# Python on Docker Production Quickstart

Itamar Turner-Trauring

June 17, 2020

# Contents

# Legal disclaimer



This quickstart and the information it contains are provided “as is”, without warranties of any kind, including without limitation the implied warranties of merchantability or fitness for any particular purpose. In no event shall the author, Itamar Turner-Trauring, or the copyright owner, Hyphenated Enterprises LLC, or any of their employees, agents or affiliates, be liable for any claim, costs, expenses, damages or other liability, whether under principles of contract, tort or otherwise, arising from, out of or in connection with the use of or reliance on the information contained in this checklist.

# Introduction

Building production-ready Docker images isn't easy: there's a huge list of details you need to remember and get right. The goal of this quickstart guide is therefore to get you going as quickly as possible.

How to read this guide:

1. Read the next chapter, which presents a possible implementation plan.
2. Skim the book, so you get a sense of all of the best practices; not all of them are listed in the implementation plan.
3. Go back to the implementation and create your own, situation-specific plan with the best practices you actually care about.
4. Start packaging!
5. Refer back to the guide when you need additional details.

A few of the best practices are marked with ∞ (infinity) symbols, indicating they are ongoing processes. These tasks require periodic attention, even after you've finished creating and deploying your image.

If you have any questions or suggestions, please email me at itamar@pythonspeed.com.

—Itamar Turner-Trauring

# 1 A phased implementation plan

There are many best practices to follow, and some are more important than others. What follows is a proposed order of implementation for your Docker packaging, with the presumption that your goal is to get something working as quickly as possible.

To summarize the proposed order:

1. Get something minimal working, so that it's buildable on your development laptop or workstation.
2. Implement basic security.
3. If relevant, have builds run in CI or your build system, so other people can use it.
4. Make the images easier to identify and debug.
5. Improve operational correctness.
6. Make the builds reproducible.
7. Speed up the build time.
8. Make the image smaller.

   **Important:** This plan doesn't cover all the best practices. You should also skim the book and read the other best practices, to learn about other ways to enhance your build.

## 1.1 Step 1: It works

The first step is just getting something working, so you have something you can improve. That means:

1. Writing your initial `Dockerfile`.
2. Writing the script that will run `docker image build`; for now it might be quite simple, but it's likely to grow over time.

I suggest starting with the following best practices:

- Choose a stable base image and tag (ref), which is not Alpine Linux (ref).

Since you'll need to choose a base image anyway, you may as well start with a good one.
- For network servers, listen on `0.0.0.0` (ref). Your image isn't working if you can't connect to it.
- Write logs to `stdout` (ref), so you can see what's going when you run the container.
- If you're writing any shell scripts, make sure they're not broken (ref). In theory you can wait to do this later, but my experience is that if you don't do this from the start, it can be difficult to add the necessary checks later on.

If you're installing system packages using `apt-get`, you'll want to:

- Make sure `apt-get` never runs in interactive mode (ref).

### Partial example

At this point the `Dockerfile` for a simple application might look like this:

```
FROM python:3.8-slim-buster
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
ENTRYPOINT ["python", "server.py"]
```

And `build.sh` will look like this:

```
#!/bin/bash
set -euo pipefail
docker image build -t yourimage .
```

## 1.2 Step 2: Basic security

Now that you have something working, the next step is to make your image sufficiently secure to run somewhere public.

- Don't listen on ports < 1024 (ref).
- Don't run as root (ref).
- Run your container with no capabilities (ref).
- Update system packages in your `Dockerfile` (ref).
- Set up the technical and organizational processes you need to apply security fixes when they come out (ref).

In some cases you might need to apply other best practices:

- If you're using build secrets, make sure you're not leaking them (ref).
- If you have any files you don't want to be in the image for security reasons, make sure they're in your `.dockerignore` (ref).

### Partial example

The `Dockerfile` might now look like this:

```
FROM python:3.8-slim-buster

# Install security updates:
RUN apt-get update && apt-get -y upgrade

WORKDIR /app
COPY . .
RUN pip install -r requirements.txt

# Run as non-root user:
RUN useradd --create-home appuser
USER appuser
```

```
ENTRYPOINT ["python", "server.py"]
```

## 1.3 Step 3: Running in CI

Now that you have a working image with basic security, usually the next step is to build the images automatically in your CI or build system. You'll typically want your build system to build the image and push it to your chosen image registry—perhaps on every commit, perhaps on pull requests, depending on your particular development workflow.

As you're doing so, follow these best practices:

- Tag images based on the version control branch (ref).
- Don't rely on the `latest` tag (ref).
- Set up an automatic rebuild-and-redeploy once a week so you get security updates to the image's system packages (ref).
- Run security scanners on your code (ref).

You can put these off until you're working on faster builds, or do them here:

- For faster builds, warm up the build cache (ref).
- Warm up the build cache for per-branch builds (ref).

### Partial example

Your build script will now look something like this:

```
#!/bin/bash
set -euo pipefail

YOURIMAGE=registry.example.com/yourorg/yourserver
```

```
GIT_BRANCH=$(git rev-parse --abbrev-ref HEAD)
GIT_COMMIT=$(git rev-parse --short HEAD)

# Pull previous version, and use with --cache-now
# for build caching:
docker pull $YOURIMAGE:$GIT_BRANCH || true

# Use branch+commit for tagging:
docker build -t "$IMAGE_NAME:$GIT_BRANCH" \
             -t "$IMAGE_NAME:$GIT_COMMIT" \
             --cache-from=$YOURIMAGE:$GIT_BRANCH .

# Security scanners:
docker run --entrypoint=bash $YOURIMAGE:$GIT_BRANCH \
           -c "pip install safety && safety check"
trivy image --ignore-unfixed --exit-code 1 \
    $YOURIMAGE:$GIT_BRANCH

# Push to the registry:
docker push "$IMAGE_NAME:$GIT_BRANCH"
docker push "$IMAGE_NAME:$GIT_COMMIT"
```

You'll also need additional code to ensure a complete rebuild (`docker build --pull --no-cache`) once a week.

## 1.4 Step 4: Easier to debug

Now that your image is building automatically, you will start accumulating many more images. Now is the time to make images identifiable and easier to debug.

- Prepare for C crashes (ref). It's one extra line in your `Dockerfile` that can save

hours or even days of debugging later on.
- Record the build's version control revision and branch in the image itself (ref).
- Add a smoke test to your CI build (ref).

You might also want to:

- Pre-install useful tools (ref).

#### Partial example

Your `Dockerfile` will add the following line:

```
ENV PYTHONFAULTHANDLER=1
```

And the build script's `docker build` line will look like this:

```
docker build -t "$IMAGE_NAME:$GIT_BRANCH" \
             -t "$IMAGE_NAME:$GIT_COMMIT" \
             --label git-commit=$GIT_COMMIT \
             --label git-branch=$GIT_BRANCH \
             --cache-from=$YOURIMAGE:$GIT_BRANCH .
python smoketest.py $YOURIMAGE:$GIT_BRANCH
```

## 1.5 Step 5: Additional operational correctness

Next, you should make your images behave correctly.

- Make sure your image shuts down correctly (ref).
- Make sure your image can handle zombie processes correctly (ref).
- If it's a long-running process, implement health checks (ref).
- If it's a server, make sure the server can handle health checks and slow queries at the same time (ref).

- Pre-compile bytecode for faster startup (ref).

You may also want some application-specific and deployment environment configuration:

- Gunicorn requires certain options (ref).
- uWSGI requires certain options (ref).
- Heroku and Google Cloud Run require listening on a configurable port (ref).

### Partial example

Your `Dockerfile` might now look like this:

```
FROM python:3.8-slim-buster

RUN apt-get update && apt-get -y upgrade
# Install tini:
RUN export DEBIAN_FRONTEND=noninteractive && \
    apt-get install -y tini

WORKDIR /app
COPY . .
RUN pip install -r requirements.txt



RUN useradd --create-home appuser
USER appuser

ENV PYTHONFAULTHANDLER=1
# Use tini as an init process:
ENTRYPOINT ["tini", "--", "python", "server.py"]
```

## 1.6 Step 6: Reproducible builds

As time goes on, the software you depend on will change—and that can break your build. So next you should ensure your builds are reproducible, always installing the software versions you specifically asked for.

- Pin your Python dependencies (ref).
- Set up the technical and organizational processes to update all pinned dependencies once a month (ref).

You might also want to:

- Pin your system package dependencies (ref).
- Create a custom base image (ref).

## 1.7 Step 7: Faster builds

Now that you have good images, it's time to look at optimizations. Your time is expensive, so it's worth spending some effort to get faster builds, by optimizing Docker's build caching.

If this is the second or third time you're packaging an application, you might have done these automatically earlier on, but if not:

- `COPY` in files only when needed (ref).
- Use `ARG` only when needed (ref).
- Install dependencies separately from your code (ref).

### Partial example

```
FROM python:3.8-slim-buster

RUN apt-get update && apt-get -y upgrade
RUN export DEBIAN_FRONTEND=noninteractive && \
```

```
    apt-get install -y tini

# Only copy in requirements.txt for now:
COPY requirements.txt .
RUN pip install -r requirements.txt

RUN useradd --create-home appuser
USER appuser

WORKDIR /app
# Copy in the rest of the files:
COPY . .

ENV PYTHONFAULTHANDLER=1
ENTRYPOINT ["tini", "--", "python", "server.py"]
```

## 1.8 Step 8: Smaller images

Finally, your images might be quite large at this point, so it's good to make them smaller.

