Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Intro to Containers

15/06/2023

BSC-CES - MWT

Bruno P. Kinoshita

Research Engineer

Computational Earth Sciences | Earth Sciences Department
Barcelona Supercomputing Center - Centro Nacional de Supercomputación
Address: C/ Jordi Girona, 31, 08034 Barcelona | Torre Girona, 2nd Floor
BSC Website: https://www.bsc.es/

Agenda

Containerization

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Prerequisites

If you would like to try the examples in these slides, you will need to:

- Install Docker
- Be able to run `` `docker run hello-world` ``
- Have Singularity (CE > 3.7) installed (optional)
- Have Docker and Singularity compose tools (optional)
- Have ~10 GB at least to spare, as well as good Internet connectivity.

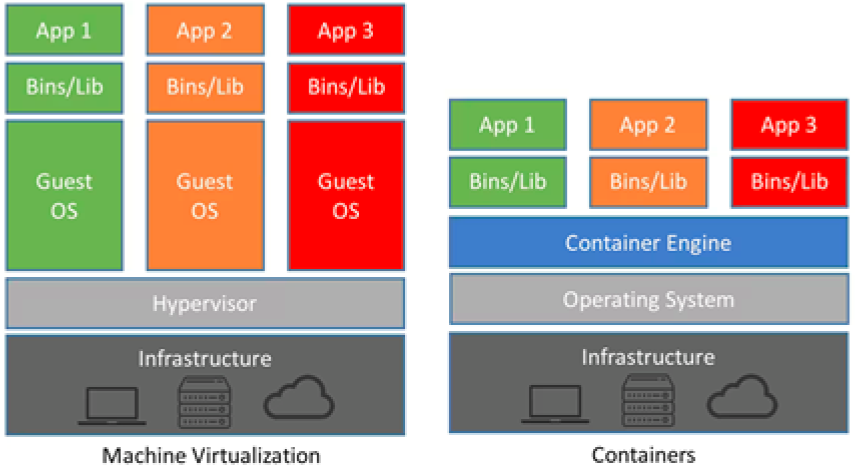
> ※ NOTE: The example code snippets in this presentation may take several minutes to complete. If you are attending a session of this presentation, try running `` `docker pull image_name:version` `` before the presentation begins.

Containerization

Containerization is a way to run applications in an isolated environment.

It can take place at **OS level** and at **application level**.

This talk is about **containerization at OS level**, more specifically about containers with Docker (with a little about Singularity).



[1]

1. Trend Micro (2022). The Difference Between Virtual Machines and Containers ↩

Some context and history

Containers are based on features of Kernel that have been available for quite some years (`chroot`, and Linux namespaces such as `cgroups`).

Containers did not start with Docker, but Docker was responsible for its recent widespread use in software development.

Refs: [1][2][3]

- 1997: Unix V7 adds `chroot`
- 2000: FreeBSD jails
- 2004: Solaris containers
- 2005: Open VZ (Open Virtuoso) (unreleased patch of Linux Kernel)
- 2006: Process Containers (Google)
- 2008: LXC (first container manager!)
- 2013: Docker (used LXC initially, then replaced by libcontainer)
- 2015: Open Container Initiative (Linux Foundation, `runc`)
- 2015: Singularity (Lawrence Berkeley National Lab)
- 2016: udocker (Python tool to execute Docker containers without root)
- 2017: Charliecloud (daemonless, rootless for HPC, created in 2015, but paper from 2017)
- 2018: Podman (daemonless, OS, uses OCI containers and images - compatible with Docker)
- 2019: Spack added the `spack containerize` command (#14202)

1. https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016 ↩
2. https://en.wikipedia.org/wiki/Linux_namespaces ↩
3. https://github.com/opencontainers/runc ↩

