Python on Docker Production Handbook

Itamar Turner-Trauring

January 26, 2022

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 handbook.

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 handbook is therefore to get you going as quickly as possible, while still providing a reference to all the relevant details.

How to read this guide:

1. Read the next chapter, which presents a possible implementation plan.
2. Skim the guide, 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 chapter doesn't cover all the best practices. You should also skim the guide 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).

See also the best practices for Conda (ref) and best practices for Poetry (ref) and Pipenv (ref) if you're using those tools.

Partial example

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

```
FROM python:3.9-slim-bullseye
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. If it's not secure, you can't run it in production; if you can't run it, what's the point?

- 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).
- Don't leak runtime secrets (ref) or build secrets (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), and likewise for runtime secrets (ref).
- If you have any files you don't want to be in the image for security reasons, make sure they're not leaked (ref).

Partial example

The `Dockerfile` might now look like this:

```
FROM python:3.9-slim-bullseye

# 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. This will allow other people on your team to get images built automatically, and reduce human error caused by manual builds.

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. You might also start thinking about automatic deploys at this point, depending how your production environment runs.

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

IMAGE_NAME=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 $IMAGE_NAME:$GIT_BRANCH || true

# Use branch+commit for tagging:
docker build -t "$IMAGE_NAME:$GIT_BRANCH" \
             -t "$IMAGE_NAME:$GIT_COMMIT" \
             --build-arg BUILDKIT_INLINE_CACHE=1 \
```

```
            --cache-from=$IMAGE_NAME:$GIT_BRANCH .

# Security scanners:
trivy image --ignore-unfixed --exit-code 1 \
    $IMAGE_NAME:$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 \
             --build-arg BUILDKIT_INLINE_CACHE=1 \
             --cache-from=$IMAGE_NAME:$GIT_BRANCH .
python smoketest.py $IMAGE_NAME:$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:3.9-slim-bullseye

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

In the first week or so of packaging and development, it's unlikely that any major changes will happen to your Python dependencies. As time goes on, the software you depend is more and more likely to change. And that can break your build, or break your application.

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:3.9-slim-bullseye

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. This will save download time, as well as bandwidth and disk space.

- Keep temporary files from ending up in a layer (ref).

- Add files to `.dockerignore` (ref).
- Add `.git` to `.dockerignore`, or consider alternatives when you can't (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:3.9-slim-bullseye

# 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 Docker versions and BuildKit

Before moving on to the details of the best practices, it's worth understanding the available versions of Docker, and what they support.

2.1 Docker 19.03 and 20.10

As of February 2021, there are two major versions of Docker available:

- Docker 19.03, which is widely available in recent operating systems, cloud CI services, and so on.
- The newly released Docker 20.10, which will become more available over time on the server side but is still quite new.

If you're using Docker Desktop on macOS and Windows you will end up getting updated to Docker 20.10 as part of the update process.

Docker 20.10 has a number of new features; I will mention them inline where relevant.

If you are using Docker 18.09 or older, I recommend upgrading: you're missing two years' worth of bug fixes and enhancements.

2.2 BuildKit

BuildKit is a re-implementation of Docker's image building system, included in both Docker 19.03 and 20.10. It includes performance enhancements, like parallel builds where possible, and new features like build secrets, SSH agent forwarding from the host, and additional caching features.

To enable it:

- You can set the environment variable `DOCKER_BUILDKIT` to `1`, e.g. `export DOCKER_BUILDKIT=1`.
- On recent versions of Docker Desktop on macOS and Windows there should be a setting in the Preferences UI; on new installs it will be enabled by default.

2.2.1 Getting complete output with BuildKit

BuildKit changes the way build progress is output. To make sure you see the output of `RUN` commands, you can run with the `--progress=plain` command-line argument.

```
$ docker image build --progress=plain .
```

2.2.2 Enabling BuildKit features in your Dockerfile

With the release of Docker 20.10, BuildKit has now stabilized, which is why it is enabled by default on new macOS and Windows installs.

Among other features, BuildKit allows configuring the builder a per-`Dockerfile` basis. On Docker 20.10 it will use BuildKit v1.3 syntax by default when BuildKit is enabled, but on Docker 19.03 you need to set this explicitly in your `Dockerfile`, like so:

```
# syntax = docker/dockerfile:1.3
FROM python:3.9-slim-bullseye

# ... rest of Dockerfile goes here
```

On Docker 20.10 this isn't strictly necessary, but it is still worthwhile: you'll automatically get the latest bugfix version downloaded (it's a Docker image!), even if it's not packaged in your particular release of Docker 20.10. If you're using BuildKit, then, you should just add that line at the top of all your `Dockerfile`s.

BuildKit mostly works pretty well on Docker 19.03, so you can use it even if you still haven't upgraded to 20.10.

Again, I will be covering BuildKit features inline where relevant.

2.2.3 Enabling BuildKit with Docker Compose

Docker Compose doesn't work out of the box with BuildKit. You can enable support for BuildKit by setting an additional environment variable beyond `DOCKER_BUILDKIT`:

```
$ export COMPOSE_DOCKER_CLI_BUILD=1
```

Unfortunately this method of using Docker results in somewhat worse error messages when builds fail. Not all BuildKit features are supported, see the best practice below on build secrets with Compose.

2.2.4 Ensuring caching works correctly with BuildKit

BuildKit has a much more sophisticated caching system; you can export the cache to the filesystem, and store it separately from images. In order for caching to work the way it previously worked with Docker when using `--cache-from`, you will need to add `--build-arg BUILDKIT_INLINE_CACHE=1` to your build script. See the best practice on warming the cache for details (ref).

2.3 Podman

On RedHat operating systems, an alternative implementation of Docker called Podman is installed. Podman is supposed to be completely compatible with Docker on the CLI side and to some extent on the API side as well, allowing support for tools like Docker Compose.

It does not yet support the BuildKit features, but it may have alternative ways to achieve the same goals. You can use BuildKit to build images and then run them on Podman, and you can use a Podman container as a BuildKit builder.

I don't have personal experience with Podman, so I can't comment on how well it works.

