cloud.google.com /architecture/best-practices-for-building-containers

Best practices for building containers

24-31 minutes

Last reviewed 2023-02-28 UTC

This article describes a set of best practices for building containers. These practices cover a wide range of goals, from shortening the build time, to creating smaller and more resilient images, with the aim of making containers easier to build (for example, with Cloud Build), and easier to run in Google Kubernetes Engine (GKE).

These best practices are not 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. Advice about running and operating containers is available in Best practices for operating containers.

Package a single app per container

Importance: HIGH

When you start working with containers, it's a common mistake to treat them as virtual machines (VMs) that can run many different things simultaneously. A container can work this way, but doing so reduces most of the advantages of the container model. For example, take a classic Apache/MySQL/PHP stack: you might be tempted to run all the components in a single container. However, the best practice is to use two or three different containers: one for Apache, one for MySQL, and potentially one for PHP if you are running PHP-FPM.

Because a container is designed to have the same lifecycle as the app it hosts, each of your containers should contain only one app. When a container starts, so should the app, and when the app stops, so should the container. The following diagram shows this best practice.

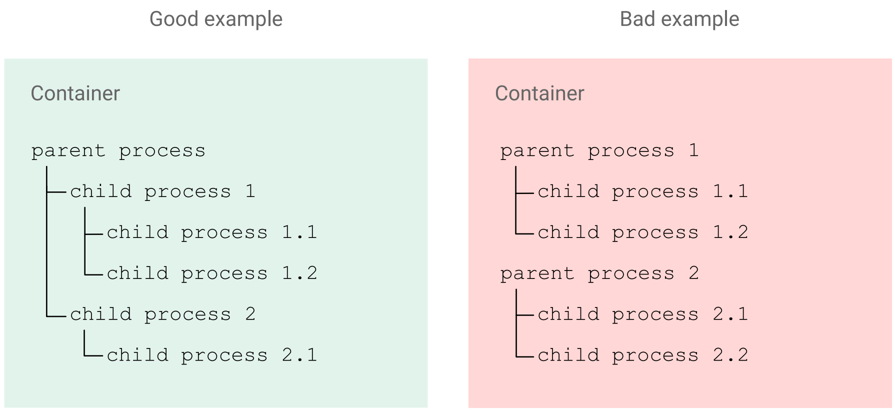

Figure 1. The container on the left follows the best practice. The container on the right doesn't.

If you have multiple apps in a container, they might have different lifecycles, or be in different states. For instance, you might end up with a container that is running, but with one of its core components crashed or unresponsive. Without an additional health check, the overall container management system (Docker or Kubernetes) cannot tell whether the container is healthy. In the case of Kubernetes, it means that if a core component is unresponsive, Kubernetes will not restart your container automatically.

You might see the following actions in public images, but do **not** follow their example:

- Using a process management system such as supervisor to manage one or several apps in the container.
- Using a bash script as an entrypoint in the container, and making it spawn several apps as background jobs. For the proper use of bash scripts in containers, see Properly handle PID 1, signal handling, and zombie processes.

Properly handle PID 1, signal handling, and zombie processes

Importance: HIGH

Linux signals are the main way to control the lifecycle of processes inside a container. In line with the previous best practice, in order to tightly link the lifecycle of your app to the container it's in, ensure that your app properly handles Linux signals. The most important Linux signal is SIGTERM because it terminates a process. Your app might also receive a SIGKILL signal, which is used to kill the process non-gracefully, or a SIGINT signal, which is sent when you type `Ctrl+C` and is usually treated like SIGTERM.

Process identifiers (PIDs) are unique identifiers that the Linux kernel gives to each process. PIDs are namespaced, meaning that a container has its own set of PIDs that are mapped to PIDs on the host system. The first process launched when starting a Linux kernel has the PID 1. For a normal operating system, this process is the init system, for example, systemd or SysV. Similarly, the first process launched in a container gets PID 1. Docker and Kubernetes use signals to communicate with the processes inside containers, most notably to terminate them. Both Docker and Kubernetes can only send signals to the process that has PID 1 inside a container.

In the context of containers, PIDs and Linux signals create two problems to consider.

