services not work properly. The following instructions should help debug Service problems.

First, verify that there are endpoints for the service. For every Service object, the apiserver makes one or more `EndpointSlice` resources available.

You can view these resources with:

```
kubectl get endpointslices -l kubernetes.io/service-name=${SERVICE_NAME}
```

Make sure that the endpoints in the EndpointSlices match up with the number of pods that you expect to be members of your service. For example, if your Service is for an nginx container with 3 replicas, you would expect to see three different IP addresses in the Service's endpoint slices.

My service is missing endpoints

If you are missing endpoints, try listing pods using the labels that Service uses. Imagine that you have a Service where the labels are:

```
...
spec:  - selector:     name: nginx     type: frontend
```

You can use:

```
kubectl get pods --selector=name=nginx,type=frontend
```

to list pods that match this selector. Verify that the list matches the Pods that you expect to provide your Service. Verify that the pod's `containerPort` matches up with the Service's `targetPort`

Network traffic is not forwarded

Please see debugging service for more information.

What's next

If none of the above solves your problem, follow the instructions in Debugging Service document to make sure that your `Service` is running, has `Endpoints`, and your `Pods` are actually serving; you have DNS working, iptables rules installed, and kube-proxy does not seem to be misbehaving.

You may also visit troubleshooting document for more information.

Windows debugging tips

Node-level troubleshooting

1. My Pods are stuck at "Container Creating" or restarting over and over

   Ensure that your pause image is compatible with your Windows OS version. See Pause container to see the latest / recommended pause image and/or get more information.

   Note:

   If using containerd as your container runtime the pause image is specified in the `plugins.plugins.cri.sandbox_image` field of the of config.toml configuration file.

2. My pods show status as `ErrImgPull` or `ImagePullBackOff`

   Ensure that your Pod is getting scheduled to a compatible Windows Node.

   More information on how to specify a compatible node for your Pod can be found in this guide.

Network troubleshooting

1. My Windows Pods do not have network connectivity

   If you are using virtual machines, ensure that MAC spoofing is **enabled** on all the VM network adapter(s).

2. My Windows Pods cannot ping external resources

   Windows Pods do not have outbound rules programmed for the ICMP protocol. However, TCP/UDP is supported. When trying to demonstrate connectivity to resources outside of the cluster, substitute `ping <IP>` with corresponding `curl <IP>` commands.

   If you are still facing problems, most likely your network configuration in cni.conf deserves some extra attention. You can always edit this static file. The configuration update will apply to any new Kubernetes resources.

   One of the Kubernetes networking requirements (see Kubernetes model) is for cluster communication to occur without NAT internally. To honor this requirement, there is an ExceptionList for all the communication where you do not want outbound NAT to occur. However, this also means that you need to exclude the external IP you are trying to query from the `ExceptionList`. Only then will the traffic originating from your Windows pods be SNAT'ed correctly to receive a response from the outside world. In this regard, your `ExceptionList` in `cni.conf` should look as follows:

   ```
   "ExceptionList": [
                   "10.244.0.0/16",  # Cluster subnet
                   "10.96.0.0/12",   # Service subnet
                   "10.127.130.0/24" # Management (host) subnet
               ]
   ```

3. My Windows node cannot access `NodePort` type Services

   Local NodePort access from the node itself fails. This is a known limitation. NodePort access works from other nodes or external clients.

4. vNICs and HNS endpoints of containers are being deleted

This issue can be caused when the `hostname-override` parameter is not passed to kube-proxy. To resolve it, users need to pass the hostname to kube-proxy as follows:

```
C:\k\kube-proxy.exe --hostname-override=$(hostname)
```

5. My Windows node cannot access my services using the service IP

   This is a known limitation of the networking stack on Windows. However, Windows Pods can access the Service IP.

6. No network adapter is found when starting the kubelet

   The Windows networking stack needs a virtual adapter for Kubernetes networking to work. If the following commands return no results (in an admin shell), virtual network creation — a necessary prerequisite for the kubelet to work — has failed:

   ```
   Get-HnsNetwork | ? Name -ieq "cbr0"
   Get-NetAdapter | ? Name -Like "vEthernet (Ethernet*"
   ```

   Often it is worthwhile to modify the InterfaceName parameter of the `start.ps1` script, in cases where the host's network adapter isn't "Ethernet". Otherwise, consult the output of the `start-kubelet.ps1` script to see if there are errors during virtual network creation.

7. DNS resolution is not properly working

   Check the DNS limitations for Windows in this section.

8. `kubectl port-forward` fails with "unable to do port forwarding: wincat not found"

   This was implemented in Kubernetes 1.15 by including `wincat.exe` in the pause infrastructure container `mcr.microsoft.com/oss/kubernetes/pause:3.6`. Be sure to use a supported version of Kubernetes. If you would like to build your own pause infrastructure container be sure to include wincat.

9. My Kubernetes installation is failing because my Windows Server node is behind a proxy

   If you are behind a proxy, the following PowerShell environment variables must be defined:

   ```
   [Environment]::SetEnvironmentVariable("HTTP_PROXY", "http://proxy.example.com:80/", [EnvironmentVariableTarget]::Machine)
   [Environment]::SetEnvironmentVariable("HTTPS_PROXY", "http://proxy.example.com:443/", [EnvironmentVariableTarget]::Machine)
   ```

Flannel troubleshooting

1. With Flannel, my nodes are having issues after rejoining a cluster

   Whenever a previously deleted node is being re-joined to the cluster, flannelD tries to assign a new pod subnet to the node. Users should remove the old pod subnet configuration files in the following paths:

   ```
   Remove-Item C:\k\SourceVip.json
   Remove-Item C:\k\SourceVipRequest.json
   ```

2. Flanneld is stuck in "Waiting for the Network to be created"

   There are numerous reports of this issue; most likely it is a timing issue for when the management IP of the flannel network is set. A workaround is to relaunch `start.ps1` or relaunch it manually as follows:

   ```
   [Environment]::SetEnvironmentVariable("NODE_NAME", "<Windows_Worker_Hostname>")
   C:\flannel\flanneld.exe --kubeconfig-file=c:\k\config --iface=<Windows_Worker_Node_IP> --ip-masq=1 --kube-subnet-mgr=1
   ```

3. My Windows Pods cannot launch because of missing `/run/flannel/subnet.env`

   This indicates that Flannel didn't launch correctly. You can either try to restart `flanneld.exe` or you can copy the files over manually from `/run/flannel/subnet.env` on the Kubernetes master to `C:\run\flannel\subnet.env` on the Windows worker node and modify the `FLANNEL_SUBNET` row to a different number. For example, if node subnet 10.244.4.1/24 is desired:

   ```
   FLANNEL_NETWORK=10.244.0.0/16
   FLANNEL_SUBNET=10.244.4.1/24
   FLANNEL_MTU=1500
   FLANNEL_IPMASQ=true
   ```

Further investigation

If these steps don't resolve your problem, you can get help running Windows containers on Windows nodes in Kubernetes through:

- StackOverflow Windows Server Container topic
- Kubernetes Official Forum discuss.kubernetes.io
- Kubernetes Slack #SIG-Windows Channel

Troubleshooting Applications

Debugging common containerized application issues.

