SCALE 23x

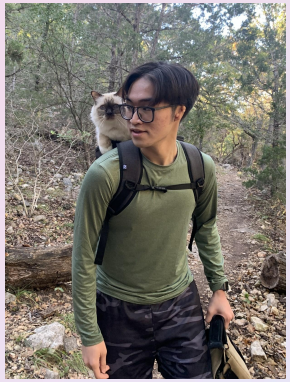
📦 Containers All the Way Down

What we learned running containers-in-containers for 🤖 AI & More

👨‍💻 Phong Nguyen Shaun Hopper

Meta

👨‍💻 Speakers



Phong Nguyen

Production Engineer

Cloud Foundation Apps

Shaun Hopper

Production Engineer

Cloud Foundation Apps

☁️ Cloud Foundation Apps

Cloud-native stack @Meta

We do “OCI” containers while the rest of Meta is on LXC containers

Meta

🗓️ Agenda

01 **What is a container?**

Deconstructing the Container

02 **User Namespace & Rootless**

What's special about userns?

03 **Container-in-Container**

Rootful-rootless Matrix

04 **Case studies: Meta lessons learned**

Meta

1. What's a Container?

Deconstructing the Container 📦

> **Goal: Containers are not magic ✨🕵️**
>
> We'll build our understanding by running a simple workloads and observing a series of follow-ups.

Meta

🧪The Container Formula

Core Components

01

⭐ Linux Namespaces

"Isolation"

Partitioning kernel resources so each process sees only its designated slice of the system.

02

⭐ Cgroups

"Resource Control"

Allocate, **prioritize**, and **limit resources** (CPU, Memory, I/O) for process groups.

03

⭐ OverlayFS

"Union Filesystem"

Merge a read-only lower layer with writable upper layer. COW ensures changes stay in the upper layer, enabling sharing base images efficiently

Step 1 & 2

PID NS: Running & Observing a Container

1. Start the Container

```
$ docker run -p 8080:80 nginx
```

We've launched a standard Nginx web server.

It seems like a normal process, but is it? 🧐

2. Where's it running? Let's check the process

Inside the Container

```
$ docker exec <container_id> ps aux
PID USER  COMMAND
1 root nginx: master process ← PID 1!
2 nginx nginx: worker process
```

On the Host

```
$ ps aux | grep nginx
PID   USER  COMMAND
28374 root  nginx: master process ← PID 28374?!
28401 nginx nginx: worker process
```

💡 **Key Insight: PID Namespace Isolation**

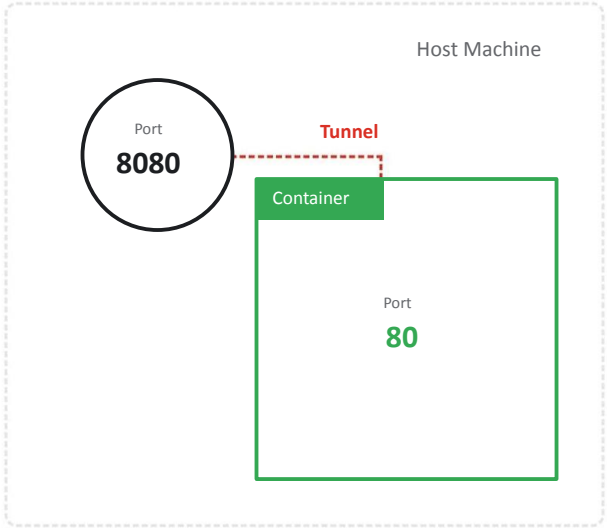
The kernel creates a separate ID space. Inside, the process thinks it's PID 1 (the first process).
Outside, it's just PID 28374, one of thousands on the host.

Step 3

Net NS: "I can't reach my nginx... how do I access it?"

```
$ docker run -p 8080:80 nginx
```

The -p 8080:80 flag creates a "tunnel" to forward traffic from the host's port 8080 to the container's port 80.



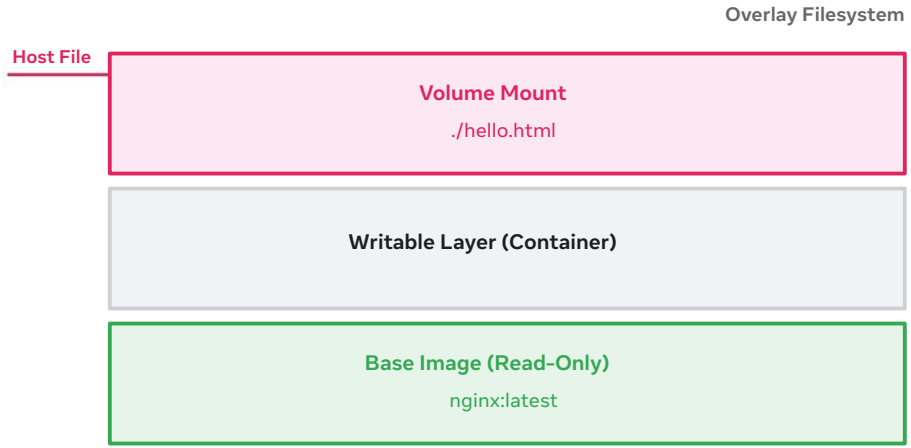
> 💡 **Key Insight: Network Namespace Isolation**
>
> The container operates as if it has its own dedicated network stack.

