Containers for Reproducible Scientific Computing

90-minute lecture for new scientific users

Casey Morean

5/15/2026

Build software into images. Mount data and configuration at runtime. Record the provenance boundary.

Where we are

The analysis environment is part of the result

The familiar failure mode

- "It worked on my laptop."
- ROOT version changed.
- Python package ABI changed.
- Compiler and C++ standard changed.
- CUDA or system libraries drifted.
- Site modules conflict.

The container goal

- Freeze the user-space software stack.
- Make commands portable across machines.
- Separate software from mutable data.
- Publish a repeatable build recipe.
- Attach provenance to the analysis.

Scientific preservation requires boundaries

Useful mental model

software environment	container image
analysis choices	config file
input data	mounted read-only or referenced by recorded identifier
outputs	mounted writable directory
execution	command line or batch job
provenance	all of the above, plus image digest and code commit

These points will be revisited throughout the presentation.

Containers are not magic preservation

Containers preserve a controlled **user-space environment**. They do not fully preserve:

- host kernel behavior;
- scheduler behavior;
- GPU driver behavior;
- external database state;
- mutable network resources;
- site authorization policy;
- unversioned calibration sources;
- implicit human decisions.

Checkpoint: The result is reproducible only when external dependencies are named, pinned, or recorded.

Where we are

Kernel, user space, and where containers fit

Linux container model

- The host provides the Linux kernel.
- The image provides user-space files: libraries, tools, binaries, Python packages, ROOT, CUDA user-space libraries, scripts.
- The runtime creates isolated processes using kernel mechanisms.

Desktop nuance

On macOS and Windows, Docker Desktop usually runs Linux containers inside a small Linux virtual machine. The container still shares that VM's Linux kernel, not a guest kernel packaged inside the image.

Scientific shorthand: image equals user-space environment; host equals kernel, devices, filesystems, and policy.

Image versus container

Core equation

$$\text{image} + \text{runtime options} + \text{host kernel} = \text{running container}$$

Image

- Packaged filesystem layers.
- Runtime metadata: entrypoint, environment, labels, working directory.
- Can be pushed to a registry.
- Usually immutable as a scientific artifact.

Container

- Running process or process tree.
- Created from an image.
- Uses runtime options: mounts, ports, GPU flags, environment, user mapping.
- Usually disposable.

A tiny Dockerfile shows the layer idea

```
FROM docker.io/ubuntu:24.04

RUN apt-get update && apt-get install -y python3

COPY app.py /app/app.py

CMD ["python3", "/app/app.py"]
```

Layer 1	Ubuntu base filesystem
Layer 2	Python packages and OS packages added by the RUN instruction
Layer 3	Application files copied into the image
Config	Default command and runtime metadata

The runtime merges those layers and presents one filesystem to the process.

Layers are content-addressed

Conceptual stack

analysis image	the name you pull or run
layer sha256:dddd...	analysis code
layer sha256:cccc...	Python packages or compiled executable
layer sha256:bbbb...	ROOT and system dependencies
layer sha256:aaaa...	base operating-system user space
config sha256:eeee...	environment, entrypoint, labels, working directory

- A filesystem-changing Dockerfile instruction normally creates a new layer.
- Layer reuse is why rebuilds can be fast.
- Layer identity is why digests matter for provenance.

Tags are names; digests are identities

Tags

```
root-python-analysis:dev
root-python-analysis:v1.0.0
root-python-analysis:sha-abc1234
```

- Human-facing names.
- Easy to teach and communicate.
- Can move if someone retags an image.

Digests

```
root-python-analysis@sha256:...
```

- Content-addressed identifiers.
- Stronger for analysis notes.
- Better for production provenance.

Registries are distribution systems

Registry roles

- Store images and metadata.
- Attach names: repository plus tag.
- Provide pull/push access control.
- Specify the registry in the pull command. i.e. docker.io/, codecr.jlab.org/, ghcr.io/
- Support automated builds and release workflows.