This doc contains a set of resources for fixing issues with containerized applications. It covers things like common issues with Kubernetes resources (like Pods, Services, or StatefulSets), advice on making sense of container termination messages, and ways to debug running containers.

Debug Pods

Debugging Kubernetes nodes with crictl

FEATURE STATE: `Kubernetes v1.11 [stable]`

`crictl` is a command-line interface for CRI-compatible container runtimes. You can use it to inspect and debug container runtimes and applications on a Kubernetes node. `crictl` and its source are hosted in the cri-tools repository.

Before you begin

`crictl` requires a Linux operating system with a CRI runtime.

Installing crictl

You can download a compressed archive `crictl` from the cri-tools release page, for several different architectures. Download the version that corresponds to your version of Kubernetes. Extract it and move it to a location on your system path, such as `/usr/local/bin/`.

General usage

The `crictl` command has several subcommands and runtime flags. Use `crictl help` or `crictl <subcommand> help` for more details.

You can set the endpoint for `crictl` by doing one of the following:

- Set the `--runtime-endpoint` and `--image-endpoint` flags.
- Set the `CONTAINER_RUNTIME_ENDPOINT` and `IMAGE_SERVICE_ENDPOINT` environment variables.
- Set the endpoint in the configuration file `/etc/crictl.yaml`. To specify a different file, use the `--config=PATH_TO_FILE` flag when you run `crictl`.

Note:

If you don't set an endpoint, `crictl` attempts to connect to a list of known endpoints, which might result in an impact to performance.

You can also specify timeout values when connecting to the server and enable or disable debugging, by specifying `timeout` or `debug` values in the configuration file or using the `--timeout` and `--debug` command-line flags.

To view or edit the current configuration, view or edit the contents of `/etc/crictl.yaml`. For example, the configuration when using the `containerd` container runtime would be similar to this:

```
runtime-endpoint: unix:///var/run/containerd/containerd.sock
image-endpoint: unix:///var/run/containerd/containerd.sock
timeout: 10
debug: true
```

To learn more about `crictl`, refer to the `crictl` documentation.

Example crictl commands

The following examples show some `crictl` commands and example output.

List pods

List all pods:

```
crictl pods
```

The output is similar to this:

```
POD ID              CREATED             STATE               NAME                            NAMESPACE           ATTEMPT
926f1b5a1d33a       About a minute ago  Ready               sh-84d7dcf559-4r2gq             default             0
4dccb216c4adb       About a minute ago  Ready               nginx-65899c769f-wv2gp          default             0
a86316e96fa89       17 hours ago        Ready               kube-proxy-gblk4                kube-system         0
919630b8f81f1       17 hours ago        Ready               nvidia-device-plugin-zgbbv      kube-system         0
```

List pods by name:

```
crictl pods --name nginx-65899c769f-wv2gp
```

The output is similar to this:

```
POD ID              CREATED             STATE               NAME                      NAMESPACE           ATTEMPT
4dccb216c4adb       2 minutes ago       Ready               nginx-65899c769f-wv2gp    default             0
```

List pods by label:

```
crictl pods --label run=nginx
```

The output is similar to this:

```
POD ID              CREATED             STATE               NAME                      NAMESPACE           ATTEMPT
4dccb216c4adb       2 minutes ago       Ready               nginx-65899c769f-wv2gp    default             0
```

List images

List all images:

```
crictl images
```

The output is similar to this:

```
IMAGE                                TAG                 IMAGE ID            SIZE
busybox                              latest              8c811b4aec35f       1.15MB
k8s-gcrio.azureedge.net/hyperkube-amd64   v1.10.3             e179bbfe5d238       665MB
k8s-gcrio.azureedge.net/pause-amd64       3.1                 da86e6ba6ca19       742kB
nginx                                latest              cd5239a0906a6       109MB
```

List images by repository:

```
crictl images nginx
```

The output is similar to this:

```
IMAGE               TAG                 IMAGE ID            SIZE
nginx               latest              cd5239a0906a6       109MB
```

Only list image IDs:

```
crictl images -q
```

The output is similar to this:

```
sha256:8c811b4aec35f259572d0f79207bc0678df4c736eeec50bc9fec37ed936a472a
sha256:e179bbfe5d238de6069f3b03fccbecc3fb4f2019af741bfff1233c4d7b2970c5
sha256:da86e6ba6ca197bf6bc5e9d900febd906b133eaa4750e6bed647b0fbe50ed43e
sha256:cd5239a0906a6ccf0562354852fae04bc5b52d72a2aff9a871ddb6bd57553569
```

List containers

List all containers:

```
crictl ps -a
```

The output is similar to this:

```
CONTAINER ID        IMAGE
1f73f2d81bf98       busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f41046e0f37d47
9c5951df22c78       busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f41046e0f37d47
87d3992f84f74       nginx@sha256:d0a8828cccb73397acb0073bf34f4d7d8aa315263f1e7806bf8c55d8ac139d5f
1941fb4da154f       k8s-gcrio.azureedge.net/hyperkube-amd64@sha256:00d814b1f7763f4ab5be80c58e98140dfc69df107f253d7fdd714b30a714260a
```

List running containers:

```
crictl ps
```

The output is similar to this:

```
CONTAINER ID        IMAGE
1f73f2d81bf98       busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f41046e0f37d47
87d3992f84f74       nginx@sha256:d0a8828cccb73397acb0073bf34f4d7d8aa315263f1e7806bf8c55d8ac139d5f
1941fb4da154f       k8s-gcrio.azureedge.net/hyperkube-amd64@sha256:00d814b1f7763f4ab5be80c58e98140dfc69df107f253d7fdd714b30a714260a
```

Execute a command in a running container

```
crictl exec -i -t 1f73f2d81bf98 ls
```

The output is similar to this:

```
bin   dev   etc   home  proc  root  sys   tmp   usr   var
```

Get a container's logs

Get all container logs:

```
crictl logs 87d3992f84f74
```

The output is similar to this:

```
10.240.0.96 - - [06/Jun/2018:02:45:49 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
10.240.0.96 - - [06/Jun/2018:02:45:50 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
10.240.0.96 - - [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
```

Get only the latest `N` lines of logs:

```
crictl logs --tail=1 87d3992f84f74
```

The output is similar to this:

```
10.240.0.96 - - [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
```

What's next

- Learn more about `crictl`.

Troubleshooting Clusters

Debugging common cluster issues.

This doc is about cluster troubleshooting; we assume you have already ruled out your application as the root cause of the problem you are experiencing. See the application troubleshooting guide for tips on application debugging. You may also visit the troubleshooting overview document for more information.

