Container Hands-on Lab - Docker Track

90-minute beginner lab - Docker/Desktop track

Casey Morean

5/15/2026

The lab starts with container literacy, then uses the JLab ROOT repository as a full reference example.

Lab outcomes

By the end of this hands-on session, participants should be able to:

1. run an existing container image;
2. bind mount input data, configuration, and output directories;
3. pass runtime environment variables and publish a port;
4. build a small analysis image from a Dockerfile;
5. explain what is inside and outside the reproducibility boundary;
6. map the same ideas onto the JLab ROOT Python/C++ reference repository;
7. recognize how the pattern changes under Apptainer and Slurm.

90-minute lab map

Time	Activity
0:00-0:10	Setup and verify Docker
0:10-0:30	Run existing images, inspect user space, observe ephemerality
0:30-0:45	Bind mounts, environment variables, and port publishing
0:45-1:05	Build a minimal analysis image with mounted data/config/output
1:05-1:20	Use the JLab ROOT repository as a worked reference example
1:20-1:30	Apptainer/HPC mapping, provenance checklist, exit ticket

Checkpoint: The main thrust for new users is the container model. The ROOT repository is the realistic worked example.

Where we are

1. Part A: run containers first
2. Part B: connect the host to the container
3. Part C: build a minimal analysis image
4. Part D: JLab ROOT reference example
5. Part E: Apptainer, Slurm, and provenance

0. Create a clean lab directory

Everyone runs this on the host:

```
mkdir -p container-lab/{data,configs,output,site,mini-analysis}
cd container-lab
pwd
```

Checkpoint: You should be in a new directory named `container-lab`.

1. Verify Docker

Use this deck if you are on Docker Desktop or Docker Engine:

```
docker --version
docker version
```

Checkpoint: The command should print a Docker client and server version. If the server/daemon check fails, fix Docker before continuing.

2. Pull and run a public image

```
docker pull python:3.12-slim

docker run --rm python:3.12-slim python --version
```

Teaching prompt

What did we just download: an image or a container?

Checkpoint: Expected: Python prints a 3.12 version and the container exits.

3. Inspect the container's user space

```
docker run --rm python:3.12-slim sh -lc '
  echo "kernel: $(uname -r)"
  echo "user:   $(id)"
  echo "os release inside image:"
  head -5 /etc/os-release
'
```

Teaching prompt

Which part came from the host, and which part came from the image?

Checkpoint: Kernel version comes from the host or VM. `/etc/os-release` comes from the image filesystem.

4. Run an interactive shell

```
docker run --rm -it python:3.12-slim sh
```

Inside the container:

```
python --version
pwd
ls /
exit
```

Instructor cue

Keep this short. The point is to see the filesystem, not to debug inside the container for 20 minutes.

5. Containers are ephemeral

Write a file inside one container:

```
docker run --rm python:3.12-slim sh -lc '
  echo "made inside the container" > /tmp/from-container.txt
  cat /tmp/from-container.txt
'
```

Look for it in a new container:

```
docker run --rm python:3.12-slim sh -lc '
  ls /tmp/from-container.txt || echo "not here"
'
```

Checkpoint: The second container starts from the image again; the temporary file is gone.

- This behavior is the same with Docker and Podman.
- `-rm` removes the stopped container after the command exits.
- Files only persist if they are written to a mounted host directory or a named volume.

Stop and name the objects

Image

- `python:3.12-slim`
- downloaded once and reused;
- packaged filesystem plus metadata.

Container

- each `run` invocation;
- a process created from the image;
- deleted after exit because of `--rm`.

Where we are

1. Part A: run containers first
2. Part B: connect the host to the container
3. Part C: build a minimal analysis image
4. Part D: JLab ROOT reference example
5. Part E: Apptainer, Slurm, and provenance

6. Create input data on the host

```
cat > data/numbers.txt <<'DATA'
1 2 3 4 5 8 13 21
DATA

cat data/numbers.txt
```

Checkpoint: This file is on the host. The container will only see it if we mount it.

7. Bind mount input data read-only

```
docker run --rm \
  -v "$PWD/data:/data:ro" \
  python:3.12-slim \
  sh -lc 'cat /data/numbers.txt'
```

Try to write to the read-only mount:

```
docker run --rm \
  -v "$PWD/data:/data:ro" \
  python:3.12-slim \
  sh -lc 'echo 99 > /data/new.txt'
```

Checkpoint: The write should fail. That failure is desirable for real input data.

8. Bind mount an output directory writable

```
docker run --rm \
  -v "$PWD/output:/output" \
  python:3.12-slim \
  sh -lc 'date > /output/container-ran.txt'

cat output/container-ran.txt
```

Checkpoint: The file survives because it was written to a host-mounted directory.