Problem 1: How the Linux kernel handles signals

The Linux kernel handles signals differently for the process that has PID 1 than it does for other processes. Signal handlers aren't automatically registered for this process, meaning that signals such as SIGTERM or SIGINT will have no effect by default. By default, you must kill processes by using SIGKILL, preventing any graceful shutdown. Depending on your app, using SIGKILL can result in user-facing errors, interrupted writes (for data stores), or unwanted alerts in your monitoring system.

Problem 2: How classic init systems handle orphaned processes

Classic init systems such as systemd are also used to remove (*reap*) orphaned, zombie processes. Orphaned processes—processes whose parents have died—are reattached to the process that has PID 1, which should reap them when they die. A normal init system does that. But in a container, this responsibility falls on whatever process has PID 1. If that process doesn't properly handle the reaping, you risk running out of memory or some other resources.

These problems have several common solutions, outlined in the following sections.

Solution 1: Run as PID 1 and register signal handlers

This solution addresses only the first problem. It is valid if your app spawns child processes in a controlled way (which is often the case), avoiding the second problem.

The easiest way to implement this solution is to launch your process with the `CMD` and/or `ENTRYPOINT` instructions in your Dockerfile. For example, in the following Dockerfile, `nginx` is the first and only process to be launched.

```
FROM debian:11

RUN apt

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

EXPOSE

80

CMD

[ "nginx", "-g", "daemon off;" ]
```

Sometimes, you might need to prepare the environment in your container for your process to run properly. In this case, the best practice is to have the container launch a shell script when starting. This shell script is tasked with preparing the environment and launching the main process. However, if you take this approach, the shell script has PID 1, not your process, which is why you must use the built-in `exec` command to launch the process from the shell script. The `exec` command replaces the script with the program you want. Your process then inherits PID 1.

Solution 2: Enable process namespace sharing in Kubernetes

When you enable process namespace sharing for a Pod, Kubernetes uses a single process namespace for all the containers in that Pod. The Kubernetes Pod infrastructure container becomes PID 1 and automatically reaps orphaned processes.

Solution 3: Use a specialized init system

As you would in a more classic Linux environment, you can also use an init system to deal with those problems. However, normal init systems such as systemd or SysV are too complex and large for just this purpose, which is why we recommend that you use an init system such as `tini`, which is created especially for containers.

If you use a specialized init system, the init process has PID 1 and does the following:

- Registers the correct signal handlers.
- Makes sure that signals work for your app.
- Reaps any eventual zombie processes.

You can use this solution in Docker itself by using the `--init` option of the `docker run` command. To use this solution in Kubernetes, you must install the init system in your container image and use it as the entrypoint for your container.

Optimize for the Docker build cache

Importance: HIGH

The Docker build cache can accelerate the building of container images. Images are built layer by layer, and in a Dockerfile, each instruction creates a layer in the resulting image. During a build, when possible, Docker reuses a layer from a previous build and skips a potentially costly step. Docker can use its build cache only if all previous build steps used it. While this behavior is usually a good thing that makes builds go faster, you need to consider a few cases.

For example, to fully benefit from the Docker build cache, you must position the build steps that change often at the bottom of the Dockerfile. If you put them at the top, Docker cannot use its build cache for the other build steps that are changing less often. Because a new Docker image is usually built for each new version of your source code, add the source code to the image as late as possible in the Dockerfile. In the following diagram, you can see that if you are changing `STEP 1`, Docker can reuse only the layers from the `FROM debian:11` step. If you change `STEP 3`, however, Docker can reuse the layers for `STEP 1` *and* `STEP 2`.