In practice you can still install normal Docker; 19.03 had some issues with the latest Fedora releases, but Docker 20.10 has fixed those.

2.3.1 References

- `docker/dockerfile` reference (hub.docker.com)
- BuildKit features overview (outdated) (docs.docker.com)
- Podman (podman.io)
- BuildKit repository (github.com)
- BuildKit in Compose (docker.com)

3 Best practices: Security

3.1 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:3.9-slim-bullseye
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`.

3.1.1 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)

3.2 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.

3.3 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.

3.3.1 References

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

3.4 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:3.9-slim-bullseye
RUN apt-get update && apt-get -y upgrade
```

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

3.5 ∞ 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
```

3.5.1 References

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

3.6 ∞ 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 scanning functionality built-in for Pip, Pipenv, and Poetry; see the documentation linked in references. 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.

Once you've realized you have a security vulnerability, either via a notification or via a security scanner (see the next best practice), you will need to:

1. Update your package dependencies with the fixed version, if you're pinning versions.
2. Rebuild the image.
3. Redeploy the image anywhere it's currently running.

3.6.1 References

- Debian security information (debian.org)
- Ubuntu security notices (usn.ubuntu.com)
- RHEL security advisories (access.redhat.com)
- GitHub dependency auto-updates (github.com)
- GitHub dependency security vulnerability alerts(github.com)

3.7 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.

3.7.1 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.
...
```

3.7.2 Known vulnerabilities in Python dependencies, system packages, and other languages

Trivy is a command-line tool that lets you scan a Docker image, the filesystem, or a remote repository, for many kinds of security vulnera-

bilities, both system packages (RHEL, Debian, Ubuntu, and more) and programming language-specific packages. `trivy` also supports checking `requirements.txt`, Poetry, and Pipenv dependency files.

Trivy reports *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.

> Note: `trivy` has some data sources it uses that are licensed for non-commercial use only. This problem should be fixed as of v0.23 (at the time of writing, January 2022, the latest release is v0.22). See the Trivy data sources page for a list of which are OK to use.
>
> 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. Note that this is only supported for some operating systems.

Additionally, by default `trivy` won't set a non-zero exit code when vulnerabilities are found. If you're using Trivy as part of CI or other automated script, you'll want to use the `--exit-code` option to make sure found vulnerabilities are noted as a failure, e.g. `--exit-code 1`.

To scan the image `yourorg/yourimage`, you can do:

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

To scan a local directory, you can do:

```
$ trivy fs --ignore-unfixed --exit-code 1 ./path/to/code/
```

The Trivy documentation explains how to use it with many CI systems (GitHub Actions, Circle CI, GitLab CI, and more); see the "Advanced" section of the docs.

An alternative Trivy is the Grype security scanner.

3.7.3 References

- Bandit documentation (bandit.readthedocs.io)
- flake8-bandit (pypi.org)
- Trivy documentation (aquasecurity.github.io)
- Grype security scanner (github.com)

3.8 Don't leak secret files

Imagine you have the following files in your build directory:

```
Dockerfile
app/
  __init__.py
  code.py
run.sh
id_rsa
```

The `id_rsa` file is a private SSH key used to access a private repository–you do not want it to end up in the image. If it does, any attacker who gains access to the image will be to extract this secret file.

If you do the following:

```
COPY . .  # INSECURE
```

Then by default `id_rsa` will get copied in to the image.

You can avoid this by explicitly copying in only the files you need:

```
COPY app/ run.sh ./
```

You can also add `id_rsa` to `.dockerignore` so it doesn't get copied in.

Finally, you can avoid having secrets in the same directory as your Docker build context.

3.9 Don't leak runtime secrets

Your Docker image may require a variety of secrets to run correctly. For example, your web application might require the password to its MySQL database. These are known as runtime secrets, and you should not store them in your image, since an attacker who gains access to the image will be able to read them.

There are a variety of runtime-specific mechanisms for passing secrets into a running container:

- Using environment variables passed in at runtime, using e.g. `docker run --env/--env-file`, Docker Compose `environment/env_file` keys, or the Kubernetes equivalents.
- Mounting a volume or host directory with the secret.
- Secret-specific mechanisms like Kubernetes secrets (which actually uses both mechanisms above).
- In some cloud environments, you can give permissions to containers to talk to the cloud environment, e.g. IAM roles for ECS tasks on AWS.
- Retrieving them from some sort of external key store like HashiCorp's Vault.

  Note that "volume" is used inconsistently in Docker. Sometimes it means "a mini-filesystem managed by Docker", or if you're using Kubernetes a "mini-filesystem managed by Kubernetes or a system it talks to." Sometimes it is used more generically as the configuration option to mount either volumes in the previous sense or bind mount directories from the host system.

  Either way, the key point is that the secret should not be in the image itself.

3.9.1 References

- Setting Docker environment variables at runtime (docs.docker.com)
- Bind mounting (docs.docker.com)
- Kubernetes secrets (kubernetes.io)
- IAM in ECS (docs.aws.amazon.com)
- HashiCorp Vault (vaultproject.io)

3.10 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 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.

For this last method of passing in secrets, via the network, see the references below. Given that BuildKit support is now stable, there is no reason to use the network variant, except perhaps with RedHat's alternative Docker implementation, Podman.

3.10.1 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.

3.10.2 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:3.9-slim-bullseye
COPY yourprivatecode .
# RUN the build, etc.
```

3.10.3 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:1.3
FROM python:3.9-slim-bullseye
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 .
```

Docker 20.10 adds the additional ability to load secrets from environment variables, not just files. For example, if you have an environment variable `MYSECRET`, you can access it like this:

```
$ export MYSECRET=theverysecretpassword
$ export DOCKER_BUILDKIT=1
$ docker build --secret --secret id=mysecret,env=MYSECRET .
```

If you're OK with the secret ID being the same as the name as the environment variable, you can replace the last command with:

```
$ docker build --secret --secret id=MYSECRET .
```

Note that it will still be exposed inside the build as a file in `/run/secrets`, it is merely read from an environment variable on the host.

3.10.4 References

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

3.11 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:1.3
FROM python:3.9-slim-bullseye
# 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
```

3.11.1 References

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

3.12 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:1.3
FROM python:3.9-slim-bullseye

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 .
```

Note that by default this will fail with an error about an unverified host; see the next best practice for details about to fix this.

3.12.1 References

- Using SSH with BuildKit (docs.docker.com)
- `ssh-add` man page (man7.org)
- `ssh-agent` man page (man7.org)

3.13 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
```

3.13.1 References

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

4 Best practices: Running in CI

4.1 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.