9. Pass environment variables at runtime

```
docker run --rm \
  -e ANALYSIS_LABEL=trial-001 \
  python:3.12-slim \
  python -c 'import os; print(os.getenv("ANALYSIS_LABEL"))'
```

Teaching prompt

Should this label be recorded in an analysis note? Why?

Checkpoint: Yes, if it changes outputs, behavior, or interpretation.

10. Publish a port

```
echo '<h1>Container lab</h1>' > site/index.html

docker run --rm \
  -p 8000:8000 \
  -v "$PWD/site:/site:ro" \
  -w /site \
  python:3.12-slim \
  python -m http.server 8000
```

Open http://localhost:8000 in a browser, then stop the container with `Ctrl-C`.

Checkpoint: The service runs inside the container; the host port makes it reachable from the browser.

Runtime summary

Runtime option	Scientific meaning
`-v data:/data:ro`	input data is visible but not mutable
`-v configs:/configs:ro`	analysis choices are explicit and versionable
`-v output:/output`	products survive container cleanup
`-e NAME=value`	runtime choice that may need to be recorded
`-p host:container`	service exposure, useful but not pure isolation
`-rm`	disposable container metadata

Where we are

1. Part A: run containers first
2. Part B: connect the host to the container
3. Part C: build a minimal analysis image
4. Part D: JLab ROOT reference example
5. Part E: Apptainer, Slurm, and provenance

The mini-analysis exercise

Goal

Build a tiny analysis image that reads numbers from `/data`, reads a JSON config from `/configs`, writes a result to `/output`, and prints the result.

Why this before ROOT?

Users learn the same boundary pattern without ROOT, CMake, PyROOT, GitLab CI, or Slurm competing for attention.

Checkpoint: The exact same mount pattern will later reappear in the JLab ROOT repository.

11. Write the analysis program

```
cat > mini-analysis/app.py <<'PY'
import argparse, json, pathlib, statistics

parser = argparse.ArgumentParser()
parser.add_argument("--config", default="/configs/config.json")
args = parser.parse_args()

cfg = json.loads(pathlib.Path(args.config).read_text())
data_path = pathlib.Path(cfg.get("input", "/data/numbers.txt"))
out_path = pathlib.Path(cfg.get("output", "/output/result.json"))
scale = float(cfg.get("scale", 1.0))

numbers = [float(x) for x in data_path.read_text().split()]
scaled = [x * scale for x in numbers]
result = {
    "label": cfg.get("label", "unnamed"),
    "n": len(scaled),
    "mean": statistics.fmean(scaled),
    "min": min(scaled),
    "max": max(scaled),
}
out_path.parent.mkdir(parents=True, exist_ok=True)
out_path.write_text(json.dumps(result, indent=2) + "\n")
print(json.dumps(result, indent=2))
PY
```

12. Write the Dockerfile

```
cat > mini-analysis/Dockerfile <<'DOCKER'
FROM python:3.12-slim

LABEL org.opencontainers.image.title="Mini reproducible analysis"
LABEL org.opencontainers.image.description="Training image for data/config/output mounts"

WORKDIR /app
COPY app.py /app/app.py

ENTRYPOINT ["python", "/app/app.py"]
CMD ["--help"]
DOCKER
```

Teaching prompt

Which line adds application code into the image? Which line defines the default process?

13. Build the image

```
docker build \
  -t mini-analysis:dev \
  mini-analysis
```

List the image:

```
docker images | grep mini-analysis
```

Checkpoint: You should see `mini-analysis` with tag `dev`.

14. Run the image with no mounts

```
docker run --rm mini-analysis:dev
```

Expected

The image prints help because the Dockerfile default command is `-help`.

Teaching prompt

Why did we not copy data into the image?

15. Write the config outside the image

```
cat > configs/config.json <<'JSON'
{
  "label": "first-container-result",
  "input": "/data/numbers.txt",
  "output": "/output/result.json",
  "scale": 2.0
}
JSON

cat configs/config.json
```

Checkpoint: The config is an analysis choice. It should travel with the result.

16. Run the full mini-analysis

```
docker run --rm \
  -v "$PWD/data:/data:ro" \
  -v "$PWD/configs:/configs:ro" \
  -v "$PWD/output:/output" \
  mini-analysis:dev \
  --config /configs/config.json
```

Inspect the output on the host:

```
cat output/result.json
```

Checkpoint: Software came from the image. Data, config, and output came from mounts.