For troubleshooting kubectl, refer to Troubleshooting kubectl.

Listing your cluster

The first thing to debug in your cluster is if your nodes are all registered correctly.

Run the following command:

```
kubectl get nodes
```

And verify that all of the nodes you expect to see are present and that they are all in the `Ready` state.

To get detailed information about the overall health of your cluster, you can run:

```
kubectl cluster-info dump
```

Example: debugging a down/unreachable node

Sometimes when debugging it can be useful to look at the status of a node -- for example, because you've noticed strange behavior of a Pod that's running on the node, or to find out why a Pod won't schedule onto the node. As with Pods, you can use `kubectl describe node` and `kubectl get node -o yaml` to retrieve detailed information about nodes. For example, here's what you'll see if a node is down (disconnected from the network, or kubelet dies and won't restart, etc.). Notice the events that show the node is NotReady, and also notice that the pods are no longer running (they are evicted after five minutes of NotReady status).

```
kubectl get nodes
```

```
NAME                     STATUS       ROLES     AGE     VERSION
kube-worker-1            NotReady     <none>    1h      v1.23.3
kubernetes-node-bols     Ready        <none>    1h      v1.23.3
kubernetes-node-st6x     Ready        <none>    1h      v1.23.3
kubernetes-node-unaj     Ready        <none>    1h      v1.23.3
```

```
kubectl describe node kube-worker-1
```

```
Name:               kube-worker-1
Roles:              <none>
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=kube-worker-1
                    kubernetes.io/os=linux
                    node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:  Thu, 17 Feb 2022 16:46:30 -0500
Taints:             node.kubernetes.io/unreachable:NoExecute
                    node.kubernetes.io/unreachable:NoSchedule
Unschedulable:      false
Lease:
  HolderIdentity:  kube-worker-1
  AcquireTime:     <unset>
  RenewTime:       Thu, 17 Feb 2022 17:13:09 -0500
Conditions:
  Type                 Status    LastHeartbeatTime                 LastTransitionTime                Reason              Message
  ----                 ------    -----------------                 ------------------                ------              -------
  NetworkUnavailable   False     Thu, 17 Feb 2022 17:09:13 -0500   Thu, 17 Feb 2022 17:09:13 -0500   WeaveIsUp           Weave pod
  MemoryPressure       Unknown   Thu, 17 Feb 2022 17:12:40 -0500   Thu, 17 Feb 2022 17:13:52 -0500   NodeStatusUnknown   Kubelet st
  DiskPressure         Unknown   Thu, 17 Feb 2022 17:12:40 -0500   Thu, 17 Feb 2022 17:13:52 -0500   NodeStatusUnknown   Kubelet st
  PIDPressure          Unknown   Thu, 17 Feb 2022 17:12:40 -0500   Thu, 17 Feb 2022 17:13:52 -0500   NodeStatusUnknown   Kubelet st
  Ready                Unknown   Thu, 17 Feb 2022 17:12:40 -0500   Thu, 17 Feb 2022 17:13:52 -0500   NodeStatusUnknown   Kubelet st
Addresses:
  InternalIP:  192.168.0.113
  Hostname:    kube-worker-1
Capacity:
```

```
  cpu:                2
  ephemeral-storage:  15372232Ki
  hugepages-2Mi:      0
  memory:             2025188Ki
  pods:               110
Allocatable:
  cpu:                2
  ephemeral-storage:  14167048988
  hugepages-2Mi:      0
  memory:             1922788Ki
  pods:               110
System Info:
  Machine ID:                 9384e2927f544209b5d7b67474bbf92b
  System UUID:                aa829ca9-73d7-064d-9019-df07404ad448
  Boot ID:                    5a295a03-aaca-4340-af20-1327fa5dab5c
  Kernel Version:             5.13.0-28-generic
  OS Image:                   Ubuntu 21.10
  Operating System:           linux
  Architecture:               amd64
  Container Runtime Version:  containerd://1.5.9
  Kubelet Version:            v1.23.3
  Kube-Proxy Version:         v1.23.3
Non-terminated Pods:          (4 in total)
  Namespace                   Name                                CPU Requests  CPU Limits  Memory Requests  Memory Limits  Age
  ---------                   ----                                ------------  ----------  ---------------  -------------  ---
  default                     nginx-deployment-67d4bdd6f5-cx2nz   500m (25%)    500m (25%)  128Mi (6%)       128Mi (6%)     23m
  default                     nginx-deployment-67d4bdd6f5-w6kd7   500m (25%)    500m (25%)  128Mi (6%)       128Mi (6%)     23m
  kube-system                 kube-proxy-dnxbz                    0 (0%)        0 (0%)      0 (0%)           0 (0%)         28m
  kube-system                 weave-net-gjxxp                     100m (5%)     0 (0%)      200Mi (10%)      0 (0%)         28m
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests     Limits
  --------           --------     ------
  cpu                1100m (55%)  1 (50%)
  memory             456Mi (24%)  256Mi (13%)
  ephemeral-storage  0 (0%)       0 (0%)
  hugepages-2Mi      0 (0%)       0 (0%)
Events:
...
```

```
kubectl get node kube-worker-1 -o yaml
```

```
apiVersion: v1
kind: Nodemetadata:  annotations:    node.alpha.kubernetes.io/ttl: "0"    volumes.kubernetes.io/controller-managed-attach-detach:
```

Looking at logs

For now, digging deeper into the cluster requires logging into the relevant machines. Here are the locations of the relevant log files. On systemd-based systems, you may need to use `journalctl` instead of examining log files.

Control Plane nodes

- `/var/log/kube-apiserver.log` - API Server, responsible for serving the API
- `/var/log/kube-scheduler.log` - Scheduler, responsible for making scheduling decisions
- `/var/log/kube-controller-manager.log` - a component that runs most Kubernetes built-in controllers, with the notable exception of scheduling (the kube-scheduler handles scheduling).

Worker Nodes

- `/var/log/kubelet.log` - logs from the kubelet, responsible for running containers on the node
- `/var/log/kube-proxy.log` - logs from `kube-proxy`, which is responsible for directing traffic to Service endpoints

Cluster failure modes

This is an incomplete list of things that could go wrong, and how to adjust your cluster setup to mitigate the problems.