- Use `dive` to find where the space is going (ref).
- Add files to `.dockerignore` (ref).
- Avoid extra `chown`s (ref).
- Minimize system package installation (ref).
- Reduce disk usage from `pip` installs (ref).

You might find that you can't have both small images and fast builds. Typically this happens when you need to install a compiler toolchain: caching enables fast builds, but also makes your images larger. In this situation you may want to consider multi-stage builds (ref).

### Partial example

Your `Dockerfile` might now look like this:

```
FROM python:3.8-slim-buster

# Clean up after installing packages:
RUN export DEBIAN_FRONTEND=noninteractive && \
    apt-get update && \
    apt-get -y upgrade && \
    apt-get install -y --no-install-recommends tini && \
    apt-get -y clean && \
    rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
# Don't cache pip-downloaded packages:
RUN pip install --no-cache-dir -r requirements.txt

RUN useradd --create-home appuser
USER appuser

WORKDIR /app
COPY . .

ENV PYTHONFAULTHANDLER=1
ENTRYPOINT ["tini", "--", "python", "server.py"]
```

# 2 Best practices: Security

## 2.1 Don't run as root

Running your Docker image as root exposes you to significant security risks: it makes it much easier for an attacker to get root on the host. Better, then, to start your container as a non-root user:

```
FROM python:3.8-slim-buster
RUN useradd --create-home appuser
WORKDIR /home/appuser
USER appuser
ENTRYPOINT ["yourprogram"]
```

Note that `USER` only affects commands after it is used, so you can still run commands as root beforehand:

```
RUN useradd --create-home appuser
WORKDIR /home/appuser
# This runs as root:
RUN chmod 777 /var/log
USER appuser
# This runs as appuser:
RUN pip install yourpackage
# And this runs as appuser:
ENTRYPOINT ["python", "-m", "yourcode"]
```

You can use tools like `gosu` to change away from root after startup, but that will prevent you from running the container with reduced capabilities (see below), which is another useful security measure. Better then to use `USER`.

### 2.1.1 References

- Less capabilities, more security: minimizing privilege escalation in Docker (pythonspeed.com)
- `Dockerfile USER` command (docs.docker.com)
- `useradd(8)` man page (www.man7.org)

## 2.2 Don't listen on ports < 1024

In order to listen on TCP or UDP ports less than 1024 on Linux, you need to either run as root or have the relevant security capability (see below). Since you don't want to run as root, nor grant your container unnecessary capabilities, you should always listen on port 1024 or higher.

But what if you do need to listen on a low port? For example, an HTTPS server needs to listen on port 443 if you want a standard `https://` URL to work.

Remember that when you run your container you are doing port-forwarding to expose the service, and the external port on the host doesn't have to match the port inside the container. So you can do:

```
$ docker run -p 443:8443 yourcontainer
```

And now port 443 on the host maps to port 8443 in the container.

Pretty much every system that can run a container has similar options. There is therefore no reason to listen on ports < 1024.

## 2.3 Run your container with no capabilities

Linux "capabilities" allow you to grant a subset of root's power to a process. A process may have some capabilities currently in effect, and some capabilities it can inherit when it executes an executable with the ability to grant capabilities.

By default when running as non-root user in Docker, the process has no effective capabilities, but it can inherit quite a few when it run an executable. For example, here you can see that running `/usr/bin/ping` granted the resulting process the ability to turn on `cap_net_raw` capability:

```
$ docker run --user=1000 -it centos
bash-4.4$ getcap /usr/bin/ping
/usr/bin/ping = cap_net_admin,cap_net_raw+p
bash-4.4$ ping example.com > /dev/null &
[1] 13
bash-4.4$ getpcaps 13
Capabilities for `13': = cap_net_raw+p
```

If `ping` has a security vulnerability that allows injecting code into it, an attacker now has access to escalated privileges. You can solve this by running the container with no capabilities:

```
$ docker run --user=1000 --cap-drop=ALL -it centos
bash-4.4$ ping example.com
ping: socket: Operation not permitted
```

Kubernetes has similar options.

Technically this best practice is about running your container, not about packaging. But as mentioned in the previous two best practices it does have an impact on how you should package your application.

### 2.3.1 References

- `capabilities(7)` man page (www.man7.org)
- `docker run` capabilities options (docs.docker.com)
- Kubernetes Pod security context (kubernetes.io)

## 2.4 Update system packages

The system packages in the base image you are using may or may not be up-to-date, which means they might not include the latest security updates. You should therefore update them at the start of your build.

```
FROM python:3.8-slim-buster
RUN apt-get update && apt-get -y upgrade
```

There are more fully fleshed-out scripts in a later checklist item.

## 2.5 ∞ Rebuild without caching and redeploy at least weekly ∞

Even if your code hasn't changed, you still need to rebuild your image and redeploy it on a regular basis.

1. The system packages you depend on will have new releases, due to security flaws and other bugs.
2. The Python base image may also have a new point release with security fixes.

For example, if your image uses `nginx` and `nginx` has a remote execution attack, you don't want to wait until a new release of your code to deploy a fixed `nginx`.

You should therefore rebuild your images once a week, and then redeploy them.

You also need to make sure these rebuilds actually get the updates. Remember that Docker build caching will skip updating a layer if the command hasn't changed. That means that if you have a cached version, `apt-get upgrade` won't run even if there are new packages!

You should therefore rebuild with `--no-cache`, so all layers are rebuilt, and with `--pull` so that you're using the latest base image:

```
$ docker build -t example.com/myorg/myimage --no-cache --pull .
$ docker push example.com/myorg/myimage
```

### 2.5.1 References

- `docker build` reference (docs.docker.com)
- Avoiding insecure images from Docker build caching (pythonspeed.com)

## 2.6 ∞ Apply security fixes when they come out ∞

Over time, software packages you depend on will get security fixes as well as fixes to critical bugs. Insofar as you've pinned your packages, you will need to apply these fixes yourself. And you want to get notified when new packages are released with these fixes, so that you know an update is needed.

GitHub has had basic support for vulnerability alerts for a while, and has acquired Dependabot, which can automatically open PRs against your application with security fixes; it supports Pipenv, Poetry and pip-tools. If you're on GitHub go to the Security tab of your project and enable Automated Security Fixes (really it's automated security fix *suggestions*).

There are also other services like requires.io, PyUp, and more.

These tools will notify you of Python vulnerabilities and updates, but will not track pinned system packages; you'll need to track those yourself. If you're using a Debian-based base image, for example, you can track the Debian security releases.

### 2.6.1 References

- Debian security information (debian.org)
- Ubuntu security notices (usn.ubuntu.com)
- CentOS/RHEL security advisories (access.redhat.com)

## 2.7 Run security scanners

In your CI system you can run security scanners to scan:

1. Your code for potential security bugs.
2. Your dependencies for known vulnerabilities.
3. System packages and packages in other languages for known vulnerabilities.

### 2.7.1 Potential vulnerabilities in your Python code

`bandit` is a tool for finding vulnerabilities in your Python code; it will search for things like SQL injection attacks, use of pickle, and many more. You might want to run it via `flake8-bandit`, so you get the benefit of `flake8`'s generic skipping and configuration mechanism.

```
$ python3 -m venv /tmp/security
$ . /tmp/security/bin/activate
$ pip install bandit
$ bandit example.py
...
>> Issue: [B403:blacklist] Consider possible security implications
 ↪  associated with pickle module.
...
```

### 2.7.2 Known vulnerabilities in your Python dependencies

For known vulnerabilities in Python dependencies you can use the `safety` tool from PyUp, though keep in mind the free version only gets its vulnerability database updated once a month. If you want to get more up-to-date security vulnerability warnings from `safety` you'll need to pay for them.

> **Important:** These are *known* vulnerabilities that someone already reported. That's different than Bandit, which looks for *possible* vulnerabilities–you could in theory also run Bandit against dependencies and find some real issues that no one knew existed.

You will typically run `safety` inside the current virtualenv or Python install, for example by doing:

```
$ docker run --entrypoint=bash yourorg/yourimage \
      -c "pip install safety && safety check"
...
```

### 2.7.3 Known vulnerabilities in system and other language packages

Trivy is a command-line tool that lets you scan a Docker image for many kinds of security vulnerabilities, both system packages and programming language-specific packages. `trivy` has limited support for checking Python dependencies, which hopefully will get better, but at the moment it only supports Poetry and Pipenv; you'll probably still want to use `safety`.

> **Important:** One caveat with `trivy` is that some of the data sources it uses are supposedly for non-commercial use only. I've filed an issue for them to document which and allow disabling those.
>
> There are a number of other security scanners you could use, e.g. Anchore and Claire + Klar. They're harder to setup, though.
>
> Many hosted image registries will also do scans for you. You should always check what a particular scanner supports; not all of them will necessarily check Python dependencies.

By default `trivy` will show vulnerabilities that don't have any fixes available, including many vulnerabilities that will *never* have any fixes available. You may wish to ignore these (if there's no fix, what can you do?) by using the `--ignore-unfixed` option.