4.1.1 References

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

4.2 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. Additionally, some of its recommendations are completely wrong; for example, it recommends not using `apt-get upgrade`, which is a very bad recommendation. So only use it if you're prepared to override many of its incorrect suggestions.

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.

4.2.1 References

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

4.3 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"
docker push "$IMAGE_NAME:$GIT_COMMIT"
```

4.4 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.

4.4.1 References

- What's wrong with the Docker `:latest` tag? (vsupalov.com)
- Kubernetes `imagePullPolicy` interaction with `latest` (kubernetes.io)

4.5 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 start 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 \
    --build-arg BUILDKIT_INLINE_CACHE=1 \
    --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.

The `--build-arg BUILDKIT_INLINE_CACHE=1` is necessary for this scheme to work with BuildKit. While this flag is not necessary in classic Docker builds, you may as well always set it so you don't forget to enable it when you turn on BuildKit.

4.5.1 References

- Speeding up Docker builds in CI (pythonspeed.com)

4.6 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:$BRANCH \
    --build-arg BUILDKIT_INLINE_CACHE=1 \
    .
docker push yourimage:$BRANCH
```

Instead of hard-coding something like `production`, you can also get the default branch for your Git repository (`master` or `main`):

```
# ... see https://stackoverflow.com/q/28666357/6214034
DEFAULT_BRANCH=$(git rev-parse --abbrev-ref origin/HEAD)
DEFAULT_BRANCH=$(basename $DEFAULT_BRANCH)
docker pull yourimage:$DEFAULT_BRANCH || true
```

4.7 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. First, make sure you enable BuildKit in general using the `DOCKER_BUILDKIT` environment variable. Second, make sure you're enabling inline caching metadata by passing in the `BUILDKIT_INLINE_CACHE` build arg. You'll want to do the latter regardless!

The build script will look like this:

```
#!/bin/bash
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`.

Note that:

1. There were some bugs in the version in 19.03, so you may wish to only use Docker 20.10 with this feature.
2. Some registries don't work with this mechanism. Notably, the deprecated GitHub Packages registry will fail, and this will break caching altogether. You may wish to try the replacement GitHub Container Registry, in beta as of May 2021 and see if it works there.

4.7.1 References

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

5 Best practices: Make debugging easier

5.1 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)
```

In fact, by default Python's `logging` will log to `stdout`, so technically you don't have to do anything.

5.2 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>
```

5.2.1 References

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

5.3 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:

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

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:3.9-slim-bullseye
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
```

5.3.1 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

5.4 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 `net-tools` so you have access to `ps` and `netstat`:

```
RUN apt-get update && apt-get -y install procps net-tools
```

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.

5.4.1 References

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

6 Best practices: Correct operation

6.1 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.

6.1.1 References

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

6.2 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 start 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.

6.2.1 References

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

6.3 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. For example, if your program shuts down just fine outside of Docker when you hit Ctrl-C and a `KeyboardInterrupt` is raised, that means it expects a `SIGINT`, and you should use `STOPSIGNAL INT`.

Another approach to Ctrl-C handling can be done with `tini`, see the next section.

6.3.1 References

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

6.4 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:3.9-slim-bullseye
RUN apt-get update && apt-get install -y tini
COPY your-entrypoint.sh .
ENTRYPOINT ["tini", "--", "./your-entrypoint.sh"]
```

One useful option to add to `tini` is the `-g` option:

```
ENTRYPOINT ["tini", "-g", "--", "./your-entrypoint.sh"]
```

When the parent `tini` process is killed, instead of just killing the top-level process (a shell running `your-entrypoint.sh` in the above example), it will send a signal to *all* descendant processes in the main process group. In practice that is likely to be all the container's processes. This is almost always the behavior you want: shutting down kills all processes immediately.

6.4.1 References

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

6.5 Set a non-interactive frontend for Debian/Ubuntu 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
```

6.6 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:3.9-slim-bullseye
HEALTHCHECK --interval=3s --timeout=1s \
  CMD ["python", "-c", \
       "from urllib.request import urlopen; \
        urlopen('http://localhost:8000').read()"]
ENTRYPOINT ["python", "-m", "http.server"]
```

6.6.1 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.

6.7 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 precompile 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/
```

6.7.1 References

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

7 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.

7.1 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 RedHat Enterprise Linux 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.

Of course, each of the distributions has different variations available, including non-LTS releases. So when you choose a base image, you'll need to make sure to specify a particular release using a tag:

- **Bad:** `FROM ubuntu`—today this might be Ubuntu 20.04, eventually it will be Ubuntu 22.04.
- **Good:** `FROM ubuntu:20.04`, `FROM debian:11` aka `FROM debian:bullseye`.

Here's a comparison between the three. Newer releases may be better in some cases, insofar as they will have more up-to-date system packages.

Distribution	Released	End-of-life	Python versions
Debian 11	Aug 2021	Aug 2025	3.9
Ubuntu 20.04	Apr 2020	Apr 2025	3.9, 3.8
RHEL 8	May 2019	May 2024	3.9, 3.8, 3.6

Even though they all include 3.9, at the time of writing (August 2021) they have different point releases, none of which match 3.9.6 which was released in late June 2021:

Image	3.9 release
Debian 11	3.9.2
Ubuntu 20.04	3.9.5
RHEL 8	3.9.2

Another option is the "official" Docker Python image, the one blessed by Docker-the-company. It has variants with pretty much every version of Python available, so you're not tied to whichever versions Linux distributions decided to include or backport. In particular, you'll want the variant based on Debian Stable, the most-up-to-date being `bullseye`. And I also recommend using the `slim` variant to get a smaller base image.