Scientific use

A registry is where a reviewed image becomes something a collaborator or batch job can pull without rebuilding locally.

Build-time versus run-time

Build time: image recipe

```
FROM docker.io/rootproject/root:6.32.04-
    ubuntu24.04
RUN apt-get update && apt-get install -y cmake
COPY . /src
RUN cmake -S /src -B /src/build && \
    cmake --build /src/build
```

Run time: analysis context

```
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: The image contains tools. The runtime supplies data, configuration, and execution context.

What belongs where?

Put in the image	Keep outside the image
ROOT, compilers, system libraries	Large data samples
Python packages, C++ executable	Analysis configuration
CLI entrypoint and default working directory	Output ROOT files, plots, logs
Small static reference files, when truly versioned	Secrets, tokens, mutable credentials
OCI labels and build metadata	Runtime command and scheduler options

Rule of thumb

Software is built into the image. State is mounted or explicitly referenced.

Where we are

Anatomy of `docker run`

```
docker run --rm -it \
  --name demo \
  -e ANALYSIS_LABEL=test-001 \
  -v "$PWD/data:/data:ro" \
  -v "$PWD/output:/output" \
  -p 8000:8000 \
  python:3.12-slim \
  python -m http.server 8000
```

`-rm`	delete the container metadata after it exits
`-it`	interactive terminal
`-e`	runtime environment variable
`-v`	bind mount host path into the container
`-p`	publish container port to host port
image name	the packaged filesystem and metadata
command	what the container process actually executes

Ephemeral by default

Disposable filesystem

A normal container's writable layer is temporary. Files written inside the container disappear when that container is removed, unless they are written to a mounted host path or another persistent storage mechanism.

Why this is good

- Repeated runs start from a clean software environment.
- Failure cleanup is simpler.
- Hidden state is less likely to accumulate.

Watch for this

Mounted data, network services, GPUs, credentials, and schedulers intentionally break pure isolation. Record them.

Mounts: the scientific workhorse

```
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
```

- `/data:ro`: input is visible but not writable.
- `/configs:ro`: analysis choices are explicit and preserved.
- `/output`: analysis artifacts survive container exit.

Checkpoint: Read-only inputs plus writable outputs make the reproducibility boundary visible.

Environment variables, ports, and secrets

Environment	Ports	Secrets
Runtime knobs such as labels, modes, paths, and feature flags.	Useful for notebooks, web UIs, event displays, and debugging services.	Treat access tokens and credentials as runtime inputs. Do not bake them into images.

Checkpoint: Every runtime input that can change a result belongs in the provenance record.

Users, ownership, and rootless execution

- Container images can define an internal user, such as UID 1000.
- Root inside a container is not always root on the host, and typically should NOT be!. Especially with rootless runtimes and user namespace remapping.
- Bind-mounted files still carry host filesystem ownership semantics.
- Apptainer commonly preserves the host user's identity more naturally for HPC shared filesystems.

Watch for this

Try to capture host level permissions and groups / UID.

Containers versus virtual machines

Feature	Container	Virtual machine
Kernel	Shared with host kernel	Guest kernel
Size	Smaller	Larger
Startup	Fast	Slower
Isolation	Process and filesystem isolation*	Stronger machine isolation
HPC fit	Common for per-job user-space packaging	Less common for per-job execution
Best use	Package scientific user-space environments	Emulate or isolate complete machines

Containers are excellent for packaging user-space scientific environments. They are not full machine snapshots.

Where we are

Docker Desktop, and Podman

Docker Desktop

Good local default on macOS/Windows. Provides a Linux VM, CLI integration, and developer convenience.

Podman

Daemonless, rootless-friendly OCI tooling. Often a better fit on shared Linux development hosts.

For beginner commands, `docker run` and `podman run` are often close enough to teach the same mental model.

Why Apptainer is common on HPC

Many HPC sites historically standardized on Apptainer or Singularity-style execution because it fits common site constraints:

- no rootful Docker daemon on compute nodes;
- user identity maps naturally into the job;
- shared filesystems are easy to bind;
- batch jobs can launch containers without elevated host control;
- SIF files are convenient single-file artifacts.

Checkpoint: The same OCI image can often become a SIF file for batch execution.

Apptainer mirrors the local runtime pattern

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

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

