Best practices for operating containers

21-26 minutes

Last reviewed 2023-02-28 UTC

This article describes a set of best practices for making containers easier to operate. These practices cover a wide range of topics, from security to monitoring and logging. Their aim is to make applications easier to run in Google Kubernetes Engine and in containers in general. Many of the practices discussed here were inspired by the twelve-factor methodology, which is a great resource for building cloud-native applications.

These best practices aren't of equal importance. For example, you might successfully run a production workload without some of them, but others are fundamental. In particular, the importance of the security-related best practices is subjective. Whether you implement them depends on your environment and constraints.

To get the most out of this article, you need some knowledge of Docker and Kubernetes. Some best practices discussed here also apply to Windows containers, but most assume that you are working with Linux containers. For advice about building containers, see Best Practices for Building Containers.

Use the native logging mechanisms of containers

Importance: HIGH

As an integral part of application management, logs contain precious information about the events that happen in the application. Docker and Kubernetes strive to make log management easier.

On a classic server, you probably need to write your logs to a specific file and handle log rotation to avoid filling up the disks. If you have an advanced logging system, you might forward those logs to a remote server in order to centralize them.

Containers offer an easy and standardized way to handle logs because you can write them to *stdout* and *stderr*. Docker captures these log lines and allows you to access them by using the `docker logs` command. As an application developer, you don't need to implement advanced logging mechanisms. Use the native logging mechanisms instead.

The platform operator must provide a system to centralize logs and make them searchable. In GKE, this service is provided by Fluent Bit and Cloud Logging. Depending on your GKE cluster master version, either Fluentd or Fluent Bit are used to collect logs. Starting from GKE 1.17, logs are collected using a Fluentbit-based agent. GKE clusters using versions prior to GKE 1.17 use a Fluentd-based agent. In other Kubernetes distributions, common methods include using an EFK (Elasticsearch, Fluentd, Kibana) stack.