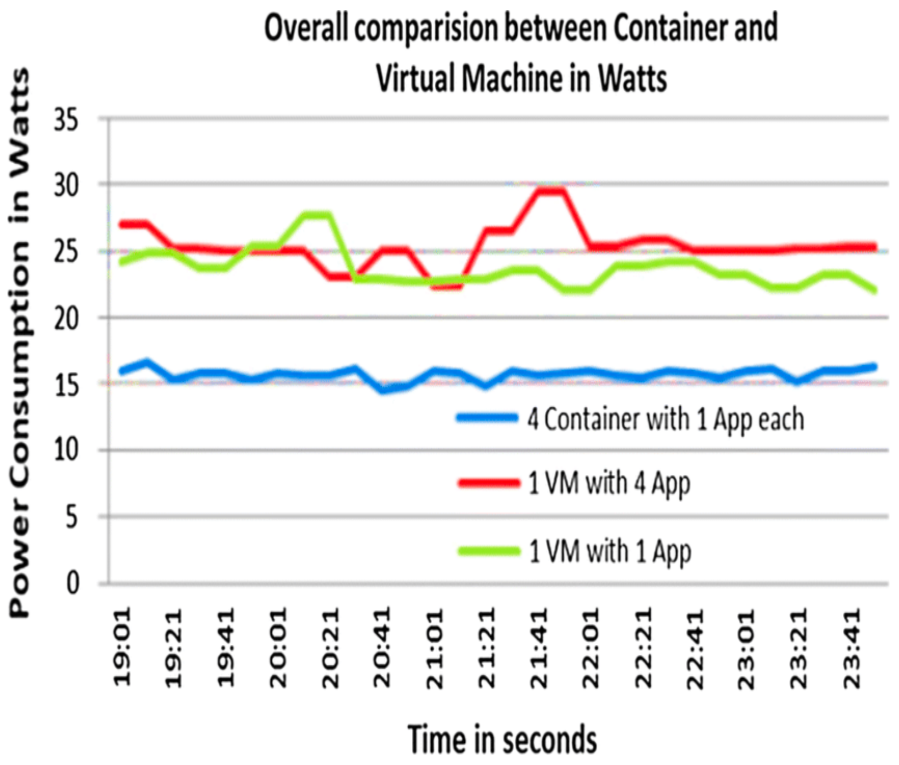
Why containers?

- Test software in a safely sandboxed environment
- Faster development lifecycle
  - Faster than managing multiple local envs or VM’s
- Isolate applications
- Create portable environments
  - More portable than VM’s [1]
- Scalability
- Cloud computing
- Disaster recovery
- Lower footprint than servers or virtual machines
  - In both memory, computing, and carbon[2] (more in the next slide)
  - (Or at least for the cloud[3], I need to find HPC studies)

1. Containers vs VMs (virtual machines) (Google Cloud) ↩
2. Anusooya, G. & Varadarajan, Vijayakumar. (2021) ↩
3. Application platform considerations for sustainable workloads on Azure ↩

Barcelona Supercomputing Center

Refs: [1][2] (networking plays an important role, valid for cloud, HPC may have different numbers.)



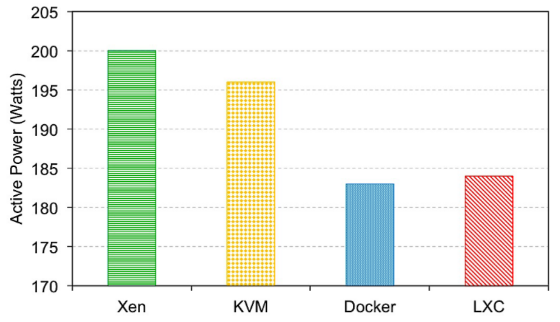
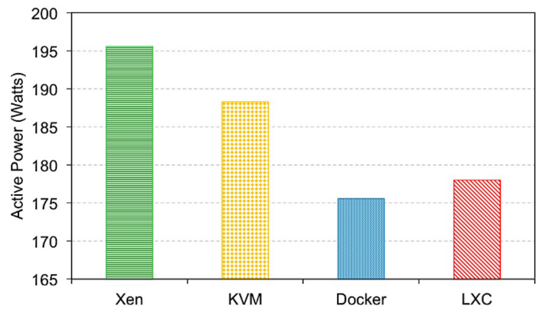


Fig. 9. Power consumption of multiple VMs/containers, which are receiving TCP traffic (MTU 1500-bytes)



Demo (follow along!)