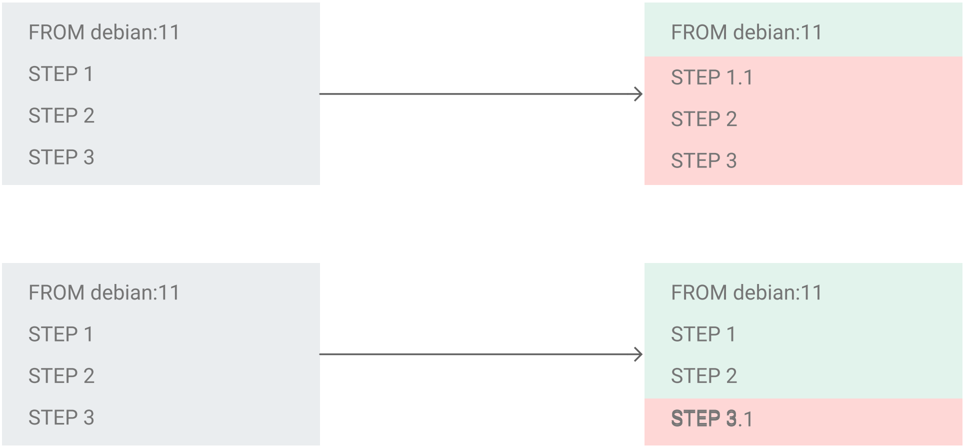

Figure 2. Examples of how to use the Docker build cache. In green, the layers that you can reuse. In red, the layers that have to be recreated.

Reusing layers has another consequence: if a build step relies on any kind of cache stored on the local file system, this cache must be generated in the same build step. If this cache isn't being generated, your build step might be executed with an out-of-date cache coming from a previous build. You see this behavior most commonly with package managers such as apt or yum: you must update your repositories in the same RUN command as your package installation.

If you change the second RUN step in the following Dockerfile, the `apt-get update` command isn't rerun, leaving you with an out-of-date apt cache.

```
FROM debian:11

RUN apt

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

Instead, merge the two commands in a single RUN step:

```
FROM debian:11

RUN apt

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

Importance: MEDIUM

To protect your apps from attackers, try to reduce the attack surface of your app by removing any unnecessary tools. For example, remove utilities like netcat, which you can use to create a reverse shell inside your system. If netcat isn't in the container, the attacker has to find another way.

This best practice is true for any workload, even if it isn't containerized. The difference is that this best practice is optimized for containers, rather than classic VMs or bare-metal servers.

When you remove unnecessary tools, this can also help you improve your debugging processes. For example, if you implement this best practice fully enough, exhaustive logs, tracing, and profiling systems can become almost mandatory. In effect, you can no longer rely on local debugging tools because they are often highly privileged.

File system content

The first part of this best practice deals with the content of the container image. Keep as few things as possible in your image. If you can compile your app into a single statically linked binary, adding this binary to the scratch image allows you to get a final image that contains *only* your app and nothing else. By reducing the number of tools packaged in your image, you reduce what a potential attacker can do in your container. For more information, see Build the smallest image possible.

File system security

Having no tools in your image isn't sufficient: you must prevent potential attackers from installing their own tools. You can combine two methods here:

- Avoid running as root inside the container: this method offers a first layer of security and could prevent, for example, attackers from modifying root-owned files using a package manager embedded in your image (such as `apt-get` or `apk`). For this method to be useful, you must disable or uninstall the `sudo` command. This topic is more broadly covered in Avoid running as root.

- Launch the container in read-only mode, which you do by using the `--read-only` flag from the `docker run` command or by using the `readOnlyRootFilesystem` option in Kubernetes.

Build the smallest image possible

Importance: MEDIUM

Building a smaller image offers advantages such as faster upload and download times, which is especially important for the cold start time of a pod in Kubernetes: the smaller the image, the faster the node can download it. However, building a small image can be difficult because you might inadvertently include build dependencies or unoptimized layers in your final image.

Use the smallest base image possible

The base image is the one referenced in the FROM instruction in your Dockerfile. Every other instruction in the Dockerfile builds on top of this image. The smaller the base image, the smaller the resulting image is, and the more quickly it can be downloaded. For example, the alpine:3.17 image is 23 MB smaller than the ubuntu:22.04 image.

You can even use the scratch base image, which is an empty image on which you can build your own runtime environment. If your app is a statically linked binary, you can use the scratch base image as follows:

```
FROM scratch
COPY mybinary /mybinary
CMD [ "/mybinary" ]
```

The following Kubernetes Best Practices video covers additional strategies for building small containers while reducing your exposure to security vulnerabilities.