Contributing causes

- VM(s) shutdown
- Network partition within cluster, or between cluster and users
- Crashes in Kubernetes software
- Data loss or unavailability of persistent storage (e.g. GCE PD or AWS EBS volume)
- Operator error, for example, misconfigured Kubernetes software or application software

Specific scenarios

- API server VM shutdown or apiserver crashing
  - Results
    - unable to stop, update, or start new pods, services, replication controller
    - existing pods and services should continue to work normally unless they depend on the Kubernetes API
- API server backing storage lost
  - Results
    - the kube-apiserver component fails to start successfully and become healthy
    - kubelets will not be able to reach it but will continue to run the same pods and provide the same service proxying
    - manual recovery or recreation of apiserver state necessary before apiserver is restarted
- Supporting services (node controller, replication controller manager, scheduler, etc) VM shutdown or crashes

    - currently those are colocated with the apiserver, and their unavailability has similar consequences as apiserver
    - in future, these will be replicated as well and may not be co-located
    - they do not have their own persistent state
- Individual node (VM or physical machine) shuts down
    - Results
        - pods on that Node stop running
- Network partition
    - Results
        - partition A thinks the nodes in partition B are down; partition B thinks the apiserver is down. (Assuming the master VM ends up in partition A.)
- Kubelet software fault
    - Results
        - crashing kubelet cannot start new pods on the node
        - kubelet might delete the pods or not
        - node marked unhealthy
        - replication controllers start new pods elsewhere
- Cluster operator error
    - Results
        - loss of pods, services, etc
        - lost of apiserver backing store
        - users unable to read API
        - etc.

Mitigations

- Action: Use the IaaS provider's automatic VM restarting feature for IaaS VMs

    - Mitigates: Apiserver VM shutdown or apiserver crashing
    - Mitigates: Supporting services VM shutdown or crashes

- Action: Use IaaS providers reliable storage (e.g. GCE PD or AWS EBS volume) for VMs with apiserver+etcd

    - Mitigates: Apiserver backing storage lost

- Action: Use high-availability configuration

    - Mitigates: Control plane node shutdown or control plane components (scheduler, API server, controller-manager) crashing
        - Will tolerate one or more simultaneous node or component failures
    - Mitigates: API server backing storage (i.e., etcd's data directory) lost
        - Assumes HA (highly-available) etcd configuration

- Action: Snapshot apiserver PDs/EBS-volumes periodically

    - Mitigates: Apiserver backing storage lost
    - Mitigates: Some cases of operator error
    - Mitigates: Some cases of Kubernetes software fault

- Action: use replication controller and services in front of pods

    - Mitigates: Node shutdown
    - Mitigates: Kubelet software fault

- Action: applications (containers) designed to tolerate unexpected restarts

    - Mitigates: Node shutdown
    - Mitigates: Kubelet software fault

What's next

- Learn about the metrics available in the Resource Metrics Pipeline
- Discover additional tools for monitoring resource usage
- Use Node Problem Detector to monitor node health
- Use `kubectl debug node` to debug Kubernetes nodes
- Use `crictl` to debug Kubernetes nodes
- Get more information about Kubernetes auditing
- Use `telepresence` to develop and debug services locally

Debug Services

An issue that comes up rather frequently for new installations of Kubernetes is that a Service is not working properly. You've run your Pods through a Deployment (or other workload controller) and created a Service, but you get no response when you try to access it. This document will hopefully help you to figure out what's going wrong.

Running commands in a Pod

For many steps here you will want to see what a Pod running in the cluster sees. The simplest way to do this is to run an interactive busybox Pod:

```
kubectl run -it --rm --restart=Never busybox --image=gcr.io/google-containers/busybox sh
```

Note:

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

If you already have a running Pod that you prefer to use, you can run a command in it using:

```
kubectl exec <POD-NAME> -c <CONTAINER-NAME> -- <COMMAND>
```

Setup

For the purposes of this walk-through, let's run some Pods. Since you're probably debugging your own Service you can substitute your own details, or you can follow along and get a second data point.

```
kubectl create deployment hostnames --image=registry.k8s.io/serve_hostname
```

```
deployment.apps/hostnames created
```

`kubectl` commands will print the type and name of the resource created or mutated, which can then be used in subsequent commands.

Let's scale the deployment to 3 replicas.

```
kubectl scale deployment hostnames --replicas=3
```

```
deployment.apps/hostnames scaled
```

Note that this is the same as if you had started the Deployment with the following YAML:

```
apiVersion: apps/v1
kind: Deploymentmetadata:  labels:    app: hostnames  name: hostnamesspec:  selector:    matchLabels:      app: hostnames  replica
```

The label "app" is automatically set by `kubectl create deployment` to the name of the Deployment.

You can confirm your Pods are running:

```
kubectl get pods -l app=hostnames
```

```
NAME                        READY     STATUS    RESTARTS   AGE
hostnames-632524106-bbpiw   1/1       Running   0          2m
hostnames-632524106-ly40y   1/1       Running   0          2m
hostnames-632524106-tlaok   1/1       Running   0          2m
```

You can also confirm that your Pods are serving. You can get the list of Pod IP addresses and test them directly.

```
kubectl get pods -l app=hostnames \
    -o go-template='{{range .items}}{{.status.podIP}}{{"\n"}}{{end}}'
```

```
10.244.0.5
10.244.0.6
10.244.0.7
```

The example container used for this walk-through serves its own hostname via HTTP on port 9376, but if you are debugging your own app, you'll want to use whatever port number your Pods are listening on.

From within a pod:

```
for ep in 10.244.0.5:9376 10.244.0.6:9376 10.244.0.7:9376; do
    wget -qO- $ep
done
```

This should produce something like:

```
hostnames-632524106-bbpiw
hostnames-632524106-ly40y
hostnames-632524106-tlaok
```

If you are not getting the responses you expect at this point, your Pods might not be healthy or might not be listening on the port you think they are. You might find `kubectl logs` to be useful for seeing what is happening, or perhaps you need to `kubectl exec` directly into your Pods and debug from there.

Assuming everything has gone to plan so far, you can start to investigate why your Service doesn't work.

Does the Service exist?

The astute reader will have noticed that you did not actually create a Service yet - that is intentional. This is a step that sometimes gets forgotten, and is the first thing to check.

What would happen if you tried to access a non-existent Service? If you have another Pod that consumes this Service by name you would get something like:

```
wget -O- hostnames
```

```
Resolving hostnames (hostnames)... failed: Name or service not known.
wget: unable to resolve host address 'hostnames'
```

The first thing to check is whether that Service actually exists:

```
kubectl get svc hostnames
```

```
No resources found.
Error from server (NotFound): services "hostnames" not found
```

Let's create the Service. As before, this is for the walk-through - you can use your own Service's details here.

```
kubectl expose deployment hostnames --port=80 --target-port=9376

service/hostnames exposed
```

And read it back:

```
kubectl get svc hostnames

NAME        TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
hostnames   ClusterIP   10.0.1.175   <none>        80/TCP    5s
```

Now you know that the Service exists.

As before, this is the same as if you had started the Service with YAML:

```
apiVersion: v1
kind: Servicemetadata:  labels:    app: hostnames  name: hostnamesspec:  selector:    app: hostnames  ports:  - name: default    p
```

In order to highlight the full range of configuration, the Service you created here uses a different port number than the Pods. For many real-world Services, these values might be the same.

Any Network Policy Ingress rules affecting the target Pods?