Running Docker containers with the command-line

Try these commands!

```
# Run a terminal in Debian Bullseye slim image with Micromamba
$ docker run -ti -u 1000:1000 mambaorg/micromamba:bullseye-slim
(base) mambauser@5df690afaa1d:/tmp$

# Run a terminal in the latest Ubuntu release
$ docker run -ti ubuntu:latest
root@51a769cec6c0:/#

# Run an old version of Linux Alpine 2.7.18 Python 2.7.3
$ docker run -ti python:2.7.18-alpine3.11
Python 2.7.18 (default, Apr 20 2020, 19:51:05)
[GCC 9.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>

# Start the terminal instead of Python
$ docker run -ti python:2.7.18-alpine3.11 /bin/ash
/ #

# Execute some Python 2 code with an environment variable
$ docker run -ti -e WORLD=Earth python:2.7.18-alpine3.11 python2 -c \
  'import os; print "Hello " + os.environ["WORLD"]'
Hello Earth
```

Running Docker containers with the command-line

```
# Look at all the mess you made!
$ docker ps -a
CONTAINER ID   IMAGE                               COMMAND                  CREATED          STATUS                       P
78ec649e58b2   python:2.7.18-alpine3.11            "python2 -c 'print \"…"   11 seconds ago   Exited (0) 10 seconds ago
18f61a30ceb4   python:2.7.18-alpine3.11            "/bin/ash"               18 seconds ago   Exited (0) 16 seconds ago
d8a7cace5eb3   python:2.7.18-alpine3.11            "python2"                20 seconds ago   Exited (0) 18 seconds ago
64bf9b4b81bc   ubuntu:latest                       "/bin/bash"              32 seconds ago   Exited (130) 29 seconds ago
f9085a740dd1   mambaorg/micromamba:bullseye-slim   "/usr/local/bin/_ent…"   50 seconds ago   Exited (0) 48 seconds ago
3ca48baf860d   mambaorg/micromamba:bullseye-slim   "/usr/local/bin/_ent…"   5 minutes ago    Up 5 minutes
```

```
# Show system information
$ docker system df
TYPE            TOTAL     ACTIVE    SIZE      RECLAIMABLE
Images          3         3         243.7MB   0B (0%)
Containers      6         1         587.6kB   587.6kB (100%)
Local Volumes   0         0         0B        0B
Build Cache     0         0         0B        0B
```

Other useful commands

```
# View all the images you have downloaded
$ docker image ls
...

# Remove the containers you have created
$ docker container prune -f
Deleted Containers:
0e121a312675fafb2933f3def49e488c0b9b20fcb93aafb094e923bcc90a9605
111f533d7cd177f7e8b65e083dfff1335ef39830d94dbf9917a7a1b10882cfc7
bec5117a015484334b9e5368c8be70f1c609d3ac93a55ab3b67a0a92954dcd73
77ee9570b2dcf055c527d7dfdd363cbb3699b66295540be14923189435b68ad4

Total reclaimed space: 587.6kB

# See everything you have running
$ docker ps -a
...

# Or volumes or networks
$ docker network list
$ docker volume list
...
```

Supercomputing Center
Centro Nacional de Supercomputación

Other useful commands

```
# You can clean up volumes, networks, images, containers, etc.

$ docker volume prune -f
Deleted Volumes:
95c54fdcf9fa352203d7c1e009a19f2cfeebcef29cca5fb54bed923df60e701c

Total reclaimed space: 478.3MB
```

```
# Or do what I do from time to time
$ docker system prune -a -f
...
Total reclaimed space: 3.189GB
```

More useful commands

```
# Run a container that will auto-delete itself when its process exits
$ docker run --rm python:2.7.18-alpine3.11

# Run a container with a name, as a daemon (in the background)
$ docker run -d --rm --name python2test python:2.7.18-alpine3.11
27909a75647fbd8bb700fcdc2e25019958a8d0ecda74227a1f74793e8caf4cab
```

Creating images with Dockerfile

File: Dockerfile

```
FROM python:2.7.18-alpine3.11
WORKDIR /app

COPY hello.txt .

RUN cat <<EOF > script.py
#!/usr/bin/python2

with open('hello.txt') as f:
  msg = f.read()
  print(msg)
  with open('/tmp/output.txt', 'w+') as o:
    o.write(msg)
    o.flush()

EOF

CMD ["python2", "script.py"]
```

`RUN`, `ADD`, and `COPY` create new layers. Layers are cached for performance.

