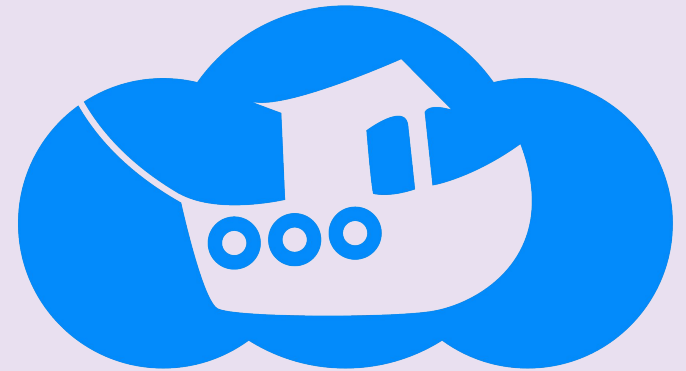


Introduction to Charliecloud

Megan Phinney
mphinney@lanl.gov
Los Alamos National Laboratory

ACES Workshop 07/14/23

LA-UR 23-27202



Charliecloud

Charliecloud Team (Current)



Reid Priedhorsky



Jordan Ogas



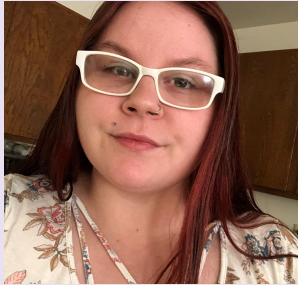
Shane Goff



Megan Phinney



Lucas Caudill



Layton McCafferty



Hank Wikle



Nick Volpe

Agenda

01 What are
containers?

02 What is
Charliecloud?

03 Fully unprivileged
build

01

What are containers?



Users need different software

Standard HPC software stacks have a specific purpose:

- Specifically: MPI-based physics simulations

What if your thing is different?

- non-MPI simulations
- Artificial intelligence
- *Spicy* software dependencies

Admins will install software for you

- **IF** there is enough demand
- Unusual software needs go unmet



User-defined software stacks

BYOS (bring your own software)

- Lets users install software of their own choice
- ... up to and including a complete Linux distribution
- ... and run it on compute resources they don't own

But, possible problems include ...

- Missing functionality
 - *high speed network, accelerators, filesystems*
- Performance
 - *many opportunities for overhead*
- Security problems
 - *multiple root exploits*
- Excessive complexity
 - *See Spack*



A container is *not*

- a lightweight virtual machine
 - or something you boot
- a container image
 - filesystem tree
- something that requires a specific tool
- the container runtime itself
 - ex. Docker

A container is

- a process
 - with its own view of kernel resources
 - or perhaps a group of processes sharing that view

An **image** is: said filesystem

In whatever form it takes



Containers are just processes!

Containers are mostly for abstraction/encapsulation.

- Moving between containers is explicitly supported.
- `setns(2)`, `/proc`, etc.

Privileged/setuid containers need more to be safe.

- SELinux/AppArmor, `seccomp-bpf`, etc.
- (this is hard! Lots of CVEs)

Unprivileged containers get kernel safety measures

- Lots of smart people's time has gone into this
- You already trust the Linux kernel to keep unprivileged processes secure. Keep doing that.



Container Ingredients

01

Linux namespaces

- **Mount:** filesystem tree and mounts
 - **PID:** process IDs
 - **UTS:** host name
 - **Network:** all other network stuff
 - **IPC:** System V and POSIX
 - **User:** UID/GID/capabilities
- privileged**
need root to create
- unprivileged**

02

cgroups: limit resource consumption per process

03

`prctl(PR_SET_NO_NEW_PRIVS)`




04

`seccomp(2)`

05

SELinux, AppArmor, etc.

Charliecloud privilege taxonomy

type	namespace	setup	IDs in container	examples
	mount	privileged	shares UID and GID with host	Docker, Singularity, Podman
	mount + privileged user	privileged	arbitrary UIDs and GIDs separate from host	Singularity, Podman (rootless)
	mount + unprivileged user	unprivileged	only 1 UID and 1 GID in container	Charliecloud

Priedhorsky, Canon, Randles, Younge. SC21. <https://dx.doi.org/10.1145/3458817.3476187>

Reproducibility

Distros have been working on bit-identical software builds for years and (*plot twist*) it's still not done

- e.g., timestamps get embedded everywhere

Prescriptive builds do help.

