Slacker Outline

Background

- Containers: lightweight isolation
- Docker: file-system provisioning

Container Workloads

Default Driver: AUFS

Our Driver: Slacker

Evaluation

Conclusion

Why use containers?

Why use containers?

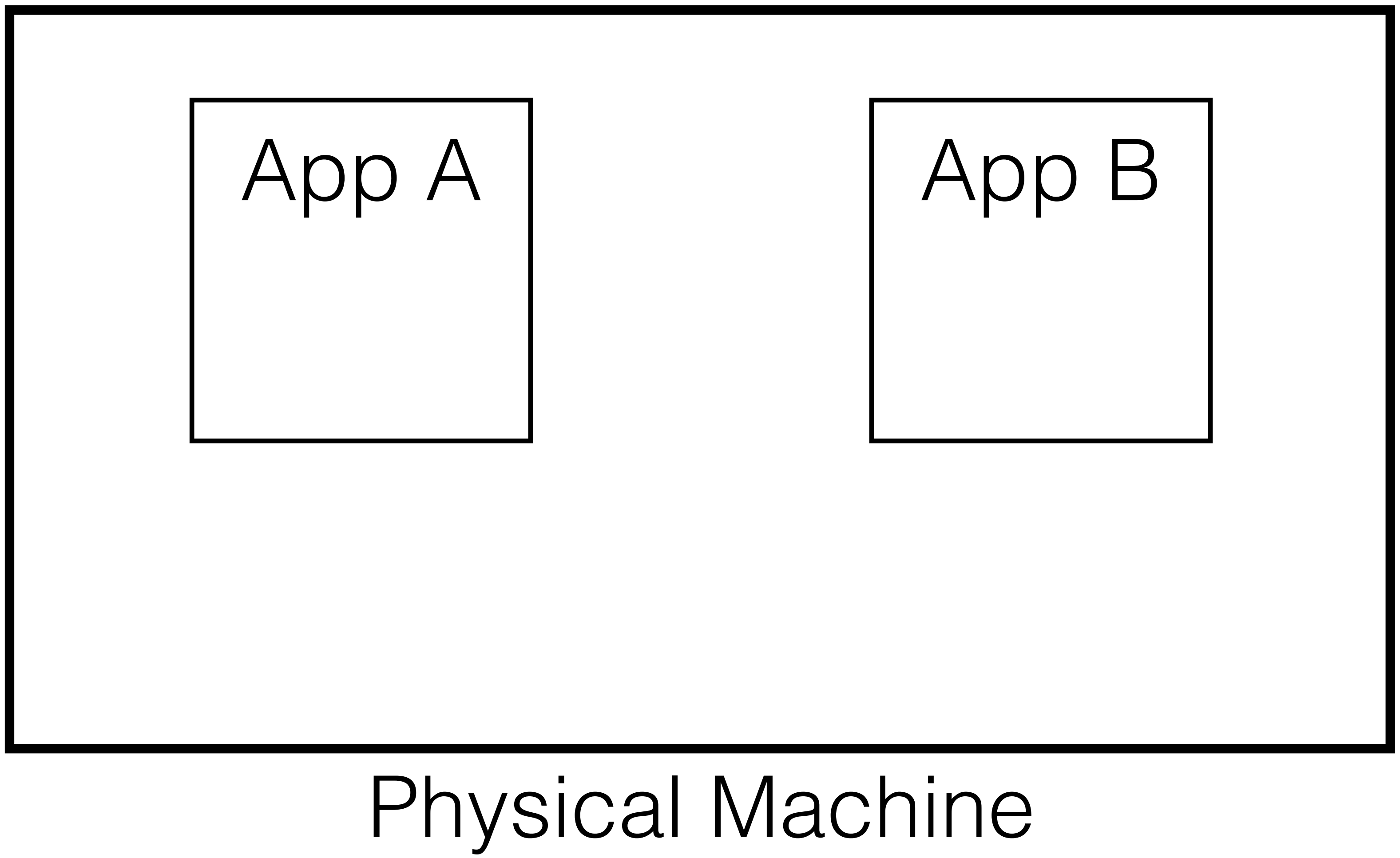
(it’s trendy)

Why use containers?

~~(it's trendy)~~

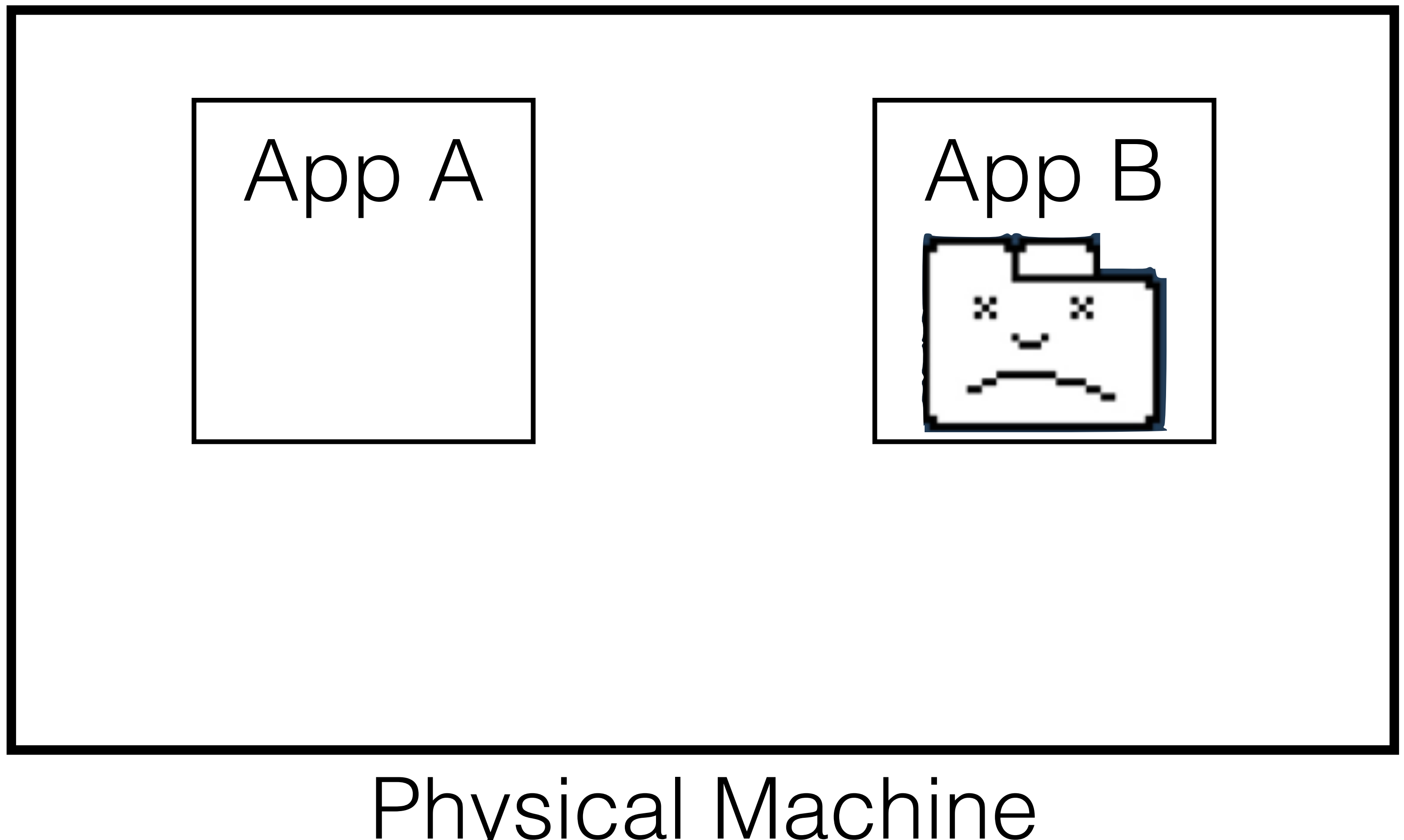
(efficient solution to classic problem)