Given these choices, you can choose different levels of reproducibility:

- `python:3.9-slim-bullseye` is the latest sub-release of Python 3.9, installed on top of Debian "Bullseye" 11. At the time of writing this will be 3.9.6, later it will be 3.9.7, and so on.
- `python:3.9.6-slim-bullseye` is a specific sub-release, Python 3.9.6. This tag might still point to different images over time, however; it might be updated with newer releases of `pip`, for example, or newer system packages.
- `python@sha256:6331fb167811d3bf7a7a33eaf4ac233ae63fa5d0e21cfd341a22bce2905ec5b5` 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.

Note that at the moment the Python packaged by the official images is not as fast as some of the alternatives. For example, Python runs 10-20% faster using Ubuntu and Debian's version of Python 3.9. So if performance is critical you may wish to use Ubuntu or Debian.

7.1.1 References

- "Official" Python base image (hub.docker.com)
- Debian base image (hub.docker.com)
- Ubuntu base image (hub.docker.com)
- RedHat Universal Base Images (developers.redhat.com)
- Pulling an image by digest (docs.docker.com)
- Choosing a base image for your Python application (python-speed.com)
- Performance comparison between different Python builds (python-speed.com)

7.2 RedHat-compatible base images

If you want to use a RedHat-compatible base image, but not pay for RedHat Enterprise Linux, you would in the past have used CentOS. CentOS 8 is however no longer a stable, maintained distribution; you should not be using it.

Supported alternatives include:

- RedHat's own Universal Base Images. Not all RedHat packages are available in these images, but they should include most popular packages.

- Oracle Linux is a pre-existing clone of RedHat Enterprise Linux, maintained by Oracle. You can pay for commercial support, but you can also just use it for free.
- AlmaLinux was created by CloudLinux, a commercial Linux vendor who used to base their product on CentOS.
- RockyLinux was started by one of the original creators of CentOS.

Oracle Linux has been in existence for much longer, so it has better tooling support from things like security scanners. Otherwise, the last three options don't seem much different.

7.2.1 References

- RedHat Universal Base Images (developers.redhat.com)
- `redhat/ubi8` on Docker Hub (hub.docker.com)
- RedHat's container image registry (catalog.redhat.com)
- `oraclelinux` image (hub.docker.com)
- `almalinux` image (hub.docker.com)
- `rockylinux` image (hub.docker.com)

7.3 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`.

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
```

7.3.1 References

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

7.4 ∞ 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.

7.5 Optional: Pin system packages

One of the benefits of using a stable base operating system like Debian, Ubuntu LTS, or RHEL 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+deb10u3
```

On RedHat-based images, `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.

7.5.1 References

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

7.6 Optional: Create a custom base image

If your `Dockerfile` looks like this:

```
FROM python:3.9-slim-bullseye
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 reproducibility.

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:3.9-slim-bullseye
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 ...
```

8 Best practices: Faster builds

8.1 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, like a slower standard C library; see the reference below for details.

8.1.1 References

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

8.2 `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:3.9-slim-bullseye
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:3.9-slim-bullseye
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.

8.2.1 References

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

8.3 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:3.9-slim-bullseye
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
RUN pip install .
```

See the best practices for Poetry (ref), Pipenv (ref), and Conda (ref) for details on installing dependencies with those tools.

8.3.1 References

- `requirements.txt` file format (pip.pypa.io)

8.4 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:3.9-slim-bullseye
# 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:3.9-slim-bullseye
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
```

8.5 Use `docker build --label` instead of `LABEL`

If you're passing in a build argument only so you can use it for a `LABEL`, e.g.

```
# ...
ARG git_commit
LABEL git_commit=$git_commit
```

Better to just set the label using the `--label` argument to `docker build`, e.g.

```
$ docker build --label=git_commit=$GIT_COMMIT
```

This way you don’t have to worry about the impact of the ARG on layer caching.

9 Best practices: Small images

9.1 Keep temporary files from ending up in a layer

Each RUN and COPY command adds another layer to your Docker image, rather like commits in the Git history. As a result deleting files that are in a previous layer won't shrink your image, since they will still be present in the previous layer:

```
RUN wget https://example.com/largefile.tar.gz
RUN tar xvfz largefile.tar.gz
RUN largefile/install.sh
# BAD, This will not shrink your image:
RUN rm -rf largefile.tar.gz largefile/
```

Instead, you can combine these commands into a single layer, and then the temporary files won't end up in the image:

```
# GOOD, temporary files deleted before RUN ends:
RUN wget https://example.com/largefile.tar.gz && \
    tar xvfz largefile.tar.gz && \
    largefile/install.sh && \
    rm -rf largefile.tar.gz largefile/
```

This works because files are only stored in a layer at the end of the RUN. If the file doesn't exist when the RUN finishes, it won't get stored.

9.1.1 References

- Shrinking your Python application's Docker image: an overview (pythonspeed.com)

9.2 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.

9.2.1 References

- `dive` (github.com)

9.3 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, copying 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__
```

9.3.1 References

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

9.4 Add `.git` to `.dockerignore` (and some alternatives when you can't)

If you're using Git, a useful entry to add to your `.dockerignore` is the `.git` directory: by default it includes the full history of your repository, which can be quite large.

In some cases, this may not be possible.

If you're using a tool like `setuptools-scm` or `versioneer` to set your Python application's version using Git tags. In this case you can try one of the following options to minimize the size of the `.git` directory:

- Minimize the size of the `.git` directory by doing a "shallow" clone, e.g. `git clone --depth=50 yourrepo.git`, where you only download the last N revisions instead of the full history. Many CI services (GitHub Actions, GitLab CI) already do this by default. The downside is that tags that are in older revisions outside the shallowly clone history won't show up, which can break the auto-versioning tools mentioned above.
- Use `git clone --filter=blob:none yourrepo.git`. This only downloads the contents of history when you need access to it. That means tags will all be visible, but the `.git` folder will be much smaller unless you explicitly check out lots of tags and branches.

You can also rely on a multi-stage Docker build to ensure the `.git` directory doesn't make it into the runtime image, at least.

If your application's runtime logic relies on `.git` to figure out its version, you can make this logic conditional, with a fallback to environment variables which can then be provided by the Docker image.

9.4.1 References

- `git clone` documentation (git-scm.com)

9.5 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 .
```

This will do the `chown` as part of the copy, ensuring only one copy of the files is kept.