17. Change a runtime choice without rebuilding

Change only the config:

```
python3 - <<'PY'
import json, pathlib
p = pathlib.Path("configs/config.json")
cfg = json.loads(p.read_text())
cfg["label"] = "scaled-by-ten"
cfg["scale"] = 10.0
p.write_text(json.dumps(cfg, indent=2) + "\n")
PY
```

Run the same image again:

```
docker run --rm \
  -v "$PWD/data:/data:ro" \
  -v "$PWD/configs:/configs:ro" \
  -v "$PWD/output:/output" \
  mini-analysis:dev --config /configs/config.json
```

Checkpoint: The image did not change. The analysis choice changed.

18. Inspect metadata

```
docker image inspect mini-analysis:dev \
  --format '{{.Id}}'

docker image inspect mini-analysis:dev \
  --format '{{json .Config.Entrypoint}}'

docker image inspect mini-analysis:dev \
  --format '{{json .Config.Labels}}'
```

Checkpoint: Image ID, entrypoint, and labels are part of the image artifact.

19. Observe rebuild behavior

Add one visible code change:

```
python3 - <<'PY'
p = "mini-analysis/app.py"
text = open(p).read()
text = text.replace('"max": max(scaled),', '"max": max(scaled),\n    "software_note": "rebuilt
    image",')
open(p, "w").write(text)
PY
```

Rebuild and rerun:

```
docker build -t mini-analysis:dev mini-analysis
docker run --rm \
  -v "$PWD/data:/data:ro" \
  -v "$PWD/configs:/configs:ro" \
  -v "$PWD/output:/output" \
  mini-analysis:dev --config /configs/config.json
```

Checkpoint: A software change requires a new image build.

Mini-analysis provenance record

Record	Example from the exercise
image reference	`mini-analysis:dev`
image identity	image ID or digest from `image inspect`
code commit	not available unless the lab directory is in Git
input data	`data/numbers.txt` plus checksum if practical
config	`configs/config.json`
runtime command	the complete `run` command
outputs	`output/result.json` plus checksum if practical

Checkpoint: This same table works for ROOT, C++, GPUs, and Slurm jobs.

Where we are

1. Part A: run containers first
2. Part B: connect the host to the container
3. Part C: build a minimal analysis image
4. Part D: JLab ROOT reference example
5. Part E: Apptainer, Slurm, and provenance

Repository role in this workshop

Use it as the full example

The JLab tutorial repository demonstrates the same boundary pattern using a realistic scientific stack: ROOT, PyROOT, CMake, C++, GitLab CI, GitHub Actions, GHCR, Slurm, and Apptainer.

Do not make it the first beginner task

New users should first understand `run`, mounts, ephemerality, build recipes, and provenance using the small mini-analysis.

20. Clone the reference repository

```
cd ..
git clone https://github.com/mrcmor100/jlab-container-tutorial.git
cd jlab-container-tutorial
```

Look at the top-level structure:

```
ls
find . -maxdepth 2 -type f | sort | head -40
```

Checkpoint: Find `Dockerfile`, `containers/cpp/Dockerfile`, `configs/`, `scripts/`, `slurm/`, and `docs/`.

21. Inspect the Python Dockerfile

```
sed -n '1,160p' Dockerfile
```

Look for

- ROOT base image tag;
- virtual environment setup;
- PyROOT validation import;
- non-root user setup;
- `ENTRYPOINT ["analyze-root"]`.

22. Build the Python image

```
export CONTAINER_RUNTIME=docker

./scripts/build-python.sh
```

Teaching prompt

Which image name did the script build? What base ROOT image did it use?

Checkpoint: Local default image name: `root-python-analysis:dev`.

23. Generate toy ROOT data

```
./scripts/generate-sample-data.sh

ls -lh data/
```

Teaching prompt

Why does the data file live outside the image even though the generator runs inside a container?

Checkpoint: Expected output: `data/example.root`.

24. Inspect the config files

```
cat configs/example.yaml
cat configs/example.json
```

Ask participants to identify

- analysis choices;
- data locations;
- output artifact names;
- fields that must be preserved with the result.

25. Run the Python/PyROOT analysis

```
./scripts/run-python-analysis.sh

ls -lh output/
```

Equivalent manual command:

```
docker run --rm \
  -v "$PWD/data:/data:ro" \
  -v "$PWD/configs:/configs:ro" \
  -v "$PWD/output:/output" \
  root-python-analysis:dev \
  --config /configs/example.yaml
```

Checkpoint: Expected output: `output/pyroot_hists.root`.

26. Build and run the C++ image

```
./scripts/build-cpp.sh
./scripts/run-cpp-analysis.sh

ls -lh output/
```

Equivalent manual command:

```
docker run --rm \
  -v "$PWD/data:/data:ro" \
  -v "$PWD/configs:/configs:ro" \
  -v "$PWD/output:/output" \
  root-cpp-analysis:dev \
  --config /configs/example.json
```

Checkpoint: Expected output: `output/cpp_hists.root`.