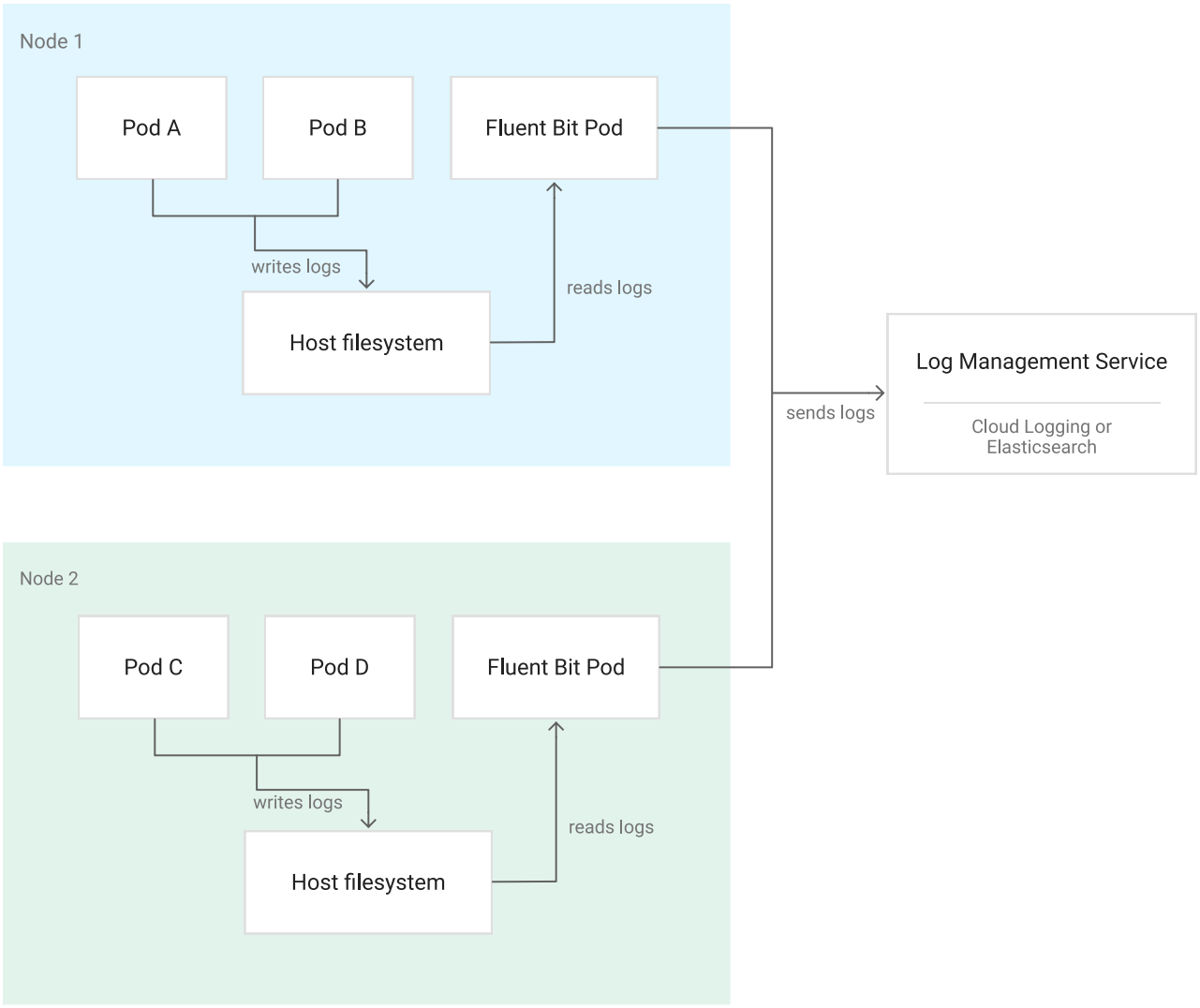

Figure 1. Diagram of a typical log management system in Kubernetes

JSON logs

Most log management systems are actually time-series databases that store time-indexed documents. Those documents can usually be provided in JSON format. In Cloud Logging and in EFK, a single log line is stored as a document, along with some metadata (information about pod, container, node, and so on).

You can take advantage of that behavior by logging directly in JSON format with different fields. You can then search your logs more effectively based on those fields.

For example, consider transforming the following log into JSON format:

```
[2018-01-01 01:01:01] foo - WARNING - foo.bar - There is something
wrong.
```

Here's the transformed log:

```
{
  "date": "2018-01-01 01:01:01",
  "component": "foo",
  "subcomponent": "foo.bar",
  "level": "WARNING",
```

```
  "message": "There is something wrong."
}
```

This transformation allows you to search easily in your logs for all `WARNING`-level logs or all logs from subcomponent `foo.bar`.

If you decide to write JSON-formatted logs, be aware that you must write each event on a single line for it to be correctly parsed. In reality, it looks something like the following:

```
{"date":"2018-01-01
01:01:01","component":"foo","subcomponent":"foo.bar","level":
"WARNING","message": "There is something wrong."}
```

As you can see, the result is far less readable than a normal line of a log. If you decide to use this method, make sure that your teams don't rely heavily on manual log inspection.

Log aggregator sidecar pattern

Some applications (such as Tomcat) can't be easily configured to write logs to *stdout* and *stderr*. Because such applications write to different log files on disk, the best way to handle them in Kubernetes is to use the sidecar pattern for logging. A sidecar is a small container that runs in the same pod as your application. For a more detailed look at sidecars, see the official Kubernetes documentation.

In this solution, you add a logging agent in a sidecar container to your application (in the same pod) and share an emptyDir volume between the two containers, as shown in this YAML example on GitHub. You then configure the application to write its logs to the shared volume and configure the logging agent to read and forward them where needed.

In this pattern, because you aren't using the native Docker and Kubernetes logging mechanisms, you must deal with log rotation. If your logging agent doesn't handle log rotation, another sidecar container in the same pod can handle the rotation.