Big Goal: Sharing and Isolation



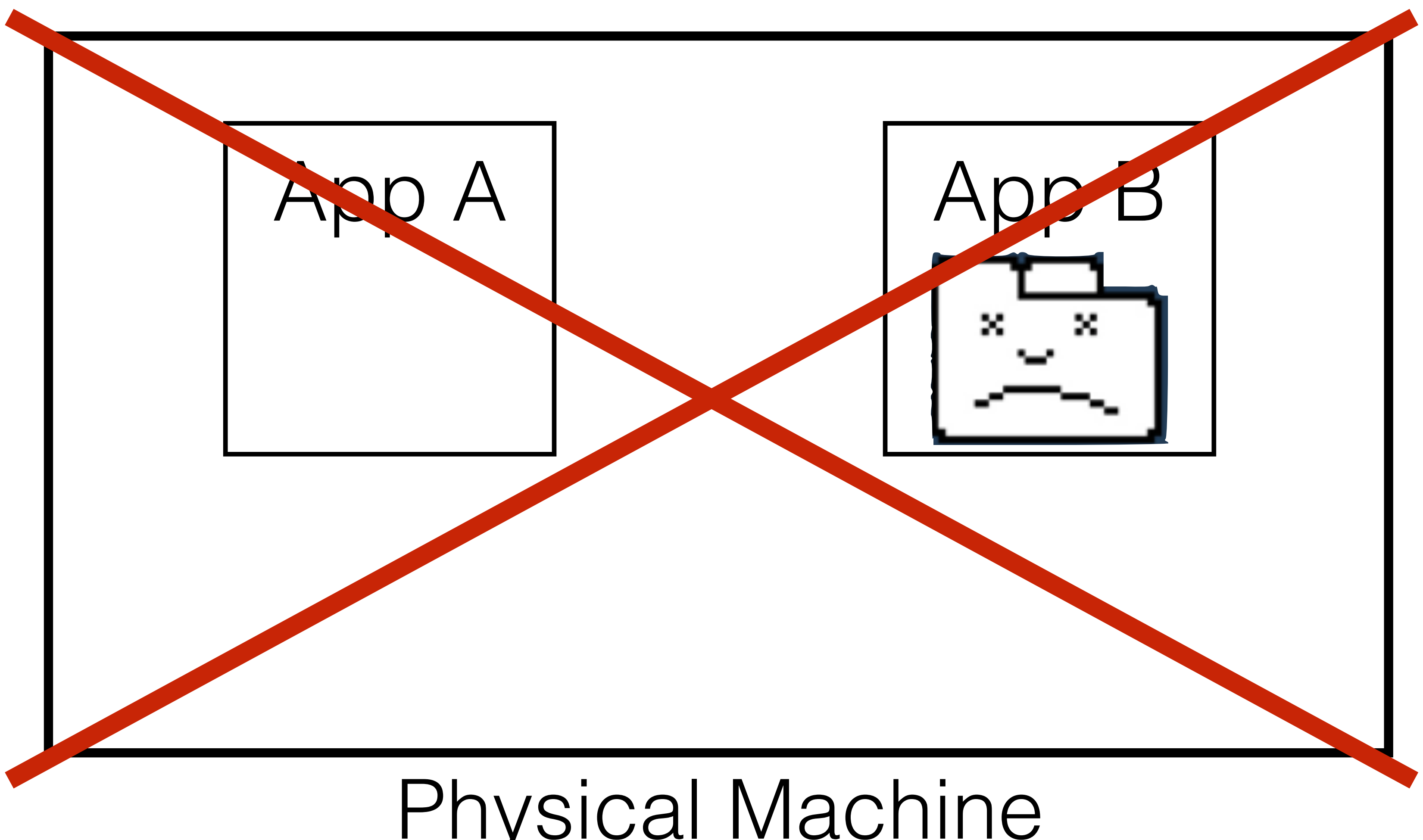
want: multitenancy

Big Goal: Sharing and Isolation



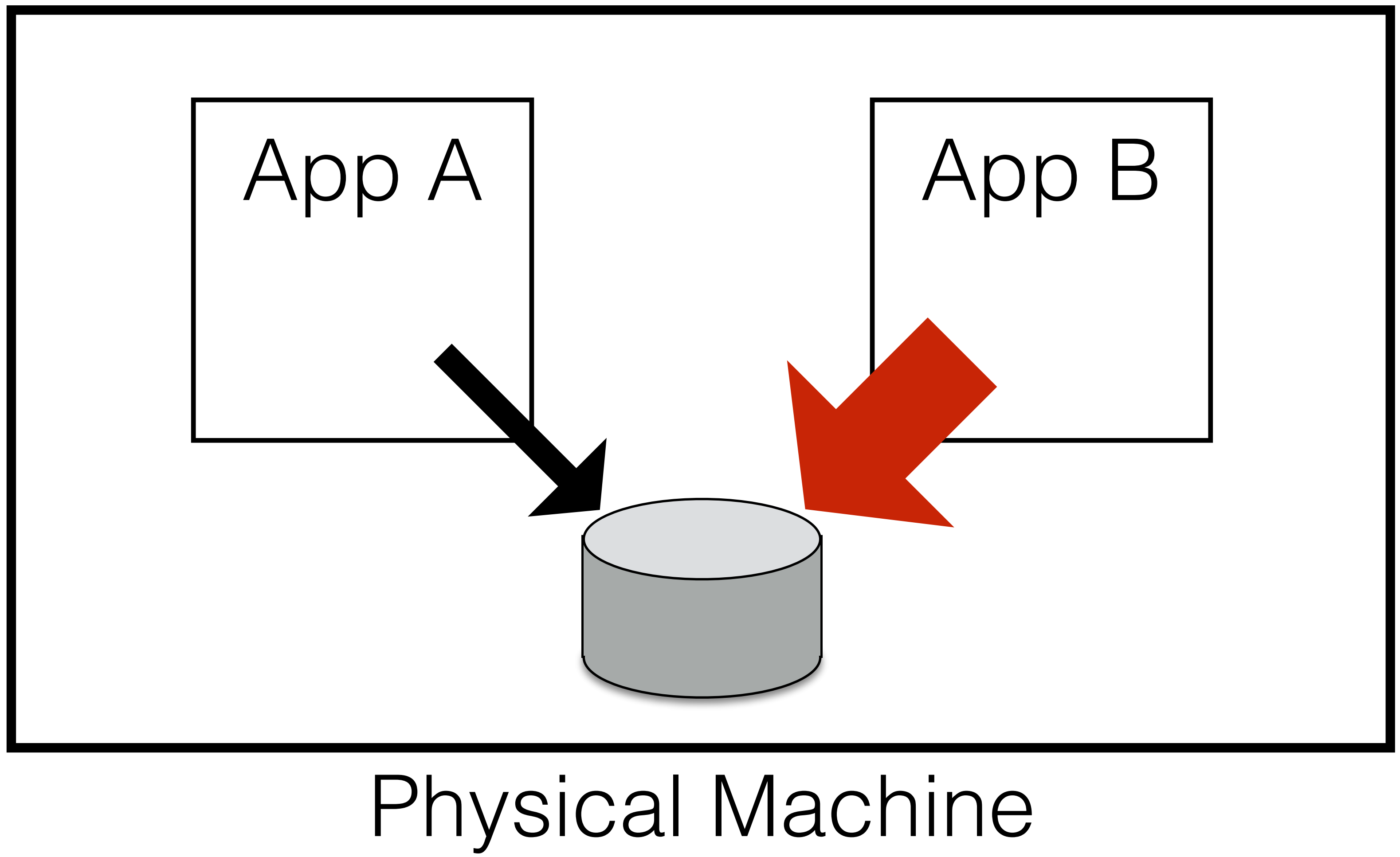
don't want: crashes

Big Goal: Sharing and Isolation



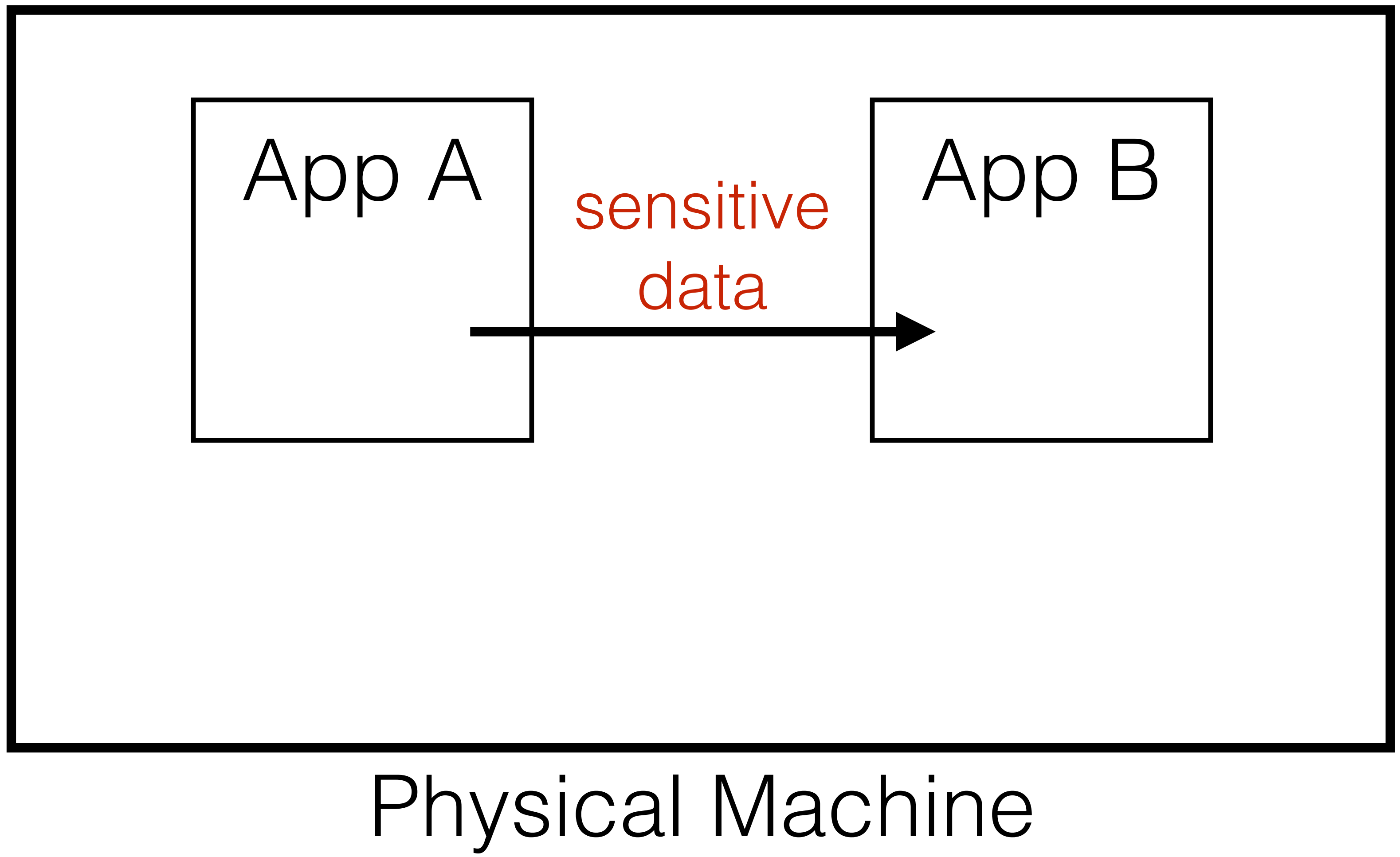
don't want: crashes

Big Goal: Sharing and Isolation



don't want: unfairness

