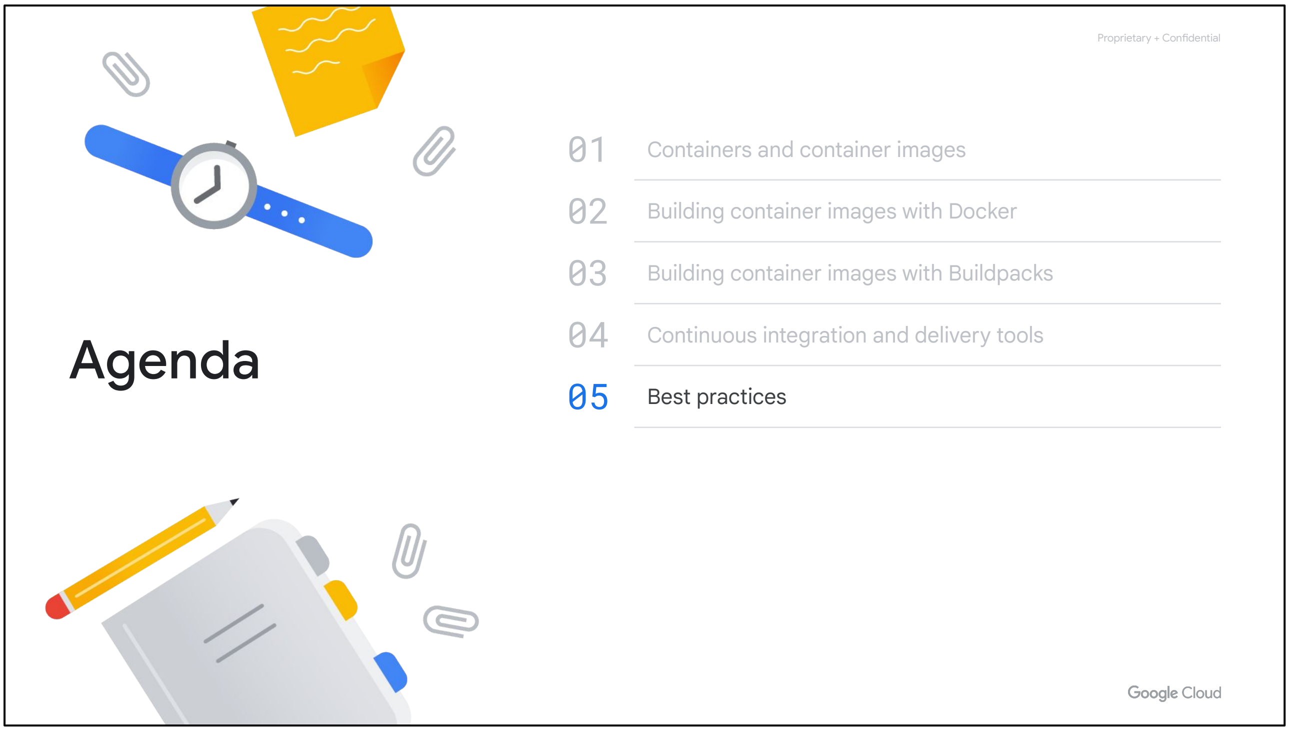


Proprietary + Confidential

Agenda

01 Containers and container images

02 Building container images with Docker

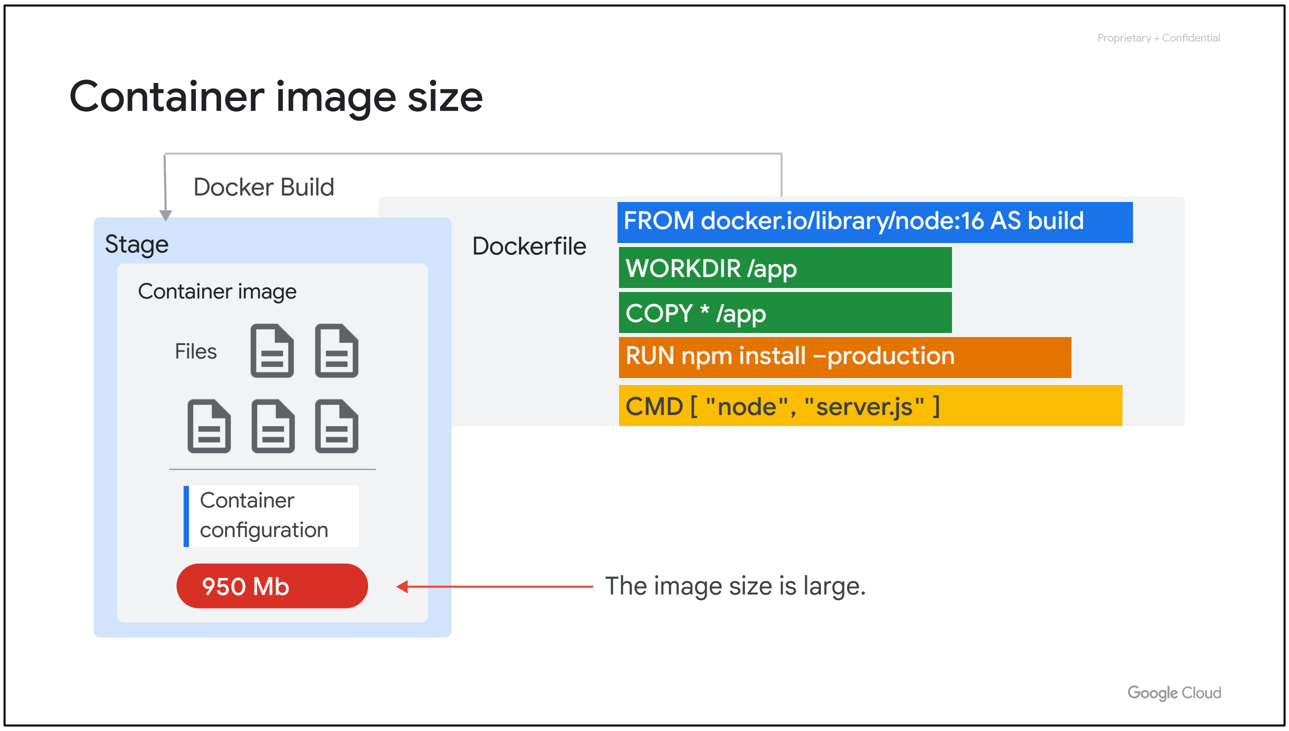
03 Building container images with Buildpacks

04 Continuous integration and delivery tools

05 Best practices

Google Cloud

Let’s now discuss some of the best practices when building and securing containers.