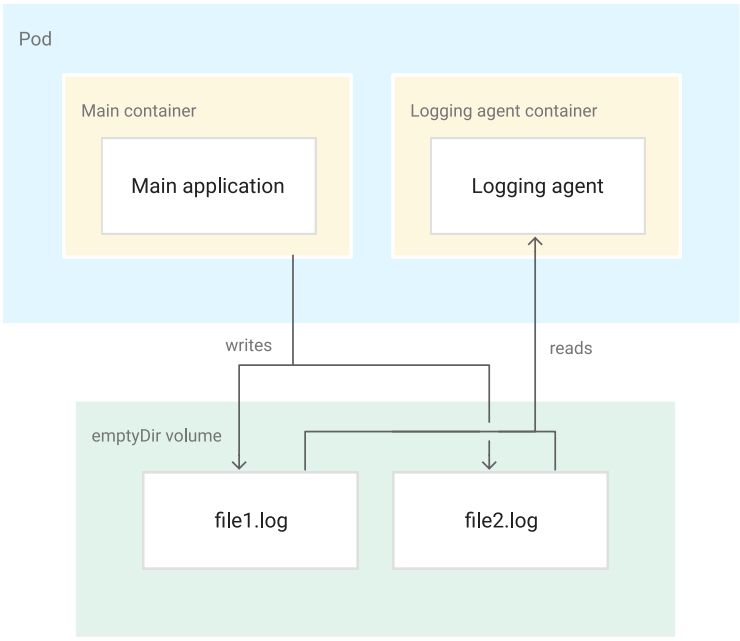

Figure 2. Sidecar pattern for log management

Ensure that your containers are stateless and immutable

Importance: HIGH

If you are trying out containers for the first time, don't treat them as traditional servers. For example, you might be tempted to update your application inside of a running container, or to patch a running container when vulnerabilities arise.

Containers are fundamentally *not* designed to work this way. They are designed to be *stateless* and *immutable*.

Statelessness

Stateless means that any state (persistent data of any kind) is stored outside of a container. This external storage can take several forms, depending on what you need:

- To store files, we recommend using an object store such as Cloud Storage.
- To store information such as user sessions, we recommend using an external, low-latency, key-value store, such as Redis or Memcached.
- If you need block-level storage (for databases, for example), you can use an external disk attached to the container. In the case of GKE, we recommend using persistent disks.

By using these options, you remove the data from the container itself, meaning that the container can be cleanly shut down and destroyed at any time without fear of data loss. If a new container is created to replace the old one, you just connect the new container to the same datastore or bind it to the same disk.

Immutability

Immutable means that a container won't be modified during its life: no updates, no patches, no configuration changes. If you must update the application code or apply a patch, you build a new image and redeploy it. Immutability makes deployments safer and more repeatable. If you need to roll

back, you simply redeploy the old image. This approach allows you to deploy the same container image in every one of your environments, making them as identical as possible.

To use the same container image across different environments, we recommend that you externalize the container configuration (listening port, runtime options, and so on). Containers are usually configured with environment variables or configuration files mounted on a specific path. In Kubernetes, you can use both Secrets and ConfigMaps to inject configurations in containers as environment variables or files.

If you need to update a configuration, deploy a new container (based on the same image), with the updated configuration.