Big Goal: Sharing and Isolation



don’t want: leaks

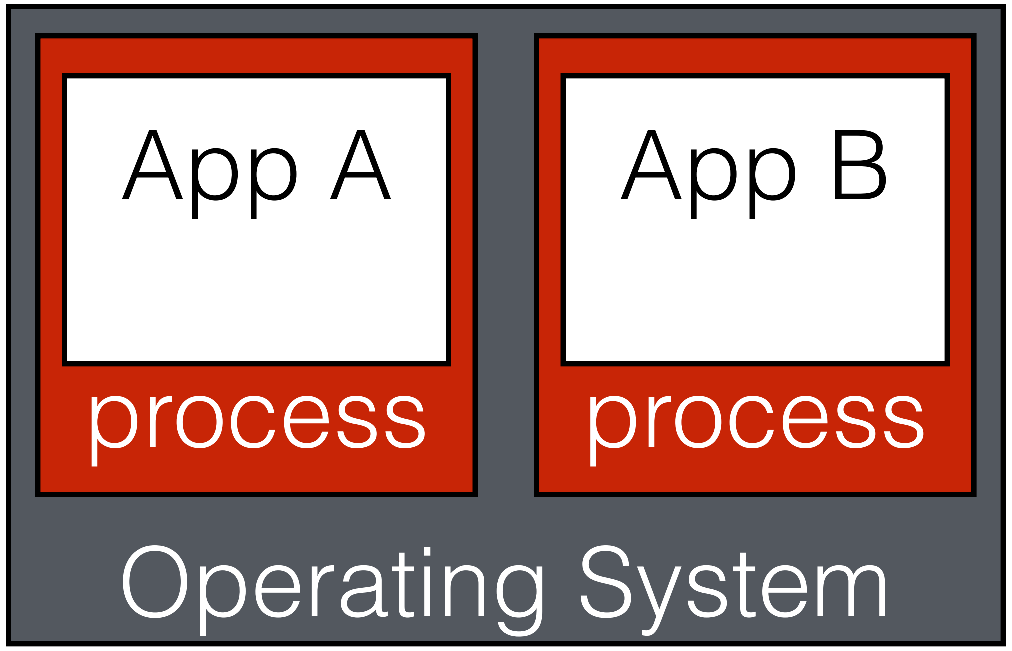
Solution: Virtualization

namespaces and **scheduling** provide illusion of private resources

Evolution of Virtualization

1st generation: process virtualization

- isolate within OS (e.g., virtual memory)
- **fast**, but incomplete (missing ports, file system, etc.)