Step 4

Mount NS: "I want to serve my own HTML page..."

```
$ docker run -v ./hello.html:/usr/share/nginx/html/index.html nginx
```

> 💡 **Key Insight: Mount Namespace Isolation**
>
> The container sees a layered filesystem. The -v flag mounts a file from the host, injecting content without modifying the image.



Meta

Step 5

Cgroups:

"Oh no this container is eating all my resources 🤕"

```
$ docker run --memory 512m --cpus 1.5 nginx
```

512MB

Memory Limit

Container cannot use more than 512MB of RAM.

1.5

CPU Cores

Container cannot use more than 1.5 CPU cores.

Cgroups

Control Groups

Kernel feature to allocate, prioritize, and limit resources.

Meta

The Seven Pillars of Linux Namespaces

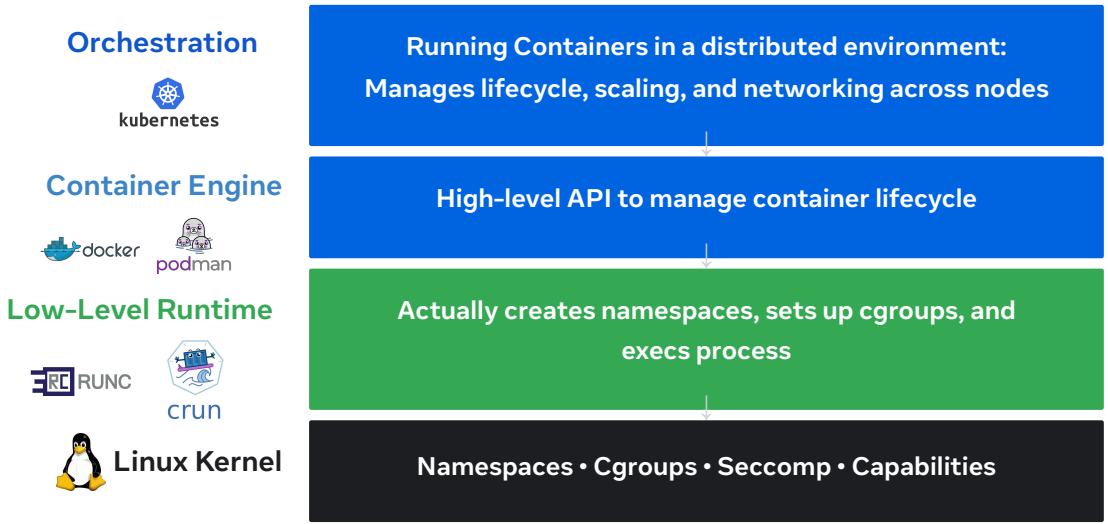
Linux namespaces = partitioning kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources.

Namespace	Target	What It Means for Containers
PID	**Process IDs**	**Separate process tree; completely "blind" to host processes.**
Network	**Network stack**	**Own routing tables, firewall rules, port space, etc**
Mount	**Filesystem mounts**	**Own view of the filesystem hierarchy (rootfs).**
Cgroup	**Resource limit**	**Limiting amount of resources it can use**
User	**User/group IDs**	**UID & GID remapping (i.e: UID 0 inside ≠ UID 0 outside)****
UTS	Hostname & domain	Container gets its own hostname identity.
IPC	Inter-process comms	Separate shared memory segments, semaphores, message queues.