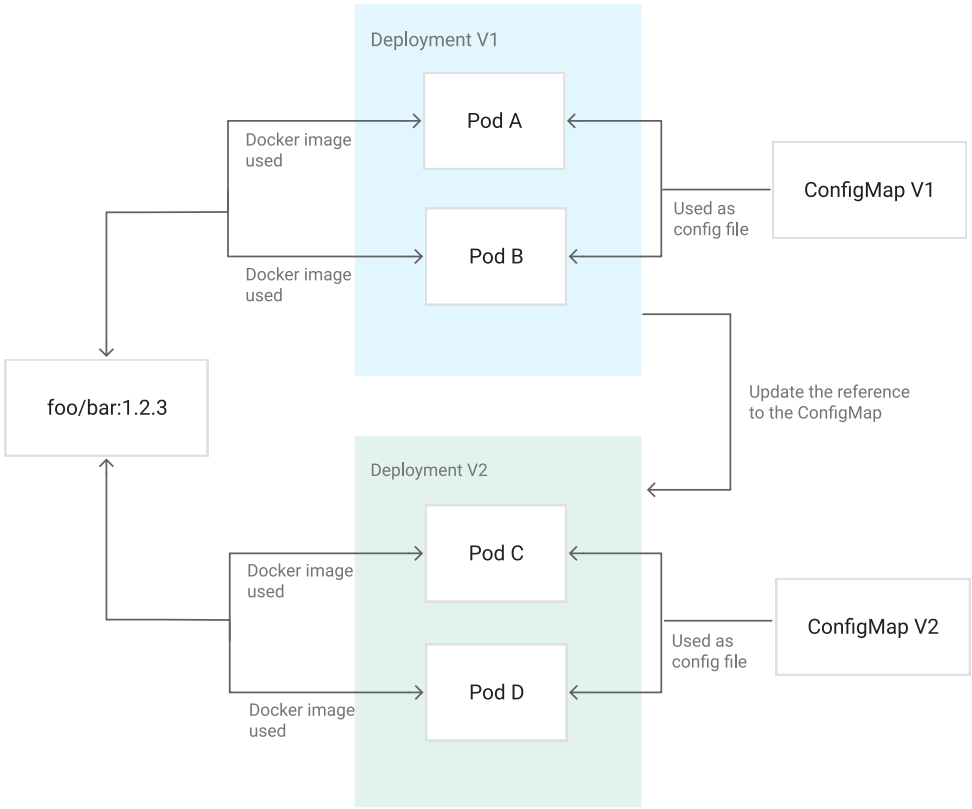

Figure 3. Example of how to update the configuration of a Deployment using a ConfigMap mounted as a configuration file in the pods

The combination of statelessness and immutability is one of the selling points of container-based infrastructures. This combination allows you to automate deployments and increase their frequency and reliability.

Avoid privileged containers

Importance: HIGH

In a virtual machine or a bare-metal server, you avoid running your applications with the root user for a simple reason: if the application is compromised, an attacker would have full access to the server. For the same reason, avoid using privileged containers. A privileged container is a container that has access to all the devices of the host machine, bypassing almost all the security features of containers.

If you believe that you need to use privileged containers, consider the following alternatives:

- Give specific capabilities to the container through the securityContext option of Kubernetes or the `--cap-add` flag of Docker. The Docker documentation lists both the capabilities enabled by default and the ones that you must explicitly enable.
- If your application must modify the host settings in order to run, modify those settings either in a sidecar container or in an init container. Unlike your application, those containers don't need to be exposed to either internal or external traffic, making them more isolated.
- If you need to modify sysctls in Kubernetes, use the dedicated annotation.

You can forbid privileged containers in Kubernetes by using Policy Controller. In the Kubernetes cluster, you cannot create pods that violate the policies configured using Policy Controller.

Make your application easy to monitor

Importance: HIGH

Like logging, monitoring is an integral part of application management. In many ways, monitoring containerized applications follows the same principles that apply to monitoring of non-containerized applications. However, because containerized infrastructures tend to be highly dynamic, with containers that are frequently created or deleted, you cannot afford to reconfigure your monitoring system each time this happens.

You can distinguish two main classes of monitoring: *black-box monitoring* and *white-box monitoring*. Black-box monitoring refers to examining your application from outside, as if you were an end user. Black-box monitoring is useful if the final service you want to offer is available and working. Because it is external to the infrastructure, black-box monitoring doesn't differ between traditional and containerized infrastructures.

White-box monitoring refers to examining your application with some kind of privileged access, and to gather metrics on its behavior that an end user cannot view. Because white-box monitoring must examine the deepest layers of your infrastructure, it differs significantly for traditional and containerized infrastructures.

A popular option in the Kubernetes community for white-box monitoring is Prometheus , a system that can automatically discover the pods it has to monitor. Prometheus scrapes the pods for metrics; it expects a specific format for them. Google Cloud offers Google Cloud Managed Service for Prometheus, a service that lets you globally monitor and alert on your workloads without having to manually manage and operate Prometheus at scale. By default, Google Cloud Managed Service for Prometheus is configured to collect system metrics from GKE clusters and send them to Cloud Monitoring. For more information, see Observability for GKE.

To benefit from either Prometheus or Monitoring, your applications need to expose metrics. The following two methods show how to do that.