If you have deployed any Network Policy Ingress rules which may affect incoming traffic to `hostnames-*` Pods, these need to be reviewed.

Please refer to Network Policies for more details.

Does the Service work by DNS name?

One of the most common ways that clients consume a Service is through a DNS name.

From a Pod in the same Namespace:

```
nslookup hostnames

Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      hostnames
Address 1: 10.0.1.175 hostnames.default.svc.cluster.local
```

If this fails, perhaps your Pod and Service are in different Namespaces, try a namespace-qualified name (again, from within a Pod):

```
nslookup hostnames.default

Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      hostnames.default
Address 1: 10.0.1.175 hostnames.default.svc.cluster.local
```

If this works, you'll need to adjust your app to use a cross-namespace name, or run your app and Service in the same Namespace. If this still fails, try a fully-qualified name:

```
nslookup hostnames.default.svc.cluster.local

Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      hostnames.default.svc.cluster.local
Address 1: 10.0.1.175 hostnames.default.svc.cluster.local
```

Note the suffix here: "default.svc.cluster.local". The "default" is the Namespace you're operating in. The "svc" denotes that this is a Service. The "cluster.local" is your cluster domain, which COULD be different in your own cluster.

You can also try this from a Node in the cluster:

Note:

10.0.0.10 is the cluster's DNS Service IP, yours might be different.

```
nslookup hostnames.default.svc.cluster.local 10.0.0.10

Server:         10.0.0.10
Address:        10.0.0.10#53

Name:   hostnames.default.svc.cluster.local
Address: 10.0.1.175
```

If you are able to do a fully-qualified name lookup but not a relative one, you need to check that your `/etc/resolv.conf` file in your Pod is correct. From within a Pod:

```
cat /etc/resolv.conf
```

You should see something like:

```
nameserver 10.0.0.10
search default.svc.cluster.local svc.cluster.local cluster.local example.com
options ndots:5
```

The `nameserver` line must indicate your cluster's DNS Service. This is passed into `kubelet` with the `--cluster-dns` flag.

The `search` line must include an appropriate suffix for you to find the Service name. In this case it is looking for Services in the local Namespace ("default.svc.cluster.local"), Services in all Namespaces ("svc.cluster.local"), and lastly for names in the cluster ("cluster.local"). Depending on your own install you might have additional records after that (up to 6 total). The cluster suffix is passed into `kubelet` with the `--cluster-domain` flag. Throughout this document, the cluster suffix is assumed to be "cluster.local". Your own clusters might be configured differently, in which case you should change that in all of the previous commands.

The `options` line must set `ndots` high enough that your DNS client library considers search paths at all. Kubernetes sets this to 5 by default, which is high enough to cover all of the DNS names it generates.

Does any Service work by DNS name?

If the above still fails, DNS lookups are not working for your Service. You can take a step back and see what else is not working. The Kubernetes master Service should always work. From within a Pod:

```
nslookup kubernetes.default
```

```
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      kubernetes.default
Address 1: 10.0.0.1 kubernetes.default.svc.cluster.local
```

If this fails, please see the kube-proxy section of this document, or even go back to the top of this document and start over, but instead of debugging your own Service, debug the DNS Service.

Does the Service work by IP?

Assuming you have confirmed that DNS works, the next thing to test is whether your Service works by its IP address. From a Pod in your cluster, access the Service's IP (from `kubectl get` above).

```
for i in $(seq 1 3); do
    wget -qO- 10.0.1.175:80
done
```

This should produce something like:

```
hostnames-632524106-bbpiw
hostnames-632524106-ly40y
hostnames-632524106-tlaok
```

If your Service is working, you should get correct responses. If not, there are a number of things that could be going wrong. Read on.

Is the Service defined correctly?

It might sound silly, but you should really double and triple check that your Service is correct and matches your Pod's port. Read back your Service and verify it:

```
kubectl get service hostnames -o json
```

```
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "hostnames",
        "namespace": "default",
        "uid": "428c8b6c-24bc-11e5-936d-42010af0a9bc",
        "resourceVersion": "347189",
        "creationTimestamp": "2015-07-07T15:24:29Z",
        "labels": {
            "app": "hostnames"
        }
    },
    "spec": {
        "ports": [
            {
                "name": "default",
                "protocol": "TCP",
                "port": 80,
                "targetPort": 9376,
                "nodePort": 0
            }
        ],
        "selector": {
            "app": "hostnames"
        },
        "clusterIP": "10.0.1.175",
        "type": "ClusterIP",
        "sessionAffinity": "None"
    },
    "status": {
        "loadBalancer": {}
    }
}
```

- Is the Service port you are trying to access listed in `spec.ports[]`?
- Is the `targetPort` correct for your Pods (some Pods use a different port than the Service)?

- If you meant to use a numeric port, is it a number (9376) or a string "9376"?
- If you meant to use a named port, do your Pods expose a port with the same name?
- Is the port's `protocol` correct for your Pods?

Does the Service have any EndpointSlices?

If you got this far, you have confirmed that your Service is correctly defined and is resolved by DNS. Now let's check that the Pods you ran are actually being selected by the Service.

Earlier you saw that the Pods were running. You can re-check that:

```
kubectl get pods -l app=hostnames
```

```
NAME                        READY     STATUS    RESTARTS   AGE
hostnames-632524106-bbpiw   1/1       Running   0          1h
hostnames-632524106-ly40y   1/1       Running   0          1h
hostnames-632524106-tlaok   1/1       Running   0          1h
```

The `-l app=hostnames` argument is a label selector configured on the Service.

The "AGE" column says that these Pods are about an hour old, which implies that they are running fine and not crashing.

The "RESTARTS" column says that these pods are not crashing frequently or being restarted. Frequent restarts could lead to intermittent connectivity issues. If the restart count is high, read more about how to debug pods.

Inside the Kubernetes system is a control loop which evaluates the selector of every Service and saves the results into one or more EndpointSlice objects.

```
kubectl get endpointslices -l k8s.io/service-name=hostnames
```

```
NAME              ADDRESSTYPE   PORTS   ENDPOINTS
hostnames-ytpni   IPv4          9376    10.244.0.5,10.244.0.6,10.244.0.7
```

This confirms that the EndpointSlice controller has found the correct Pods for your Service. If the `ENDPOINTS` column is `<none>`, you should check that the `spec.selector` field of your Service actually selects for `metadata.labels` values on your Pods. A common mistake is to have a typo or other error, such as the Service selecting for `app=hostnames`, but the Deployment specifying `run=hostnames`, as in versions previous to 1.18, where the `kubectl run` command could have been also used to create a Deployment.

Are the Pods working?

At this point, you know that your Service exists and has selected your Pods. At the beginning of this walk-through, you verified the Pods themselves. Let's check again that the Pods are actually working - you can bypass the Service mechanism and go straight to the Pods, as listed by the Endpoints above.