Additionally, by default `trivy` won't set a non-zero exit code when vulnerabilities are found, so you'll want to use the `--exit-code` option.

```
$ trivy image --ignore-unfixed --exit-code 1 yourorg/yourimage
```

### 2.7.4 References

- Bandit documentation (bandit.readthedocs.io)
- flake8-bandit (pypi.org)
- Safety documentation (pyup.io)
- Trivy (github.com)

## 2.8 Don't leak build secrets

Sometimes during a build you need to download source code or packages using a secret: a password, or an SSH private key. This is not the same as a *runtime* secret. Rather, it's a secret you will only need during the Docker image build, to download some build dependencies.

The obvious mechanism to pass secrets in is `docker build --build-arg`, but this is insecure: the secret will be embedded in the image, so anyone who has access to the image can see the secret by running `docker history`. You can also `COPY` in secrets, but then they will be leaked via the images.

To solve this you have a number of alternatives:

1. Use a short-term key or access token that will expire after a few minutes.
2. Pre-download the necessary packages or source code outside of the Docker build, then `COPY` the files in as normal. In this case you don't need to pass the secret in to the Docker build at all.
3. Use the new experimental BuildKit backend for Docker builds, which supports passing in secrets as well as SSH authentication-agent forwarding.
4. Pass in the secrets over the network.

### 2.8.1 A short term expiring access token

Some package repositories support the creation of short term access tokens. If your access token expires after 5 minutes, it doesn't matter if you leak it in your image so long as the push happens more than 5 minutes after token creation. You can then use build args or `COPY` to get the secret in.

### 2.8.2 Pre-download necessary files

Here's how this option would work if, for example, you need a private SSH key to download some code:

```
$ eval $(ssh-agent)
$ ssh-add ~/.ssh/id_rsa
$ git clone git@github.com:yourorg/yourprivatecode.git
$ docker build -t yourimage .
```

And the `Dockerfile` would just need to copy in the code:

```
FROM python:3.8-slim-buster
COPY yourprivatecode .
# RUN the build, etc.
```

### 2.8.3 BuildKit secrets

Here's how you would use BuildKit. Let's say you have a file with a secret:

```
$ cat secret-file
THIS IS SECRET
```

First, configure your `Dockerfile` to use BuildKit, and add a flag to `RUN` telling it to expose a particular secret:

```
# syntax = docker/dockerfile:1.1-experimental
FROM python:3.8-slim-buster
COPY build-script.sh .
RUN --mount=type=secret,id=mysecret ./build-script.sh
```

The `build-script.sh` will be able to find the secret at `/run/secrets/mysecret`.

Then, to build your image with the secret set the appropriate environment variable and pass in the newly enabled command-line arguments:

```
$ export DOCKER_BUILDKIT=1
$ docker build --secret id=mysecret,src=secret-file .
```

For the last method of passing in secrets, via the network, see the references below.

### 2.8.4 References

- Docker BuildKit backend documentation (docs.docker.com)
- Docker build secrets, via the network (pythonspeed.com)
- Using BuildKit secrets with Docker Compose (pythonspeed.com)

## 2.9 Using BuildKit secrets from Docker Compose

Docker Compose does not yet support using BuildKit secrets. Until that is supported (see the GitHub issue in the references) it is possible to use a workaround.

The basic idea is to have code that supports either secrets file or environment variables (via build args). When using `docker build` to build the released image, you use the secure secrets file; when building locally in Docker Compose, you use the insecure build argument.

Let's say you have a script `use_secret.sh` that wants to download a file using a password. You'd write it like this:

```
#!/bin/bash
set -euo pipefail

# Support both secrets file and an env variable:
if [ -f /run/secrets/thepassword ]; then
   export THEPASSWORD=$(cat /run/secrets/thepassword)
fi

wget -o download.zip https://admin:$THEPASSWORD@example.com/download.zip
```

The `Dockerfile` would look like this:

```
# syntax = docker/dockerfile:1.1-experimental
FROM python:3.8-slim-buster
# Only use the build arg for local development:
ARG THEPASSWORD
COPY use_secret.sh .
# Mount the secret to /run/secrets:
RUN --mount=type=secret,id=thepassword ./use_secret.sh
```

When you did a normal build, you would pass the secret in using a secure mechanism:

```
#!/bin/bash
set -euo pipefail
export DOCKER_BUILDKIT=1
docker build -t myimage \
    --secret id=thepassword,src=mypassword.txt .
```

Your Compose file in contrast would use build arguments:

```
version: "3.7"
services:
  yourapp:
    build:
      context: "."
      args:
        THEPASSWORD: "$THEPASSWORD"
```

And to use it you set `$THEPASSWORD` and the environment variables needed to make Compose use BuildKit:

```
$ export THEPASSWORD=$(cat mypassword.txt)
$ export DOCKER_BUILDKIT=1
$ export COMPOSE_DOCKER_CLI_BUILD=1
$ docker-compose up
```

### 2.9.1 References

- Build secrets in Docker Compose, the secure way (pythonspeed.com)
- BuildKit support in Docker Compose 1.25.1+ (docker.com)
- Issue #6358: Support for BuildKit secrets in Docker Compose (github.com)

## 2.10 Use host SSH keys with BuildKit `ssh-agent` forwarding

If you have an SSH private key on your host, perhaps password-protected, and you want to make it available to multiple processes without having to type that password each time, you can use `ssh-agent`.

```
$ eval `ssh-agent`
$ ssh-add ~/.ssh/id_rsa
Password: ******
```

Now, all future `ssh` calls will be able to use that private key without having to type in the password, since the agent has it cached in memory.

If you're using BuildKit, you can give the Docker build access to the private SSH keys on the host by talking the `ssh-agent` running on the host, and without leaking them into the image.

Your `Dockerfile` will look like this:

```
# syntax = docker/dockerfile:1.1-experimental
FROM python:3.8-slim-buster

RUN apt-get update && \
    apt-get -y install --no-install-recommends openssh-client git

RUN --mount=type=ssh git clone git@github.com:yourorg/private.git
```

And then you'd build it like this:

```
#!/bin/bash
set -euo pipefail

# Enable the ssh-agent:
eval `ssh-agent`
ssh-add

# Build with BuildKit and ssh-agent forwarding:
export DOCKER_BUILDKIT=1
docker build --ssh default -t yourimage .
```

### 2.10.1 References

- Using SSH with BuildKit (docs.docker.com)

- `ssh-add` man page (man7.org)
- `ssh-agent` man page (man7.org)

## 2.11 Optional: Pre-populate SSH known hosts

If you are using SSH within your Docker build, SSHing to a new host will ask you to verify the host's key, which will fail because Docker builds don't allow interactive input. You can disable checking the key with the `StrictHostKeyChecking=no` option, but this puts you at risk of man-in-the-middle attacks.

Better to copy in a `.ssh/known_hosts` prepopulated with the public key of the SSH host you will be accessing. You can download a key as follows:

```
$ ssh-keyscan -t rsa ssh-host.example.com > ssh_known_hosts
```

If you're worried about attackers, compare it to the value in your existing `.ssh/known_hosts`. Some services like GitHub also post their SSH public key fingerprint, so you can check the value out-of-band.

Then in your `Dockerfile` you can copy in the `known_hosts` file:

```
RUN mkdir /root/.ssh && chmod 700 /root/.ssh
COPY ssh_known_hosts /root/.ssh/known_hosts
RUN chmod 600 /root/.ssh/known_hosts
```

### 2.11.1 References

- Automatically add new host to SSH known_hosts (serverfault.com)
- OpenSSH documentation (openssh.com)
- GitHub SSH public key fingerprints (help.github.com)

# 3 Best practices: Running in CI

## 3.1 Add a smoke test for your image

Your application might have bugs, and you'll have unit tests and end-to-end tests to catch those. But you might also have issues with your Docker image: it might not start at all, for example.

So before you push your newly built image to the image registry, you should implement a smoke test. For example, if your image is a web server, you can run the newly built image and make sure you can send a successful HTTP query to the status endpoint:

```
import time
from subprocess import check_call
from urllib.request import urlopen

check_call(
    "docker run --rm --name=mycontainer -p 8080:80 -d httpd".split()
)
# Wait for the server to start. A better implementation would
# poll in a loop:
time.sleep(5)
# Check if the server started (it'll throw an exception if not):
try:
    urlopen("http://localhost:8080").read()
finally:
    check_call("docker kill mycontainer".split())
```

The smoke test won't catch all problems, but it will ensure you don't push a completely broken image.

Notice the use of `--rm`, to ensure you're not leaking containers. In a build system that spins up an empty environment each time this won't matter, but in a persistent

setup you don't want to leak resources.

### 3.1.1 References

- Your Docker build needs a smoke test (pythonspeed.com)

## 3.2 Additional checks: `hadolint`, size checks

You can do additional checks on both your `Dockerfile` and image.

First, the `hadolint Dockerfile` linter will catch some problems, though very definitely not all.

Second, it's useful to check your image size isn't any higher than a particular value: if you expect images to be 300-400MB and they're suddenly 1GB, something has gone wrong. You can get the image size by running:

```
$ docker image inspect --format={{.Size}} yourorg/yourimage
237117212
```

If you're current image is 250MB, check for 300MB; better to be a little lenient so you don't get spurious failures.

