Using Alpine can make Python Docker builds 50× slower

by Itamar Turner-Trauring
Last updated 19 Aug 2020, originally created 29 Jan 2020

When you're choosing a base image for your Docker image, Alpine Linux is often recommended. Using Alpine, you're told, will make your images smaller and speed up your builds. And if you're using Go that's reasonable advice.

But if you're using Python, Alpine Linux will quite often:

1. Make your builds much slower.
2. Make your images bigger.
3. Waste your time.
4. On occassion, introduce obscure runtime bugs.

Let's see why Alpine is recommended, and why you probably shouldn't use it for your Python application.

Why people recommend Alpine

Let's say we need to install gcc as part of our image build, and we want to see how Alpine Linux compares to Ubuntu 18.04 in terms of build time and image size.

First, I'll pull both images, and check their size:

```
$ docker pull --quiet ubuntu:18.04
docker.io/library/ubuntu:18.04
$ docker pull --quiet alpine
docker.io/library/alpine:latest
$ docker image ls ubuntu:18.04
REPOSITORY          TAG        IMAGE ID         SIZE
ubuntu              18.04      ccc6e87d482b     64.2MB
$ docker image ls alpine
REPOSITORY          TAG        IMAGE ID         SIZE
alpine              latest     e7d92cdc71fe     5.59MB
```

As you can see, the base image for Alpine is much smaller.

Next, we'll try installing gcc in both of them. First, with Ubuntu:

```
FROM ubuntu:18.04
RUN apt-get update && \
    apt-get install --no-install-recommends -y gcc && \
    apt-get clean && rm -rf /var/lib/apt/lists/*
```

added complexity would obscure the main point of the article. To ensure you're writing secure, correct, fast Dockerfiles, consider my quickstart guide, which includes 60+ best practices, from security to performance.

We can then build and time that:

```
$ time docker build -t ubuntu-gcc -f Dockerfile.ubuntu --qui
et .
sha256:b6a3ee33acb83148cd273b0098f4c7eed01a82f47eeb8f5bec775
c26d4fe4aae

real    0m29.251s
user    0m0.032s
sys     0m0.026s
$ docker image ls ubuntu-gcc
REPOSITORY   TAG      IMAGE ID      CREATED         SIZE
ubuntu-gcc   latest   b6a3ee33acb8  9 seconds ago   150MB
```

Now let's make the equivalent Alpine `Dockerfile`:

```
FROM alpine
RUN apk add --update gcc
```

And again, build the image and check its size:

```
$ time docker build -t alpine-gcc -f Dockerfile.alpine --qui
et .
sha256:efd626923c1478ccde67db28911ef90799710e5b8125cf4ebb2b2
ca200ae1ac3

real    0m15.461s
user    0m0.026s
sys     0m0.024s
$ docker image ls alpine-gcc
REPOSITORY   TAG      IMAGE ID       CREATED         SIZE
alpine-gcc   latest   efd626923c14   7 seconds ago   105MB
```

As promised, Alpine images build faster and are smaller: 15 seconds instead of 30 seconds, and the image is 105MB instead of 150MB. That's pretty good!

But when we switch to packaging a Python application, things start going wrong.

Let's build a Python image

We want to package a Python application that uses `pandas` and `matplotlib`. So one option is to use the Debian-based official Python image (which I

```
RUN pip install --no-cache-dir matplotlib pandas
```

And when we build it:

```
$ docker build -f Dockerfile.slim -t python-matpan.
Sending build context to Docker daemon  3.072kB
Step 1/2 : FROM python:3.8-slim
 ---> 036ea1506a85
Step 2/2 : RUN pip install --no-cache-dir matplotlib pandas
 ---> Running in 13739b2a0917
Collecting matplotlib
  Downloading matplotlib-3.1.2-cp38-cp38-manylinux1_x86_64.w
hl (13.1 MB)
Collecting pandas
  Downloading pandas-0.25.3-cp38-cp38-manylinux1_x86_64.whl
 (10.4 MB)
...
Successfully built b98b5dc06690
Successfully tagged python-matpan:latest

real    0m30.297s
user    0m0.043s
sys     0m0.020s
```

The resulting image is 363MB.