Meta

Summary

(OCI OPEN CONTAINER INITIATIVE) Container Stack



Meta

SECTION 02

User Namespaces & Rootless Containers

Meta

User Namespaces Remap Identity

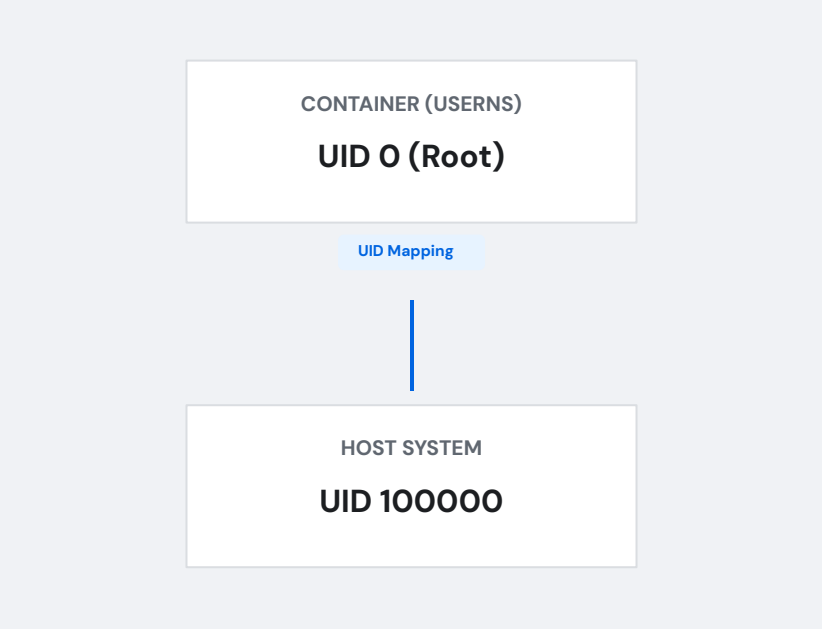
User namespaces don't isolate a resource — they isolate *who you are*.

Inside the Container

You are **root (UID 0)**.

Outside the Container

You are **UID 100000**. Unprivileged.



Meta

How UID Mapping Works

UID mapping is defined by two simple files on the host.

```
# /etc/subuid
shopper:100000:65536

# /etc/subgid
shopper:100000:65536
```

This means user **shopper** can map **65,536** UIDs starting at **100000**.

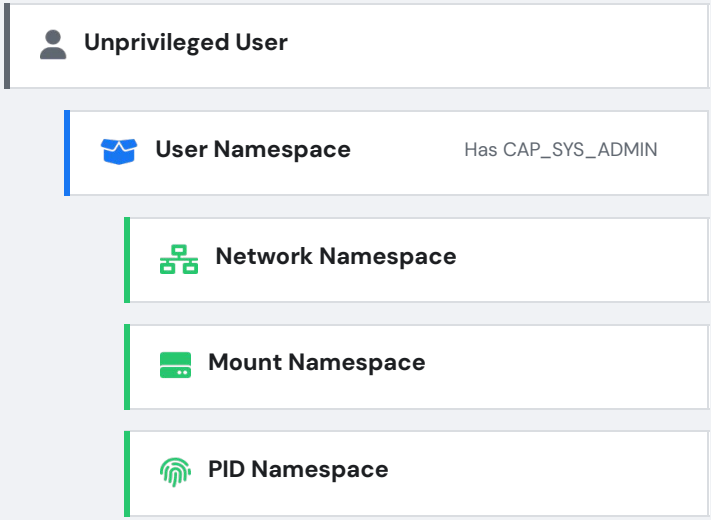
CONTAINER UID		HOST UID
0 (Root)	→	100000
1	→	100001
2	→	100002
...		...
65535	→	165535

Meta

Unprivileged Namespace Creation

Normally, creating namespaces (PID, network, mount) requires **CAP_SYS_ADMIN** (root). User namespaces flip this (kinda):

1. Any unprivileged user can create a user namespace.
2. Inside that namespace, they gain **CAP_SYS_ADMIN**.
3. That capability lets them create **ALL** the other namespaces.



SECTION 03

Now lets nest them

What happens when you run containers inside containers?

Meta

The Rootful-Rootless Matrix

	OUTER: ROOTFUL	OUTER: ROOTLESS
INNER: ROOTFUL	**Scenario A** Container started as root, container process is root. "docker run"	**Scenario C** Container started by unprivileged user (rootless), container process is (user-namespaced) root. "docker daemon not running as root"
INNER: ROOTLESS	**Scenario B** Container started as root, container process is an unprivileged user (in a user namespace) "docker run with good hygiene"	**Scenario D** Container started as unprivileged user (rootless), container process is unprivileged user (in a user namespace) "docker daemon not running as root with good hygiene"

Why Would You Do This?

Two compelling reasons to run containers inside containers.

1. Multi-Tenant Compute

Researchers sharing HPC clusters need to build and run their own containers. You can't give them host root, so you give them rootless containers inside their environment.