Reduce the amount of clutter in your image

To reduce the size of your image, install only what is strictly needed inside it. It might be tempting to install extra packages, and then remove them at a later step. However, this approach isn't sufficient. Because each instruction of the Dockerfile creates a layer, removing data from the image in a later step than the step that created it doesn't reduce the size of the overall image (the data is still there, just hidden in a deeper layer). Consider this example:

Bad Dockerfile	Good Dockerfile

See more code actions.
`FROM debian:11`
 	
`RUN apt-get update && \`
`    apt-get install -y \`
`    [buildpackage]`
	See more code actions.
	`FROM debian:11`
	
	`RUN apt-get update && \`
	`    apt-get install -y \`
	`    [buildpackage] && \`

Bad Dockerfile	Good Dockerfile

`RUN [build my app]`
`RUN apt-get autoremove --purge \`
`    -y [buildpackage] && \`
`    apt-get -y clean && \`
`    rm -rf /var/lib/apt/lists/*`
	`    [build my app] && \`
	`    apt-get autoremove --purge \`
	`    -y [buildpackage] && \`
	`    apt-get -y clean && \`
	`    rm -rf /var/lib/apt/lists/*`

In the bad version of the Dockerfile, the `[buildpackage]` and files in `/var/lib/apt/lists/*` still exist in the layer corresponding to the first RUN. This layer is part of the image and must be uploaded and downloaded with the rest, even if the data it contains isn't accessible in the resulting image.

In the good version of the Dockerfile, everything is done in a single layer that contains only your built app. The `[buildpackage]` and files in `/var/lib/apt/lists/*` don't exist anywhere in the resulting image, not even hidden in a deeper layer.

For more information on image layers, see Optimize for the Docker build cache.

Another great way to reduce the amount of clutter in your image is to use *multi-staged builds* (introduced in Docker 17.05). Multi-stage builds allow you to build your app in a first "build" container and use the result in another container, while using the same Dockerfile.

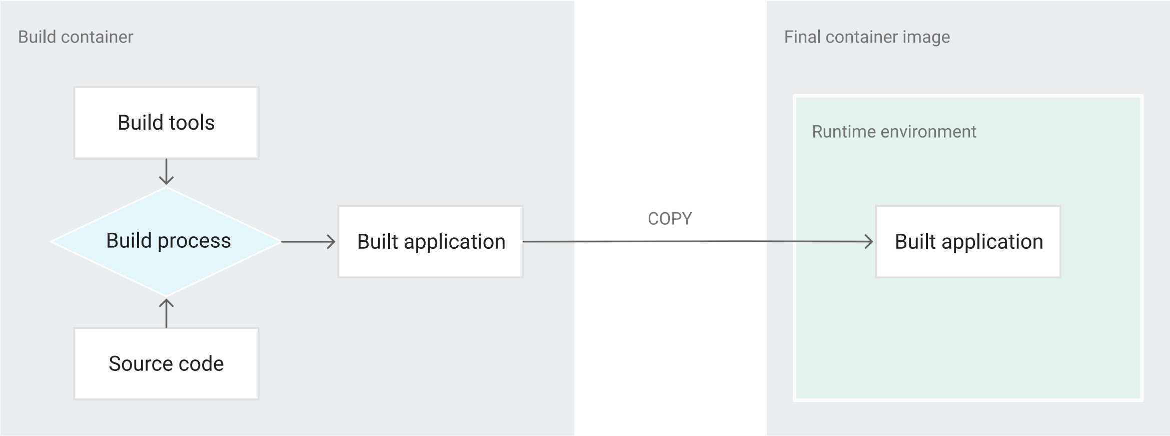

Figure 3. The Docker multi-stage build process.

In the following Dockerfile, the `hello` binary is built in a first container and injected in a second one. Because the second container is based on scratch, the resulting image contains only the `hello` binary and not the source file and object files needed during the build. The binary must be statically linked in order to work without the need for any outside library in the scratch image.

```
FROM golang:1.20 as builder

WORKDIR

/tmp/go
```

```
COPY hello.go ./
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -a -ldflags '-s' -o
hello

FROM scratch
CMD

[ "/hello" ]
COPY --from=builder /tmp/go/hello /hello
```

Try to create images with common layers

If you must download a Docker image, Docker first checks whether you already have some of the layers that are in the image. If you do have those layers, they aren't downloaded. This situation can occur if you previously downloaded another image that has the same base as the image you are currently downloading. The result is that the amount of data downloaded is much less for the second image.

At an organizational level, you can take advantage of this reduction by providing your developers with a set of common, standard, base images. Your systems must download each base image only once. After the initial download, only the layers that make each image unique are needed. In effect, the more your images have in common, the faster they are to download.