### 3.2.1 References

- `hadolint` (github.com)
- `docker image inspect` help (docs.docker.com)

## 3.3 Tag images based on the version control branch

You might have a CI system that automatically builds images from Git branches:

1. A developer pushes to branch `mybranch`.

2. The CI/build system automatically builds a new image, and pushes it to an image registry.

You want to ensure that a developer working on a feature branch won't accidentally overwrite the stable image used in production. The simplest solution is to have the CI script choose the image tag based on the branch.

While you're at it, it's worth tagging based on Git commit so you can also distinguish different images on the same branch:

```
GIT_BRANCH=$(git rev-parse --abbrev-ref HEAD)
GIT_COMMIT=$(git rev-parse --short HEAD)
IMAGE_NAME="imageregistry.example.com/org/theapp"
docker build -t "$IMAGE_NAME:$GIT_BRANCH" \
             -t "$IMAGE_NAME:$GIT_COMMIT" .
docker push "$IMAGE_NAME:$GIT_BRANCH" "$IMAGE_NAME:$GIT_COMMIT"
```

## 3.4 Don't rely on the `latest` tag

Using the `latest` tag as your main branch name is problematic:

- Its name is confusing, it's not actually the latest image. From Docker's perspective it's just the default tag, it may well be an old image.
- It's easy to overwrite by mistake if you're doing manual pushes, by forgetting to omit the tag.
- If you're using Kubernetes, it behaves differently for the `latest` tag vs other tags, which can lead to confusing behavior.

If you want a default tag for production, better to choose some other name.

### 3.4.1 References

- What's wrong with the Docker `:latest` tag? (vsupalov.com)

- Kubernetes `imagePullPolicy` interaction with `latest` (kubernetes.io)

## 3.5 Warm up the build cache

If you're building your Docker image in your CI/build setup (GitLab CI, Jenkins, GitHub Actions, Azure Pipelines, etc.), in many cases each build will with an empty local image cache. So you'll want to make sure the previous version of the image is available locally so that it can be used when rebuilding. Otherwise your build might not use cached layers, and will always rebuild from scratch.

You'll want to pull the image, and then use `--cache-from` to make sure Docker uses it:

```
#!/bin/bash
set -euo pipefail
docker pull yourimage || true
docker build -t yourimage --cache-from=yourimage .
docker push yourimage
```

The `|| true` allows you to keep going even if the pull fails; this will be the case the first time you create the image.

## 3.6 Warm up the build cache for per-branch builds

If you're building multiple branches in parallel, and tagging images correspondingly, warming the cache won't work the first time you build a new branch, even though it's identical to your main branch. That is, if you do `docker pull yourimage:newbranch` and this is your first build of `newbranch`, that base image won't be available. The solution is to pull and use `--cache-from` on multiple tags, both the branch and your default build.

Presuming your default production build is tagged with `production`:

```
#!/bin/bash
set -euo pipefail

BRANCH=$(git rev-parse --abbrev-ref HEAD)

docker pull yourimage:production || true
docker pull yourimage:$BRANCH || true
docker build -t yourimage:$BRANCH \
    --cache-from=yourimage:production \
    --cache-from=yourimage:$BRANCH .
docker push yourimage:$BRANCH
```

## 3.7 Optional: Self-warming cache with BuildKit

As an alternative to explicitly pulling the previous version of an image, you can also have Docker automatically pull only those layers it actually needs. In theory this is faster because layers that can't be reused won't need to be pulled.

You can do this using BuildKit, the alternative and improved Docker build backend. There are two necessary steps:

1. Enable BuildKit in general using the `DOCKER_BUILDKIT` environment variable.
2. Build the image with inline caching metadata by passing in the `BUILDKIT_INLINE_CACHE` build arg.

The build script will look like this:

```
set -euo pipefail
export DOCKER_BUILDKIT=1
docker build -t yourimage --cache-from=yourimage \
       --build-arg BUILDKIT_INLINE_CACHE=1 .
docker push yourimage
```

Notice there is no need for a `docker pull`.

### 3.7.1 References

- External cache sources in `docker build` (docs.docker.com)

# 4 Best practices: Make debugging easier

## 4.1 Write logs to `stdout` or `stderr`

In order to debug problems in a running application, you'll want to make sure the logs from your application are captured.

Docker runtime environments will capture logs from `stdout` and `stderr`, so just make sure that's where your logs go. You can then read the logs using tools like `docker logs` or `kubectl logs`, depending how you're running your image, or redirect your logs elsewhere.

With Python's built-in `logging` library, you can do:

```
import sys, logging
logging.basicConfig(stream=sys.stdout)
```

## 4.2 Prepare for C crashes

If your Python program crashes due to a segfault or some other bug in C code, you won't get a traceback by default. And silent crashes are hard to debug.

To fix this, set the `PYTHONFAULTHANDLER` environment variable in your `Dockerfile`:

```
ENV PYTHONFAULTHANDLER=1
```

And now you'll get tracebacks from C crashes:

```
$ docker run -it crasher
About to crash...
Fatal Python error: Segmentation fault

Current thread 0x00007f2f75f98740 (most recent call first):
  File "/usr/local/lib/python3.7/ctypes/__init__.py", line 505 in string_at
```

```
  File "crash.py", line 3 in crash
  File "crash.py", line 5 in <module>
```

### 4.2.1 References

- Python's `faulthandler` library (docs.python.org)

## 4.3 Record the build's version control revision and branch

It's useful to know what exact revision of your application is running in production—or on your laptop—when you're trying to reproduce a problem. You should therefore record the revision of your application's source code, as well as the branch, in the image itself.

First, you can do this by adding metadata labels to the image:

```
$ GIT_COMMIT=$(git rev-parse --short HEAD)
$ GIT_BRANCH=$(git rev-parse --abbrev-ref HEAD)
$ docker build -t myimage:latest --label git-commit=$GIT_COMMIT --label
↪  git-branch=$GIT_BRANCH .
```

You can inspect these labels using `docker inspect myimage`.

Second, you can also pass the git commit and branch into your image, by using the `ARG` command in your `Dockerfile`:

```
FROM python:3.8-slim-buster
ARG git_commit
RUN echo $git_commit > /git-commit.txt
```

And then passing the information in using `--build-arg`:

```
$ docker build -t myimage --build-arg git_commit=$GIT_COMMIT .
```

You can then have the status API in your web application include this information, where it can be read by your monitoring infrastructure.

By default build arguments are exposed as environment variables only during the build. If you want the `ARG` to be available as an environment variable at runtime, you can do:

```
ARG git_commit
ENV git_commit=$git_commit
```

### 4.3.1 References

- What's running in production? Making your Docker images identifiable (pythonspeed.com)
- Docker image labels (nickjanetakis.com)
- Docker build arguments (docs.docker.com)
- `git rev-parse` documentation (git-scm.com); personally I rely on StackOverflow search results because Git is so awful

## 4.4 Optional: Pre-install useful tools

Unless you're super-worried about security, it's useful to have a few common debugging tools installed on your image. Since you shouldn't run as root, you won't be able to install these packages once the container is running.

For example, if you're using the `slim` variants of the official Python image, you'll want to install packages like `procps` and `nettools` so you have access to `ps` and `netstat`:

```
RUN apt-get update && apt-get -y install procps nettools
```

Another useful tool, albeit with potential security risks, is the `manhole` library: it gives you a Python prompt into your running process. See the reference for details.

### 4.4.1 References

- A Python prompt into your runner process: debugging with Manhole (pythonspeed.com)
- `manhole` documentation (python-manhole.readthedocs.io)

# 5 Best practices: Correct operation

## 5.1 Have public ports listen on `0.0.0.0`

If your server listens on `127.0.0.1`, you won't be able to access it from the outside of the container. Unless this is your goal, make sure to listen on `0.0.0.0` so that it binds to the container's external IP.

### 5.1.1 References

- Connection refused? Docker networking and how it impacts your image (pythonspeed.com)

## 5.2 Avoid bash, or at least use bash strict mode and `shellcheck`

Shell scripting is a recipe for failure. Whereas errors in a Python script will cause an exception, in shell scripts the default is to silently continue. The following script, for example, will print "Success!", which is probably not what you want.

```
#!/bin/bash
export VAR=$(echo hello | nonexistentprogram)
echo "Success!"
```

You can make many errors cause the script to stop by using bash strict mode (`set -euo pipefail`), but even that won't work in this case:

```
#!/bin/bash
set -euo pipefail
export VAR=$(set -euo pipefail; echo hello | nonexistentprogram)
echo "Success!"
```

This will still print "Success!". Do you know how to fix that?

There are two solutions to the limits of shell scripting:

1. Use `set -euo pipefail` at the of every `bash` script, and use the `shellcheck` tool to lint your shell script.
2. Replace the shell script with a Python script, since you already have Python installed.

### 5.2.1 References

- bash strict mode (redsymbol.net)
- The `shellcheck` shell linter (shellcheck.net)

## 5.3 Ensure fast shutdowns

If you don't configure your image correctly, signals won't be delivered to your process, and shutdowns will take 10 seconds: first the original signal will be used, then after a timeout SIGKILL will be used, shutting it down with extreme prejudice (the same way `kill -9` does).