- e.g., Dockerfile \Rightarrow standard

But unsolved challenges remain

- FROM centos:7 \Rightarrow maybe different tomorrow
- FROM centos:9f38484 \Rightarrow maybe gone tomorrow



02

What is Charliecloud?



Charliecloud Philosophy

1) transparent;
not opaque

Treat containers as regular files

**Examine/debug containers with
standard UNIX tools**

Things should be explicit

Charliecloud Philosophy

2) simple;
not complex

Everything is a user process

**Implement the right features;
Minimize dependencies**

**Use mount and user namespaces
only**

**Embrace UNIX: *make each
program do one thing well***

Charliecloud Philosophy

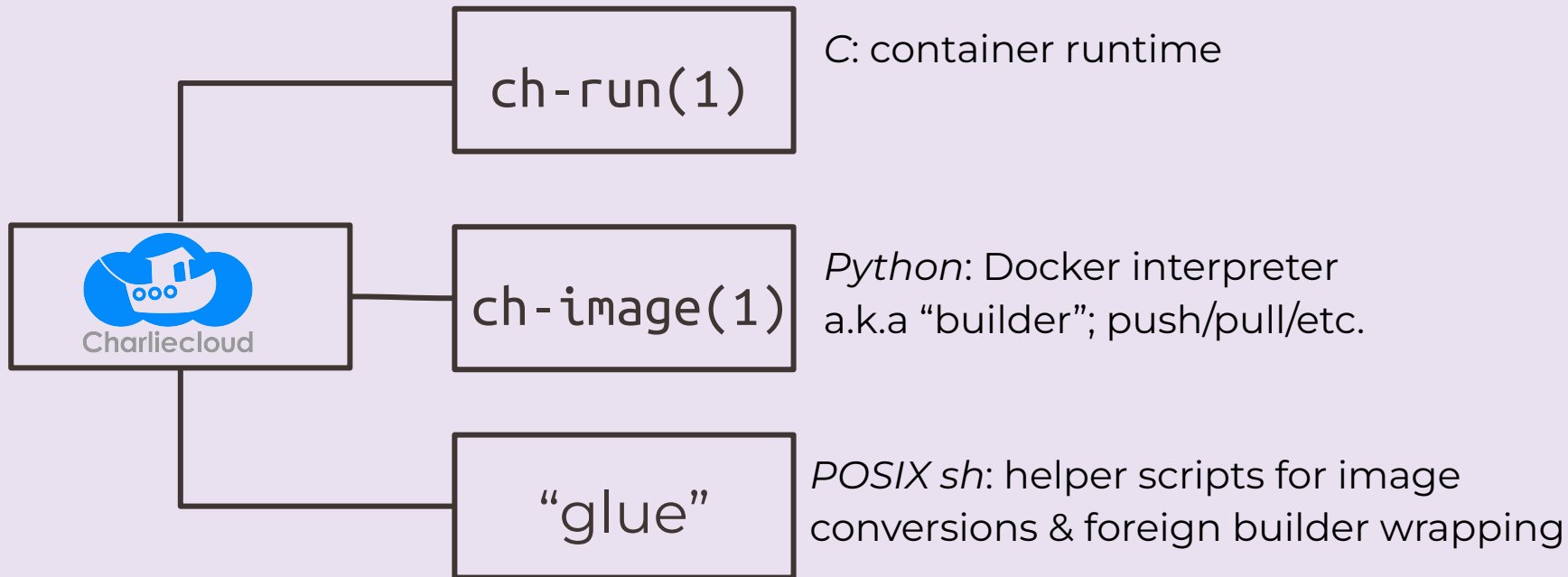
3) trust the kernel

Don't maintain a security boundary

Stay unprivileged

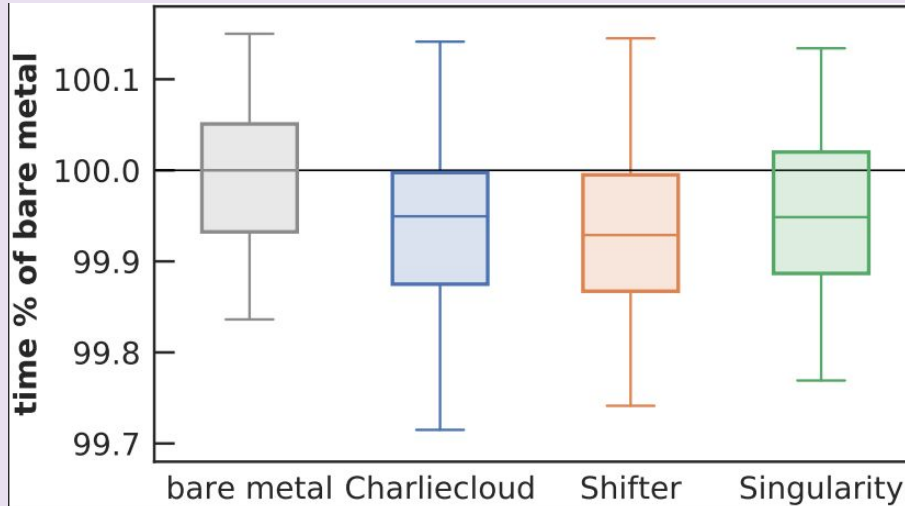
Avoid responsibility

Charliecloud Components

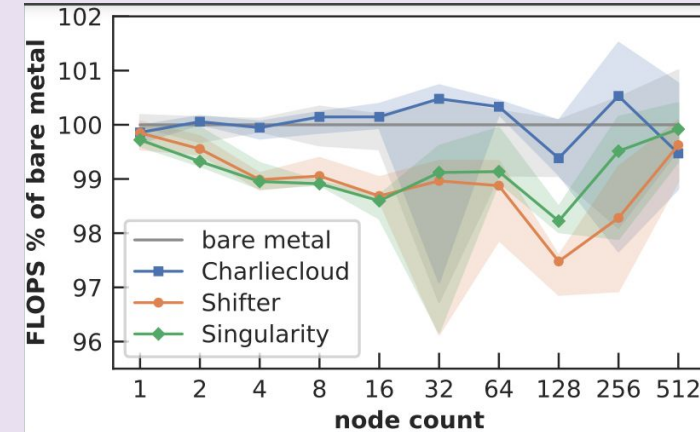
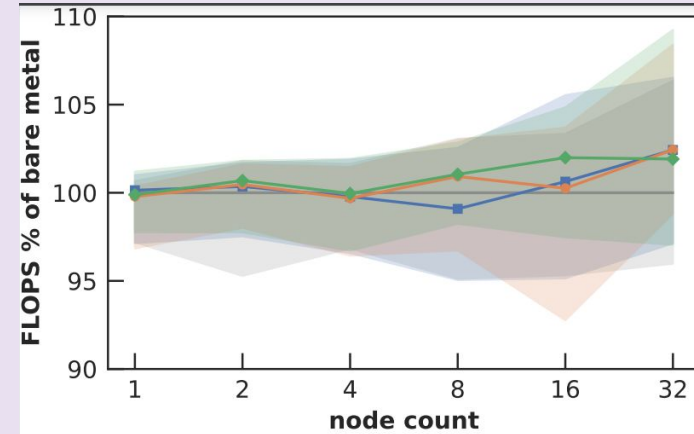