process virtualization

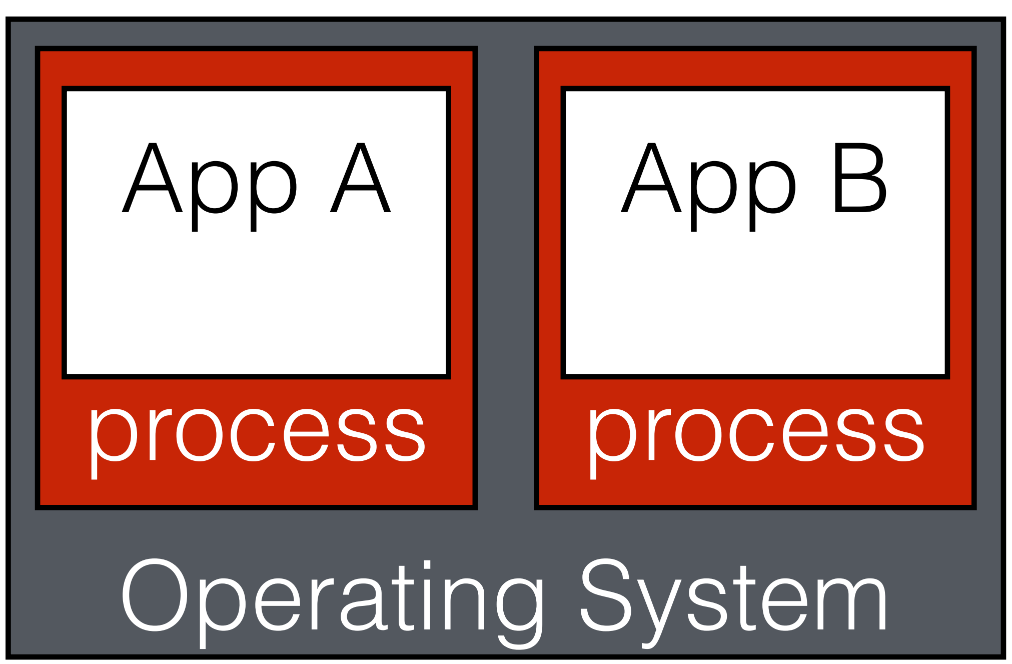
Evolution of Virtualization

1st generation: process virtualization

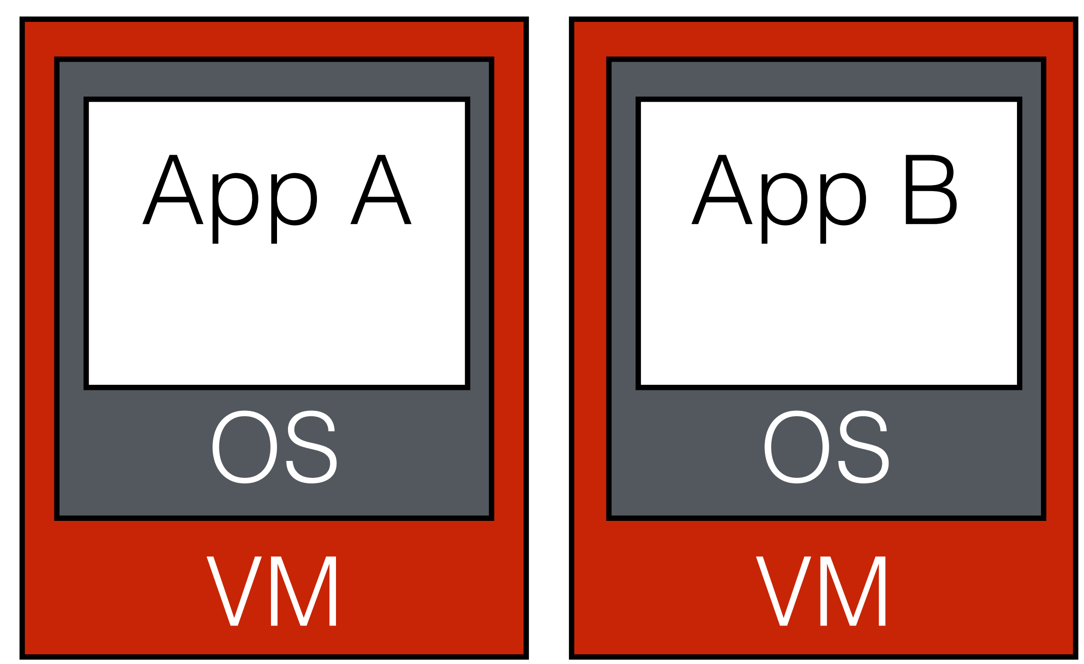
- isolate within OS (e.g., virtual memory)
- **fast**, but incomplete (missing ports, file system, etc.)

2nd generation: machine virtualization

- isolate around OS
- **complete**, but slow (redundancy, emulation)



process virtualization



machine virtualization

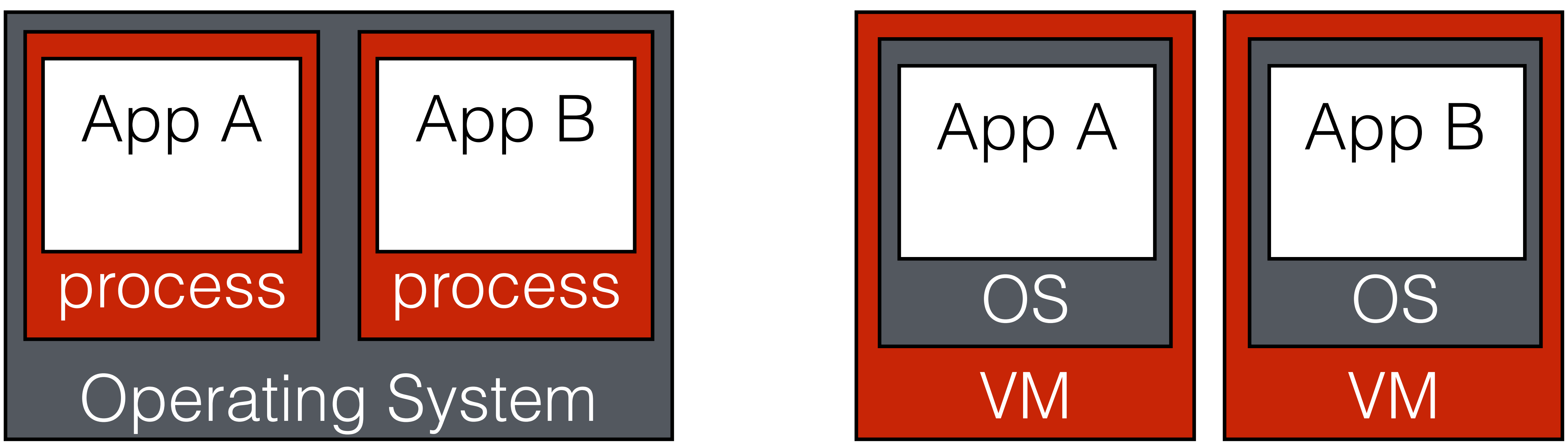
Evolution of Virtualization

1st generation: process virtualization

- isolate within OS (e.g., virtual memory)
- **fast**, but incomplete (missing ports, file system, etc.)

2nd generation: machine virtualization

- isolate around OS
- **complete**, but slow (redundancy, emulation)



process virtualization

machine virtualization

Evolution of Virtualization

1st generation: process virtualization

- isolate within OS (e.g., virtual memory)
- **fast**, but incomplete (missing ports, file system, etc.)

2nd generation: machine virtualization

- isolate around OS
- **complete**, but slow (redundancy, emulation)

3rd generation: container virtualization

- extend process virtualization: ports, file system, etc.
- **fast** and **complete**

Evolution of Virtualization

1st generation: process virtualization

- isolate within OS (e.g., virtual memory)
- **fast**, but incomplete (missing ports, file system, etc.)

2nd generation: machine virtualization

- isolate around OS
- **complete**, but slow (redundancy, emulation)

3rd generation: container virtualization

- extend process virtualization: ports, file system, etc.
- **fast** and **complete**???

many storage challenges

New Storage Challenges

Crash isolation

Physical Disentanglement in a Container-Based File System.
Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. **OSDI ‘14.**

Performance isolation

Split-level I/O Scheduling For Virtualized Environments.
Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau,
Remzi H. Arpaci-Dusseau. **SOSP ‘15.**

File-system provisioning

Slacker: Fast Distribution with Lazy Docker Containers.
Tyler Harter, Brandon Salmon, Rose Liu,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. **FAST ‘16.**

today

Slacker Outline

Background
- Containers: lightweight isolation
- Docker: file-system provisioning

Container Workloads

Default Driver: AUFS

Our Driver: Slacker

Evaluation

Conclusion

Docker Background

Deployment tool built on containers

An application is defined by a file-system image

- application binary
- shared libraries
- etc.

Version-control model

- **extend** images by committing additional files
- **deploy** applications by pushing/pulling images

Containers as Repos

LAMP stack example

- commit 1: **L**inux packages (e.g., Ubuntu)
- commit 2: **A**pache
- commit 3: **M**ySQL
- commit 4: **P**HP