2. AI Sandboxing

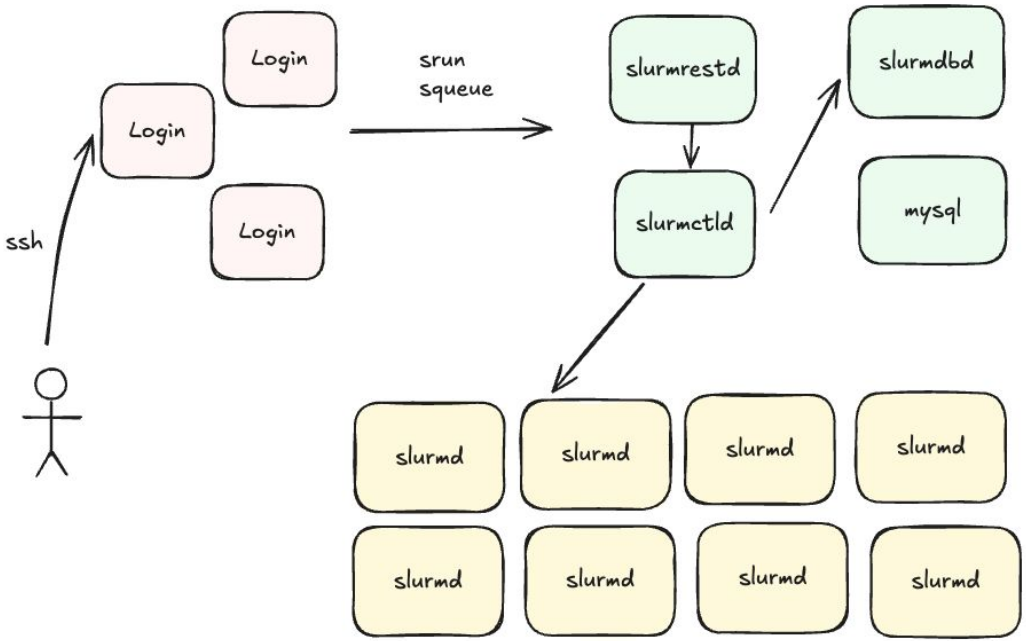
AI agents execute arbitrary code. You want to sandbox execution and network access.

SECTION 04.1

Case Studies — Real ~~Pain~~ Lessons from ~~endured at~~ Meta

Meta

A typical slurm research cluster



Meta

HPC Login Pods

Researchers get a containerized "Linux box" - we want to let them run containers inside of it.

Familiar Environment

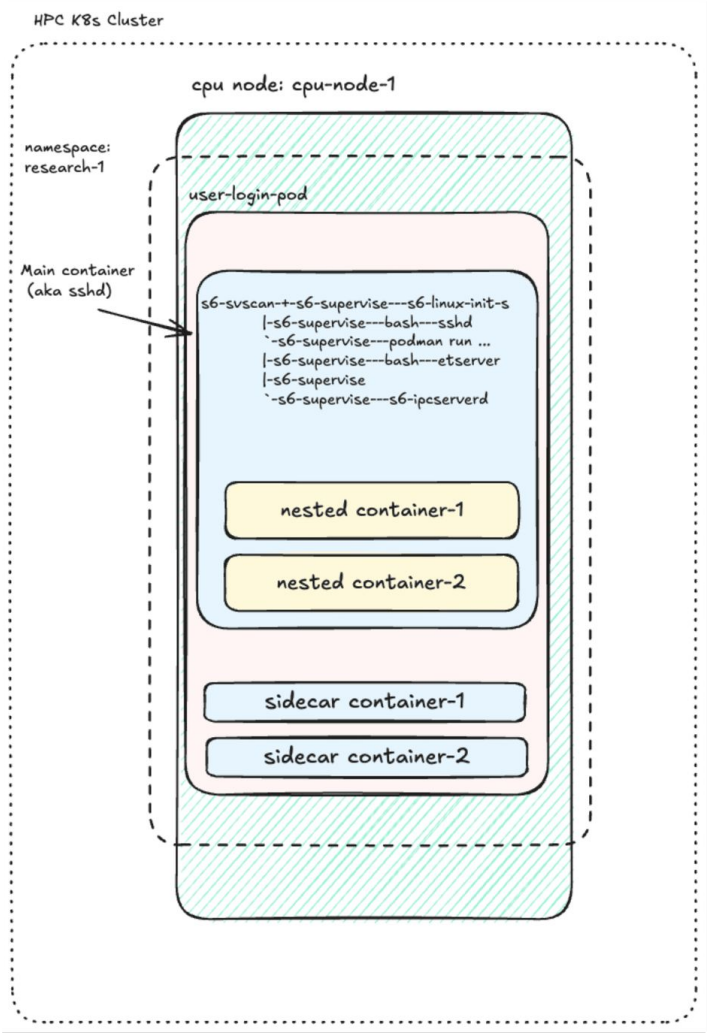
Researchers get a shell that feels like a regular Linux machine.

Shared Storage

NFS home directories are mounted and shared across many users.