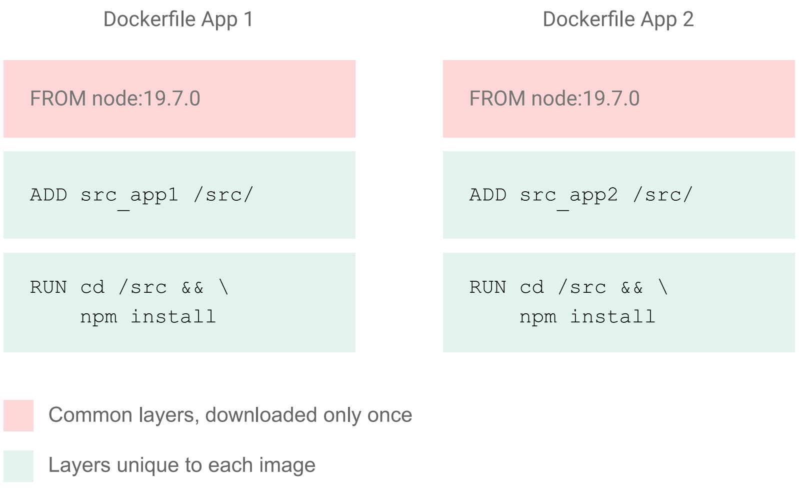
Dockerfile App 1
Dockerfile App 2
FROM node:19.7.0
FROM node:19.7.0
ADD src_app1 /src/
ADD src_app2 /src/
RUN cd /src && \ npm install
RUN cd /src && \ npm install
🟥 Common layers, downloaded only once
🟩 Layers unique to each image

Figure 4. Creating images with common layers.

Scan images for vulnerabilities

Importance: MEDIUM

Software vulnerabilities are a well-understood problem in the world of bare-metal servers and virtual machines. A common way to address these vulnerabilities is to use a centralized inventory system that lists the packages installed on each server. Subscribe to the vulnerability feeds of the upstream operating systems to be informed when a vulnerability affects your servers, and patch them accordingly.

However, because containers are supposed to be immutable (see statelessness and immutability of containers for more details), don't patch them in place in case of a vulnerability. The best practice is to rebuild the image, patches included, and redeploy it. Containers have a much shorter lifecycle and a less well-defined identity than servers. So using a similar centralized inventory system is a poor way to detect vulnerabilities in containers.

To help you address this problem, Artifact Analysis can scan your images for security vulnerabilities in publicly monitored packages. The following options are available:

Automatic vulnerability scanning

> When enabled, this feature identifies package vulnerabilities in your container images. Images are scanned when they are uploaded to Artifact Registry or Container Registry and the data is continuously monitored to find new vulnerabilities for up to 30 days after pushing the image. You can act on the information reported by this feature in several ways:
>
> - Create a cron-like job that lists vulnerabilities and triggers the process to fix them, where a fix exists.
> - As soon as a vulnerability is detected, use the Pub/Sub integration to trigger the patching process that your organization uses.

On-Demand Scanning API

> When enabled, you can manually scan local images or images stored in Artifact Registry or Container Registry. This feature helps you to detect and address vulnerabilities early in your build pipeline. For example, you can use Cloud Build to scan an image after it is built and then block upload to Artifact Registry if the scan detects vulnerabilities at a specified severity level. If you also enabled automatic vulnerability scanning, Artifact Registry also scans images you upload to the registry.

We recommend automating the patching process and relying on the existing continuous integration pipeline initially used to build the image. If you are confident in your continuous deployment pipeline, you might also want to automatically deploy the fixed image when ready. However, most people prefer a manual verification step before deployment. The following process achieves that:

1. Store your images in Artifact Registry and enable vulnerability scanning.
2. Configure a job that regularly fetches new vulnerabilities from Artifact Registry and triggers a rebuild of the images if needed.
3. When the new images are built, have your continuous deployment system deploy them to a staging environment.
4. Manually check the staging environment for problems.
5. If no problems are found, manually trigger the deployment to production.

Properly tag your images

Importance: MEDIUM

Docker images are generally identified by two components: their name and their tag. For example, for the `google/cloud-sdk:419.0.0` image, `google/cloud-sdk` is the name and `419.0.0` is the tag. The tag `latest` is used by default if you don't provide one in your Docker commands. The name and tag pair is unique at any given time. However, you can reassign a tag to a different image as needed.