Proprietary + Confidential

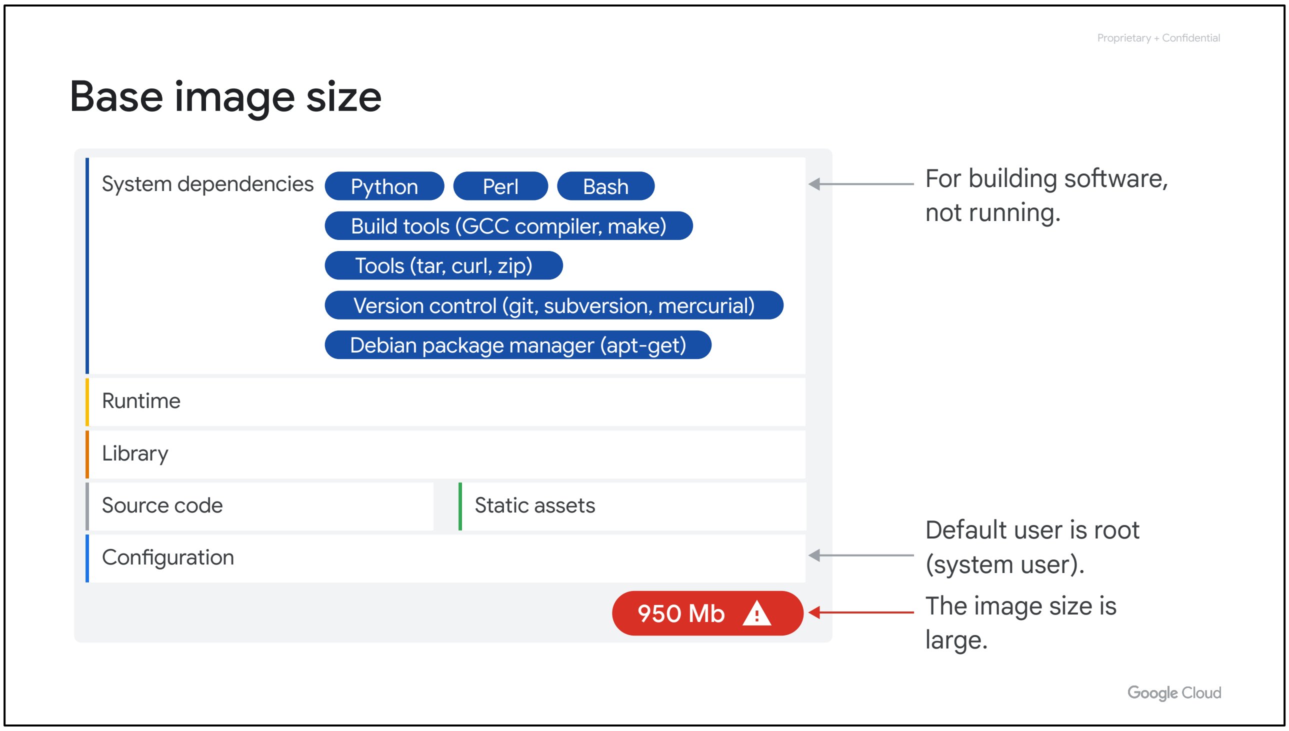
Container image size

```
FROM docker.io/library/node:16 AS build
WORKDIR /app
COPY * /app
RUN npm install –production
CMD [ "node", "server.js" ]
```

Google Cloud

In a previous lesson, we containerized a Node.js application with Docker.

The image that was created is almost 1 *GB* in size. What is the reason for this?


Proprietary + Confidential

Base image size

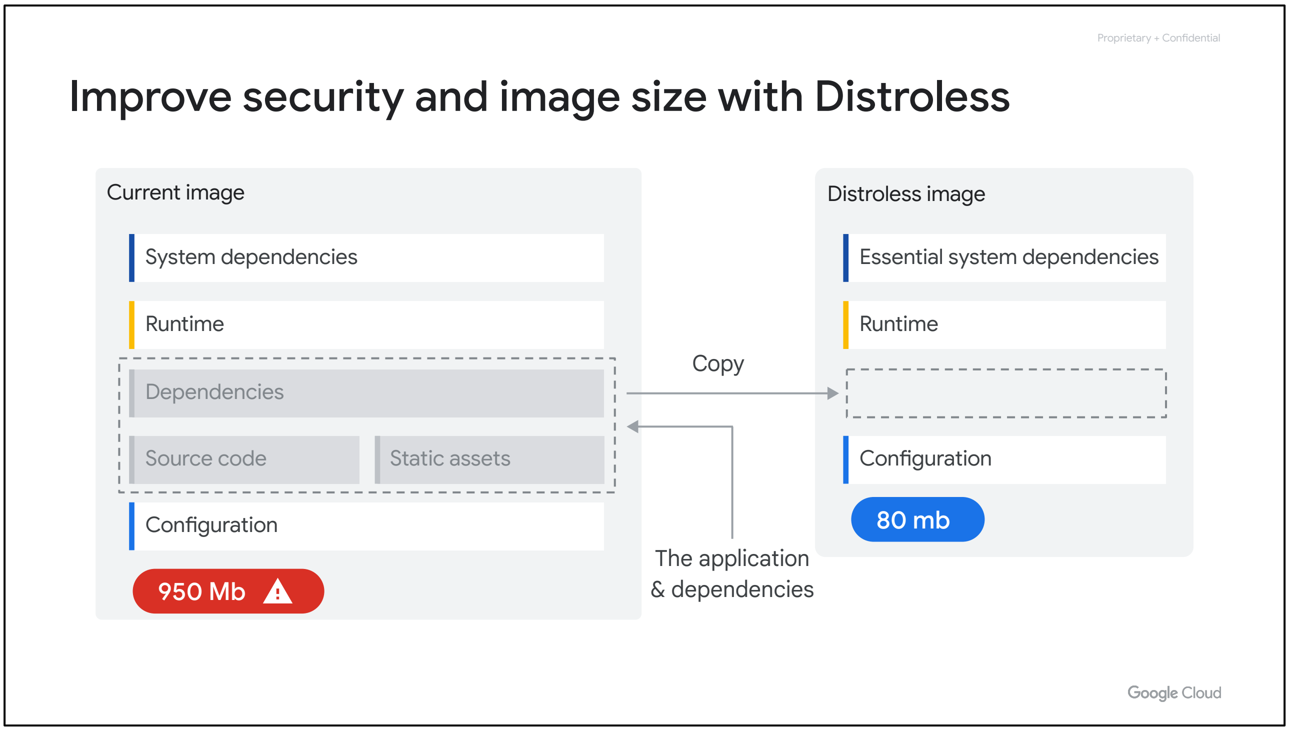
System dependencies
Python
Perl
Bash
Build tools (GCC compiler, make)
Tools (tar, curl, zip)
Version control (git, subversion, mercurial)
Debian package manager (apt-get)
Runtime
Library
Source code
Static assets
Configuration
950 Mb
For building software, not running.
Default user is root (system user).
The image size is large.