The Reality

It's actually a container on a Kubernetes node. And researchers want to run **more containers** inside it.



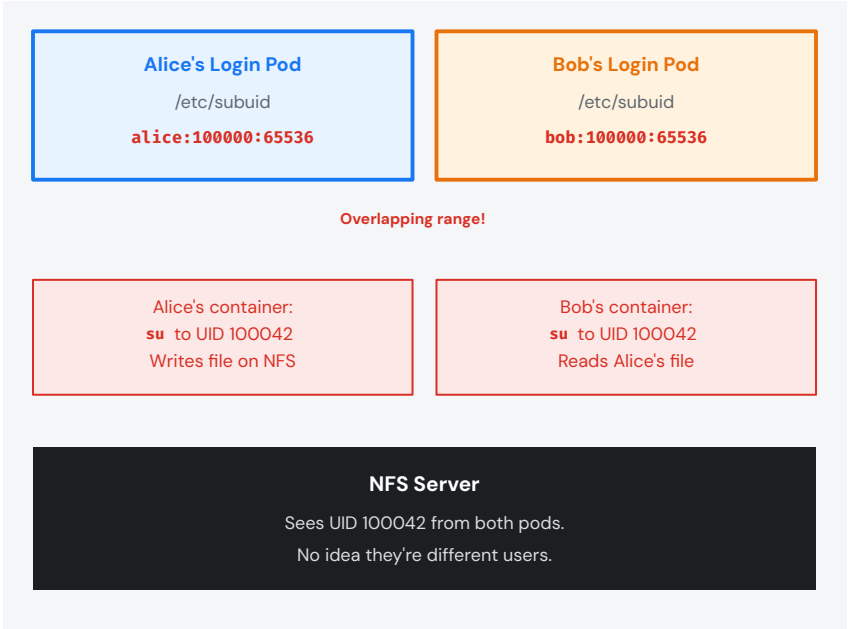
Meta

Case Study — NFS + Overlapping Subuid Ranges

Each login pod has its own `/etc/subuid` file. Since pods are personalized per user, it is easy to accidentally assign **overlapping subuid ranges** to different users.

NFS doesn't know about user namespaces — it just trusts UIDs on the wire. Every login pod mounts the same NFS, so NFS effectively acts as a **global UID namespace** whether we want it to or not.

The problem: If Alice and Bob share an overlapping subuid range, a nested container can **su** to a common UID in that overlap and read or write the other user's files on NFS.



Meta

Fix — Disjoint Subuid Ranges

FEATURE	BEFORE	AFTER
/etc/subuid	Per-pod, uncoordinated Overlap risk	Globally assigned, unique per user Disjoint
Subuid Range	`alice:100000:65536` `bob:100000:65536`	`alice:200000:65536` `bob:300000:65536`
Cross-user NFS risk	HIGH Shared UIDs across pods	NONE No range contains another user's UIDs
User-namespaced root	YES Still a feature	YES Still a feature

THE LESSON

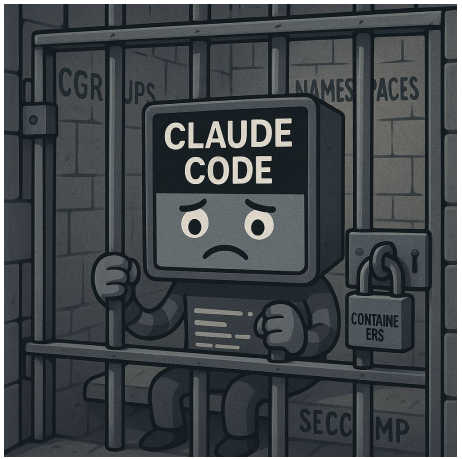
User namespaces are powerful, but NFS predates them. Any protocol that trusts UIDs on the wire will ignore your namespace boundaries.

Meta

Case Study — Jailing Claude Code

THE GOAL

Sandbox Claude Code so it can only reach the inference endpoint — nothing else. And do it without any elevated privileges.



THE THREAT MODEL

- Claude Code should **NOT** have the researcher's network access.
- The login pod can reach internal services, databases, and APIs.
- An AI agent executing arbitrary code should not.