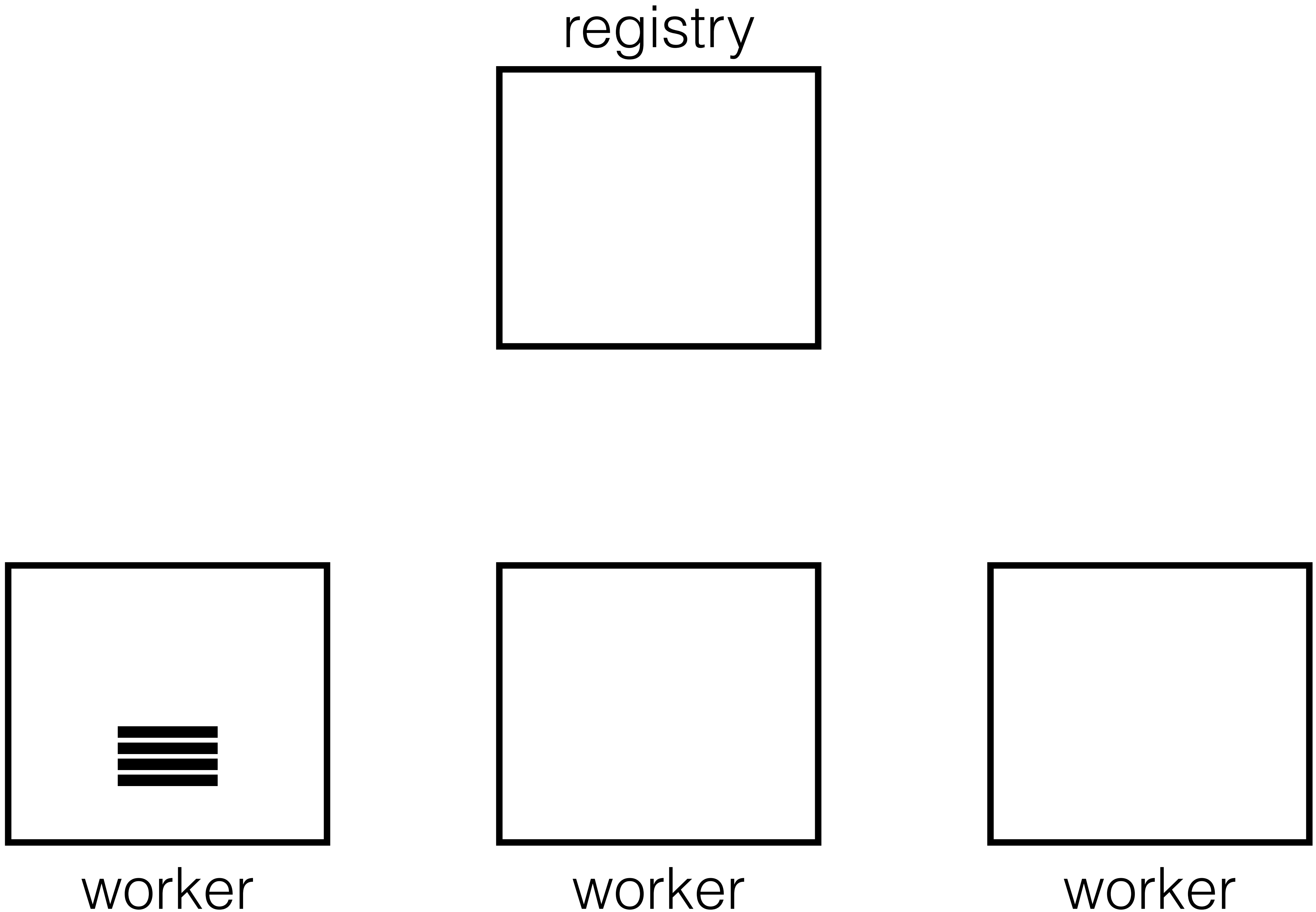
Central registries

- Docker HUB
- private registries

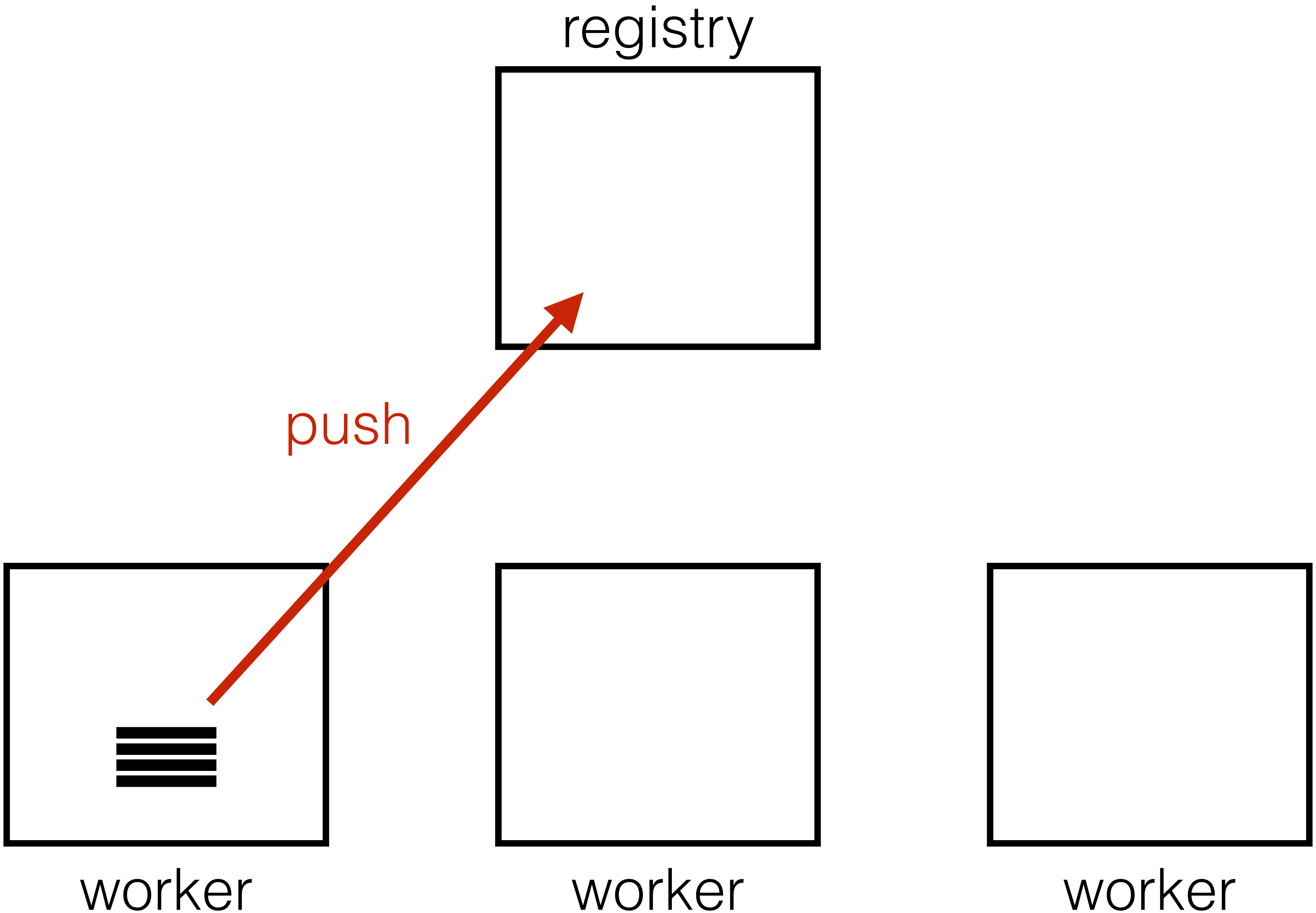
Docker “layer”