Google Cloud

The Node.js runtime is approximately 80Mb, and the app and library dependencies are not more than 1 MB. What's in the other 869 MB?

As it turns out, this is because the image was based on the Docker Hub *node* image—which comes packed with the system packages you need to build software. Even the Debian package manager apt-get is included so you can install even more system packages.

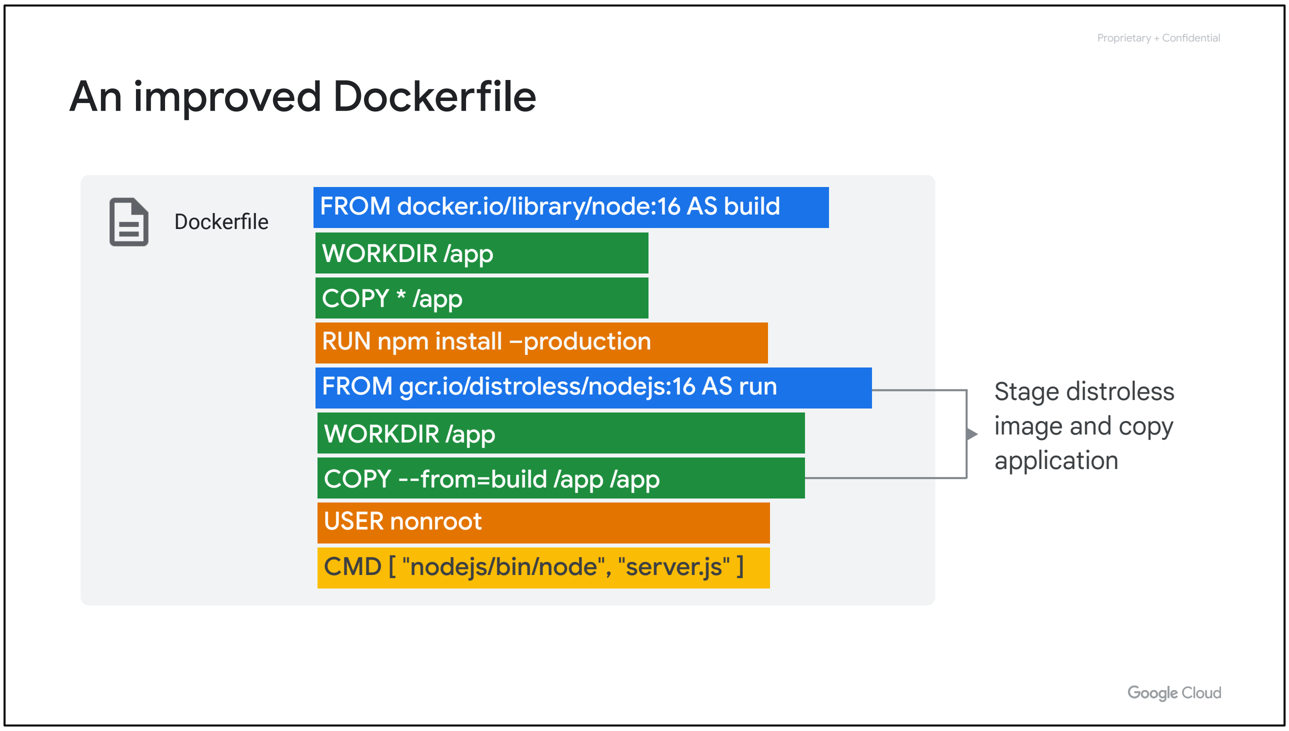
This *base image* is more suitable for building than for running software. In production, smaller images are better for security reasons. There might be security vulnerabilities in the additional system packages.



The solution that Docker has for this problem is a multi-stage image build.

It involves downloading a new image after you've completed your initial build, and copying all application files, assets, and library dependencies.

We review this process with an image from the Distroless project that is optimized for running applications, as it contains only the application and its runtime (not build time) dependencies.


Proprietary + Confidential

An improved Dockerfile

Dockerfile

```
FROM docker.io/library/node:16 AS build
WORKDIR /app
COPY * /app
RUN npm install –production
FROM gcr.io/distroless/nodejs:16 AS run
WORKDIR /app
COPY --from=build /app /app
USER nonroot
CMD [ "nodejs/bin/node", "server.js" ]
```