Performance impact: probably zero

SysBench



HPCG



03

Fully Unprivileged Builds

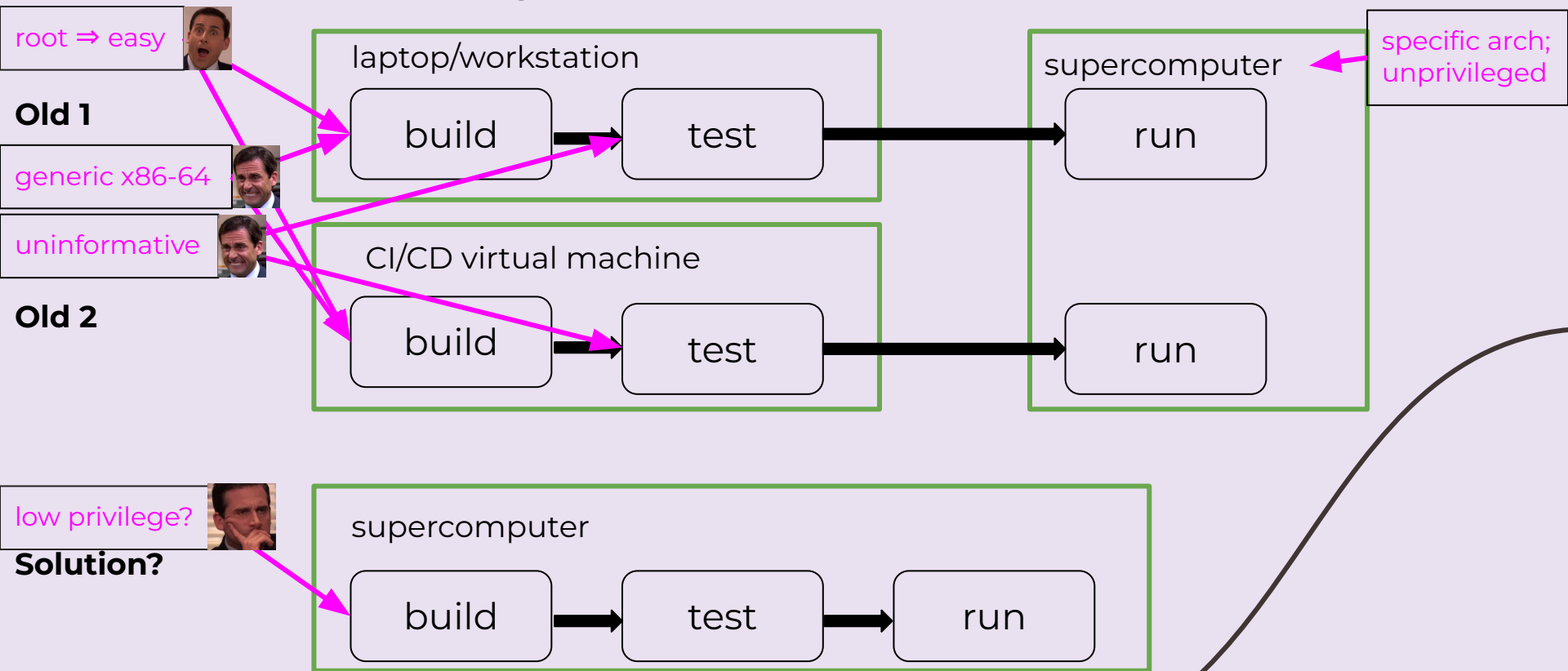


Basic Pitch


- Users want more flexibility ⇒ containers
- Container build needs root ⇒ HPC management mismatch
- Build on generic x86 VMs ⇒ HPC hardware mismatch
- Low-privilege containers ⇒ build directory on HPC
- **The Key:** Linux user namespaces
- New taxonomy of container privilege
- OSS implementations
 - Fully-unprivileged Charliecloud
- Better workflow now & future is bright



Container image workflow



Charliecloud privilege taxonomy

type	namespace	setup	IDs in container	examples
I	mount	privileged	shares UID and GID with host	Docker, Singularity, Podman
II	mount + privileged user	privileged	arbitrary UIDs and GIDs separate from host	Singularity, Podman (rootless)
	mount + unprivileged user	unprivileged	only 1 UID and 1 GID in container	Charliecloud

Priedhorsky, Canon, Randles, Younge. SC21. <https://dx.doi.org/10.1145/3458817.3476187>