9.5.1 References

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

9.6 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/*
```

In practice, modern Debian base images (both the official Debian and official Ubuntu images) will actually do at least part of the clean-up step automatically themselves, so you can skip that if you want, but can't hurt to make sure.

For RPM-based systems like RHEL:

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

```
# Install security updates, bug fixes and enhancements only.
# --nodocs means documentation isn't installed, leading to a slightly
↪  smaller image.
dnf -y --nodocs upgrade-minimal

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

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

9.6.1 References

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

9.7 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:3.9-slim-bullseye
RUN pip install --no-cache-dir flask
```

9.8 Optional: Caching installed packages using BuildKit

By default `pip` caches downloaded packages so that new virtualenvs won't require you to redownload everything. With classic Docker you are forced to redownload the same packages every time `requirements.txt` changed.

BuildKit, however, allows you to cache a directory outside of your image and across builds, which allows you to get a similar set of benefits. This level of caching is distinct from Docker's layer caching. If you change `requirements.txt`, `pip` will still need to be rerun when you rebuild your Docker image, but at least it won't have to download everything from scratch.

If we're using this feature we do not need to use best practices like `pip install --no-cache`: saving downloaded packages is actually desirable because they're cached outside the image. And since the official Debian Docker base image actually deletes downloaded packages and archive indexes automatically, we want to disable that feature too because we're making sure to cache outside the image.

Here's how to use this form of caching:

```
# syntax = docker/dockerfile:1.3
FROM python:3.9-slim-bullseye

# Disable auto-cleanup after install:
RUN rm /etc/apt/apt.conf.d/docker-clean
# Cache downloaded packages across runs:
RUN --mount=type=cache,target=/var/cache/apt,id=apt \
  apt-get update && apt-get -y upgrade

COPY requirements.txt .
```

```
# Cache downloaded packages across runs:
RUN --mount=type=cache,target=/root/.cache,id=pip \
  pip install -r requirements.txt
```

If you rebuild this (or another!) image on the same machine and have to redo the `apt-get` or `pip install`, the already downloaded packages will be available to the installer and won't need to be redownloaded.

Make sure to enable BuildKit with `export DOCKER_BUILDKIT=1`.

For the `pip` packages I'm caching the `~/.cache` directory, since that is also where Pipenv and Poetry will store their files; `pip` uses `~/.cache/pip` by default. You'll want to use the absolute path version of `~/.cache`, e.g. `/home/youruser/.cache`.

9.8.1 References

- `Dockerfile` BuildKit extensions (hub.docker.com)
- Debian base image apt config scripts (github.com)
- Speed up pip downloads in Docker with BuildKit's new caching (pythonspeed.com)

9.9 Optional: Remove files you don't need

Your Docker image may end up with files in it that you don't need, and whose existence you can't prevent with other techniques. You have a number of options to get rid of these extraneous files, if they're taking up too much space:

- Multi-stage builds; see the relevant chapter of the handbook.

- The `docker-squash` tool allows you to merge multiple layers into one.
- The `docker-slim` tool uses runtime instrumentation to figure out which files your container actually uses (probably a tiny subset!) and creates a new image with only those files.

9.9.1 References

- `docker-squash` (github.com)
- `docker-slim` (github.com)

10 Best practices: Application and tool-specific

10.1 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"
```

10.1.1 References

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

10.2 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.

10.2.1 References

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

10.3 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.

10.4 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:3.9-slim-bullseye
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:3.9-slim-bullseye \
    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"]
```

10.4.1 References

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

10.5 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=- --bind 0.0.0.0:8000 yourmodule:yourapp
```

10.5.1 References

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

10.6 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.

10.6.1 References

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

10.7 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 variable.

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:3.9-slim-bullseye
COPY entrypoint.sh .
ENTRYPOINT ["./entrypoint.sh"]
```

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

10.7.1 References

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

11 Best practices: Conda

11.1 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.

11.1.1 Activating with `conda run`

One approach 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 .
# We give it a custom name, "myenv":
RUN conda env create -n myenv

# Make RUN commands use the new environment:
SHELL ["conda", "run", "--no-capture-output", "-n", "myenv", "/bin/bash",
↪  "-c"]

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

# The code to run when container is started:
COPY run.py .
ENTRYPOINT ["conda", "run", "--no-capture-output", "-n", "myenv", "python",
↪  "run.py"]
```

The `--no-capture-output` ensures the stdout and stderr output from Python isn't hidden by `conda run`.

11.1.2 Activating with shell startup

Another approach takes advantage of the fact that `continuumio/miniconda3` configures `bash` to load the Conda startup shell scripts, and activate the environment in the entrypoint.

There's one tricky point here: typically I would recommend using bash strict mode. However, some Conda activation scripts break with this mode is enabled! So what we have to do is disable strict mode while activating the script.

We have an `entrypoint.sh` to run the program:

```
#!/bin/bash --login
# The --login ensures the bash configuration is loaded,
# enabling Conda.

# Enable strict mode.
set -euo pipefail
# ... Run whatever additional commands ...

# Temporarily disable strict mode and activate conda:
set +euo pipefail
conda activate myenv

# Re-enable strict mode:
```

```
set -euo pipefail

# Run the program:
exec python run.py
```

And here's the `Dockerfile`:

```
FROM continuumio/miniconda3

WORKDIR /app

# Create the environment:
COPY environment.yml .
# We give it a custom name, "myenv":
RUN conda env create -n myenv

# Make RUN commands use the new environment:
RUN echo "conda activate myenv" >> ~/.bashrc
SHELL ["/bin/bash", "--login", "-c"]

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

# The code to run when container is started:
COPY run.py entrypoint.sh ./
ENTRYPOINT ["./entrypoint.sh"]
```

11.1.3 References

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

11.2 Conda environments require activation

You should not simply call binaries in the Conda `bin/` directory without activating the environment. This will work fine for virtualenvs, but might fail for Conda in certain cases.