Creating images with Dockerfile

```
$ echo "Hello World" > hello.txt
# The dot at the end is for the local dir, where Dockerfile is
$ docker build -t test-python:latest .
[+] Building 0.7s (9/9) FINISHED
 ⇒ [internal] load .dockerignore                                       0.0s
 ⇒ ⇒ transferring context: 2B                                          0.0s
 ⇒ [internal] load build definition from Dockerfile                    0.0s
 ⇒ ⇒ transferring dockerfile: 231B                                     0.0s
 ⇒ [internal] load metadata for docker.io/library/python:2.7.18-alpine3.  0.0s
 ⇒ CACHED [1/4] FROM docker.io/library/python:2.7.18-alpine3.11        0.0s
 ⇒ [internal] load build context                                       0.0s
 ⇒ ⇒ transferring context: 48B                                         0.0s
 ⇒ [2/4] WORKDIR /app                                                  0.2s
 ⇒ [3/4] COPY hello.txt .                                              0.0s
 ⇒ [4/4] RUN cat <<EOF > script.py                                     0.3s
 ⇒ exporting to image                                                  0.1s
 ⇒ ⇒ exporting layers                                                  0.1s
 ⇒ ⇒ writing image sha256:de797699e542f0fd08cd4321d434754334df8399d5b3c  0.0s
 ⇒ ⇒ naming to docker.io/library/test-python:latest                    0.0s
```

Creating images with Dockerfile

What if you run that command again?

Look at the lines that contain `CACHED`!

```
$ docker build -t test-python:latest .
[+] Building 0.1s (9/9) FINISHED
⇒ [internal] load build definition from Dockerfile                   0.0s
⇒ ⇒ transferring dockerfile: 231B                                    0.0s
⇒ [internal] load .dockerignore                                      0.0s
⇒ ⇒ transferring context: 2B                                         0.0s
⇒ [internal] load metadata for docker.io/library/python:2.7.18-alpine3.  0.0s
⇒ [1/4] FROM docker.io/library/python:2.7.18-alpine3.11              0.0s
⇒ [internal] load build context                                      0.0s
⇒ ⇒ transferring context: 30B                                        0.0s
⇒ CACHED [2/4] WORKDIR /app                                          0.0s
⇒ CACHED [3/4] COPY hello.txt .                                      0.0s
⇒ CACHED [4/4] RUN cat <<EOF > script.py                             0.0s
⇒ exporting to image                                                 0.0s
⇒ ⇒ exporting layers                                                 0.0s
⇒ ⇒ writing image sha256:de797699e542f0fd08cd4321d434754334df8399d5b3c  0.0s
⇒ ⇒ naming to docker.io/library/test-python:latest                   0.0s
```

Creating images with Dockerfile

Your image should be available in your local environment now.

```
$ docker image ls test-python
REPOSITORY    TAG       IMAGE ID       CREATED           SIZE
test-python   latest    39b47fff6145   45 minutes ago    71.1MB
```

But no container.

```
$ docker ps -a --filter ancestor=test-python:latest
CONTAINER ID   IMAGE     COMMAND    CREATED    STATUS     PORTS     NAMES
```

Yet!

```
# You can use `docker container run` too (newer syntax)
$ docker run --name test-python2 test-python:latest
Hello World

# Or `docker container ps` ...
$ docker ps -a --filter ancestor=test-python:latest
CONTAINER ID   IMAGE                COMMAND                CREATED              STATUS                          PORTS     NAME
48ec9a91370e   test-python:latest   "python2 script.py"    About a minute ago   Exited (0) About a minute ago             test
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Volumes and persisting data

The container runs a script with Python 2. This script writes to the stdout and to a file on `/tmp/output.txt`.

But what if you wanted to access `output.txt`?

```
$ ls
Dockerfile  hello.txt
```

You can use a volume (similar to volumes in VM’s like VirtualBox!), and map the `/tmp` inside the container to a local directory.

```
$ docker run --rm -ti -v ${PWD}:/tmp test-python:latest
Hello World

$ ls
Dockerfile  hello.txt  output.txt
```

Volumes and persisting data

You can even modify the `hello.txt` file.