```
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
```

- exec requires a container *and* a command.
- Inputs and config are still mounted read-only.
- Outputs are still mounted writable.

GPU containers: host driver still matters

Apptainer pattern

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

Docker pattern

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

- The image can include CUDA user-space libraries and application code.
- The host still provides the kernel driver and device access.
- AMD and Intel accelerator paths are site-specific and runtime-specific.

Watch for this

Do not teach GPU containers as if the driver lives entirely inside the image.

Where we are

Package managers inside images

Tool family	Common use in containers
`apt`	Debian and Ubuntu package install/update operations
`dnf`	Fedora, RHEL, AlmaLinux, and related package operations
`dpkg` / `rpm`	Low-level package file tools, less often used directly in Dockerfiles
`pip` / `venv`	Python environment construction inside the image
`cmake` / `make` / `ninja`	Build compiled scientific code into the image

Prefer deterministic build recipes. Interactive installation inside a running container is convenient but usually not reproducible.

Dockerfile hygiene for scientific images

```
FROM docker.io/rootproject/root:6.32.04-ubuntu24.04

LABEL org.opencontainers.image.source="https://example.org/project"
LABEL org.opencontainers.image.revision="<git-commit>"

RUN apt-get update && apt-get install -y --no-install-recommends \
    ca-certificates cmake g++ git ninja-build \
 && rm -rf /var/lib/apt/lists/*

COPY src/ /src/
RUN cmake -S /src -B /src/build -DCMAKE_BUILD_TYPE=Release \
 && cmake --build /src/build
```

Pin base images, label provenance, reduce hidden state, and keep build steps readable.

Why CI-built containers are better artifacts

Manual local build

- Easy for exploration.
- Depends on local context.
- Easy to forget exact commit or build args.
- Harder to audit.

CI build

- Triggered by merge, tag, or manual workflow.
- Runs from a recorded commit.
- Pushes predictable tags or digests.
- Can enforce policy and review.

Checkpoint: For production analyses, use reviewed version tags or digests, not unreviewed local images.

A recommended CI image workflow

1. Developers work on branches.
2. Merge requests build images without pushing them.
3. Protected `main` builds and pushes `sha-<short-sha>` images.
4. Git tags build and push versioned release images.
5. Manual container pipelines may push branch-scoped test images.
6. Production analyses use version tags or digests.

Release teaching pattern

`git tag v0.1.0` then `git push origin v0.1.0` creates a named release image in the registry.

Where we are

Reference repository: full example, not the beginner core

What the repository demonstrates

- Python/PyROOT CLI driven by YAML or JSON configuration.
- C++/ROOT/CMake executable driven by JSON configuration.
- Dockerfiles for both analysis paths.
- GitLab CI with Kaniko and GitHub Actions for GHCR.
- Slurm and Apptainer examples for HPC execution.

In the hands-on session, this becomes the worked reference example after users understand run, mount, build, and provenance basics.

Repository quick-start command sequence

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

export CONTAINER_RUNTIME=podman
./scripts/build-python.sh
./scripts/build-cpp.sh
./scripts/generate-sample-data.sh
./scripts/run-python-analysis.sh
./scripts/run-cpp-analysis.sh
```

Generated analysis outputs appear under `output/`. Use `CONTAINER_RUNTIME=podman` for Podman.