Metrics HTTP endpoint

The metrics HTTP endpoint works in a similar way to the endpoints mentioned later in expose the health of your application. It exposes the internal metrics of the application, usually on a `/metrics` URI. A response looks like this:

```
http_requests_total{method="post",code="200"} 1027
http_requests_total{method="post",code="400"}    3
http_requests_total{method="get",code="200"} 10892
http_requests_total{method="get",code="400"}    97
```

In this example, `http_requests_total` is the metric, `method` and `code` are labels, and the right-most number is the value of this metric for those labels. Here, since its startup, the application has responded to an HTTP GET request 97 times with a 400 error code.

Generating this HTTP endpoint is made easy by the Prometheus client libraries that exist for many languages. OpenCensus can also export metrics using this format (among many other features). Don't expose this endpoint to the public internet.

The official Prometheus documentation goes into deeper detail on this topic. You can also read Chapter 6 of *Site Reliability Engineering* to learn more about white-box (and black-box) monitoring.

Sidecar pattern for monitoring

Not all applications can be instrumented with a `/metrics` HTTP endpoint. To keep standardized monitoring, we recommend using the sidecar pattern to export metrics in the right format.

The Log aggregator sidecar pattern section explained how to use a sidecar container to manage application logs. You can use the same pattern for monitoring: the sidecar container hosts a monitoring agent that translates the metrics as they are exposed by the application to a format and protocol that the global monitoring system understands.

Consider a concrete example: Java applications and Java Management Extensions (JMX). Many Java applications expose metrics using JMX. Rather than rewriting an application to expose metrics in the Prometheus format, you can take advantage of the jmx_exporter. The jmx_exporter gathers metrics from an application through JMX and exposes them through a `/metrics` endpoint that Prometheus can read. This approach also has the advantage of limiting the exposure of the JMX endpoint, which can be used to modify application settings.

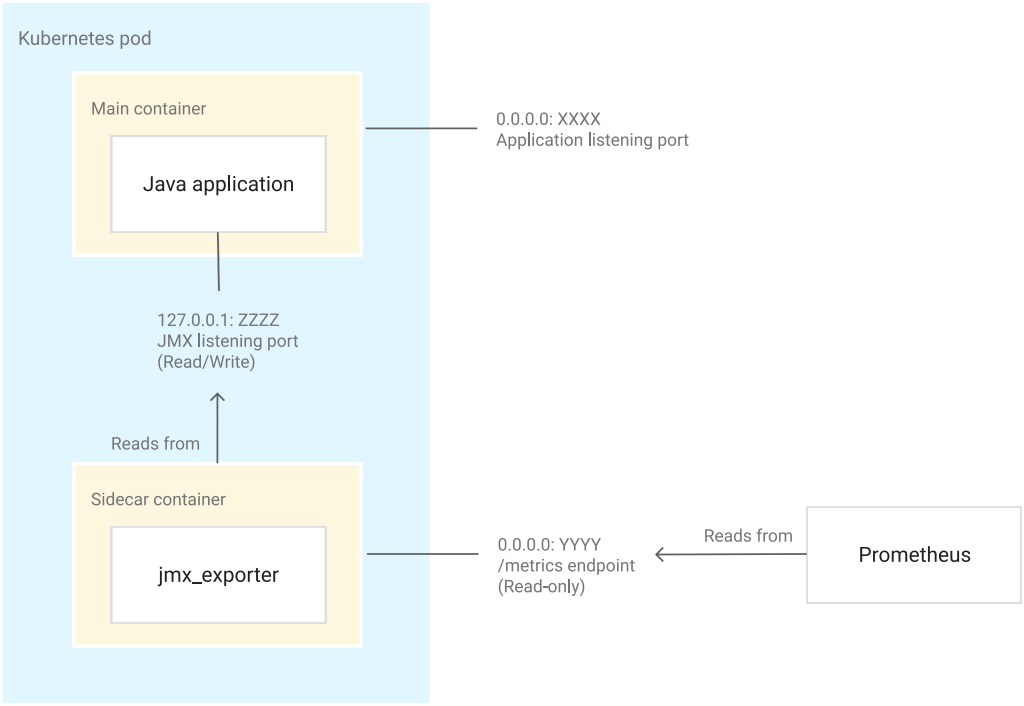

Figure 4. Sidecar pattern for monitoring

Expose the health of your application

Importance: MEDIUM

To facilitate its management in production, an application must communicate its state to the overall system: is the application running? Is it healthy? Is it ready to receive traffic? How is it behaving?