Note:

These commands use the Pod port (9376), rather than the Service port (80).

From within a Pod:

```
for ep in 10.244.0.5:9376 10.244.0.6:9376 10.244.0.7:9376; do
    wget -qO- $ep
done
```

This should produce something like:

```
hostnames-632524106-bbpiw
hostnames-632524106-ly40y
hostnames-632524106-tlaok
```

You expect each Pod in the endpoints list to return its own hostname. If this is not what happens (or whatever the correct behavior is for your own Pods), you should investigate what's happening there.

Is the kube-proxy working?

If you get here, your Service is running, has EndpointSlices, and your Pods are actually serving. At this point, the whole Service proxy mechanism is suspect. Let's confirm it, piece by piece.

The default implementation of Services, and the one used on most clusters, is kube-proxy. This is a program that runs on every node and configures one of a small set of mechanisms for providing the Service abstraction. If your cluster does not use kube-proxy, the following sections will not apply, and you will have to investigate whatever implementation of Services you are using.

Is kube-proxy running?

Confirm that `kube-proxy` is running on your Nodes. Running directly on a Node, you should get something like the below:

```
ps auxw | grep kube-proxy
```

```
root  4194  0.4  0.1 101864 17696 ?    Sl Jul04  25:43 /usr/local/bin/kube-proxy --master=https://kubernetes-master --kubeconfig=/
```

Next, confirm that it is not failing something obvious, like contacting the master. To do this, you'll have to look at the logs. Accessing the logs depends on your Node OS. On some OSes it is a file, such as /var/log/kube-proxy.log, while other OSes use `journalctl` to access logs. You should see something like:

```
I1027 22:14:53.995134    5063 server.go:200] Running in resource-only container "/kube-proxy"
I1027 22:14:53.998163    5063 server.go:247] Using iptables Proxier.
I1027 22:14:54.038140    5063 proxier.go:352] Setting endpoints for "kube-system/kube-dns:dns-tcp" to [10.244.1.3:53]
```

```
I1027 22:14:54.038164    5063 proxier.go:352] Setting endpoints for "kube-system/kube-dns:dns" to [10.244.1.3:53]
I1027 22:14:54.038209    5063 proxier.go:352] Setting endpoints for "default/kubernetes:https" to [10.240.0.2:443]
I1027 22:14:54.038238    5063 proxier.go:429] Not syncing iptables until Services and Endpoints have been received from master
I1027 22:14:54.040048    5063 proxier.go:294] Adding new service "default/kubernetes:https" at 10.0.0.1:443/TCP
I1027 22:14:54.040154    5063 proxier.go:294] Adding new service "kube-system/kube-dns:dns" at 10.0.0.10:53/UDP
I1027 22:14:54.040223    5063 proxier.go:294] Adding new service "kube-system/kube-dns:dns-tcp" at 10.0.0.10:53/TCP
```

If you see error messages about not being able to contact the master, you should double-check your Node configuration and installation steps.

Kube-proxy can run in one of a few modes. In the log listed above, the line `Using iptables Proxier` indicates that kube-proxy is running in "iptables" mode. The most common other mode is "ipvs".

Iptables mode

In "iptables" mode, you should see something like the following on a Node:

```
iptables-save | grep hostnames
```

```
-A KUBE-SEP-57KPRZ3JQVENLNBR -s 10.244.3.6/32 -m comment --comment "default/hostnames:" -j MARK --set-xmark 0x00004000/0x00004000
-A KUBE-SEP-57KPRZ3JQVENLNBR -p tcp -m comment --comment "default/hostnames:" -m tcp -j DNAT --to-destination 10.244.3.6:9376
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -s 10.244.1.7/32 -m comment --comment "default/hostnames:" -j MARK --set-xmark 0x00004000/0x00004000
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -p tcp -m comment --comment "default/hostnames:" -m tcp -j DNAT --to-destination 10.244.1.7:9376
-A KUBE-SEP-X3P2623AGDH6CDF3 -s 10.244.2.3/32 -m comment --comment "default/hostnames:" -j MARK --set-xmark 0x00004000/0x00004000
-A KUBE-SEP-X3P2623AGDH6CDF3 -p tcp -m comment --comment "default/hostnames:" -m tcp -j DNAT --to-destination 10.244.2.3:9376
-A KUBE-SERVICES -d 10.0.1.175/32 -p tcp -m comment --comment "default/hostnames: cluster IP" -m tcp --dport 80 -j KUBE-SVC-NWV5X2
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostnames:" -m statistic --mode random --probability 0.33332999982 -j KU
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostnames:" -m statistic --mode random --probability 0.50000000000 -j KU
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostnames:" -j KUBE-SEP-57KPRZ3JQVENLNBR
```

For each port of each Service, there should be 1 rule in `KUBE-SERVICES` and one `KUBE-SVC-<hash>` chain. For each Pod endpoint, there should be a small number of rules in that `KUBE-SVC-<hash>` and one `KUBE-SEP-<hash>` chain with a small number of rules in it. The exact rules will vary based on your exact config (including node-ports and load-balancers).

IPVS mode

In "ipvs" mode, you should see something like the following on a Node:

```
ipvsadm -ln
```

```
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
...
TCP  10.0.1.175:80 rr
  -> 10.244.0.5:9376              Masq    1      0          0
  -> 10.244.0.6:9376              Masq    1      0          0
  -> 10.244.0.7:9376              Masq    1      0          0
...
```

For each port of each Service, plus any NodePorts, external IPs, and load-balancer IPs, kube-proxy will create a virtual server. For each Pod endpoint, it will create corresponding real servers. In this example, service hostnames(`10.0.1.175:80`) has 3 endpoints(`10.244.0.5:9376`, `10.244.0.6:9376`, `10.244.0.7:9376`).

Is kube-proxy proxying?

Assuming you do see one the above cases, try again to access your Service by IP from one of your Nodes:

```
curl 10.0.1.175:80
```

```
hostnames-632524106-bbpiw
```

If this still fails, look at the `kube-proxy` logs for specific lines like:

```
Setting endpoints for default/hostnames:default to [10.244.0.5:9376 10.244.0.6:9376 10.244.0.7:9376]
```

If you don't see those, try restarting `kube-proxy` with the `-v` flag set to 4, and then look at the logs again.

Edge case: A Pod fails to reach itself via the Service IP

This might sound unlikely, but it does happen and it is supposed to work.

This can happen when the network is not properly configured for "hairpin" traffic, usually when `kube-proxy` is running in `iptables` mode and Pods are connected with bridge network. The `Kubelet` exposes a `hairpin-mode` flag that allows endpoints of a Service to loadbalance back to themselves if they try to access their own Service VIP. The `hairpin-mode` flag must either be set to `hairpin-veth` or `promiscuous-bridge`.