- commit
- container scratch space

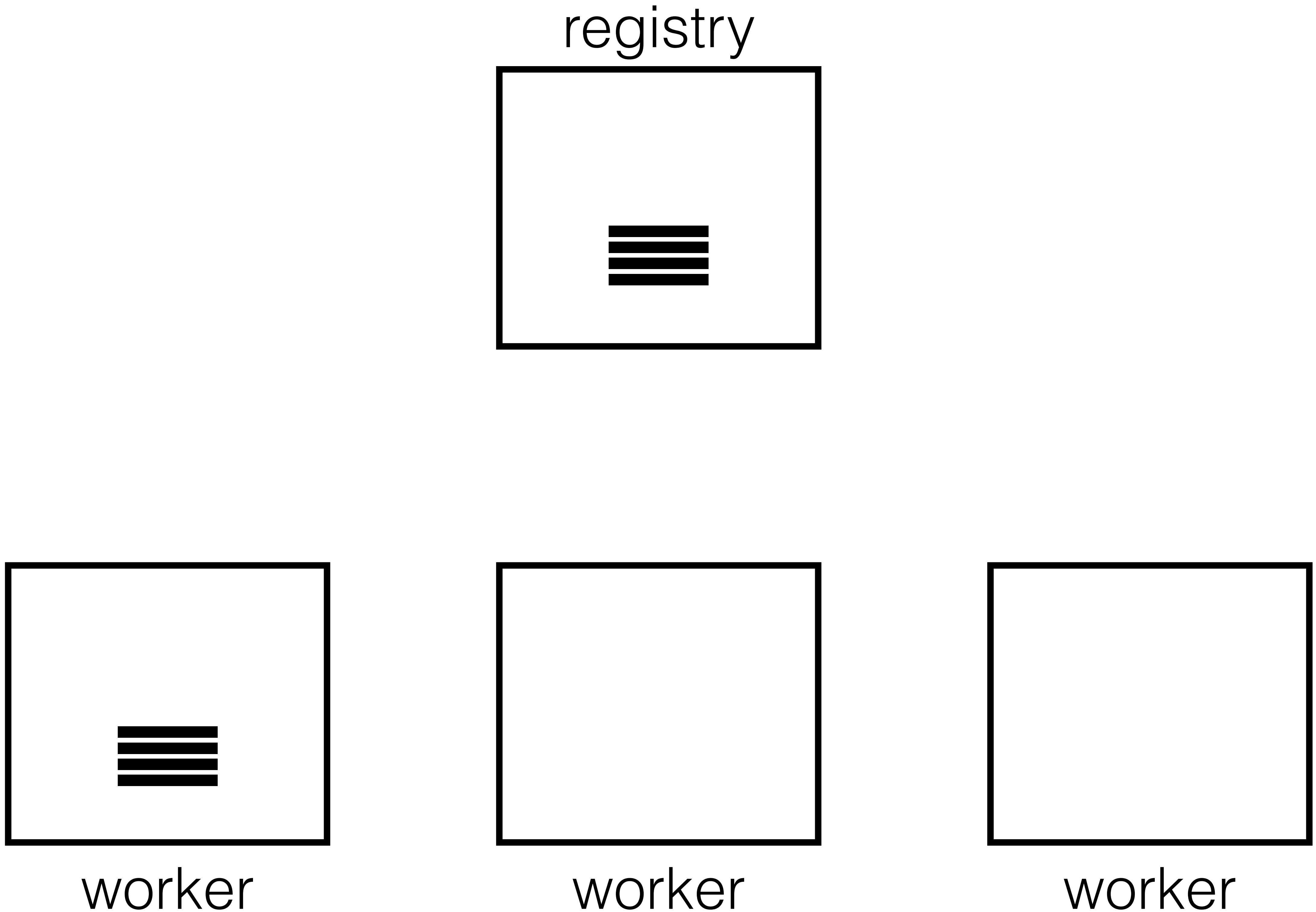
Push, Pull, Run



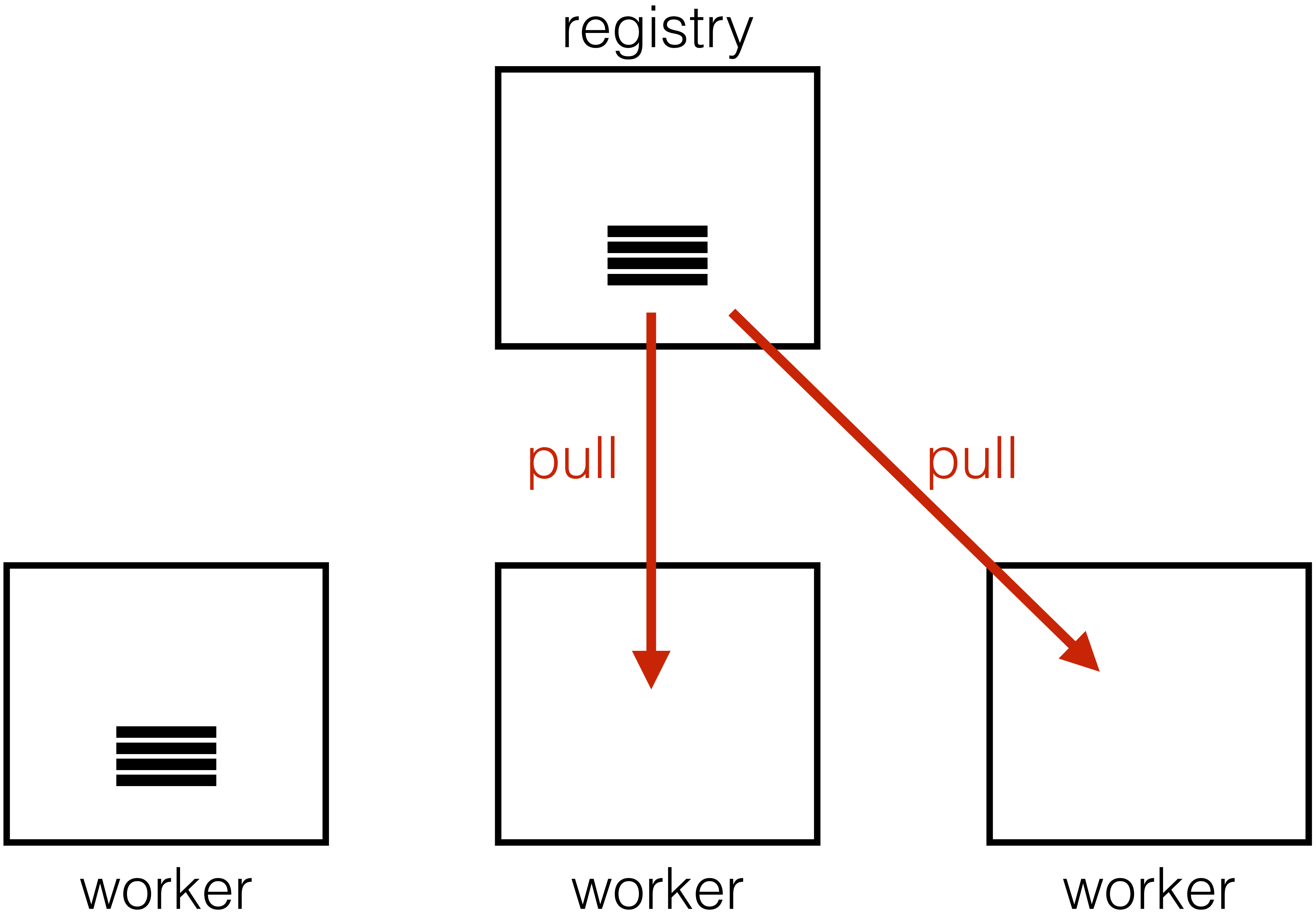
Push, Pull, Run



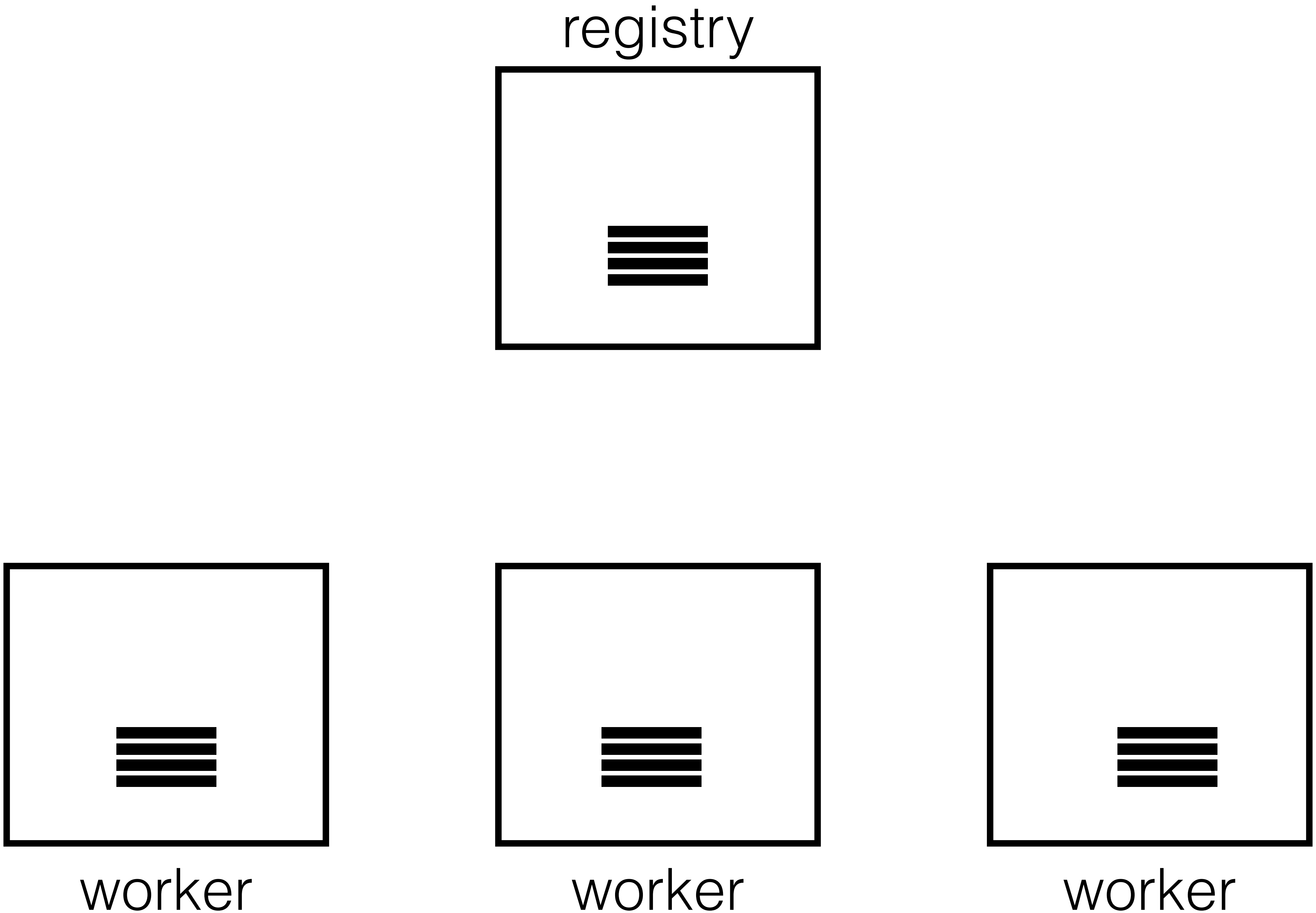
Push, Pull, Run