Kubernetes has two types of health checks: liveness probes and readiness probes. Each has a specific use, as described in this section. You can implement both in a number of ways (including running a command inside the container or checking a TCP port), but the preferred method is to use the HTTP endpoints described in this best practice. For more information on this topic, see the Kubernetes documentation.

Liveness probe

The recommended way to implement the liveness probe is for your application to expose a `/healthz` HTTP endpoint. Upon receiving a request on this endpoint, the application should send a "200 OK" response if it is considered healthy. In Kubernetes, healthy means that the container doesn't need to be killed or restarted. What constitutes healthy varies from one application to another, but it usually means the following:

- The application is running.
- Its main dependencies are met (for example, it can access its database).

Readiness probe

The recommended way to implement the readiness probe is for your application to expose a `/ready` HTTP endpoint. When it receives a request on this endpoint, the application should send a "200 OK" response if it is ready to receive traffic. Ready to receive traffic means the following:

- The application is healthy.
- Any potential initialization steps are completed.
- Any valid request sent to the application doesn't result in an error.

Kubernetes uses the readiness probe to orchestrate your application's deployment. If you update a Deployment, Kubernetes will do a rolling update of the pods belonging to that Deployment. The default update policy is to update one pod at a time: Kubernetes waits for the new pod to be ready (as indicated by the readiness probe) before updating the next one.

Avoid running as root

Importance: MEDIUM

Containers provide isolation: with default settings, a process inside a Docker container cannot access information from the host machine or from the other collocated containers. However, because containers share the kernel of the host machine, the isolation isn't as complete as it is with virtual machines, as this blog post explains. An attacker could find yet unknown vulnerabilities (either in Docker or the Linux kernel itself) that would allow the attacker to escape from a container. If the attacker does find a vulnerability and your process is running as root inside the container, they would get root access to the host machine.

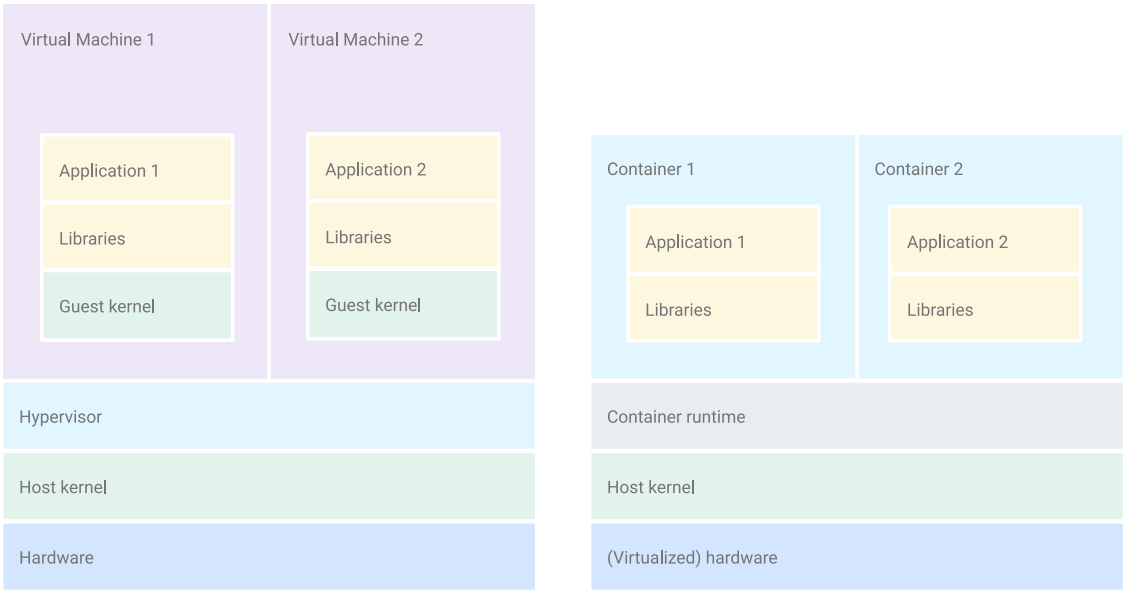

Figure 5. On the left, virtual machines use virtualized hardware. On the right, applications in containers use the host kernel.