To get signal delivery working:

1. Use the `[]` syntax of `ENTRYPOINT` and `CMD`, not the shell syntax.
2. If your entrypoint script is a shell script, make sure it ends by using the shell `exec` command to run your final program. This will replace the shell process with your program. In Python you can use `os.execve` or one of the related functions.

For example, let's say you have a `Dockerfile` that runs a script called `entrypoint.sh`. The shell script should look like this:

```
#!/bin/bash
set -euo pipefail

# BAD:
# python myserver.py
```

```
# GOOD:
exec python myserver.py
```

And the `Dockerfile`:

```
# BAD:
# ENTRYPOINT ./entrypoint.sh

# GOOD:
ENTRYPOINT ["./entrypoint.sh"]
```

Additionally, there are two other problems you might encounter:

1. If you're using a shell script there's another failure mode involving pipes: don't use them on the final command in your entrypoint script.
2. If your program expects a signal other than `SIGTERM`, use the `STOPSIGNAL` `Dockerfile` command, e.g. `STOPSIGNAL INT` to have `docker stop` deliver a `SIGINT`.

### 5.3.1 References

- Why your Dockerized application isn't receiving signals (hynek.me)
- `STOPSIGNAL` reference (docs.docker.com)
- `os.exec*` documentation (docs.python.org)

## 5.4 Add an `init` process

Unix systems are designed to have an `init` process with PID 1 to help deal with existing processes. If you're running subprocesses in your image you'll therefore want to ensure you have one setup.

With Docker you can do this on the command-line when you run an image (`docker`

`run --init`). Not all runtime environments have this option, though, so it's likely better to include one in your `Dockerfile`:

```
FROM python:3.8-slim-buster
RUN apt-get update && apt-get install -y tini
COPY your-entrypoint.sh .
ENTRYPOINT ["tini", "--", "./your-entrypoint.sh"]
```

### 5.4.1 References

- The `tini` init process (github.com)

## 5.5 Set a non-interactive frontend for Debian package installs

When installing packages via `apt-get` on Debian, Ubuntu, or Debian-based images like the official `python` image, you don't want the build trying to ask you questions about how to configure the packages you are installing. You do so by setting the `DEBIAN_FRONTEND` environment variable to `noninteractive`.

One way to do so is like this:

```
RUN export DEBIAN_FRONTEND=noninteractive && \
    apt-get install -y gcc
```

## 5.6 Add health checks

Your running container may lock up or stop working due to bugs or other problems. Most container runtimes can therefore monitor running containers and check if they're still alive—but only if you configure health checks.

There are at least two different ways to configure health checks, depending on your runtime environment:

1. The `Dockerfile` format supports one kind, which is used by tools like Docker Swarm.
2. Kubernetes has its own mechanism, and doesn't support the one in `Dockerfile`.

Here's an example of the `Dockerfile` variant, which uses the `HEALTHCHECK` command:

```
FROM python:3.8-slim-buster
HEALTHCHECK --interval=20s --timeout=3s \
  CMD ["python", "-c", \
       "from urllib.request import urlopen; \
        urlopen('http://localhost:8000').read()"]
ENTRYPOINT ["python", "-m", "http.server"]
```

### 5.6.1 References

- `Dockerfile`'s `HEALTHCHECK` command (docs.docker.com)
- Kubernetes health checks (kubernetes.io)
- docker-autoheal is a tool that will automatically restart Docker containers that failed their healthcheck.

## 5.7 Pre-compile bytecode for faster startup

Python compiles source code to `.pyc` files, the corresponding bytecode. If `.pyc` files aren't available in your Docker image, your application will need to compile them, and this can lead to much slower startup. This leads to a tradeoff between startup performance and image size:

- If you want your container to start as quickly as possible, you will want to pre-compile the `.pyc` files.
- If you want your image to be as small as possible, you want to have no `.pyc` files at all in your image.

Presuming you want fast startup, add the following to the end of your `Dockerfile` to create `.pyc` files for all installed packages:

```
RUN python -c "import compileall; compileall.compile_path(maxlevels=10)"
```

If you also have code in the current directory you want to compile, you can compile it like so:

```
RUN python -m compileall yourpackage/
```

### 5.7.1 References

- The `compileall` module (docs.python.org)

# 6 Best practices: Reproducible builds

Imagine you start with a certain revision of your source code and build a Docker image. A month later, you start with the same revision, fix a minor bug, and build a new image from scratch.

If your build is not reproducible, you might end up installing different versions of your Python dependencies, system packages, and perhaps even a different version of the operating system. The resulting image might have new bugs, behave in unexpected ways, or even fail to work completely due to incompatible changes. A minor bug fix has now spiraled out of control.

But if your build is reproducible, your new image will be mostly the same as your old image: the only difference will be the bug fix.

Your goal then is to have a reproducible build: the same inputs should result in the same output.

## 6.1 Choose a stable base image and tag

When choosing a base image for your `Dockerfile`, you will likely want:

- **Stability:** You want a build today to give you the same basic set of libraries, directory structure, and infrastructure as a build tomorrow, otherwise your application will randomly break.
- **Security updates:** You want the base image to be well-maintained, so that you get security updates for the base operating system in a timely manner.
- **Up-to-date dependencies:** Unless you're building a very simple application, you will likely depend on operating system-installed libraries and applications, for example a C compiler. Ideally these dependencies would be fairly modern.
- **Extensive dependencies:** Some applications will require less popular dependencies—a base image with access to a large number of libraries makes this easier.

- **Up-to-date Python:** Having an up-to-date Python available out of the box saves you some effort.
- **Small images:** All things being equal, it's better to have a smaller Docker image than a bigger Docker image.

Ubuntu Long Term Support (LTS) releases, Debian Stable, and CentOS are all reasonable candidates, since they aim for backwards compatibility while still providing security updates and critical bug fixes. As I discuss later on, I recommend avoiding Alpine Linux.

When you choose a base image, make sure to specify a particular release using a tag:

- **Bad:** `FROM ubuntu`—today this might be Ubuntu 19.10, tomorrow it might be Ubuntu 20.04.
- **Good:** `FROM ubuntu:18.04`, `FROM debian:10` aka `FROM debian:buster`, `FROM centos:8`.

My recommendation is to use the "official" Docker Python image, which is based on Debian Stable but still allows you to get different versions of Python. In particular, you should specify which Debian Stable release you want, the most-up-to-date being `buster`. And I also recommend using the `slim` variant to get a smaller base image.

Depending how strict you want to be, you can choose different levels of reproducibility:

- `python:3.8-slim-buster` is the latest sub-release of Python 3.8, installed on top of Debian "Buster" 10. At the time of writing this will be 3.8.1, later it will be 3.8.2, and so on.
- `python:3.8.1-slim-buster` is a specific sub-release, Python 3.8.1. This image might still change over time, however; it might be updated with newer releases of `pip`, for example, or newer system packages.
- `python@sha256:89d719142de465e7c80195dff820a0bbbbba49b148fbd97abf4b58889372b5e3` is a specific image, unchanging even if the tags get pointed at new images.

There's no guarantee the Hub will keep old images around though, so you may wish to copy the image into your registry, or even create your own custom base image. The latter is covered below.

### 6.1.1 References

- The "official" Python Docker image (hub.docker.com)
- Pulling an image by digest (docs.docker.com)
- Choosing a base image for your Python application (pythonspeed.com)

## 6.2 Pin your Python dependencies

If you run:

```
RUN pip install flask
```

You will get one version today, and potentially a very different version in 6 months. So when you install Python dependencies you want to install "pinned" versions, specific versions of the package and its transitive dependencies; `flask` depends on other libraries, and we want them to be pinned too.

In general you want to maintain two lists of dependencies:

1. The high-level dependencies of your application, the particular libraries you're importing. For example, `flask` and `pandas`.
2. The pinned transitive dependencies, which you use to install the dependencies as part of the Docker build.

The high-level dependencies are used to regenerate the transitive dependencies; see below in the section on updating dependencies.

There are three tools for handling these two sets of dependency files: `pip-tools` (the simplest of the three), `poetry`, and `pipenv`. As of January 2020, `pipenv` development seems stalled; it hasn't had a release since November 2018.

With `pip-tools`, for example, you would have a `requirements.in` file that looks like this:

```
flask
pandas
```

You would then compile it to a `requirements.txt` by running `pip-compile --generate-hashes requirements.in`. Hashes are useful to ensure you're getting the exact same package, and that it hasn't been replaced on PyPI by a malicious attacker. The resulting `requirements.txt` would look like this:

```
click==7.0 \
    --
↪   hash=sha256:2335065e6395b9e67ca716de5f7526736bfa6ceead690adf616d925bdc622b13
↪   \
    --
↪   hash=sha256:5b94b49521f6456670fdb30cd82a4eca9412788a93fa6dd6df72c94d5a8ff2d7
↪   \
    # via flask
flask==1.1.1 \
    --
↪   hash=sha256:13f9f196f330c7c2c5d7a5cf91af894110ca0215ac051b5844701f2bfd934d52
↪   \
    --
↪   hash=sha256:45eb5a6fd193d6cf7e0cf5d8a5b31f83d5faae0293695626f539a823e93b13f6
itsdangerous==1.1.0 \
    --
↪   hash=sha256:321b033d07f2a4136d3ec762eac9f16a10ccd60f53c0c91af90217ace7ba1f19
↪   \
    --
↪   hash=sha256:b12271b2047cb23eeb98c8b5622e2e5c5e9abd9784a153e9d8ef9cb4dd09d749
↪   \
```

```
    # via flask
...
```

Your `Dockerfile` would do:

```
RUN pip install -r requirements.txt
```

### 6.2.1 References

- Dear package managers: dependency resolution results should be in version control (blog.ometer.com)
- Faster Docker builds with pipenv, poetry, and pip-tools (pythonspeed.com)
- `pip-tools` documentation (github.com)
- `poetry` documentation (python-poetry.org)
- `pipenv` documentation (pipenv.kennethreitz.org)

## 6.3 ∞ Update all pinned dependencies once a month ∞

As we discussed above, in order to ensure reproducible builds you want to keep the following from changing by pinning them to specific versions:

1. Base image.
2. System packages (optional).
3. Version of Python.
4. Python dependencies.

It can be tempting to leave these versions unchanged for long periods of time, to ensure a stable baseline for your application. This would be a mistake:

1. You need to get security updates and other critical bug fixes that you've missed.
2. A series of small upgrades are much safer and easier than one massive upgrade.