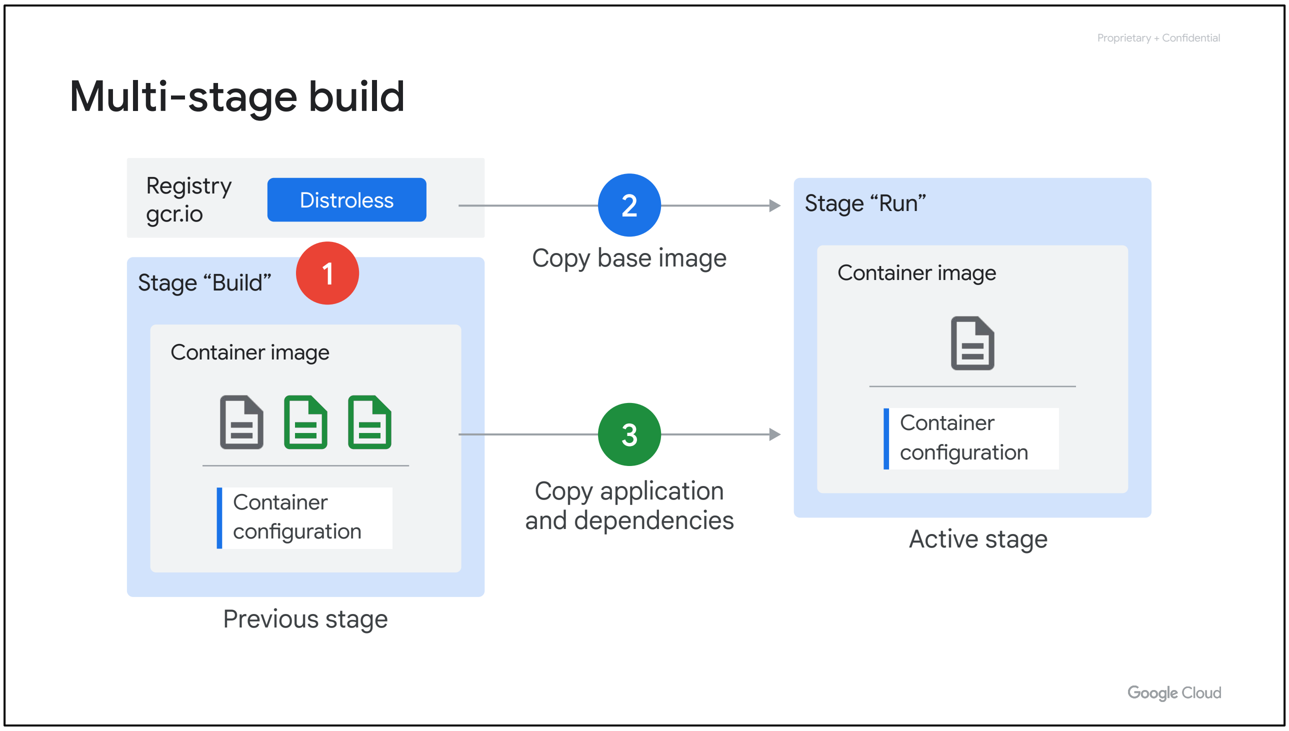
Google Cloud

Here's an improved version of the same example that you saw earlier.

After running npm install, it has a repeated FROM instruction, this time from the Distroless project.

The FROM instruction is followed by a COPY instruction, that pulls the application from the previously staged image onto the active stage.

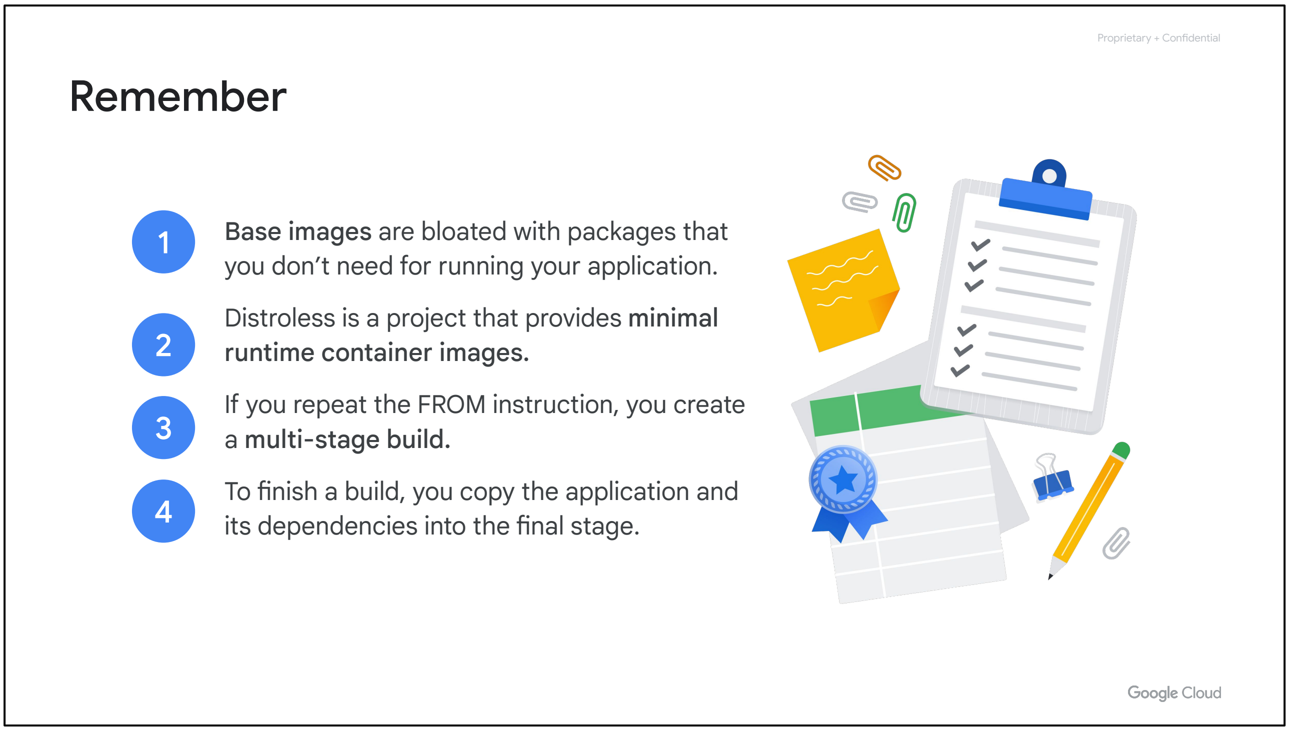
The Dockerfile includes additional container configuration, which sets the *nonroot* user - a user without system administrator permissions.



Use multi-stage builds to create a minimal container image.

In a multi-stage build:

1. The first stage is built that contains the Node.js image along with the application code using the first set of FROM and COPY instructions.
2. The second FROM instruction stages a new container image from the Distroless project.
3. The second COPY command copies the application and dependencies from the previous stage onto the active stage.



Proprietary + Confidential

Remember

1. **Base images** are bloated with packages that you don't need for running your application.
2. Distroless is a project that provides **minimal runtime container images.**
3. If you repeat the FROM instruction, you create a **multi-stage build.**
4. To finish a build, you copy the application and its dependencies into the final stage.