Can we do better with Alpine? Let's try:

```
FROM python:3.8-alpine
RUN pip install --no-cache-dir matplotlib pandas
```

And now we build it:

```
$ docker build -t python-matpan-alpine -f Dockerfile.alpine
.
Sending build context to Docker daemon  3.072kB
Step 1/2 : FROM python:3.8-alpine
 ---> a0ee0c90a0db
Step 2/2 : RUN pip install --no-cache-dir matplotlib pandas
 ---> Running in 6740adad3729
Collecting matplotlib
  Downloading matplotlib-3.1.2.tar.gz (40.9 MB)
    ERROR: Command errored out with exit status 1:
     command: /usr/local/bin/python -c 'import sys, setuptoo
ls, tokenize; sys.argv[0] = '"'"'/
tmp/pip-install-a3olrixa/matplotlib/setup.py'"'"'; __file__
='"'"'/tmp/pip-install-a3olrixa/matplotlib/setup.py'"'"';f=g
etattr(tokenize, '"'"'open'"'"', open)(__file__);code=f.read
().replace('"'"'\r\n'"'"', '"'"'\n'"'"');f.close();exec(comp
ile(code, __file__, '"'"'exec'"'"'))' egg_info --egg-base /t
```

```
ERROR: Command errored out with exit status 1: python setup.
py egg_info Check the logs for full command output.
The command '/bin/sh -c pip install matplotlib pandas' retur
ned a non-zero code: 1
```

What's going on?

Standard PyPI wheels don't work on Alpine

If you look at the Debian-based build above, you'll see it's downloading `matplotlib-3.1.2-cp38-cp38-manylinux1_x86_64.whl`. This is a pre-compiled binary wheel. Alpine, in contrast, downloads the source code (`matplotlib-3.1.2.tar.gz`), because standard Linux wheels don't work on Alpine Linux.

Why? Most Linux distributions use the GNU version (`glibc`) of the standard C library that is required by pretty much every C program, including Python. But Alpine Linux uses `musl`, those binary wheels are compiled against `glibc`, and therefore Alpine disabled Linux wheel support.

Most Python packages these days include binary wheels on PyPI, significantly speeding install time. **But if you're using Alpine Linux you need to compile all the C code in every Python package that you use.**

Which also means you need to figure out every single system library dependency yourself. In this case, to figure out the dependencies I did some research, and ended up with the following updated `Dockerfile`:

```
FROM python:3.8-alpine
RUN apk --update add gcc build-base freetype-dev libpng-dev
 openblas-dev
RUN pip install --no-cache-dir matplotlib pandas
```

And then we build it, and it takes...

... 25 minutes, 57 seconds! And the resulting image is 851MB.

Here's a comparison between the two base images:

Base image	Time to build	Image size	Research required
`python:3.8-slim`	30 seconds	363MB	No
`python:3.8-alpine`	1557 seconds	851MB	Yes

Alpine builds are vastly slower, the image is bigger, and I had to do a bunch of research.

Can't you work around these issues?

Build time

release does not include these popular packages.

Even when they are available, however, system packages almost always lag what's on PyPI, and it's unlikely that Alpine will ever package everything that's on PyPI. In practice most Python teams I know don't use system packages for Python dependencies, they rely on PyPI or Conda Forge.

Image size

Some readers pointed out that you can remove the originally installed packages, or add an option not to cache package downloads, or use a multi-stage build. One reader attempt resulted in a 470MB image.

So yes, you can get an image that's in the ballpark of the slim-based image, but the whole motivation for Alpine Linux is smaller images and faster builds. With enough work you may be able to get a smaller image, but you're still suffering from a 1500-second build time when they you get a 30-second build time using the `python:3.8-slim` image.

But wait, there's more!

Alpine Linux can cause unexpected runtime bugs

While in theory the `musl` C library used by Alpine is mostly compatible with the `glibc` used by other Linux distributions, in practice the differences can cause problems. And when problems do occur, they are going to be strange and unexpected.

Some examples:

1. Alpine has a smaller default stack size for threads, which can lead to Python crashes.
2. One Alpine user discovered that their Python application was much slower because of the way musl allocates memory vs. glibc.
3. I once couldn't do DNS lookups in Alpine images running on minikube (Kubernetes in a VM) when using the WeWork coworking space's WiFi. The cause was a combination of a bad DNS setup by WeWork, the way Kubernetes and minikube do DNS, and musl's handling of this edge case vs. what glibc does. musl wasn't *wrong* (it matched the RFC), but I had to waste time figuring out the problem and then switching to a glibc-based image.
4. Another user discovered issues with time formatting and parsing.

Most or perhaps all of these problems have already been fixed, but no doubt there are more problems to discover. Random breakage of this sort is just one more thing to worry about.

Don't use Alpine Linux for Python images

Unless you want massively slower build times, larger images, more work, and the potential for obscure bugs, you'll want to avoid Alpine Linux as a base image. For some recommendations on what you should use, see my article on choosing a good base image.

Next: When to switch to Python 3.9
Previous: A deep dive into the official Docker image for Python