```
$ echo "Hola Món" > earth.txt
# In the host machine, it is `earth.txt`, but mapped as `hello.txt` in the guest.
$ docker run --rm -ti -v ${PWD}/earth.txt:/app/hello.txt test-python:latest
Hola Món
```

> ※ NOTE: Avoid persisting data inside your container, see best practices slides.

Organize your container so that users can control input and output
(imagine your container is a stateless math function)

If a container goes down, for whatever reason, start a new one using the same shared volume. You can have multiple containers sharing the same volume, you can mount volumes as readonly, and more.

Networking

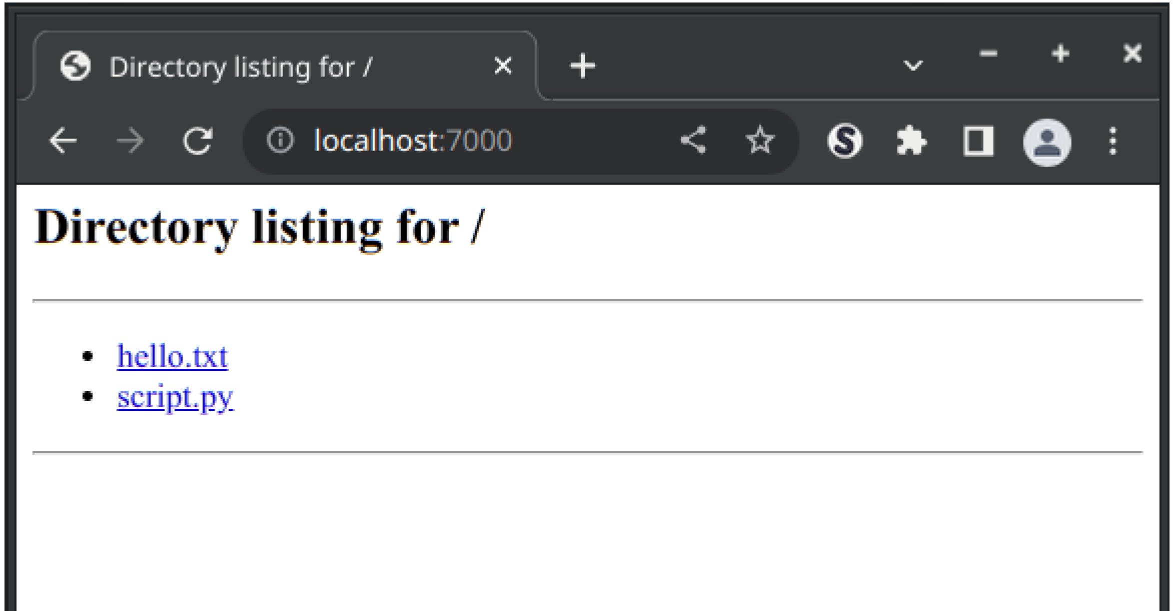
Quick example to show how to expose a port in Docker.

You can modify the command (`CMD`) executed by a container.

```
$ docker run --rm -ti test-python:latest python2 -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
# CTRL + C

# Run is now as a daemon
$ docker run --rm -d test-python:latest python2 -m SimpleHTTPServer
cc38293a5265e609d76cd440b95e775175e6da72d03cccc098d8a35483e3878a

# Note the `PORT` column!
$ docker ps
CONTAINER ID   IMAGE                COMMAND                   CREATED          STATUS          PORTS     NAMES
cc38293a5265   test-python:latest   "python2 -m SimpleHT…"   34 seconds ago   Up 33 seconds             nifty_elion

$ docker stop cc38293a5265
cc38293a5265
```

Networking

```
# Specify a port to bind with `-p $HOST:$GUEST`
$ docker run --rm -d -p 7000:8000 test-python:latest python2 -m SimpleHTTPServer
34be3a39dc27516805d9698581e34edbc0da6f1f0e7843090cd3218830dd1c88

# Look at `PORT` now!
$ docker ps
CONTAINER ID   IMAGE                COMMAND                  CREATED              STATUS             PORTS
34be3a39dc27   test-python:latest   "python2 -m SimpleHT…"   About a minute ago   Up About a minute  0.0.0.0:7000→8000/tcp,
```



Directory listing for /

- hello.txt
- script.py

Creating a Singularity container

You can write a Singularity file, download from Singularity Hub or Docker Hub, or use a local Docker image.