The common steps to trouble shoot this are as follows:

- Confirm `hairpin-mode` is set to `hairpin-veth` or `promiscuous-bridge`. You should see something like the below. `hairpin-mode` is set to `promiscuous-bridge` in the following example.

```
ps auxw | grep kubelet
```

```
root      3392  1.1  0.8 186804 65208 ?        Sl   00:51  11:11 /usr/local/bin/kubelet --enable-debugging-handlers=true --config=/
```

- Confirm the effective `hairpin-mode`. To do this, you'll have to look at kubelet log. Accessing the logs depends on your Node OS. On some OSes it is a file, such as /var/log/kubelet.log, while other OSes use `journalctl` to access logs. Please be noted that the effective hairpin mode may not match `--hairpin-mode` flag due to compatibility. Check if there is any log lines with key word `hairpin` in kubelet.log. There should be log lines indicating the effective hairpin mode, like something below.

```
I0629 00:51:43.648698    3252 kubelet.go:380] Hairpin mode set to "promiscuous-bridge"
```

- If the effective hairpin mode is `hairpin-veth`, ensure the `Kubelet` has the permission to operate in `/sys` on node. If everything works properly, you should see something like:

```
for intf in /sys/devices/virtual/net/cbr0/brif/*; do cat $intf/hairpin_mode; done

1
1
1
1
```

- If the effective hairpin mode is `promiscuous-bridge`, ensure `Kubelet` has the permission to manipulate linux bridge on node. If `cbr0` bridge is used and configured properly, you should see:

```
ifconfig cbr0 |grep PROMISC

UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1460  Metric:1
```

- Seek help if none of above works out.

Seek help

If you get this far, something very strange is happening. Your Service is running, has EndpointSlices, and your Pods are actually serving. You have DNS working, and `kube-proxy` does not seem to be misbehaving. And yet your Service is not working. Please let us know what is going on, so we can help investigate!

Contact us on Slack or Forum or GitHub.

What's next

Visit the troubleshooting overview document for more information.

Monitoring, Logging, and Debugging

Set up monitoring and logging to troubleshoot a cluster, or debug a containerized application.

Sometimes things go wrong. This guide helps you gather the relevant information and resolve issues. It has four sections:

- Debugging your application - Useful for users who are deploying code into Kubernetes and wondering why it is not working.
- Debugging your cluster - Useful for cluster administrators and operators troubleshooting issues with the Kubernetes cluster itself.
- Logging in Kubernetes - Useful for cluster administrators who want to set up and manage logging in Kubernetes.
- Monitoring in Kubernetes - Useful for cluster administrators who want to enable monitoring in a Kubernetes cluster.

You should also check the known issues for the release you're using.

Getting help

If your problem isn't answered by any of the guides above, there are variety of ways for you to get help from the Kubernetes community.

Questions

The documentation on this site has been structured to provide answers to a wide range of questions. Concepts explain the Kubernetes architecture and how each component works, while Setup provides practical instructions for getting started. Tasks show how to accomplish commonly used tasks, and Tutorials are more comprehensive walkthroughs of real-world, industry-specific, or end-to-end development scenarios. The Reference section provides detailed documentation on the Kubernetes API and command-line interfaces (CLIs), such as `kubectl`.

Help! My question isn't covered! I need help now!

Stack Exchange, Stack Overflow, or Server Fault

If you have questions related to *software development* for your containerized app, you can ask those on Stack Overflow.

If you have Kubernetes questions related to *cluster management* or *configuration*, you can ask those on Server Fault.

There are also several more specific Stack Exchange network sites which might be the right place to ask Kubernetes questions in areas such as DevOps, Software Engineering, or InfoSec.

Someone else from the community may have already asked a similar question or may be able to help with your problem.

The Kubernetes team will also monitor posts tagged Kubernetes. If there aren't any existing questions that help, **please ensure that your question is on-topic on Stack Overflow, Server Fault, or the Stack Exchange Network site you're asking on**, and read through the guidance on how to ask a new question, before asking a new one!

Slack

Many people from the Kubernetes community hang out on Kubernetes Slack in the `#kubernetes-users` channel. Slack requires registration; you can request an invitation, and registration is open to everyone). Feel free to come and ask any and all questions. Once registered, access the Kubernetes organisation in

Slack via your web browser or via Slack's own dedicated app.

Once you are registered, browse the growing list of channels for various subjects of interest. For example, people new to Kubernetes may also want to join the `#kubernetes-novice` channel. As another example, developers should join the `#kubernetes-contributors` channel.

There are also many country specific / local language channels. Feel free to join these channels for localized support and info:

Country	Channels
China	`#cn-users`, `#cn-events`
Finland	`#fi-users`
France	`#fr-users`, `#fr-events`
Germany	`#de-users`, `#de-events`
India	`#in-users`, `#in-events`
Italy	`#it-users`, `#it-events`
Japan	`#jp-users`, `#jp-events`
Korea	`#kr-users`
Netherlands	`#nl-users`
Norway	`#norw-users`
Poland	`#pl-users`
Russia	`#ru-users`
Spain	`#es-users`
Sweden	`#se-users`
Turkey	`#tr-users`, `#tr-events`

Forum

You're welcome to join the official Kubernetes Forum: discuss.kubernetes.io.

Bugs and feature requests

If you have what looks like a bug, or you would like to make a feature request, please use the GitHub issue tracking system.

Before you file an issue, please search existing issues to see if your issue is already covered.

If filing a bug, please include detailed information about how to reproduce the problem, such as:

- Kubernetes version: `kubectl version`
- Cloud provider, OS distro, network configuration, and container runtime version
- Steps to reproduce the problem

Debugging Kubernetes Nodes With Kubectl

This page shows how to debug a node running on the Kubernetes cluster using `kubectl debug` command.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be at or later than version 1.2.

To check the version, enter `kubectl version`.

You need to have permission to create Pods and to assign those new Pods to arbitrary nodes. You also need to be authorized to create Pods that access filesystems from the host.

Debugging a Node using `kubectl debug node`

Use the `kubectl debug node` command to deploy a Pod to a Node that you want to troubleshoot. This command is helpful in scenarios where you can't access your Node by using an SSH connection. When the Pod is created, the Pod opens an interactive shell on the Node. To create an interactive shell on a Node named “mynode”, run:

```
kubectl debug node/mynode -it --image=ubuntu
```

```
Creating debugging pod node-debugger-mynode-pdx84 with container debugger on node mynode.
If you don't see a command prompt, try pressing enter.
root@mynode:/#
```

The debug command helps to gather information and troubleshoot issues. Commands that you might use include `ip`, `ifconfig`, `nc`, `ping`, and `ps` and so on. You can also install other tools, such as `mtr`, `tcpdump`, and `curl`, from the respective package manager.

Note:

The debug commands may differ based on the image the debugging pod is using and these commands might need to be installed.