This is because Conda packages can include activation scripts that are necessary for the packages to work correctly. You therefore need to make sure those activation scripts get called, which is part of what Conda environment activation does.

11.2.1 References

- Activate scripts in Conda packages

11.3 Reproducible builds with `conda-lock`

In order to get reproducible builds, you will want to transitively pin your dependencies. For example, the following `environment.yml` will install different packages each time:

```
name: example
channels:
  - conda-forge
dependencies:
  - python=3.8
  - numpy
```

Even if you pinned specific packages, that won't necessarily install the same dependencies, you need transitive pinning.

`conda-lock` is a tool that lets you pin packages transitively, creating what's known as a spec list, or lockfile, a list of packages to download. The additional benefit of a spec list is that you can skip the slow "Solving dependencies" stage you get when you install an `environment.yml`.

There are two steps. First, you pin the dependencies, which you can do on any computer with any operating system (Windows or macOS work too). Note that you can also install `conda-lock` using `pip`.

```
$ conda install -c conda-forge conda-lock
$ conda-lock -f environment.yml -p linux-64
```

This will generate a `conda-linux-64.lock` file. Any time you want to update your dependencies you'll need to regenerate this file by rerunning `conda-lock`, and it will get the latest version of dependencies in `environment.yml`.

You can then use this file to create a Conda environment in your Docker image, using a slightly different method than what you'd use with an `environment.yml`:

```
FROM continuumio/miniconda3

COPY conda-linux-64.lock .
# Create environment called "yourenv":
RUN conda create --name yourenv --file conda-linux-64.lock
```

One limitation of this approach is that `pip` packages aren't supported by `conda-lock` quite yet. The next major release of `conda-lock` will include support for `pip` packages (at the time of writing the current release was v0.13.2).

In the interim:

- List required `pip` packages outside of `environment.yml`, in a `requirements.txt` for example.
- Install `pip` packages separately, with `pip`.
- Use the `pip`-specific package pinning mechanisms to keep them locked to specific versions.

11.3.1 References

- `conda-lock` (github.com)
- Reproducible and upgradable Conda environments: dependency management with conda-lock (pythonspeed.com)

11.4 Make Conda images smaller with `conda clean`

By default, Conda will cache a variety of files: index files, downloaded packages, and so on. If you don't anticipate doing any further installs, you can delete all these files using `conda run` to save some space.

Make sure to do this in the same `RUN` command where you install the packages, to ensure the files aren't saved in a previous layer.

```
FROM continuumio/miniconda3

COPY environment.yml .
RUN conda env create -n myenv && \
    conda clean -afy
```

11.4.1 References

- `conda clean` documentation (docs.conda.io)

11.5 Make Conda images smaller by using OpenBLAS

If you're using NumPy, or packages like Pandas that depends on NumPy, you need to choose which BLAS linear algebra library to use. The two choices are MKL, which is often faster, and OpenBLAS, which uses much less disk space.

Switching to OpenBLAS depends on where you're installing packages from:

- The packages you install with `pip` use OpenBLAS.
- If you're installing packages from the Conda-Forge channel, OpenBLAS is the default, so you don't need to take any further action to get smaller images.
- If you're using the default Anaconda packages, MKL is the default, and if you want OpenBLAS you'll want to add `nomkl` to the packages you're installing. This will shave something like 700MB off your Docker image!

```
name: example
dependencies:
  - python=3.8
  - numpy
  - nomkl
```

11.5.1 References

- Omitting `mkl` in Anaconda (docs.anaconda.com)
- Smaller Docker images with Conda (jcristharif.com)

11.6 Even smaller Conda images with `conda-pack` and multi-stage builds

In addition to the Conda environment you create, by default your Conda-based Docker image will ship with the base environment, including a full install of Python. By using `conda-pack`, a tool that lets you convert a Conda environment into a standalone directory, you can drop that extra environment and any other unneeded files from the final runtime image.

```
# The build-stage image:
FROM continuumio/miniconda3 AS build

# Install the package as normal:
COPY environment.yml .
RUN conda env create -n myenv

# Install conda-pack:
RUN conda install -c conda-forge conda-pack

# Use conda-pack to create a standalone enviornment
# in /venv:
RUN conda-pack -n myenv -o /tmp/env.tar && \
  mkdir /venv && cd /venv && tar xf /tmp/env.tar && \
  rm /tmp/env.tar
```

```
# We've put venv in same path it'll be in final image,
# so now fix up paths:
RUN /venv/bin/conda-unpack


# The runtime-stage image; we can use Debian as the
# base image since the Conda env also includes Python
# for us.
FROM debian:bullseye AS runtime

# Copy /venv from the previous stage:
COPY --from=build /venv /venv

# When image is run, run the code with the environment
# activated:
SHELL ["/bin/bash", "-c"]
ENTRYPOINT source /venv/bin/activate && \
           python -c "import numpy; print('success!')"
```

`conda-pack` may have issues if you are installing many packages with `pip`.

11.6.1 References

- Shrink your Conda Docker images with conda-pack (python-speed.com)
- `conda-pack` documentation (conda.github.io)

11.7 Faster installs with Mamba

Mamba is a re-implementation of the Conda package manager that runs much faster. It also supports the same command-line options as Conda, so once you've installed it you just need to replace `conda` with `mamba`:

```
FROM continuumio/miniconda3
RUN conda install -c conda-forge mamba
# Now proceed as normal, just using mamba instead:
RUN mamba env create -n myenv
```

11.7.1 References

- Mamba (github.com)

11.8 Security scans for Conda environments

Conda packages can include more than just Python packages, so if you want to scan your dependencies for security vulnerabilities you need a security scanner that understands this. One such tool is Jake.