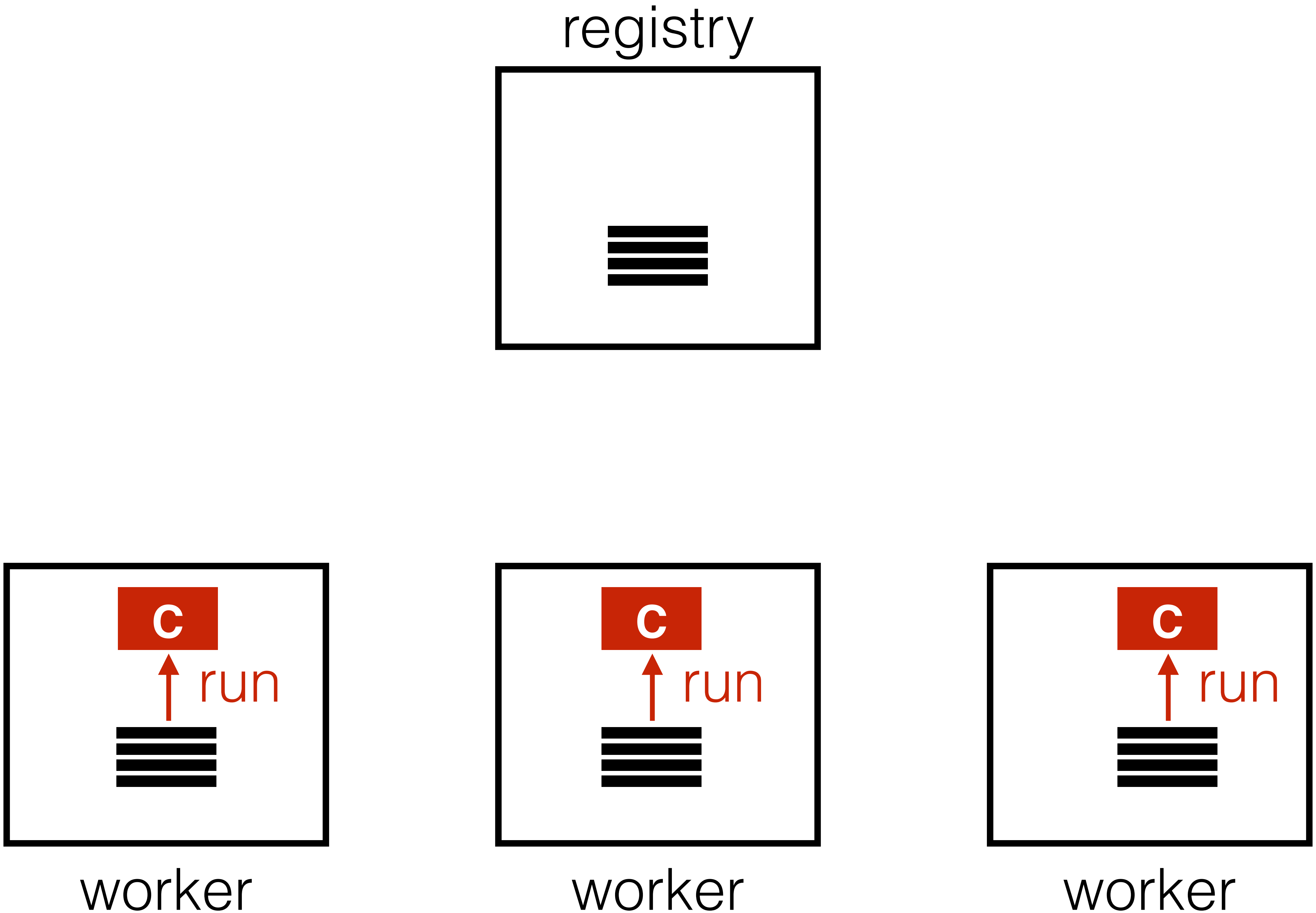
Push, Pull, Run



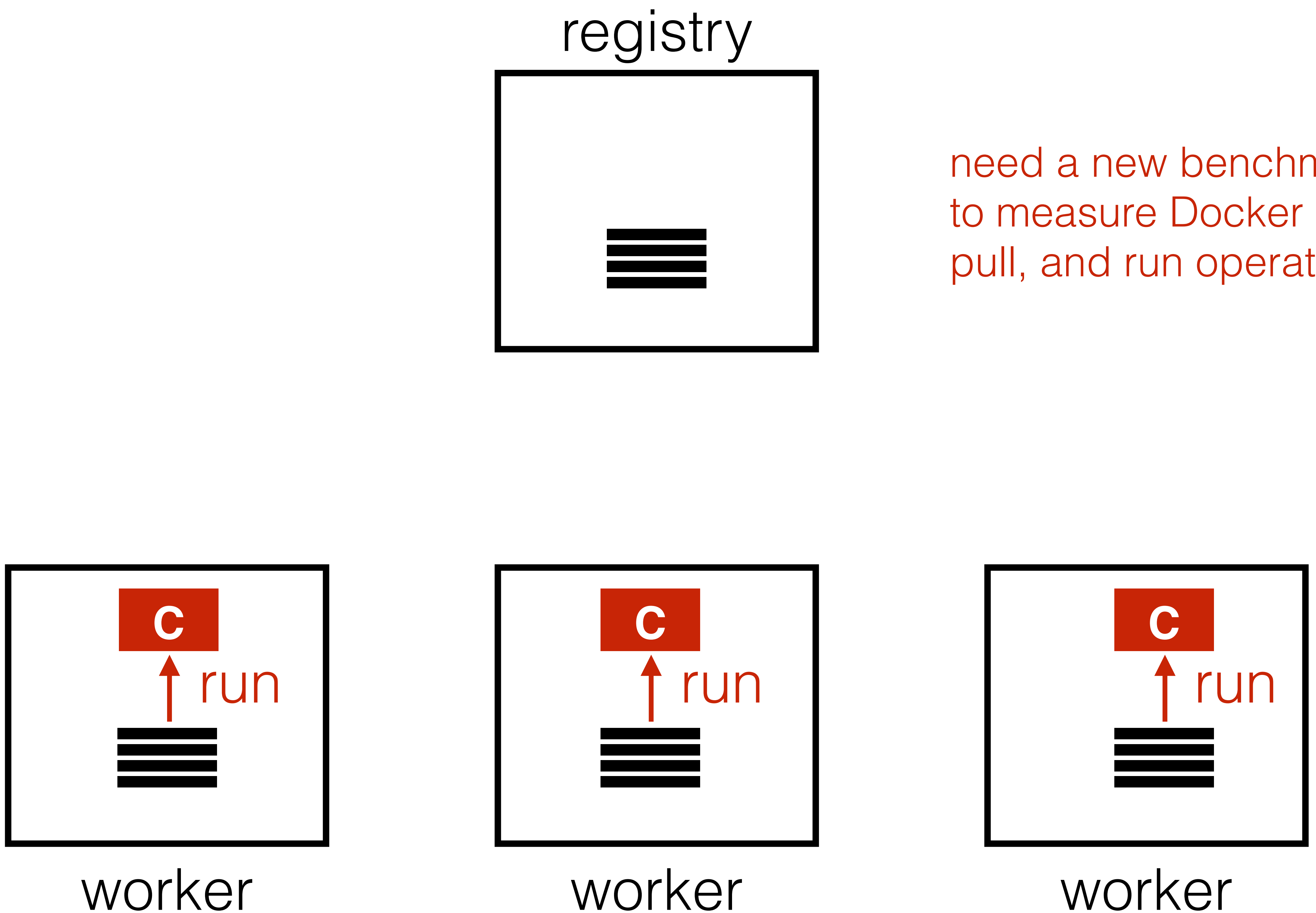
Push, Pull, Run



Push, Pull, Run



Push, Pull, Run



need a new benchmark to measure Docker push, pull, and run operations.