Google Cloud

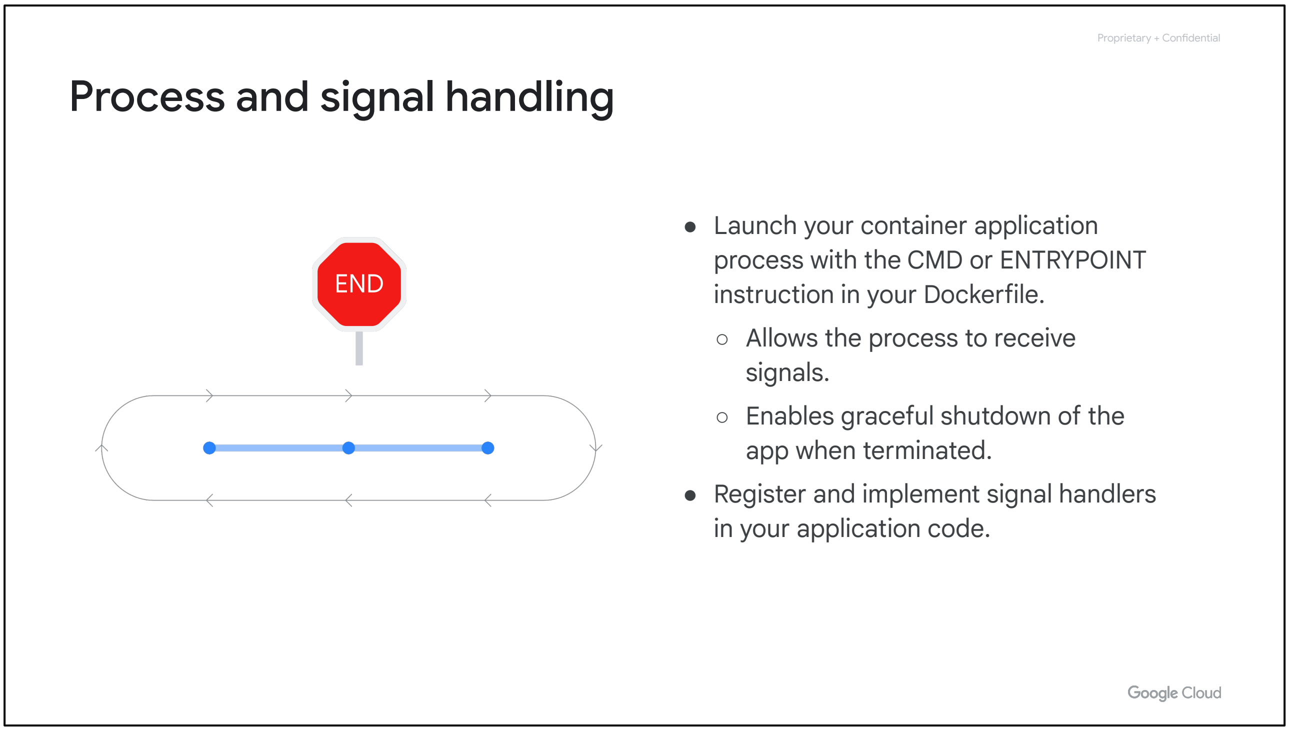
Here's what's important to remember about building secure and small container images with Dockerfiles.

Base images are bloated with packages that you don't need for running your application, but do need to build your application. For example, they contain package managers and version control software. It's a risk to add these binaries to the container image you deploy, because they might contain security vulnerabilities.

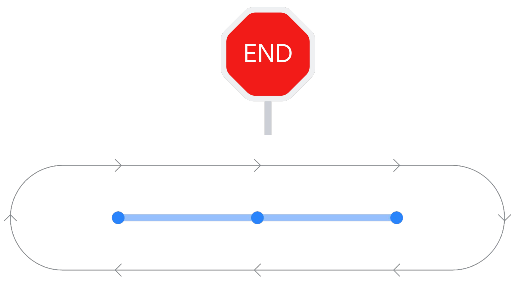
Distroless is a project that provides minimal runtime container images. These images can't be used to *build* your application, but do contain only a minimal set of system dependencies to *run* your application.

If you repeat the FROM instruction, you create a multi-stage build.

To finish a build, you copy the application and its dependencies into the final stage.


Proprietary + Confidential

Process and signal handling



- Launch your container application process with the CMD or ENTRYPOINT instruction in your Dockerfile.
  - Allows the process to receive signals.
  - Enables graceful shutdown of the app when terminated.
- Register and implement signal handlers in your application code.

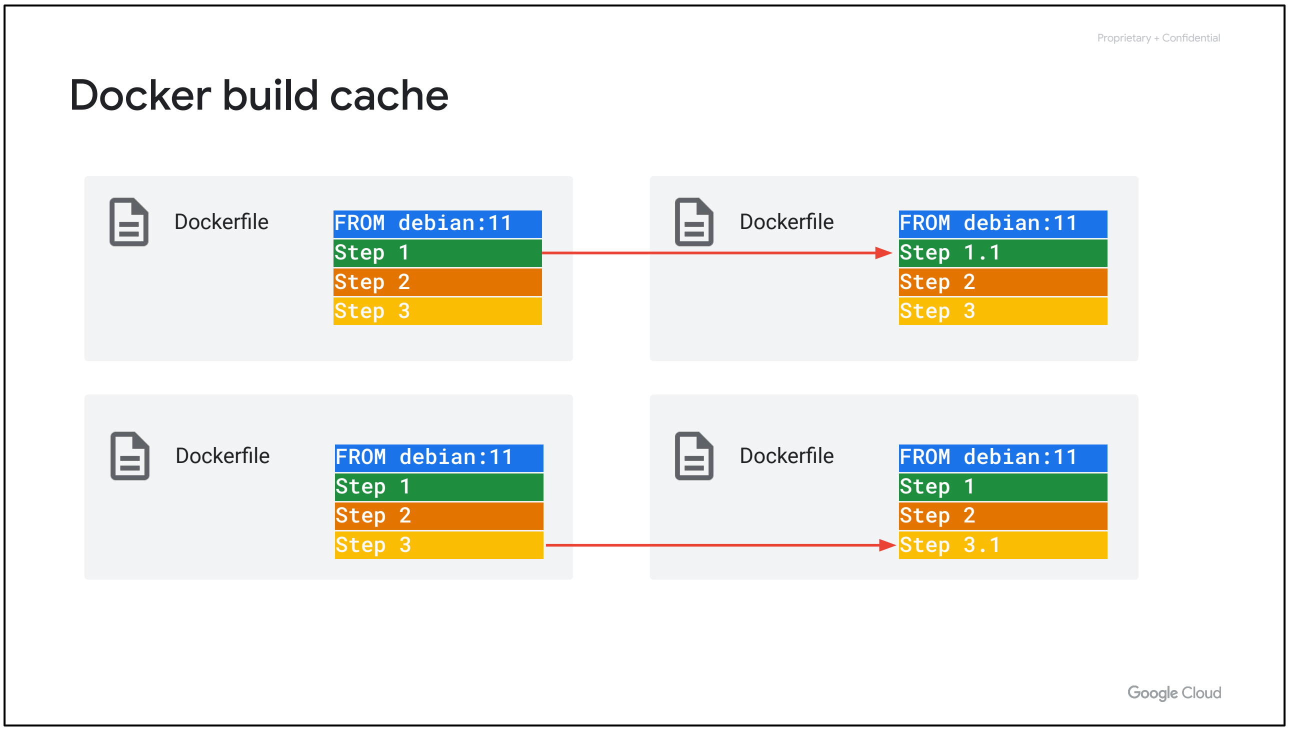
Google Cloud

Process identifiers (PIDs) are unique identifiers that the Linux kernel gives to each process. The first process launched in a container gets PID 1.