Assuming you have an environment named `myenv` you create earlier in the `Dockerfile`, during the build, you can scan for vulnerabilities like so:

```
# ... create a Conda env called "myenv" with your packages ...

# Make virtualenv for Jake, run a security scan, and then clean up:
RUN python3 -m venv /tmp/venv-jake && \
    /tmp/venv-jake/bin/pip install jake && \
```

```
conda run -n myenv conda list | (set -e && \
/tmp/venv-jake/bin/jake ddt -c && rm -rf /tmp/venv-jake)
```

Note that Jake support for Conda-Forge may be limited.

11.8.1 References

- Scanning your Conda environment for security vulnerabilities (pythonspeed.com)
- Jake (github.com)
- Jake + Conda-Forge issue (github.com)

12 Best practices: Pipenv

12.1 Install Pipenv separately from your application code

If you're going to use Pipenv inside your Docker build, you need to install it. Since it has a variety of dependencies, you don't want to install it into the same place as your code, in case the dependencies conflict.

As you'll see in the next best practice, you can either:

1. Install Pipenv as a system package, and have Pipenv install dependencies in a virtualenv.
2. Export dependencies and install them using `pip`. If you follow this path, you could just uninstall Pipenv once you've done the export.

12.1.1 References

- `pipenv` documentation (pipenv.pypa.io)

12.2 Installing dependencies inside a container

There are two methods to install your dependencies inside your Docker build using Pipenv. The first method involves exporting the dependencies to a separate `requirements.txt` file, and then relying on `pip` to install them:

```
FROM python:3.9-slim-bullseye
RUN pip install pipenv

COPY Pipfile* .
```

```
RUN pipenv lock --keep-outdated --requirements > /tmp/requirements.txt

RUN pip install -r /tmp/requirements.txt

# ... install application code, set entrypoint ...
```

The `--keep-outdated` makes sure you use the pinned versions in the `Pipfile.lock`, and don't update the lock file as a side-effect of exporting the packages.

This mechanism does not validate downloaded packages' hashes, a potential security risk; see the issue linked in the references for details.

The other method involves creating a virtualenv and installing the packages there using Pipenv. Pipenv wants to only install in a virtualenv, and creating our own means it's exactly where we want it:

```
FROM python:3.9-slim-bullseye
RUN pip install pipenv

# Create and activate virtualenv:
RUN python3 -m venv /venv
ENV PATH=/venv/bin:$PATH
ENV VIRTUAL_ENV=/venv

COPY Pipfile.lock .

# Install inside virtualenv:
RUN pipenv install --keep-outdated --ignore-pipfile
```

The two options to `pipenv install` allow us to only copy in the `Pipfile.lock`.

12.2.1 References

- `pipenv lock` (pipenv.pypa.io)
- Issue: Generating hashes in `pipenv lock` (github.com)
- `pipenv install` (pipenv.pypa.io)

13 Best practices: Poetry

13.1 Install Poetry separately from your application code

If you're going to use Poetry inside your Docker build, you need to install it. Since it has a variety of dependencies, you don't want to install it into the same place as your code, in case the dependencies conflict.

Some options:

- If you're installing your code into a virtualenv, you can just install Poetry as a system package and it won't conflict with the virtualenv's libraries.
- You can use Poetry's installer to install it into its own isolated directory.

Here's how you do the latter:

```
FROM python:3.9-slim-bullseye

RUN apt-get update && apt-get install -y curl
ENV POETRY_HOME=/tmp/poetry
RUN curl -sSL https://raw.githubusercontent.com/python-
↪  poetry/poetry/master/get-poetry.py | python -
↪  --no-modify-path
ENV PATH=$POETRY_HOME/bin:$PATH
```

The `--no-modify-path` keeps Poetry from modifying `.profile` in order to modify the `PATH`; instead, we do that ourselves in the `Dockerfile`.

13.1.1 References

- Installing `poetry` (python-poetry.org)

13.2 Installing outside of Poetry's managed virtualenvs

By default, Poetry will install code in a virtualenv that it creates and manages. If you want to install either in the normal system packages location, or in a virtualenv of your choosing, add this to your `Dockerfile` before installing any packages:

```
RUN poetry config virtualenvs.create false
```

This will ensure `poetry install` doesn't create its own virtual environment. Instead, it will install in either the normal system packages location, or the current activated virtualenv if there is one.

13.2.1 References

- `poetry's virtualenv.create` (python-poetry.org)

13.3 Don't install development dependencies (unless you want to)

By default Poetry will install development dependencies, things like `flake8` that you've configured in `pyproject.toml` as only being required for development. To disable installing dev dependencies, use the `--no-dev` option.

```
RUN poetry install --no-dev
```

13.3.1 References

- `poetry install` reference (python-poetry.org)

13.4 Speed up rebuilds by installing dependencies separately

For Poetry, you can install dependencies separately to get better build caching by first doing `poetry install --no-root` to just install the dependencies:

```
FROM python:3.9-slim-bullseye

# Install Poetry:
RUN apt-get update && apt-get install -y curl
ENV POETRY_HOME=/tmp/poetry
RUN curl -sSL https://raw.githubusercontent.com/python-
↪  poetry/poetry/master/get-poetry.py | python -
↪  --no-modify-path
ENV PATH=$POETRY_HOME/bin:$PATH

# Don't create virtualenvs:
RUN poetry config virtualenvs.create false

# Copy in the config files:
WORKDIR /myapp
```

```
COPY pyproject.toml poetry.lock ./

# Install only dependencies:
RUN poetry install --no-dev --no-root

# Copy in everything else and install app code:
COPY . .
RUN poetry install --no-dev
```

Important: When your application code is installed, it is *not* copied in, as would be the case when you install some other package like Flask, Django, or NumPy. Instead, Poetry creates a link to the directory where your code resides; in the example above, your code is expected to stay in `/myapp`. **That means you can't delete the original code even though you've installed it.**

If you do want to actually install the code in a way where the code is copied in, instead of using `poetry install` to install your code, do `poetry build` to create a wheel. You can then `pip install` the wheel—you can find it in the `dist/` directory of your project.