To avoid this possibility, it is a best practice to not run processes as root inside containers. You can enforce this behavior in Kubernetes by using Policy Controller. When creating a pod in Kubernetes, use the runAsUser option to specify the Linux user that is running the process. This approach overrides the `USER` instruction of the Dockerfile.

In reality, there are challenges. Many well-known software packages have their main process run as root. If you want to avoid running as root, design your container so that it can be run with an unknown, non-privileged user. This practice often means that you have to tweak permissions on various folders. In a container, if you are following the single application per container best practice and you are running a single application with a single user—preferably not root, you can grant all users write permissions to the folders and files that need to be written in, and make all the other folders and files only writable by root.

A simple way to check if your container complies with this best practice is to run it locally with a random user and test if it works properly. Replace `[YOUR_CONTAINER]` with your container name.

```
docker run --user $((RANDOM+1)) [YOUR_CONTAINER]
```

If your container needs an external volume, you can configure the fsGroup Kubernetes option to give ownership of this volume to a specific Linux group. This configuration solves the problem of the ownership of external files.

If your process is run by a non-privileged user, it will not be able to bind to ports below 1024. This is usually not an issue, because you can configure Kubernetes Services to route traffic from one port to another. For example, you can configure an HTTP server to bind to port 8080 and redirect the traffic from port 80 with a Kubernetes Service.

Carefully choose the image version

Importance: MEDIUM

When you use a Docker image, whether as a base image in a Dockerfile or as an image deployed in Kubernetes, you must choose the tag of the image you are using.

Most public and private images follow a tagging system similar to the one described in Best Practices for Building Containers. If the image uses a system close to Semantic Versioning, you must consider some tagging specifics.

Most importantly, the "latest" tag can be moved frequently from image to image. The consequence is that you cannot rely on this tag for predictable or reproducible builds. For example, take the following Dockerfile:

```
FROM debian:latest

RUN apt-get -y update && \
    apt-get -y install nginx
```

If you build an image from this Dockerfile twice, at different times, you can end up with two different versions of Debian and NGINX. Instead, consider this revised version:

```
FROM debian:11.6

RUN apt-get -y update && \
    apt-get -y install nginx
```

By using a more precise tag, you ensure that the resulting image will always be based on a specific minor version of Debian. Because a specific Debian version also ships a specific NGINX version, you have much more control over the image that is being built.

This outcome is not only true at build time, but also at run time. If you reference the "latest" tag in a Kubernetes manifest, you have no guarantee of the version that Kubernetes will use. Different nodes of your cluster might pull the same "latest" tag at different moments. If the tag was updated at one point between the pulls, you can end up with different nodes running different images (that were all tagged "latest" at one point).

Ideally, you should always use an immutable tag in your FROM line. This tag allows you to have reproducible builds. However, there are some security tradeoffs: the more you pin the version you want to use, the less automated the security patches will be in your images. If the image you are using is using proper semantic versioning, the patch version (that is, the "Z" in "X.Y.Z") should not have backward-incompatible changes: you can use the "X.Y" tag and get bug fixes automatically.

Imagine a piece of software called "SuperSoft." Suppose the security process for SuperSoft is to fix vulnerabilities through a new *patch* version. You want to customize SuperSoft, and you have written the following Dockerfile:

```
FROM supersoft:1.2.3

RUN a-command
```

After a while, the vendor discovers a vulnerability and releases version 1.2.4 of SuperSoft to address the problem. In this case, it is up to you to stay informed on SuperSoft's patches and to update your Dockerfile accordingly. If instead you use `FROM supersoft:1.2` in your Dockerfile, the new version is pulled automatically.

In the end, you have to carefully examine the tagging system of each external image you are using, decide how much you trust the people who build those images, and decide what tag you will use.

What's next

- Learn about Best practices for building containers.
- Build your first containers with Cloud Build.
- Spin up your first GKE cluster.
- Speed up your Cloud Build builds.
- Docker Inc. has its own set of best practices. Some of them are covered this document, but others are not.

Explore reference architectures, diagrams, and best practices about Google Cloud. Take a look at our Cloud Architecture Center.