Container platforms such as Docker, Kubernetes, and Cloud Run use signals to communicate with the processes inside containers, most notably to terminate them.

Because these platforms can only send signals to the process that has PID 1 inside a container, you must launch your process with the *CMD* or *ENTRYPOINT* instruction in your Dockerfile. This allows the process to receive signals and gracefully shutdown the app when it’s terminated.

Also, because signal handlers aren't automatically registered for the process with PID 1, you must implement and register these signal handlers in your application code.



Proprietary + Confidential

Docker build cache

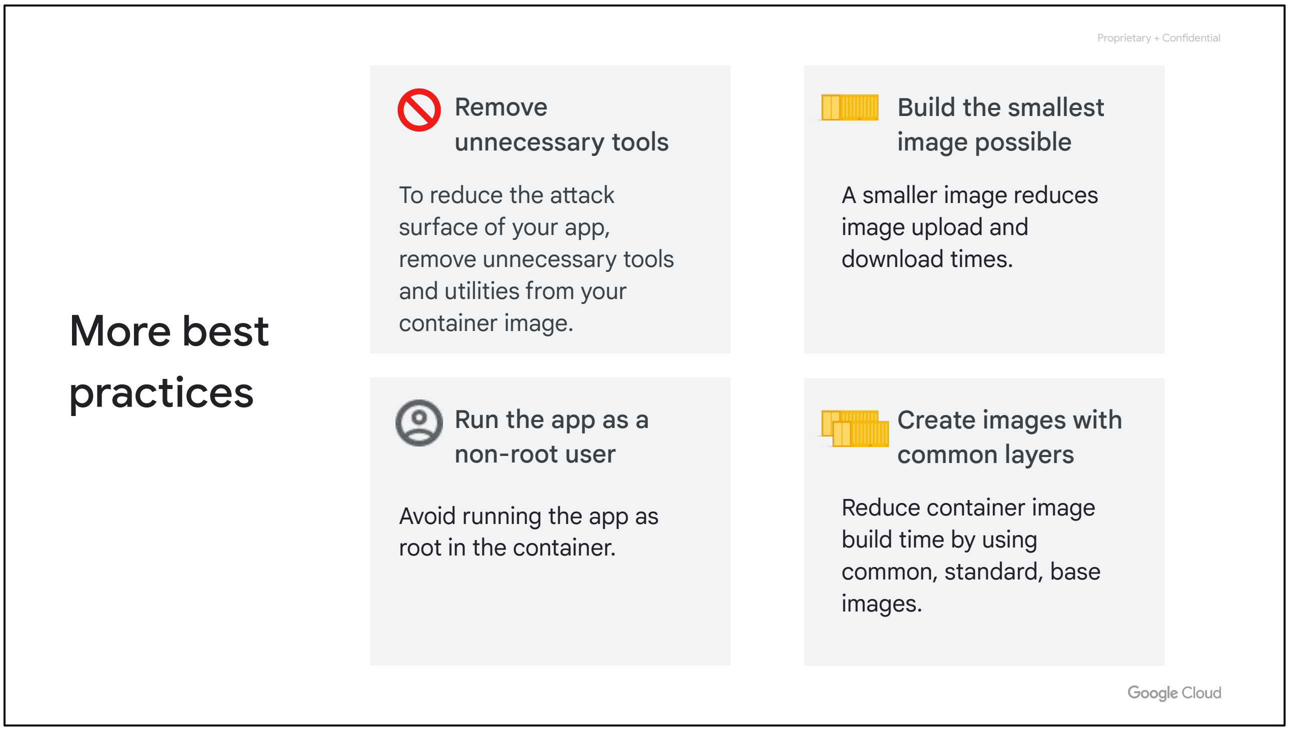
Google Cloud

When building a container image, Docker steps through the instructions in your Dockerfile executing them in the specified order. Each instruction creates a layer in the resulting image. For each instruction, Docker looks for existing image layers in its cache that can be reused.

Docker can use its build cache for an image only if all previous build steps used it. By positioning build steps that involve frequent changes at the bottom of the Dockerfile, you can enable faster builds by utilizing the Docker build cache.

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 example, if STEP 1 involves changes, Docker can reuse only the layers from the FROM debian:11 step. If STEP 3 involves changes, Docker can also reuse the layers for STEP 1 *and* STEP 2.



Proprietary + Confidential

More best practices

Remove unnecessary tools

To reduce the attack surface of your app, remove unnecessary tools and utilities from your container image.

Build the smallest image possible

A smaller image reduces image upload and download times.

Run the app as a non-root user

Avoid running the app as root in the container.