Or, you can use a so-called "PEP 517" install. Basically that means you install via `pip`, but `pip` then figures out from `pyproject.toml` how to get the information it needs from Poetry. As a side-effect, it also builds your application code as a wheel and installs it, so you can then delete the original directory.

If you created your `pyproject.toml` with a sufficiently new version of Poetry and `poetry new/init`, this will already be set up. Otherwise make sure `pyproject.toml` has the following section (see the Poetry docs linked below for details):

```
[build-system]
requires = ["poetry_core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

Then, you can change the last part of the `Dockerfile` to install a different way:

```
# ...
# Install Poetry and your dependencies as above.
# ...

# Copy in everything else and install app code:
COPY . .
# Instead of `poetry install`:
RUN pip install .
```

13.4.1 References

- `poetry install` documentation (python-poetry.org)
- `poetry build` documentation (python-poetry.org)
- Poetry and PEP 517 (python-poetry.org)

13.5 Application version changes can lead to slow rebuilds

If you're using Poetry in its default configuration, your package/application version is stored in `pyproject.toml`. That means that every time you change the version, `pyproject.toml` will change. Which means you'll have to reinstall dependencies from scratch when you

rebuild your Docker image, even if the dependencies haven't changed: the new `pyproject.toml` will invalidate the Docker build cache.

How can you fix this?

First, you can choose not to care. If you don't update application versions very often, re-installing all dependencies on a image build won't happen that often.

Second, you can choose not to rely on Poetry's application versioning. Either you can choose not to set versions at all for your package, or you can use something like the `poetry-dynamic-versioning` package that lets you set versions from Git tags (or some other version control). This introduces a dependency on your Git repository during Docker build time, which has its own set of issues (ref).

Third, you can install dependencies via `pip`. Outside of your Docker build, in your build script, you can run `poetry export` to dump the dependencies to a `requirements.txt` file:

```
#!/bin/bash
poetry export -o requirements.txt
docker image build -t myimage .
```

In your `Dockerfile` you can then install dependencies using `pip`:

```
FROM python:3.9-slim-bullseye
COPY requirements.txt .
RUN pip install -r requirements.txt

# etc..
```

13.5.1 References

- `poetry export` documentation (python-poetry.org)

14 Best practices: Multi-stage builds

14.1 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.

14.1.1 References

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

14.2 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:3.9-slim-bullseye 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:3.9-slim-bullseye 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.

14.2.1 References

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

14.3 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.

14.3.1 References

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

14.4 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 possible. 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.

14.5 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:3.9-slim-bullseye 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:3.9-slim-bullseye 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 for Pip, Poetry, or Pipenv-based projects that will allow you to get going in just an hour or two. You can also purchase a template designed for Conda. As a purchaser of the handbook, you can use the discount code `BPQUICKSTART` to get a 15% discount off the templates.

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

—Itamar Turner-Trauring

Changelog

January 26, 2022

- New best practices:
  - Security scans for Conda with Jake.
  - CentOS alternatives.
- Fix `conda activate` in the face of scripts with semi-broken activation scripts, by disabling bash strict mode.
- Update with some changes to `trivy` security scanner.
- Removed the `safety` security scanner, since the non-commercial usage restriction is too limiting.
- Note that `conda-lock` will support pip packages in the near future.

August 31, 2021

- Debian "Bullseye" 11 has replaced Buster as the stable Debian of choice.
- Latest BuildKit backend is now v1.3.
- Documented `tini -g`.
- Added note on dealing with `pip` packages when using `conda-lock`.
- New best practices:
  - Conda environments require activation.
  - Faster Conda installs with Mamba.

June 7, 2021

- Clarified `STOPSIGNAL` usage.

June 1, 2021

- Added best practices:
  - Don't leak secret files.
  - Don't leak runtime secrets.
  - Don't store temporary files in layers.
  - Get rid of unneeded files.
- Added `--nodocs` to `dnf` instructions, for even smaller images.
- Added an alternative method for activating Conda environments.

March 16, 2021

- When using BuildKit you should *always* set `--build-arg BUILDKIT_INLINE_CACHE=1` if you want `--cache-from` to work.
- Noted that official Python Docker image is slower than Ubuntu.

February 8, 2021

- The Quickstart has been renamed to the Handbook; at 100 pages, it's getting more than just introductory.
- New best practices:
  - Installing packages with `pipenv` without exporting.
  - Keeping `pipenv` separate from application code.
  - Recommend `docker build --label` over `LABEL`.

February 3, 2021

- Removed references to CentOS, as it is no longer a stable base image.
- Added link to RedHat's Docker base image.
- Noted Podman can be used with BuildKit.
- Noted need for `--keep-outdated` when using `pipenv lock`.
- Documented how to make Docker Compose use BuildKit.
- New best practices:
  - Caching package downloads across builds using BuildKit.
  - Avoiding dev dependency installs in Poetry.
  - Conda dependency locking using `conda-lock`.

January 28, 2021

- Noted need to pass `--no-capture-output` to `conda run` (thanks to Joe Selvik).

December 17, 2020

Updates for Docker 20.10 and a stable BuildKit.

- Added new chapter covering different Docker releases and BuildKit.
- Switched all BuildKit examples to use the new stable `docker/dockerfile:1.2` version.
- Documented getting BuildKit secrets from environment variables.

November 13, 2020

- Documented PEP 517 Poetry installation usage.
- A large number of minor code fixes throughout, as well as some typos in the text.
- HTML version is now included, for easier copy/pasting.
- Noted an issue with Conda-Pack that will hopefully be fixed soon.

October 22, 2020

- Expanded and more accurate Poetry two-step install, plus added three more Poetry-related best practices.
- Added best practice on adding `.git` to `.dockerignore`, and what to do when you can't.

September 22, 2020

Added best practices:

- Using `conda clean` for smaller Conda-based Docker images.
- Using OpenBLAS instead of MKL for smaller Docker images.
- Using `conda-pack` and multi-stage builds for even smaller Conda-based Docker images.

Noted that the `safety` vulnerability scanner is not licensed for commercial use.

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.