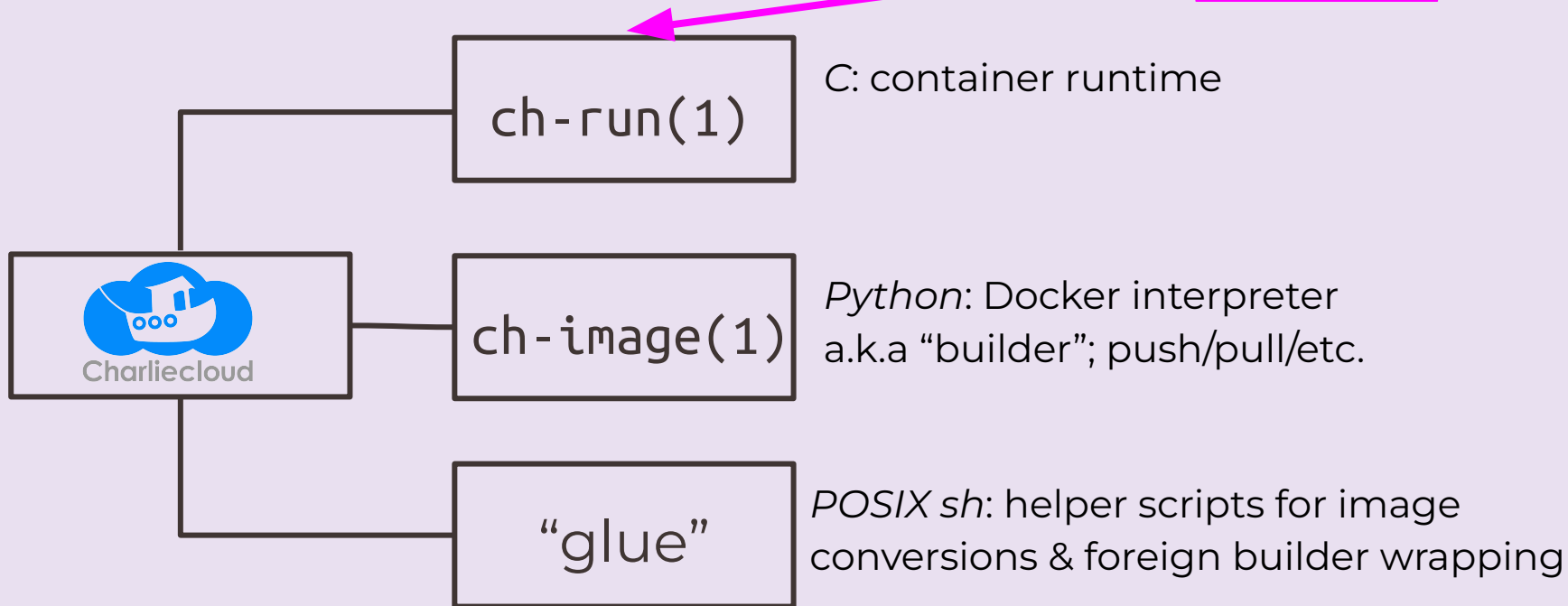
Only **Type III containers** are fully unprivileged throughout the container lifetime

Build options

type	namespace	setup	IDs in container	approach
I	mount	privileged	shares UID and GID with host	sandboxed build system
II	mount + privileged user	privileged	arbitrary UIDs and GIDs separate from host	privileged helper tools; careful configuration
III	mount + unprivileged user	unprivileged	only 1 UID and 1 GID in container	fakeroot(1) wrapper

Charliecloud Components

Type III



New Root Emulation Mode: `seccomp`

- Why do we need this?
 - We need to tell programs that we have real root privileges even though we are running as a normal user
- Uses the kernel's `seccomp(2)` system call filtering to intercept certain privileged system calls, do absolutely nothing, and return success to the program



New Root Emulation Mode: seccomp

- Advantages:
 - Simpler
 - Faster
 - Completely agnostic to libc
 - Mostly agnostic to distribution
- Disadvantages:
 - Lacks consistency
- Our previous root emulation mode, fakeroot, has already been adopted by SingularityCE and Apptainer.



Type II vs. Type III build

type	Unprivileged?	File Ownership	ID Management on Host	Works with Network FS	No fakeroot(1) Wrapper
II	mostly	preserved	security boundary	no	yes
III	fully	flattened	only 1 UID and 1 GID in container	yes	no

Recommendations



Type II implementations:

- add Type III
- fix shared FS (xattrs on NFS, Lustre, GPFS?)



Type III implementations:

- robustify fakeroot(1)
- use its ownership data

Distributions:

- add unprivileged mode to package managers

Linux kernel:

- move ID maps into kernel
- make supplemental groups mappable



Los Alamos
NATIONAL LABORATORY

Introduction to Charliecloud

Megan Phinney
mphinney@lanl.gov
Los Alamos National Laboratory

ACES Workshop 07/14/23

LA-UR 23-27202



Charliecloud

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, infographics & images by **Freepik**

Thanks

Do you have any questions?
your email@freepik.com
+91 620 421 838
yourcompany.com



CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, infographics & images by **Freepik**

Please keep this slide for attribution