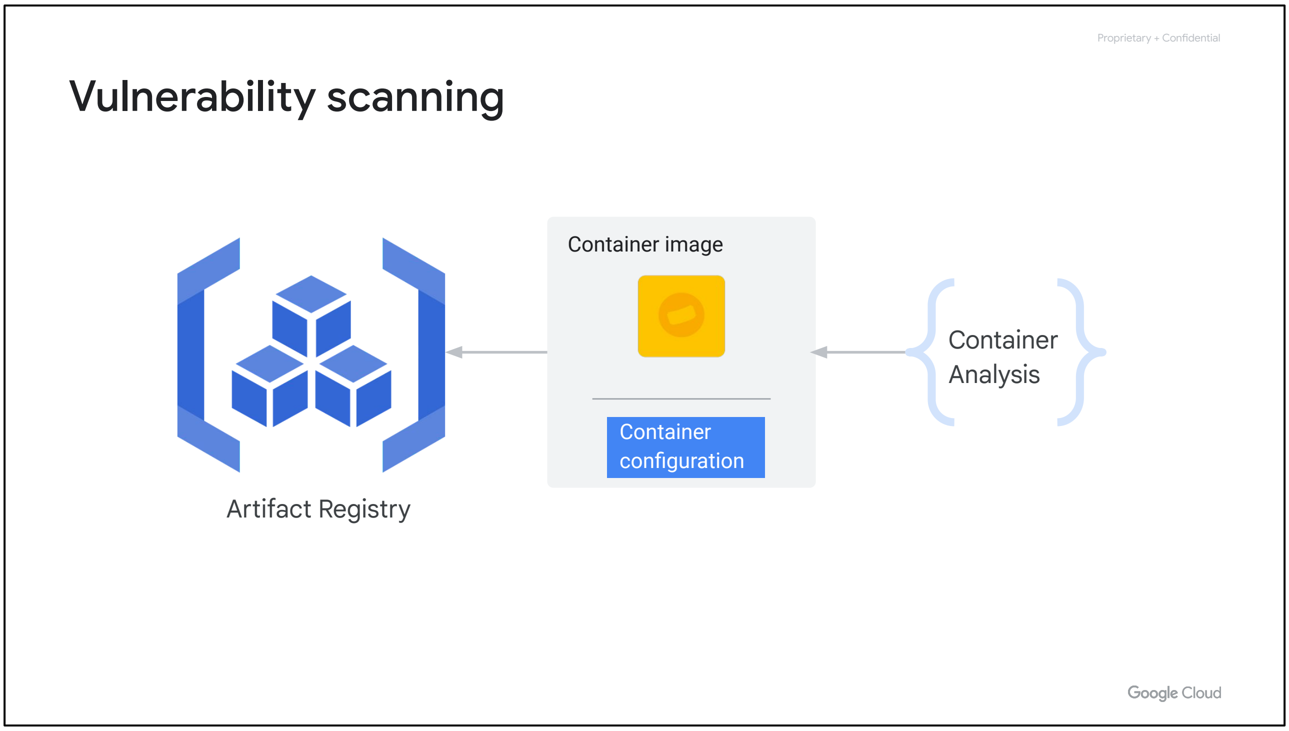
Create images with common layers

Reduce container image build time by using common, standard, base images.

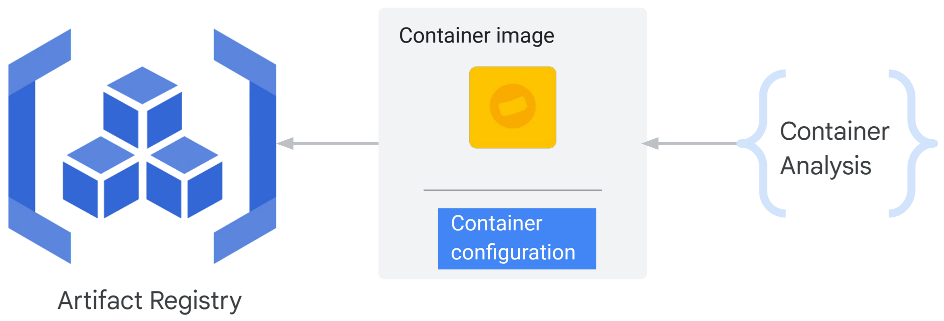
Google Cloud

Here are some additional best practices to follow when building container images:

- You should keep as few things as possible in your container image, ideally only your app, and remove any unnecessary tools or utilities. This reduces the attack surface of your app and protects it from potential harm from attackers.
- To prevent attackers from modifying root-owned files in your container image using a package-manager such as *apt-get*, avoid running the app as the *root* user inside the container. You must also disable or uninstall the *sudo* command. Also, consider launching the container in read-only mode.
- Build a smaller image to reduce image upload and download times. The smaller the image, the faster a node can download it. By referencing a smaller base image in the FROM instruction of your Dockerfile, you can control the overall size of the resulting container image.
- Provide developers within your organization with a set of common, standard, base images by downloading each base image only once. After the initial download, only the layers that make each container image unique are needed, thus reducing the amount of time needed to build the developer's container image.


Proprietary + Confidential

Vulnerability scanning



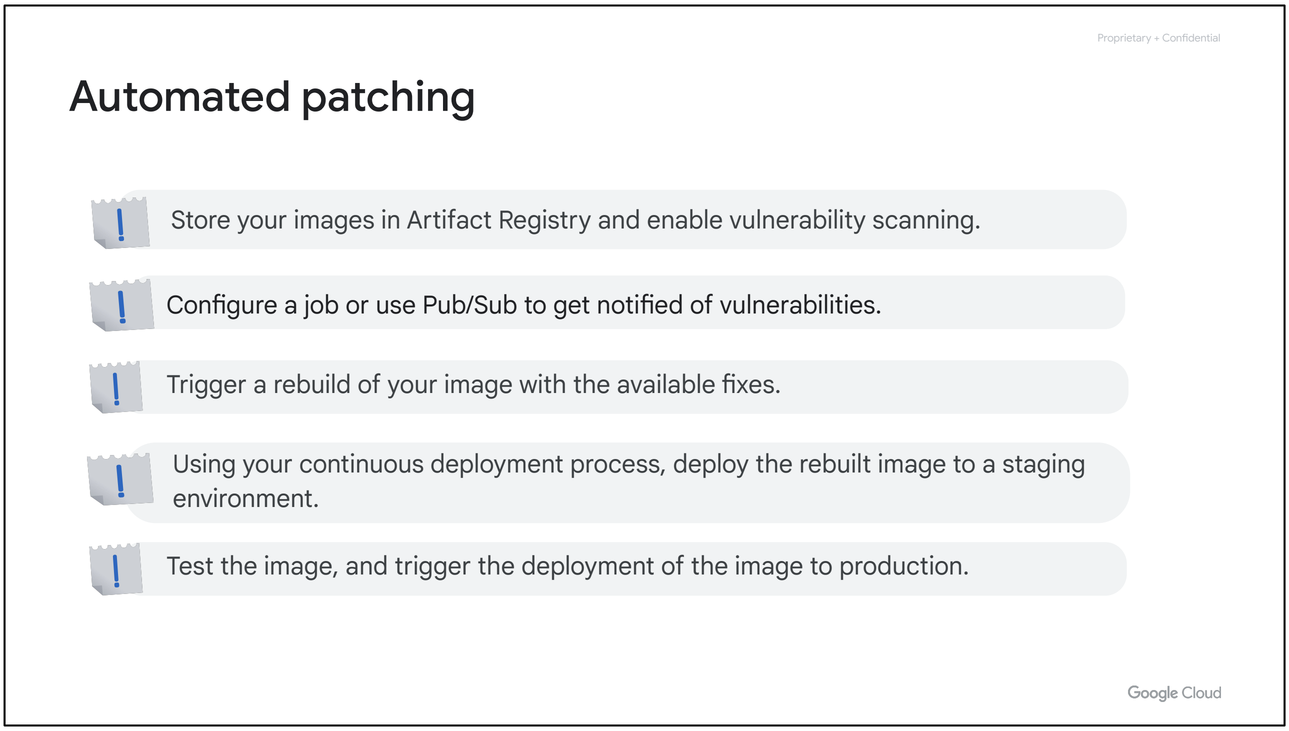
Google Cloud

It’s a best practice to scan your container images for software vulnerabilities, and if found, rebuild the image to include any patches that fix the vulnerabilities, and then redeploy your container.