Meta

Jailing Claude Code — Rootless Podman + slirp4netns

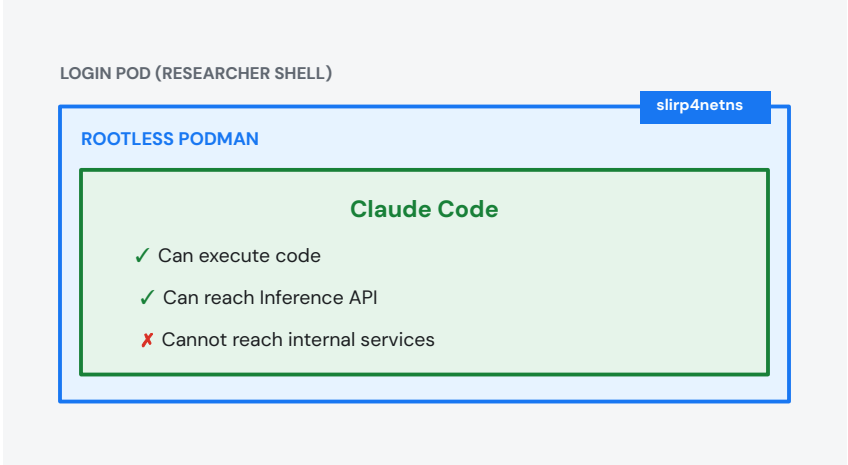
Sandbox Claude Code so it can only reach the inference endpoint — nothing else. Without elevated privileges.

THREAT MODEL

- ✗ Claude Code should **not** have the researcher's network access.
- ✗ The login pod can reach internal services, databases, and APIs. An AI agent should not.

User Namespace
UID 0 inside = unprivileged outside.

Network Namespace
Isolated via `slirp4netns` . iptables: ALLOW → inference, DENY → everything else.



slirp4netns provides userspace networking. It creates a network namespace without needing root.

Meta

Case Study — Container Building in Login Pods

Researchers want to iterate on containers: write a Dockerfile, build, test, repeat.

- **podman build** without RUN instructions works fine.
- **podman build** with RUN instructions fails immediately.

```
$ podman build -t testbuild .
STEP 1/2: FROM ubuntu:latest
STEP 2/2: RUN echo "hello"

error running container: error from /usr/bin/crun creating
container:
mount '/proc' to '/proc': Operation not permitted
```

Why? Every RUN instruction creates a temporary container. That container needs to mount /proc. And the kernel says no.

Meta

Why the Kernel Says No

Kubelet masks sensitive **/proc** paths to prevent information leaks.

Masked Paths

/proc/kcore	*bind mount /dev/null*
/proc/keys	*bind mount /dev/null*
/proc/sched_debug	*bind mount /dev/null*
/proc/acpi	*tmpfs overmount*
/proc/scsi	*tmpfs overmount*

KERNEL LOGIC CHECK

1. Check: Is there a fully visible procfs mount in this namespace?

↓

2. Inspect /proc: Found submounts hiding content.

↓

3. Conclusion: NOT fully visible.

↓

EPERM (Operation not permitted)

Meta

The Fix(?) — A Phantom Bind Mount

We discovered something strange. If you create a **bind mount** of `/proc` to a new location, suddenly everything works.

```
# This fails:
$ podman build -t test .
STEP 1/2: FROM ubuntu:latest
STEP 2/2: RUN echo "hello"
error running container: mount '/proc' to '/proc':
Operation not permitted

# But if you do THIS first:
$ mount --bind /proc /pureproc

#  ...then it works?!
$ podman build -t test .
STEP 1/2: FROM ubuntu:latest
STEP 2/2: RUN echo "hello"
COMMIT test
--> Success
```

Meta

The Fix — Non-Recursive Bind Mount

`mount_too_revealing()` scans every procfs instance in the namespace. If all of them have masking submounts, it blocks new `/proc` mounts with `EPERM`.

A non-recursive bind mount of `/proc` copies the mount point but **not** the masking submounts on top. This gives the kernel a fully visible procfs to find.

```
mount --bind /proc /pureproc
```

Run this at pod startup (e.g., init container).
`podman build` succeeds.

SECTION 4.2

💻 Non-HPC Use Cases

> *We've seen how nested containers solve problems in HPC environments. Now let's look at how the same patterns apply to a completely different domain: building container images at scale.*

Meta

🗂️CASE STUDY

📦 Centralized Container Build Platform

We run a centralized, managed container build platform — a shared service for internal teams to build OCI container images without managing their own build infrastructure. Our customers just define three things:

01

🔨 Build Configs / Specs

What internal packages and dependencies to include, resource requirements (disk, memory), and target architecture (x86_64, arm64).

02

📝 Dockerfile

The recipe that describes what goes into the image.