When you build an image, it's up to you to tag it properly. Follow a coherent and consistent tagging policy. Document your tagging policy so that image users can easily understand it.

Container images are a way of packaging and releasing a piece of software. Tagging the image lets users identify a specific version of your software in order to download it. For this reason, tightly link the tagging system on container images to the release policy of your software.

Tagging using semantic versioning

A common way of releasing software is to "tag" (as in the `git tag` command) a particular version of the source code with a version number. The Semantic Versioning Specification provides a clean way of handling version numbers. In this system, your software has a three-part version number: `X.Y.Z`, where:

- `X` is the major version, incremented only for incompatible API changes.
- `Y` is the minor version, incremented for new features.
- `Z` is the patch version, incremented for bug fixes.

Every increment in the minor or patch version number must be for a backward-compatible change.

If you use this system, or a similar one, tag your images according to the following policy:

- The `latest` tag always refers to the most recent (possibly stable) image. This tag is moved as soon as a new image is created.
- The `X.Y.Z` tag refers to a specific version of your software. Don't move it to another image.
- The `X.Y` tag refers to the latest patch release of the `X.Y` minor branch of your software. It's moved when a new patch version is released.
- The `X` tag refers to the latest patch release of the latest minor release of the `X` major branch. It's moved when either a new patch version or a new minor version is released.

Using this policy offers users the flexibility to choose which version of your software they want to use. They can pick a specific X.Y.Z version and be guaranteed that the image will never change, or they can get updates automatically by choosing a less specific tag.

Tagging using the Git commit hash

If you have an advanced continuous delivery system and you release your software often, you probably don't use version numbers as described in the Semantic Versioning Specification. In this case, a common way of handling version numbers is to use the Git commit SHA-1 hash (or a short version of it) as the version number. By design, the Git commit hash is immutable and references a specific version of your software.

You can use this commit hash as a version number for your software, but also as a tag for the Docker image built from this specific version of your software. Doing so makes Docker images traceable: because in this case the image tag is immutable, you instantly know which specific version of your software is running inside a given container. In your continuous delivery pipeline, automate the update of the version number used for your deployments.

Carefully consider whether to use a public image

Importance: N/A

One of the great advantages of Docker is the sheer number of publicly available images, for all kinds of software. These images allow you to get started quickly. However, when you are designing a container strategy for your organization, you might have constraints that publicly provided images aren't able to meet. Here are a few examples of constraints that might render the use of public images impossible:

- You want to control exactly what is inside your images.
- You don't want to depend on an external repository.
- You want to strictly control vulnerabilities in your production environment.
- You want the same base operating system in every image.

The response to all those constraints is the same, and is unfortunately costly: you must build your own images. Building your own images is fine for a limited number of images, but this number has a tendency to grow quickly. To have any chance of managing such a system at scale, think about using the following:

- An automated way to build images, in a reliable way, even for images that are built rarely. Build triggers in Cloud Build are a good way to achieve that.
- A standardized base image. Google provides some base images that you can use.
- An automated way to propagate updates to the base image to "child" images.
- A way to address vulnerabilities in your images. For more information, see Scan images for vulnerabilities.
- A way to enforce your internal standards on images created by the different teams in your organization.

Several tools are available to help you enforce policies on the images that you build and deploy:

- container-diff can analyze the content of images and even compare two images between them.
- container-structure-test can test whether the content of an image complies with a set of rules that you define.
- Grafeas is an artifact metadata API, where you store metadata about your images to later check whether those images comply with your policies.
- Kubernetes has admission controllers that you can use to check a number of prerequisites before deploying a workload in Kubernetes.

You might also want to adopt a hybrid system: using a public image such as Debian or Alpine as the base image and building everything on top of that image. Or you might want to use public images for some of your noncritical images, and build your own images for other cases. Those questions have no right or wrong answers, but you have to address them.

A note about licenses

Before you include third-party libraries and packages in your Docker image, ensure that the respective licenses allow you to do so. Third-party licenses might also impose restrictions on redistribution, which apply when you publish a Docker image to a public registry.

What's next

- Learn about Best practices for operating containers.
- Build your first containers with Cloud Build.
- Spin up your first GKE cluster.
- Speed up your Cloud Build builds.
- Docker has its own set of best practices, some of which are covered in this document.

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