27. Inspect ROOT output interactively with X11

For interactive ROOT graphics on the farm, connect with X11 forwarding first:

```
XAUTHORITY_HOST="${XAUTHORITY:-$HOME/.Xauthority}"
echo "DISPLAY=$DISPLAY"
```

Pass the display and Xauthority file into the container:

```
podman run --rm -it "${PODMAN_USER_FLAGS[@]}" \
  --net=host \
  -e DISPLAY="$DISPLAY" \
  -e XAUTHORITY=/tmp/.container.xauth \
  -v "$XAUTHORITY_HOST:/tmp/.container.xauth:ro" \
  -v /tmp/.X11-unix:/tmp/.X11-unix:ro \
  -v "$PWD/output:/output:ro" \
  --entrypoint root \
  root-python-analysis:dev \
  /output/pyroot_hists.root
```

Inside ROOT:

```
.ls
h_p->Draw()
```

Contained versus not contained in the ROOT image

Contained in image

- ROOT;
- Python package or C++ executable;
- Python dependencies;
- default entrypoint;
- selected OS packages.

Not contained in image

- input ROOT file;
- analysis config;
- output ROOT file;
- runtime command;
- host kernel and scheduler;
- GPU driver, if used.

28. Review CI and registry files

```
sed -n '1,200p' .gitlab-ci.yml
find .gitlab/ci -type f -maxdepth 3 -print
find .github/workflows -type f -maxdepth 2 -print
```

Teaching prompt

What should happen on merge requests, protected `main`, and version tags?

Checkpoint: The purpose of CI is to turn source commits into controlled image artifacts.

29. Release image pattern

```
git tag v0.1.0
git push origin v0.1.0
```

Then users pull a release image, not a local one-off build:

```
docker pull ghcr.io/<owner>/<repo>-root-python-analysis:v0.1.0
```

Checkpoint: For production analyses, version tags or digests beat unreviewed local images.

Where we are

1. Part A: run containers first
2. Part B: connect the host to the container
3. Part C: build a minimal analysis image
4. Part D: JLab ROOT reference example
5. Part E: Apptainer, Slurm, and provenance

30. Pull a release image with Apptainer

Authenticate to the registry if the image is private:

```
apptainer registry login --username cmorean docker://codecr.jlab.org
```

Pull the OCI image into a local SIF artifact:

```
apptainer pull \
  root-cpp-analysis_v0.1.0.sif \
  docker://codecr.jlab.org/scicomp/software/jlab-container-tutorial/root-cpp-analysis:v0.1.0
```

Checkpoint: `pull` creates the SIF file. `exec` requires both a container and a command.

31. Run with Apptainer binds

```
mkdir -p output

apptainer exec \
  --bind "$PWD/data:/data:ro" \
  --bind "$PWD/configs:/configs:ro" \
  --bind "$PWD/output:/output" \
  root-cpp-analysis_v0.1.0.sif \
  root-cpp-analysis --config /configs/example.json
```

Teaching prompt

What changed from Docker or Podman? What stayed exactly the same?

Checkpoint: The runtime tool changed. The data/config/output boundary did not.

33. GPU command patterns

Apptainer NVIDIA check

```
apptainer exec --nv image.sif nvidia-smi
```

Docker NVIDIA check

```
docker run --rm --gpus all \
  nvidia/cuda:TAG nvidia-smi
```

Watch for this

The host driver and device access still matter. The image alone does not preserve GPU execution.

Final provenance checklist

For any result produced in the lab, record:

- image reference and digest or image ID;
- source repository commit or release tag;
- data identifiers and checksums when practical;
- config file path and content;
- complete runtime command or batch script;
- output files and hashes when practical;
- scheduler, GPU, calibration, database, or network dependencies.

Checkpoint: A reproducible result is a recorded boundary, not just a container image.

Exit ticket

Try and answer each of these in one sentence:

1. What was inside your mini-analysis image?
2. What was outside the image but required to reproduce the result?
3. Which command line would you paste into an analysis note?
4. What would change when moving from Docker to Apptainer on a Slurm cluster?

Clean up

```
# Optional cleanup for local lab images and containers.
docker image rm mini-analysis:dev
docker image rm python:3.12-slim
```

Watch for this

You can run a prune command, but this will broadly delete unrelated containers, images, and caches.

References and source material

- JLab tutorial GitHub mirror: https://github.com/mrcmor100/jlab-container-tutorial
- JLab GitLab repository: https://code.jlab.org/scicomp/software/jlab-container-tutorial/
- Workshop notes provided with this deck: lecture plan, container theory, hands-on lab, CI/registry, HPC/Apptainer, instructor notes.
- Docker documentation: https://docs.docker.com/
- Podman documentation: https://docs.podman.io/
- Apptainer documentation: https://apptainer.org/docs/