```
# The chroot way... you can write to it with --writable
$ singularity build --sandbox test-python docker-daemon://test-python:latest
...

# Or build a single container file to be loaded into memory... (what I normally use)
$ sudo singularity build test-python.sif docker-daemon://test-python:latest
INFO:    Starting build...
2023/06/14 18:29:13  info unpack layer: sha256:c1f002e71ff26f48d9071266c88531cc7740de5f12a710395e68ea604ed4ff6a
2023/06/14 18:29:13  info unpack layer: sha256:7dfeae6c1959458377be7cb43c8d4bba4d64c09f2aa569b9806ef90ffa4bfae8
2023/06/14 18:29:13  info unpack layer: sha256:eae0303b3277085a9830077a7ff6e679ef772961676015d0418df72ac5de3582
2023/06/14 18:29:13  info unpack layer: sha256:e1a35eb1d0f6bcf41aef81ad625a2024de331991fc453573b6a4ab644f42aaf7
2023/06/14 18:29:14  info unpack layer: sha256:cdbb2ced0e763d99157788e6b3b1b04f63f442467a34cbfb35cd7ab6825f70da
2023/06/14 18:29:14  info unpack layer: sha256:f1cfa610d290fd7eaac1a5f60f8494c9703e042a2240b3b36eafa5e37c3f8ff5
2023/06/14 18:29:14  info unpack layer: sha256:fe2555b8984cba0a403a87269a257667fddc0f527231de75245d8fe9f2cf3c0d
INFO:    Creating SIF file...
INFO:    Build complete: test-python.sif
```

Creating a Singularity container

The latter command gives you `test-python.sif` file. You can run the container now.

```
$ singularity run test-python.sif
/usr/local/bin/python2: can't open file 'script.py': [Errno 2] No such file or directory
```

> ※ NOTE: You can convert Docker to Singularity. That does not mean everything will work automatically. Portability requires careful design (other examples: MPI, GPU.)

```
$ singularity run --pwd /app test-python.sif
Hello World

$ singularity shell test-python.sif
Singularity>
```

Singularity will mount your `$HOME` or `$PWD` (depending on the version) and use as the working directory. In the default configuration, the system default bind points are `$HOME`, `/sys:/sys`, `/proc:/proc`, `/tmp:/tmp`, `/var/tmp:/var/tmp`, `/etc/resolv.conf:/etc/resolv.conf`, `/etc/passwd:/etc/passwd`, and `$PWD`.

Barcelona Supercomputing

Testing on the HPC

An exercise for you. Try running this, or another Docker → Singularity container on your HPC.

Something like

```
# Upload it to some offline node, for example (or save to a NFS partition to use with computing nodes.)
bruno@bscearth000:~> scp test-python.sif bsc32...@mn2.bsc.es:/home/bsc32/bsc32..../

# Jump onto that host and load the Singularity module
bruno@bscearth000:~> ssh bsc32....@mn2.bsc.es
bsc32....@login2:~> module load Singularity