03

🔫 How to Trigger

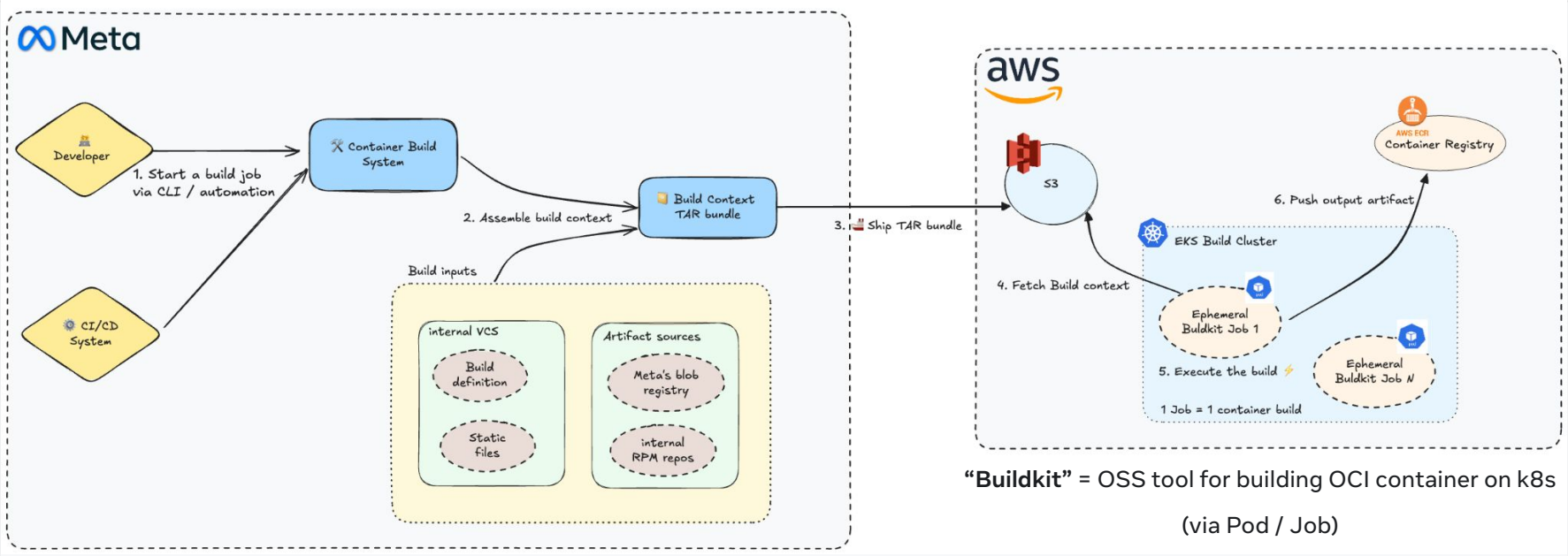
- internal CLI for ad-hoc builds
- CI/CD for automated production builds.

Meta

Meta

🛠️ Build System Pipeline

From local context assembly to remote execution on AWS EKS.



"Buildkit" = OSS tool for building OCI container on k8s (via Pod / Job)

1. Build Context Assembly

CLI/Automation gathers build inputs into a single TAR bundle.

2. Transport

The context TAR is shipped to an S3 bucket

3. Execution

Ephemeral BuildKit jobs on EKS: ***fetch the context***, **execute the build**, and **push the artifact to ECR**

📙History

Kaniko Served Us Well, But We Hit Its Ceiling

Kaniko GitHub repo was archived in June 2025 — no more patches 😢
We needed a tool that could keep up with our scale and security requirements.
Buildkit checks all the boxes

Speed

Parallel execution

BuildKit transforms a Dockerfile into a graph and solves it concurrently; Kaniko is strictly sequential.

Instant filesystem change detection

Uses kernel-level overlayfs instead of Kaniko's brute-force file scanning (the single biggest bottleneck).

Smarter caching

Content-addressable checksums only rebuild what actually changed. Kaniko's registry-based cache forces full rebuilds on any miss.

Security

User-namespaced & rootless modes

BuildKit supports advanced isolation modes; Kaniko only runs as root.

Stricter Dockerfile semantics

Enforces better practices and reduces security risks from ambiguous instructions.

Meta

DEEP DIVE

🔍 Buildkit internal: Where is Nested Container?