To expand on the second point: if you only upgrade dependencies once a year, you now potentially have a new base operating system, a new version of Python, and major changes to three libraries you depend on. If these updates cause your program to have issues, it can be difficult to figure out what caused them.

It's also difficult to convince management that you should spend a week upgrading your dependencies; what about all those features and bug fixes on the product plan?

On the other hand, if every month you update your dependencies, you will only be changing one or two things at a time. That means problems can be easily pinpointed: if you've only updated Flask this month, it's clear what caused the regression in your application.

What's more, the chunks of time you spend on these upgrades will also be much shorter, causing less disruption to other work.

This doesn't mean that every time a new major, incompatible release comes out you should immediately rewrite your software. "Once a month" is a starting point, not the correct timespan for every dependency. But if you decide not to upgrade a dependency for now, make it an explicit decision with an explicit and ideally short-term deadline for when you will upgrade.

## 6.4 Optional: Pin system packages

One of the benefits of using a stable base operating system like Debian, Ubuntu LTS, or CentOS is compatibility over time. As long as you stick to a major release, the maintainers will try to release critical bug fixes and security updates to libraries without making incompatible changes.

This is the theory.

In practice, that might not be good enough for you. If you really want to ensure specific package versions get installed, instead of doing:

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

You can install a specific release:

```
RUN apt-get install -y nginx=1.14.2-2+deb10u1
```

On CentOS, `dnf` and `yum` support a similar syntax.

If you want to be even more paranoid about stability, you may want to look into Nix, which is a completely different approach to software packaging.

### 6.4.1 References

- APT User's Guide (debian.org)
- DNF Documentation (dnf.readthedocs.io)
- The Nix package manager (nixos.org)

## 6.5 Optional: Create a custom base image

If your `Dockerfile` looks like this:

```
FROM python:3.8-slim-buster
RUN apt-get update && apt-get -y upgrade

# Your actual application:
# ...
```

Then rebuilding your image from scratch with security updates once a week will lead to a lack of reproducibility: the base image will change over time, as will the security updates.

You can just assume that security updates won't make too much of a semantic difference, and live with it. Or, you can create a custom base image to ensure repro-

ducibility.

Here is an example of how you might do that; you might need to modify this scheme to meet your particular workflow. You create a `Dockerfile.base` that looks like this:

```
FROM python:3.8-slim-buster
RUN apt-get update && apt-get -y upgrade
# ...whatever else you want in a base image...
```

And then build it with both a tag and label that store some permanent identifier, the current date for example. You make sure never to change the image for the tag once it's created:

```
$ TODAY=2020-05-29
$ docker build -f Dockerfile.base -t yourorg/baseimage \
    -t yourorg/baseimage:$TODAY --label BASE_BUILT=$TODAY .
$ docker push yourorg/baseimage
$ docker push yourorg/baseimage:$TODAY
```

Now you can build your normal image using that base image:

```
FROM yourog/baseimage
COPY requirements.txt .
RUN pip install -r requirements.txt
# ... etc ...
```

If you ever need to know which base image was used to create the image, you can just look for the label:

```
$ docker inspect myapp | grep BASE_BUILT
       "BASE_BUILT": "2020-05-29"
```

And you can always rebuild with that exact version of the base image if need be:

```
FROM yourorg/baseimage:2020-05-29
COPY requirements.txt .
RUN pip install -r requirements.txt
# ... etc ...
```

# 7 Best practices: Faster builds

## 7.1 Don't use Alpine Linux

When installing Python packages from PyPI, `pip` can usually speed up installation by downloading pre-compiled binary wheels provided by the package maintainers. However, these wheels don't work on Alpine Linux, which means you have to rebuild every single package yourself.

For example, if you want to install `pandas` and `matplotlib` from PyPI, build time will go from ~30 seconds on a Debian-based image to ~1500 seconds on Alpine, a 50× increase. Alpine Linux has other issues as well; see the reference below for details.

### 7.1.1 References

- Using Alpine can make Python Docker builds 50× slower (pythonspeed.com)

## 7.2 `COPY` in files only when needed

To a first approximation, each command in the `Dockerfile` creates a new layer in the new image. Docker builds can use cached layers to speed up the build: if the command hasn't changed, or input files haven't changed if you're doing a `COPY`, the cached layer can be reused.

If a layer can't be loaded from the cache, none of the later steps in the `Dockerfile` can use caching either. So you want to ensure you don't invalidate the cache unnecessarily.

Consider the following `Dockerfile`:

```
FROM python:3.8-slim-buster
COPY requirements.txt .
RUN apt-get update && apt-get install -y gcc
RUN pip install -r requirements.txt
```

If `requirements.txt` changes, that will invalidate the `apt-get` command too, even though `apt-get` doesn't need `requirements.txt`. Better to copy in the file only when you need it:

```
FROM python:3.8-slim-buster
RUN apt-get update && apt-get install -y gcc
COPY requirements.txt .
RUN pip install -r requirements.txt
```

More broadly, you'll want commands that don't depend on `COPY`, like `apt-get` or `dnf`, to run before commands that do.

### 7.2.1 References

- Faster or slower: the basics of Docker build caching (pythonspeed.com)

## 7.3 Use `ARG` only when needed

Just like changes to files can invalidate the cache, so can changing build arguments passed in to a `ARG` command. So much like you want to only `COPY` in files when you actually need them, you also only want to add the `ARG` to the `Dockerfile` at the point where you'll actually be using it.

For example, let's say you want to pass in the Git commit to the build. The Git commit will change on every build, so don't do the following or you'll completely disable build caching:

```
FROM python:3.8-slim-buster
# Bad location, it's not used yet and will invalidate cache:
ARG git_commit
RUN apt-get install ...
RUN pip install ...
RUN echo $git_commit > /var/run/git-commit.txt
```

Instead, do this:

```
FROM python:3.8-slim-buster
RUN apt-get install ...
RUN pip install ...

# Good location, won't invalidate cache of apt-get or pip:
ARG git_commit
RUN echo $git_commit > /var/run/git-commit.txt
```

## 7.4 Install dependencies separately from your code

Expanding on the previous point, if you're installing your Python application's dependencies via `setup.py`, any change to your application code will invalidate the list of Python dependencies you installed. So that means downloading and installing all the packages from PyPI again.

Instead of listing dependencies in `setup.py`, you should list dependencies in a separate file, either `requirements.txt` or the configuration files used by `poetry` or `pipenv`. Then you can install 3rd party libraries first and have that layer cached:

```
FROM python:3.8-slim-buster
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
RUN pip install .
```

For Poetry, you can do it like this:

```
FROM python:3.8-slim-buster
RUN pip install poetry
```

```
WORKDIR /tmp/myapp
COPY pyproject.toml poetry.lock .
RUN cd /tmp && poetry install --no-root
# Copy in everything else:
COPY . .
RUN poetry install
```

For Pipenv:

```
FROM python:3.8-slim-buster
RUN pip install pipenv
COPY Pipfile* /tmp
RUN cd /tmp && pipenv lock --requirements > requirements.txt
RUN pip install -r /tmp/requirements.txt
COPY . /tmp/myapp
RUN pip install /tmp/myapp
CMD flask run exampleapp:app
```

### 7.4.1 References

- `requirements.txt` file format (pip.pypa.io)
- `poetry` documentation (python-poetry.org)
- `pipenv` documentation (pipenv.kennethreitz.org)

# 8 Best practices: Small images

## 8.1 Find large layers and files with `dive`

To figure out why your image is too large, you can use `docker image history yourimage`, but that just tells you which layers are using a lot of disk space. The `dive` tools shows you per-layer sizes, and shows the differences and what files were added, so you can get a much more nuanced view of where image size is coming from.

### 8.1.1 References

- `dive` (github.com)

## 8.2 Add files to `.dockerignore`

When you run `docker build`, all files in the source directory are copied in to the context. The context is then used as the source of files copied in with the `COPY` or `ADD` commands. Usually you'll pass in the current directory as the source directory.

This has some issues:

1. If there are many large files, coping them into the context can slow down your builds.
2. If you're `COPY`ing whole directories, you might copy files you didn't mean to, which can mean larger images or even leaking secret information.
3. Files you don't care about may invalidate the cache.