# Run the container
bsc32....@login2:~> singularity run --pwd /app test-python.sif
Hello World
```

That’s that! You have a container that you can use on your laptop, in the cloud, or in an HPC. For other solutions that involve MPI or GPU they may not be as portable.

But containerization is a useful skill to have in your tool-belt, especially if you collaborate with people from external institutions.

Deploying containers

- You can deploy images locally (as you have seen in this talk)
- You can deploy images to Docker Hub or Singularity Hub
- You can deploy Singularity images to Singularity Registries (they can be installed as modules)
- You can deploy images to Quay.io
- You can deploy images to Biocontainers
- AWS has a registry for containers, as well as Google Cloud, OpenStack
- You can deploy to Docker Swarm
- You can deploy images to Kubernetes
- You can deploy images to GitHub Packages (limitations for free projects apply)
- …

Container orchestration ({Docker, Singularity} Compose, Kubernetes, ...)

Docker Swarm and Kubernetes are example of container orchestration tools.

They provide tools for deploying pods and using containers, and for scaling, networking, securing, and maintaining containerized applications.

Docker Compose, and Singularity Compose are tools that also can be used to manage clusters of containers. Users can declare how the containers must be created and interlinked (networking, volumes). Useful when deploying more than 1 container locally or remotely (especially if sharing with others).

There are other tools, some offered by cloud providers like AWS.

Best practices

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Docker development

1. Keep your images small
   - (For Cloud environments it is important, but for HPC not so much)
   - Use a smaller image for production, but additional debug tooling added for troubleshooting
   - Use multistage builds (※ [1]!) and leverage the build cache
   - Use `docker image history $image` to view the layers in your image
2. Use official images where possible
3. Create **ephemeral containers**
4. Manage your build context (see `.dockerignore`)
5. Avoid storing application data in the container
   - Use volumes or bind mounts
6. Limit each container to one process and application (whenever that is possible)
7. Add labels to add metadata to your image (license, author, release information, etc.)
8. Use `RUN set -o pipefail && cmd1 | cmd2` if you want to use pipes (Docker uses `/bin/sh -c`)
9. Declare network ports used in `EXPOSE`
10. Expose environment variables for the software with `ENV`
11. `COPY` is preferred over `ADD` whenever possible
12. Portability and reproducibility are hard. Test it! (Developing containers is software development!)

1. This is omitted from the current version of this presentation, but it is an important Docker concept. ↩

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

References

These slides are based on (links with underscore):

- Docker development best practices
- Best practices for writing Dockerfiles
- Image-building best practices
- Security best practices
- DestinE Barcelona Coding Sprint CSC presentation
- Best practices for building and running Docker and Singularity containers

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you

bruno.depaulakinoshita@bsc.es

Security

- Use official images
- Use the right context
- Use secrets to store sensitive application data
- If you are using a `` `:latest` `` base image, rebuild your image to get the latest updates
- Use security scanning tools to check your image for vulnerabilities
- WIP, more to come!

Singularity and Docker

Pro Singularity

- Singularity has a single file, simplified I/O
- Simplified security model, more compatible with HPC
- Easier to work with Slurm (Docker has a daemon)

Con Singularity

- Docker has a wider user/knowledge base
- Docker is used by many types of software developers
- Docker has more containers ready-to-use

Other possible issues

- Singularity had a big rewrite from C to Go
- Singularity modified its file format (OCI should change it)
- Singularity, Singularity CE & Apptainer

Most HPC centers have their own set of instructions for users to use Singularity, e.g. Running MPI apps on Singularity at BSC MN4

HPC Examples

- hzdr.de recipe for Slurm in Docker https://codebase.helmholtz.cloud/fwcc/slurm-in-docker
- Sweden's PDC center for HPC instructions on using Singularity https://www.pdc.kth.se/support/documents/software/singularity.html
- NeSI (New Zealand) Knowledge Base entry on using Singularity https://support.nesi.org.nz/hc/en-gb/articles/360001107916-Singularity
- NCAR recipes for WRF with Docker https://github.com/NCAR/WRF_DOCKER & https://github.com/NCAR/container-wrf
- Singularity docs for MPI https://docs.sylabs.io/guides/latest/user-guide/mpi.html
- Auburn HPC docs about MPI and Slurm with Singularity
  https://hpc.auburn.edu/hpc/docs/hpcdocs/build/html/easley/containers.html#parallel-mpi-code
- Pawsey repository with container recipes (conda, mpi, OpenFoam, etc.) https://github.com/PawseySC/pawsey-containers/tree/master
- IFCA (ES) Docker containers for Distributed Training of Deep Learning Algorithms in HPC Clusters https://github.com/IFCA/workflow-DL-HPC/
- CANOPIE-HPC 2023 https://canopie-hpc.org/cfp/

Workflow Managers

- cwltool, reference CWL runner, uses containers (docker/udocker/podman/singularity) to isolate tasks (Cylc does so without containers)
- Snakemake supports Docker and singularity too
- Workflow Managers normally support Docker, and then add support to Singularity later. Science/Research Workflow Managers may be an exception to this rule.
- In some workflow managers users must decide if, and how to use containers (e.g. Autosubmit, Cylc, ecFlow)

Other

- https://container-in-hpc.org/ (they have a Slack channel too)
- #containers channel in Slack/Mattermost/etc.
-