Python/PyROOT image: what to point out live

```
ARG ROOT_IMAGE=docker.io/rootproject/root:6.32.04-ubuntu24.04
FROM ${ROOT_IMAGE} AS runtime

ENV VIRTUAL_ENV=/opt/venv
ENV PATH="/opt/venv/bin:${PATH}"

RUN apt-get update \
 && apt-get install -y --no-install-recommends ca-certificates git python3-venv \
 && rm -rf /var/lib/apt/lists/*

COPY python-pyroot/src/ ./python-pyroot/src/
RUN python3 -m venv --system-site-packages "${VIRTUAL_ENV}" \
 && python -m pip install --no-cache-dir ./python-pyroot \
 && python -c 'import ROOT; print("Validated PyROOT import from:", ROOT.__file__)'

ENTRYPOINT ["analyze-root"]
CMD ["--help"]
```

Teaching points: pinned ROOT base tag, virtual environment that can import PyROOT, installed CLI, default entrypoint.

C++/ROOT image: multi-stage build

```
ARG ROOT_IMAGE=docker.io/rootproject/root:6.32.04-ubuntu24.04
FROM ${ROOT_IMAGE} AS build
RUN apt-get update \
 && apt-get install -y --no-install-recommends cmake g++ ninja-build nlohmann-json3-dev \
 && rm -rf /var/lib/apt/lists/*
WORKDIR /src
COPY cpp-root-cmake/CMakeLists.txt ./CMakeLists.txt
COPY cpp-root-cmake/include/ ./include/
COPY cpp-root-cmake/src/ ./src/
RUN cmake -S . -B build -G Ninja -DCMAKE_BUILD_TYPE=Release \
 && cmake --build build

FROM ${ROOT_IMAGE} AS runtime
COPY --from=build /src/build/root-cpp-analysis /usr/local/bin/root-cpp-analysis
ENTRYPOINT ["root-cpp-analysis"]
CMD ["--help"]
```

Teaching points: compiler stays in build stage; runtime image carries ROOT plus executable.

Same runtime shape for Python and C++

Python

```
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
```

C++

```
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: The strongest demo is not the complexity of the analysis. It is the identical command pattern.

What to record in an analysis note

At minimum, record:

- image reference and digest;
- Git commit or release tag;
- config file;
- input ROOT file identifiers;
- runtime command or batch script;
- output filenames and hashes, when practical;
- external services or calibration snapshots.

Compact provenance equation

result = code commit + image digest + input identifiers + config + command + external state + output metadata

Where we are

Decision tree for users

Need	Default recommendation
Explore commands locally	Docker Desktop or farm Podman
Build a small analysis environment	Dockerfile or Containerfile with pinned base image
Avoid dependency drift	Put system tools and packages in the image
Avoid data mutation	Mount inputs read-only
Preserve analysis choices	Put them in versioned config files
Run on HPC	Convert or pull to Apptainer/SIF
Release for collaborators	Build and push from CI using tags or digests

Common audience questions

Why not just use a virtual environment?

A Python virtual environment manages Python packages. A container can also include ROOT, compilers, system libraries, command-line tools, and a controlled entrypoint.

Can I put data in the image?

Small static examples, yes. Real analysis data should usually be mounted or referenced by recorded identifiers.

Can I install packages interactively inside a running container?

Maybe for exploration, but for reproducibility put the installation in the Dockerfile or build recipe.

Transition to hands-on

1. First, we will run containers without building anything.
2. Then we will mount data, configuration, and outputs.
3. Then we will build a small analysis image.
4. Finally, we will inspect the JLab ROOT repository as a full reference example.

Lab framing

The repository is a realistic example. The beginner lab is about learning the container model before asking users to understand ROOT, CMake, CI, and HPC all at once.

References and source material

- JLab container documentation: https://pages.jlab.org/scicomp/software/jlab-container-docs
- 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.
- OCI image/distribution concepts: https://opencontainers.org/
- Apptainer documentation: https://apptainer.org/docs/
- Docker documentation: https://docs.docker.com/
- Podman documentation: https://docs.podman.io/