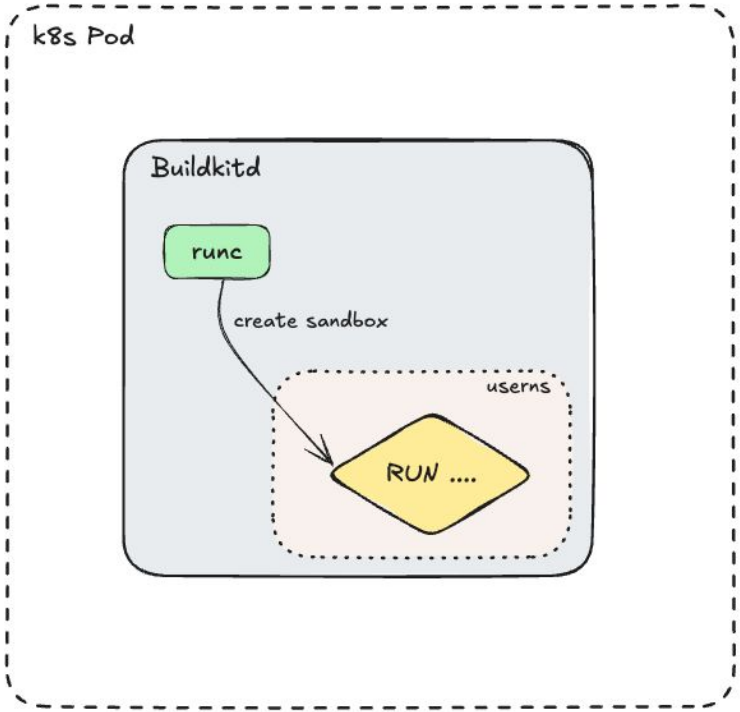
Nested Model

We run “BuildKit” container inside a container (Pod).

Inside the pod, ***BuildKit*** *creates a nested sandbox (“new userns”) for every RUN instruction in* ***Dockerfile.***

runc Invocation

BuildKit uses *runc* to create these ephemeral sandboxes on the fly



Meta

🔒 SECURITY

🤦 Our Deployment Model: “User-Namespaced BuildKit Pod”

k8s Pod Spec

✓ `hostUsers: false` + `privileged: true`

Security & Functionality Balance

Privileged, but safely contained:

Capabilities are scoped to the user namespace - not the host.

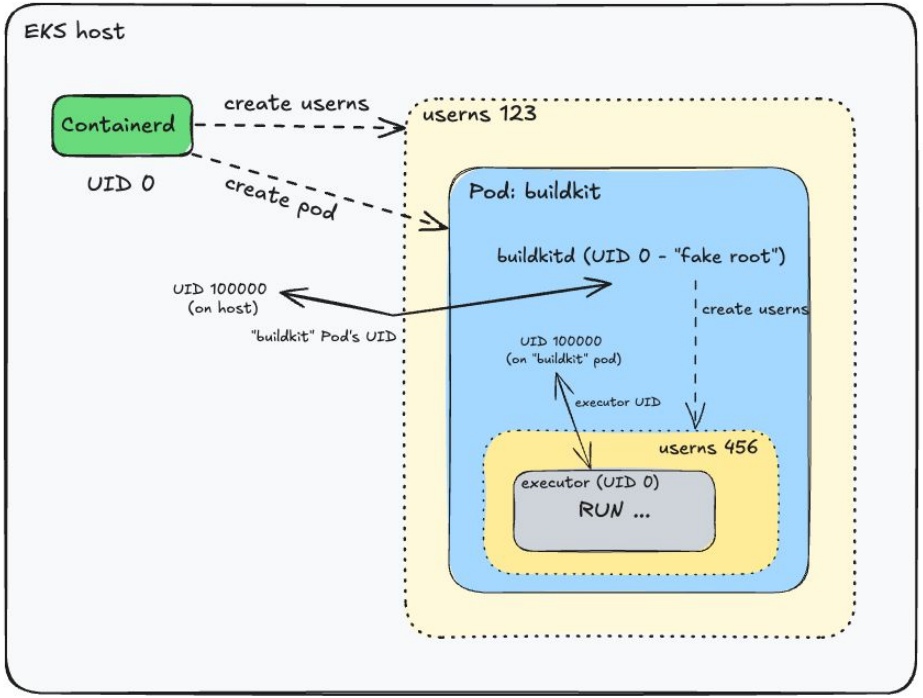
Userns & UID isolation:

UID 0 inside maps to unprivileged UID (e.g., 100000) outside.

Escape = nobody.

Process sandbox intact:

BuildKit isolates each RUN step, preventing malicious Dockerfiles from affecting the daemon.



Buildkit Usernamespace Job

Meta

Meta

⚠️ ⚠️ Challenges We Hit

1. Rootless BuildKit Is Tricky

We evaluated fully rootless alternatives, but they weren't viable:

- **Rootless BuildKit (`runAsUser: 1000`):** Requires disabling the process sandbox, allowing malicious **RUN** instructions to kill the build daemon.
- **Rootless end-to-end:** Requires node-level runtime config & k8s feature flags - impractical on managed Kubernetes (EKS, etc) where you don't control everything.

→ **This led us to choose the user-namespaced model.**

2. Dockerfile Compatibility

User namespaces change what's possible inside builds: operations on privileged filesystems（/proc , /sys ）no longer work.

Our Solution

Leveraging **feature gate** that allows teams to opt in at their own pace, along with migration guidance to help update affected Dockerfiles.

Thanks for ~~(going through this painful journey with us)~~ listening!

Thanks

Meta

for making this presentation happen!