The debugging Pod can access the root filesystem of the Node, mounted at `/host` in the Pod. If you run your kubelet in a filesystem namespace, the debugging Pod sees the root for that namespace, not for the entire node. For a typical Linux node, you can look at the following paths to find relevant logs:

`/host/var/log/kubelet.log`
: Logs from the `kubelet`, responsible for running containers on the node.

`/host/var/log/kube-proxy.log`
: Logs from `kube-proxy`, which is responsible for directing traffic to Service endpoints.

`/host/var/log/containerd.log`
: Logs from the `containerd` process running on the node.

`/host/var/log/syslog`
: Shows general messages and information regarding the system.

`/host/var/log/kern.log`
: Shows kernel logs.

When creating a debugging session on a Node, keep in mind that:

- `kubectl debug` automatically generates the name of the new pod, based on the name of the node.
- The root filesystem of the Node will be mounted at `/host`.
- Although the container runs in the host IPC, Network, and PID namespaces, the pod isn't privileged. This means that reading some process information might fail because access to that information is restricted to superusers. For example, `chroot /host` will fail. If you need a privileged pod, create it manually or use the `--profile=sysadmin` flag.
- By applying Debugging Profiles, you can set specific properties such as securityContext to a debugging Pod.

Cleaning up

When you finish using the debugging Pod, delete it:

```
kubectl get pods
```

```
NAME                          READY   STATUS       RESTARTS   AGE
node-debugger-mynode-pdx84    0/1     Completed    0          8m1s
```

```
# Change the pod name accordingly
kubectl delete pod node-debugger-mynode-pdx84 --now
```

```
pod "node-debugger-mynode-pdx84" deleted
```

Note:

The `kubectl debug node` command won't work if the Node is down (disconnected from the network, or kubelet dies and won't restart, etc.). Check debugging a down/unreachable node in that case.

Tools for Monitoring Resources

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, services, 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 or full metrics 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 controller, as well as the `kubectl top` utility. These metrics are collected by the lightweight, short-term, in-memory 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 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). 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.

Kubernetes is designed to work with OpenMetrics, which is one of the CNCF Observability and Analysis - Monitoring Projects, built upon and carefully extending Prometheus exposition format in almost 100% backwards-compatible ways.

If you glance over at the CNCF Landscape, you can see a number of monitoring projects that can work with Kubernetes by *scraping* metric data and using that to help you observe your cluster. It is up to you to select the tool or tools that suit your needs. The CNCF landscape for observability and analytics includes a mix of open-source software, paid-for software-as-a-service, and other commercial products.

When you design and implement a full metrics pipeline you can make that monitoring data available back to Kubernetes. For example, a HorizontalPodAutoscaler can use the processed metrics to work out how many Pods to run for a component of your workload.

Integration of a full metrics pipeline into your Kubernetes implementation is outside the scope of Kubernetes documentation because of the very wide scope of possible solutions.

The choice of monitoring platform depends heavily on your needs, budget, and technical resources. Kubernetes does not recommend any specific metrics pipeline; many options are available. Your monitoring system should be capable of handling the OpenMetrics metrics transmission standard and needs to be chosen to best fit into your overall design and deployment of your infrastructure platform.

What's next

Learn about additional debugging tools, including:

- Logging
- Getting into containers via exec
- Connecting to containers via proxies
- Connecting to containers via port forwarding
- Inspect Kubernetes node with crictl

Monitoring in Kubernetes

Monitoring kubernetes system components.

This page provides resources that describe monitoring in Kubernetes. You can learn how to collect system metrics and traces for Kubernetes system components:

- Metrics For Kubernetes System Components
- Traces For Kubernetes System Components

Determine the Reason for Pod Failure

This page shows how to write and read a Container termination message.

Termination messages provide a way for containers to write information about fatal events to a location where it can be easily retrieved and surfaced by tools like dashboards and monitoring software. In most cases, information that you put in a termination message should also be written to the general Kubernetes logs.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Writing and reading a termination message

In this exercise, you create a Pod that runs one container. The manifest for that Pod specifies a command that runs when the container starts:

debug/termination.yaml 

```
apiVersion: v1
kind: Podmetadata:  name: termination-demospec:  containers:  - name: termination-demo-container    image: debian    command: ["/b
```

1. Create a Pod based on the YAML configuration file:

   ```
   kubectl apply -f https://k8s.io/examples/debug/termination.yaml
   ```

   In the YAML file, in the `command` and `args` fields, you can see that the container sleeps for 10 seconds and then writes "Sleep expired" to the `/dev/termination-log` file. After the container writes the "Sleep expired" message, it terminates.

2. Display information about the Pod:

   ```
   kubectl get pod termination-demo
   ```

   Repeat the preceding command until the Pod is no longer running.

3. Display detailed information about the Pod:

```
kubectl get pod termination-demo --output=yaml
```

The output includes the "Sleep expired" message:

```
apiVersion: v1
kind: Pod...    lastState:      terminated:        containerID: ...        exitCode: 0        finishedAt: ...        message:
```

4. Use a Go template to filter the output so that it includes only the termination message:

```
kubectl get pod termination-demo -o go-template="{{range .status.containerStatuses}}{{.lastState.terminated.message}}{{end}}"
```

If you are running a multi-container Pod, you can use a Go template to include the container's name. By doing so, you can discover which of the containers is failing:

```
kubectl get pod multi-container-pod -o go-template='{{range .status.containerStatuses}}{{printf "%s:\n%s\n\n" .name .lastState.terr
```

Customizing the termination message

Kubernetes retrieves termination messages from the termination message file specified in the `terminationMessagePath` field of a Container, which has a default value of `/dev/termination-log`. By customizing this field, you can tell Kubernetes to use a different file. Kubernetes use the contents from the specified file to populate the Container's status message on both success and failure.

The termination message is intended to be brief final status, such as an assertion failure message. The kubelet truncates messages that are longer than 4096 bytes.

The total message length across all containers is limited to 12KiB, divided equally among each container. For example, if there are 12 containers (`initContainers` or `containers`), each has 1024 bytes of available termination message space.

The default termination message path is `/dev/termination-log`. You cannot set the termination message path after a Pod is launched.

In the following example, the container writes termination messages to `/tmp/my-log` for Kubernetes to retrieve:

```
apiVersion: v1
kind: Podmetadata:  name: msg-path-demospec:  containers:  - name: msg-path-demo-container    image: debian    terminationMessageP
```

Moreover, users can set the `terminationMessagePolicy` field of a Container for further customization. This field defaults to "`File`" which means the termination messages are retrieved only from the termination message file. By setting the `terminationMessagePolicy` to "`FallbackToLogsOnError`", you can tell Kubernetes to use the last chunk of container log output if the termination message file is empty and the container exited with an error. The log output is limited to 2048 bytes or 80 lines, whichever is smaller.

What's next

- See the `terminationMessagePath` field in Container.
- See ImagePullBackOff in Images.
- Learn about retrieving logs.
- Learn about Go templates.
- Learn about Pod status and Pod phase.
- Learn about container states.