Container Analysis is a service that provides vulnerability scanning and metadata storage for containers on Google Cloud. The scanning service performs vulnerability scans on images in Artifact Registry, then stores the resulting metadata and makes it available for consumption through an API.

When enabled, this service can automatically scan your container image and is triggered when a new image is pushed to Artifact Registry. With the on-demand scanning API, you can also enable manual scans of container images that are stored in these registries or stored locally.

For more information on the Container Analysis service, view the documentation.


Proprietary + Confidential

Automated patching

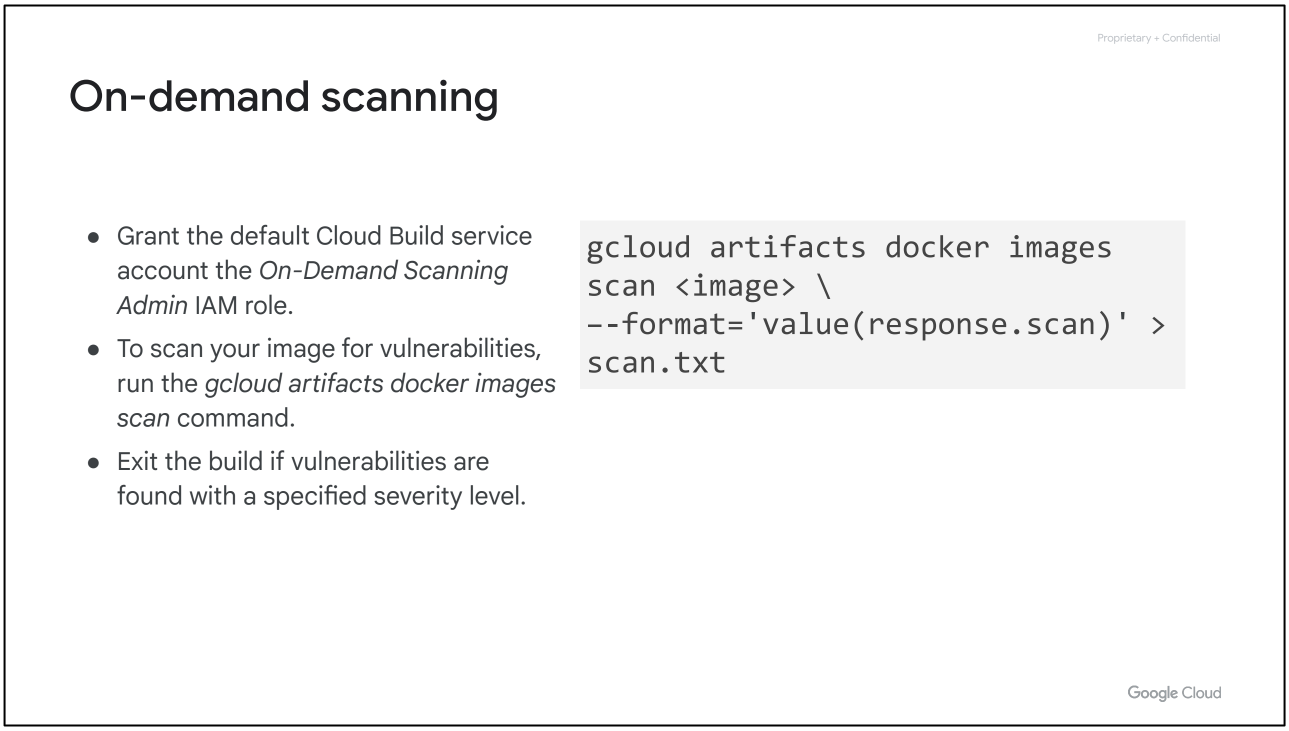
- Store your images in Artifact Registry and enable vulnerability scanning.
- Configure a job or use Pub/Sub to get notified of vulnerabilities.
- Trigger a rebuild of your image with the available fixes.
- Using your continuous deployment process, deploy the rebuilt image to a staging environment.
- Test the image, and trigger the deployment of the image to production.

Google Cloud

To fix vulnerabilities that are discovered in your container image, it's recommended to patch the image with an automated process using your continuous integration pipeline that was originally used to build the image.

To achieve this, here are a set of high-level steps:

1. Store your images in Artifact Registry and enable vulnerability scanning.
2. Configure a job to fetch vulnerability metadata from the Container Analysis service, and if a vulnerability is detected with available fixes, trigger a rebuild of your image. You can also use Pub/Sub integration to get notified of vulnerabilities and trigger a rebuild of your image.
3. Deploy the rebuilt image to a staging environment using your continuous deployment process.
4. Test the image in staging and check that the fixes are applied.
5. Trigger the deployment of the image to your production environment.


Proprietary + Confidential

On-demand scanning

- Grant the default Cloud Build service account the *On-Demand Scanning Admin* IAM role.
- To scan your image for vulnerabilities, run the *gcloud artifacts docker images scan* command.
- Exit the build if vulnerabilities are found with a specified severity level.

```
gcloud artifacts docker images
scan <image> \
--format='value(response.scan)' >
scan.txt
```

Google Cloud

As part of your Cloud Build pipeline, you can use on-demand scanning to scan a container image for vulnerabilities after it's built. If the scan detects vulnerabilities at a specified severity level, you can then block the upload of the image to Artifact Registry.

If the default Cloud Build service account is used for the build, grant it the On-Demand Scanning Admin IAM role: (`roles/ondemandscanning.admin`).

To scan your built container image for vulnerabilities, use the `gcloud artifacts docker images scan` command after the build step in your Cloud Build configuration file, and save the command output to a text file.

Exit the build if the scan command output indicates the desired severity level of any detected vulnerabilities. Otherwise continue with the build to push the built image to Artifact Registry.

For more information, view this tutorial for on-demand scanning.