The solution is to create a `.dockerignore` file, listing files you don't want to be included in the context and therefore the final Docker image. For example:

```
Dockerfile
.dockerignore
venv/
```

```
**/__pycache__
```

### 8.2.1 References

- `.dockerignore` reference (docs.docker.com)

## 8.3 Avoid extra chowns

Any time you modify a file in any way in the `Dockerfile`, a whole new copy is stored in the next layer. Recursive `chown`s can therefore result in very large images, since you're duplicating every file you change.

Therefore, instead of doing:

```
COPY code .
RUN chown -R youruser code
```

You should instead do:

```
COPY --chown=youruser code .
```

### 8.3.1 References

- `Dockerfile` `COPY` reference (docs.docker.com)

## 8.4 Don't install unnecessary system packages, and clean up when you're done

When you install system packages, whether DEB or RPM, the package manager will typically keep a copy of the original package around, and also store the package listing from the package index. This wastes space.

You can't delete these unneeded files in a separate `RUN` because Docker layers are always additive. So the solution is to have a single shell script that installs the packages, and then cleans up unnecessary files.

Additionally, when you install a package, non-essential but recommended dependencies may get installed, wasting space. Instead, you want to install only the minimal necessary dependencies to ensure smaller images.

For Debian/Ubuntu systems, the following shell script will do the trick:

```
#!/bin/bash
set -euo pipefail

# Tell apt-get we're never going to be able to give manual
# feedback:
export DEBIAN_FRONTEND=noninteractive

# Update the package listing, so we know what package exist:
apt-get update

# Install security updates:
apt-get -y upgrade

# Install a new package, without unnecessary recommended packages:
apt-get -y install --no-install-recommends YOUR_PACKAGE_LIST_GOES_HERE

# Delete cached files we don't need anymore:
apt-get clean
rm -rf /var/lib/apt/lists/*
```

For RPM-based systems like CentOS and RHEL:

```
#!/bin/bash
set -euo pipefail

# Install security updates, bug fixes and enhancements only:
dnf -y upgrade-minimal

# Install a new package, without unnecessary recommended packages:
dnf -y install --setopt=install_weak_deps=False YOUR_PACKAGE_LIST

# Delete cached files we don't need anymore:
dnf clean all
```

### 8.4.1 References

- Installing system packages in Docker with minimal bloat (pythonspeed.com)
- APT User's Guide (debian.org)
- DNF Documentation (dnf.readthedocs.io)

## 8.5 Disable pip caching

By default `pip` keeps a copy of downloaded packages. This wastes space.

To fix this, run `pip` with caching disabled:

```
FROM python:3.8-slim-buster
RUN pip install --no-cache-dir flask
```

# 9 Best practices: Application and tool-specific

## 9.1 Using a virtualenv in your `Dockerfile`

The `activate` script included in the virtualenv isn't very usable from a `Dockerfile`, because each `RUN` is a separate shell session. Instead, to activate a virtualenv, you just need to set two environment variables:

```
ENV VIRTUAL_ENV=/opt/venv
RUN python -m venv $VIRTUAL_ENV
ENV PATH="$VIRTUAL_ENV/bin:$PATH"
```

### 9.1.1 References

- Elegantly activating a virtualenv in a Dockerfile (pythonspeed.com)

## 9.2 Using a Conda environment in your `Dockerfile`

Activating a Conda environment is a much more complex task than activating a virtualenv, because environment variables won't work. The key is to use `conda run`, which allows you to run a command inside a specified environment:

```
FROM continuumio/miniconda3

WORKDIR /app

# Create the environment:
COPY environment.yml .
RUN conda env create -f environment.yml

# Make RUN commands use the new environment:
SHELL ["conda", "run", "-n", "myenv", "/bin/bash", "-c"]
```

```
# ... the rest of the build ...

# The code to run when container is started:
COPY run.py .
ENTRYPOINT ["conda", "run", "-n", "myenv", "python", "run.py"]
```

### 9.2.1 References

- Activating a Conda environment in your Dockerfile (pythonspeed.com)

## 9.3 Don't run database upgrades on startup

Your application might require database schema upgrades. You should not, however, run those schema upgrades as part of your container's startup:

1. Most schema management systems will break if you do the same schema upgrade concurrently, and it's common to run multiple copies of the same container.
2. It will encourage you to tightly couple your code to the database schema, making schema rollbacks difficult or impossible.

Instead, you should decouple the two. The usual solution for this is to make your schema changes purely additive, e.g. adding columns but not deleting them:

1. Migrate from schema S to schema S+1, with only additive changes.
2. Over time upgrade some of your processes from application version V to V+1.
3. Eventually everything is on V+1, and you don't ever expect to rollback to V.
4. Finally, migrate from schema S+1 to S+2, and now you can do destructive schema changes to anything that V+1 no longer uses.

### 9.3.1 References

- Decoupling database migrations from server startup: why and how (python-speed.com)
- Flyway (flywaydb.org) does support concurrent schema upgrades, unlike most tools
- Evolutionary Database Design (martinfowler.com)

## 9.4 Make sure slow queries don't break health checks

If you've implemented health checks for your server, the runtime environment is occasionally sending queries to your server to see if it's still alive. An HTTP server will get HTTP queries, for example.

Now, imagine you have a single-threaded HTTP server that can only handle one query at a time. And this server sometimes takes a long time to answer queries, and in particular it's blocking while handling that query. Consider the following timeline:

1. A slow query is sent to the server. The server starts processing, but doesn't respond yet.
2. The runtime environment sends a liveness check to server, and gets no response because slow query is still running.
3. The runtime environment sends another liveness check.
4. The runtime environment sends yet another liveness check.
5. The runtime environment decides the server is dead and kills it before it can finish responding to the slow query.

If you expect slow queries, you should configure your application with enough concurrency that it can handle health check liveness queries. For typical WSGI web applications this can be done by running multiple threads, or multiple processes, depending on the application configuration.

## 9.5 Enable running other commands via the command-line

The combination of `ENTRYPOINT` and `CMD` allows you to have default command-line arguments for your application, and allows users to run your container with different command-line options.

In some cases, however, you never expect the user to pass in any command-line options to the entrypoint. Instead, you want them to be able to run a *different* program when they give command-line options. This can be useful for debugging, for example.

Thus by default the official `python` image runs Python:

```
$ docker run -it python:3.8-slim-buster
Python 3.8.3 (default, Jun  9 2020, 17:49:41)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

But you can pass it other commands instead, in this case `bash`:

```
$ docker run -it python:3.8-slim-buster \
    bash -c 'echo hello, I am $(whoami)'
hello, I am root
```

You can do this with the following combination of `ENTRYPOINT` and `CMD` in your `Dockerfile`:

```
ENTRYPOINT []
CMD ["/app/entrypoint.sh"]
```

### 9.5.1 References

- `ENTRYPOINT` reference (docs.docker.com)
- `CMD` reference (docs.docker.com)

## 9.6 Configure the Gunicorn web server correctly

In addition to the need to support multiple threads (see above), Gunicorn uses `/tmp` as a default location for its internal heartbeat system for communicating with workers. On Docker that is an on-disk filesystem, which means the heartbeat system can be slow, making Gunicorn unresponsive.

You should therefore use `/dev/shm`, an in-memory filesystem, as the location for the heartbeat file.

A working configuration where logs also go to stdout might therefore look like this:

```
#!/bin/bash
set -euo pipefail
exec gunicorn --worker-tmp-dir /dev/shm --workers=2 \
              --threads=4 --worker-class gthread \
              --log-file=- ...
```

### 9.6.1 References

- Configuring Gunicorn for Docker (pythonspeed.com)
- Gunicorn documentation (docs.gunicorn.org)

## 9.7 Configure the uWSGI web server correctly

Configuring uWSGI is complex.

In addition to the need to support multiple threads and logging to `stdout` (see above), uWSGI has some oddities. Just a couple worth mentioning:

1. By default it will `fork()` workers without `exec()`ing a new Python process, which supposedly saves memory but actually just makes your Python program much more likely to crash if you use the wrong library.

2. The default behavior for environment dictionaries doesn't match the WSGI standard.

You can fix these issues in the `uwsgi.ini` file:

```
[uwsgi]
lazy-apps = true
wsgi-env-behaviour = holy
```

The Tech at Bloomberg article referenced below on uWSGI in production doesn't even mention these two, but does mention many other details you need to configure to get uWSGI working correctly.

While it's true that uWSGI is fast, in most web applications the bottleneck is the database, not the web server. And personally I feel the `fork()`-without-`exec()` design choice shows such bad judgment that I don't trust uWSGI, but that's just me; it seems like it works for some people with enough configuration tweaks.

Technically Gunicorn does an `exec()` without `fork()` too, but it does it much earlier in the process, before you load your code, so it's less likely to be an issue. Other WSGI containers may be better than either.

### 9.7.1 References

- uWSGI Preforking and Lazy Apps (engineering.ticketea.com)
- uWSGI env behaviour policies (uwsgi-docs.readthedocs.io)
- Configuring uWSGI for Production Deployment (techatbloomberg.com)

## 9.8 Respect the `PORT` environment variable on Heroku and Google Cloud Run

If you're deploying your image on Heroku or Google Cloud Run, you will be expected to have your server listen on a port number specified in the `PORT` environment vari-

able.

Note that just like binding to `0.0.0.0`, the port to listen to is specific to your server. Gunicorn, uWSGI, etc. will all have different command line or configuration file options–read the relevant documentation for your server.

As an example, you'll probably want an `entrypoint.sh`. The `${PORT:-8080}` means that if no `$PORT` environment variable was set, a default value of `8080` will be used:

```
#!/bin/bash
set -euo pipefail
exec python -m http.server ${PORT:-8080}
```

The `Dockerfile` will then use this entrypoint:

```
FROM python:3.8-slim-buster
COPY entrypoint.sh
ENTRYPOINT ["./entrypoint.sh"]
```

Heroku also requires you to run as a non-root user.

### 9.8.1 References

- Heroku's Docker image requirements (devcenter.heroku.com)
- Google Cloud Run's Docker image requirements (cloud.google.com)

# 10 Best practices: Multi-stage builds

## 10.1 Omit build dependencies from your runtime image

If you need to use a compiler to build some of your packages, installing the compiler will make your image much bigger. But that compiler isn't necessary when running the image, it's only necessary during the build phase.

One solution is multi-stage builds: create a series of images, the first with all the packages necessary to build your image, and the second with only the runtime packages installed. As you build the runtime image, you can copy files from the build image:

```
# This is the first image:
FROM ubuntu:18.04 AS compile-image
RUN apt-get update
RUN apt-get install -y --no-install-recommends gcc build-essential

WORKDIR /root
COPY hello.c .
RUN gcc -o helloworld hello.c

# This is the second and final image; it copies the compiled
# binary over but starts from the base ubuntu:18.04 image.
FROM ubuntu:18.04 AS runtime-image

COPY --from=compile-image /root/helloworld .
CMD ["./helloworld"]
```

Notice that each step has a name, `compile-image` and `runtime-image`.

If you just build normally, both stages will get built but any tags will get set on the final stage, in this case `runtime-image`. But you can also build specific stages, so if for example you to build the `compile-image` and then give it a tag you can do:

```
$ docker build --target compile-image -t myimage:compile-stage .
```

In Docker Compose, you can also have `build` services target a particular stage.

### 10.1.1 References

- Multi-stage builds: Smaller images for compiled code (pythonspeed.com) motivates the problem
- Multi-stage builds (docs.docker.com)
- Targeting multi-stage builds in Docker Compose (docs.docker.com)

## 10.2 Use a virtualenv to make copying across stages easier

If you're creating a multi-stage Python Docker image, you need to figure out how to copy images from the build image to the runtime image. Python packages can install files in a variety of locations, so this can be tricky.

One way to do that is by installing everything in a virtualenv:

```
FROM python:3.8-slim-buster AS compile-image
RUN apt-get update
RUN apt-get install -y --no-install-recommends build-essential gcc

RUN python -m venv /opt/venv
# Make sure we use the virtualenv:
ENV PATH="/opt/venv/bin:$PATH"

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY setup.py .
COPY myapp/ .
```

```
RUN pip install .

FROM python:3.8-slim-buster AS build-image
COPY --from=compile-image /opt/venv /opt/venv

# Make sure we use the virtualenv:
ENV PATH="/opt/venv/bin:$PATH"
CMD ['myapp']
```

Another way is to use `pip install --user`, and then you just need to copy over the `~/.local` directory where user installs happen.

### 10.2.1 References

- Multi-stage builds, Python specifics: `virtualenv`, `--user`, and other methods (pythonspeed.com)

## 10.3 Avoid unnecessary complete rebuilds

One problem with multi-stage builds is rebuilds. If you only stored the final runtime image, you won't have the build/compile-stage image available when you rebuild. Which means you won't get any benefit from Docker build caching: every rebuild will rebuild from scratch.

The solution is to tag both images, the compile-stage and the runtime-stage, and to store them both:

```
#!/bin/bash
set -euo pipefail
# Pull the latest version of the image, in order to
# populate the build cache:
docker pull itamarst/helloworld:compile-stage || true
```

```
docker pull itamarst/helloworld:latest        || true

# Build the runtime stage, using cached compile stage:
docker build --target runtime-image \
       --cache-from=itamarst/helloworld:compile-stage \
       --cache-from=itamarst/helloworld:latest \
       --tag itamarst/helloworld:latest .

# Build the compile stage so we tag it. In practice this will
# all come from the local cache and run quickly.
docker build --target compile-image \
       --cache-from=itamarst/helloworld:latest \
       --cache-from=itamarst/helloworld:compile-stage \
       --tag itamarst/helloworld:compile-stage .

# Push the new versions:
docker push itamarst/helloworld:compile-stage
docker push itamarst/helloworld:latest
```

We build the runtime stage first to enable parallelism when using BuildKit; see the next best practice.

### 10.3.1 References

- Multi-stage builds: Why your build is surprisingly slow, and how to speed it up (pythonspeed.com)

## 10.4 Optional: Use BuildKit for faster builds

BuildKit is a new build backend for Docker. Among other features, if you're doing multi-stage builds it can build the different stages in parallel, insofar as this is pos-

sible. That means BuildKit can make multi-stage builds run faster, especially if you build the runtime stage first (see above).

You can enable BuildKit by adding:

```
export DOCKER_BUILDKIT=1
```

to the top of your build script.

## 10.5 Optional: Optimizing image size even further

A multi-stage build gives you the opportunity to make your runtime image even smaller, by deleting arbitrary files from the build stage. Remember that because of layering, deleting files with a `RUN` command won't make the build-stage image smaller. It will however let you copy over less files.

Some examples of files you might want to delete:

- Source code in C or another compiled language that is in a directory you plan to copy over, now that you're done compiling it.
- Likewise, any build artifacts.
- `.pyc` files, but only if you're OK with slower startup.
- JavaScript source map files, if you don't care about the additional debugging they enable.

Your `Dockerfile` might therefore look like this:

```
FROM python:3.8-slim-buster AS compile-image

RUN python -m venv /venv
ENV PATH="/venv/bin:$PATH"
COPY requirements.txt .
RUN pip install -r requirements.txt
```

```
WORKDIR /app
COPY . .
RUN python setup.py build_ext --inplace

# Delete C, Cython, map files and build artifacts we
# don't want in the runtime image:
RUN rm -f `find . -iname "*.c"` && \
    rm -f `find . -iname "*.pyx"` && \
    rm -f `find /venv -iname "*.js.map"` && \
    rm -f build/

FROM python:3.8-slim-buster AS build-image
COPY --from=compile-image /opt/venv /opt/venv
COPY --from=compile-image /app /app

# ... etc. ...
```

## Some final recommendations

There are many best practices, and you won't need all of them, especially not to begin with. So start by following the plan in the first chapter and see how much you actually need in practice.

If you'd like a working setup implementing many of these best practices, either as an example or as a basis for your packaging, you can also purchase a production-ready template that will allow you to get going in just 2 to 3 hours. As a purchaser of the checklist, you can use the discount code `BPCHECKLIST` to get a 15% discount off the template.

And as always, if you have any questions or suggestions, please email me at itamar@pythonspeed.com.

—Itamar Turner-Trauring

# Changelog

### June 17, 2020

More best practices:

- Various BuildKit features that help speed up builds.
- Dropping capabilities.
- Avoiding listening on ports < 1024.
- Running a different command altogether based on command-line arguments.
- Bytecode compilation.
- Additional image size optimization in multi-stage builds.
- Warm up the build cache for per-branch builds.
- Requirements for running on Heroku and Google Cloud Run.
- BuildKit ssh-agent forwarding.
- BuildKit secrets when using Docker Compose.

Other tweaks and improvements throughout the text.

### June 8, 2020

The Checklist has been renamed, and is now known as a Quickstart. To help make that change:

- Added a new introductory chapter with a plan to help you figure out which best practices to implement when.
- Tweaked the chapter structure.

Additionally:

- Added a best practice about using `dive` to find large layers.
- Split off `init` into its own best practice.
- Explained the goal of responding to health checks quickly more broadly, rather than in specific implementation terms of process/thread pool.
- Added more nuance to the section on updating dependencies once a month.

- Restored the Pipenv instructions, since it's now being maintained again.
- Make the `DEBIAN_FRONTEND=noninteractive` best practice standalone.

### June 2, 2020

Added multiple new best practices:

- Making sure `ARG` doesn't break build caching.
- Don't use the `latest` tag.
- For better reproducibility, you can create a custom base image.
- Size checks for images.
- Security scanners: `bandit`, `safety`, `trivy`.

Also updated existing best practices:

- You can make a build arg available at runtime by using `ENV`.
- For build secrets another alternative is short term keys.

### April 27, 2020

- Added new best practice on timely security updates, with additional information on automatic notifications.
- Switched some examples from shell session transcripts to `Dockerfile` or shell script.
- Noted `docker build` and Docker Compose support for targeting named stages.

### April 1, 2020

- Documented two-stage install with Poetry.
- Added link to docker-autoheal.

February 24, 2020

Added many more examples:

- Configuring `logging`.
- A smoke test.
- Passing in secrets with BuildKit.
- `.dockerignore` file.
- System package upgrade script for CentOS/RHEL.
- `Dockerfile` healthcheck.
- And a few more expanded examples here and there.