Slacker Outline

Background

Container Workloads

- HelloBench
- Analysis

Default Driver: AUFS

Our Driver: Slacker

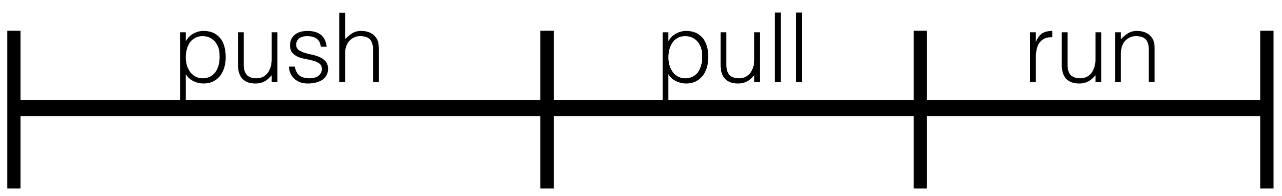
Evaluation

Conclusion

HelloBench

Goal: stress container startup

- including push/pull
- **57 container images** from Docker HUB
- run simple "hello world"-like task
- wait until it's done/ready



HelloBench

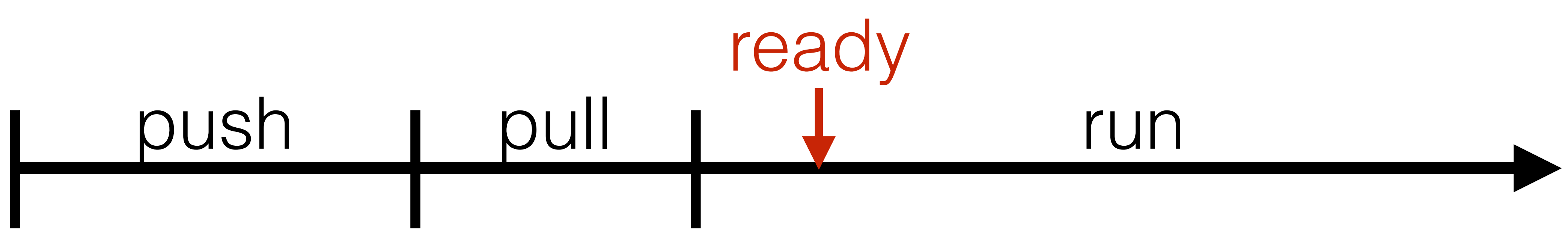
Goal: stress container startup

- including push/pull
- **57 container images** from Docker HUB
- run simple “hello world”-like task
- wait until it’s done/ready

HelloBench

Goal: stress container startup

- including push/pull
- **57 container images** from Docker HUB
- run simple "hello world"-like task
- wait until it's done/ready



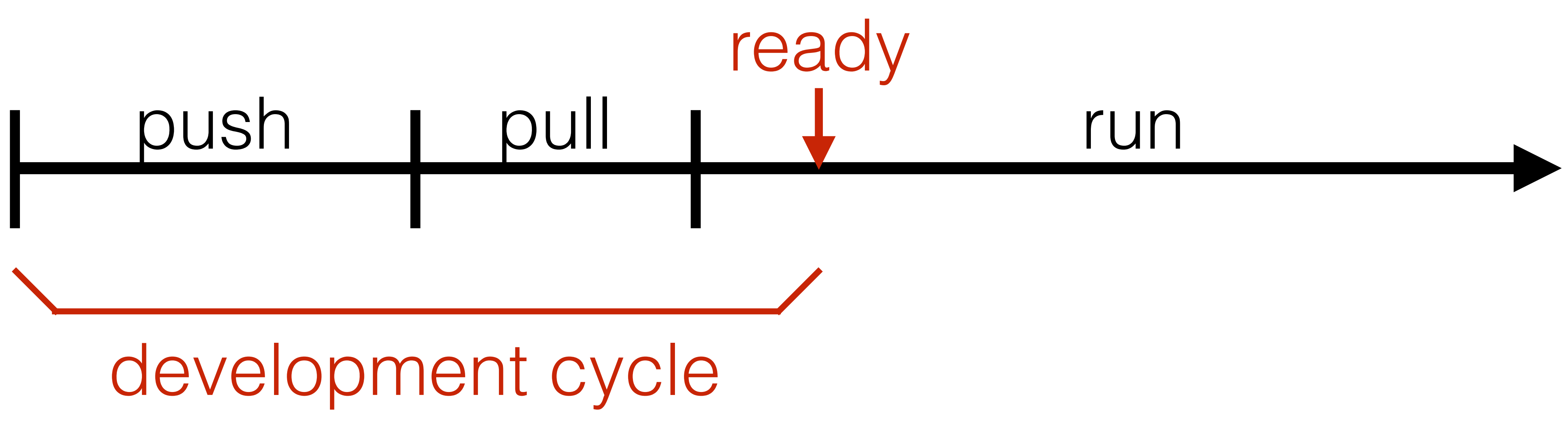
HelloBench

Goal: stress container startup

- including push/pull
- **57 container images** from Docker HUB
- run simple "hello world"-like task
- wait until it's done/ready

Development cycle

- distributed programming/testing



HelloBench

Goal: stress container startup

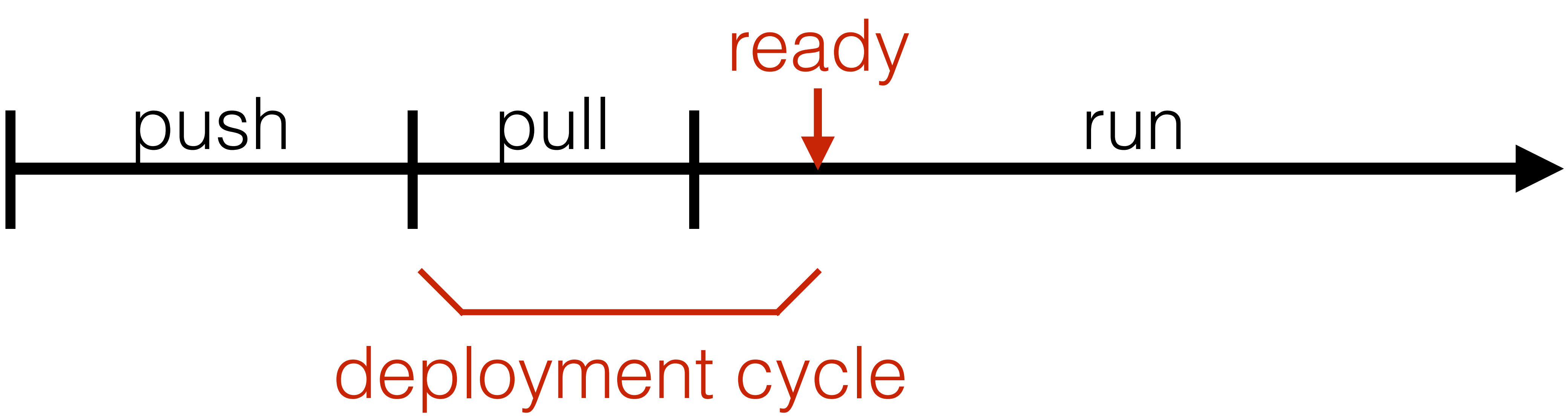
- including push/pull
- **57 container images** from Docker HUB
- run simple “hello world”-like task
- wait until it’s done/ready

Development cycle

- distributed programming/testing

Deployment cycle

- flash crowds, rebalance



Workload Categories

Language
clojure
gcc
golang
haskell
hylang
java
jruby
julia
mono
perl
php
pypy
python
r-base
rakudo-star
ruby
thrift

Linux Distro
alpine
busybox
centos
cirros
crux
debian
fedora
mageia
opensuse
oraclelinux
ubuntu
ubuntu-debootstrap
ubuntu-upstart

Database
cassandra
crate
elasticsearch
mariadb
mongo
mysql
percona
postgres
redis
rethinkdb

Web Framework
django
iojs
node
rails

Web Server
glassfish
httpd
jetty
nginx
php-zendserver
tomcat

Other
drupal
ghost
hello-world
jenkins
rabbitmq
registry
sonarqube

Slacker Outline

Background

Container Workloads

- HelloBench
- Analysis

Default Driver: AUFS

Our Driver: Slacker

Evaluation

Conclusion

Questions

How is data distributed across Docker layers?

How much image data is needed for container startup?

How similar are reads between runs?

Questions

How is data distributed across Docker layers?

How much image data is needed for container startup?

How similar are reads between runs?

HelloBench images

- **circle**: commit
- **red**: image

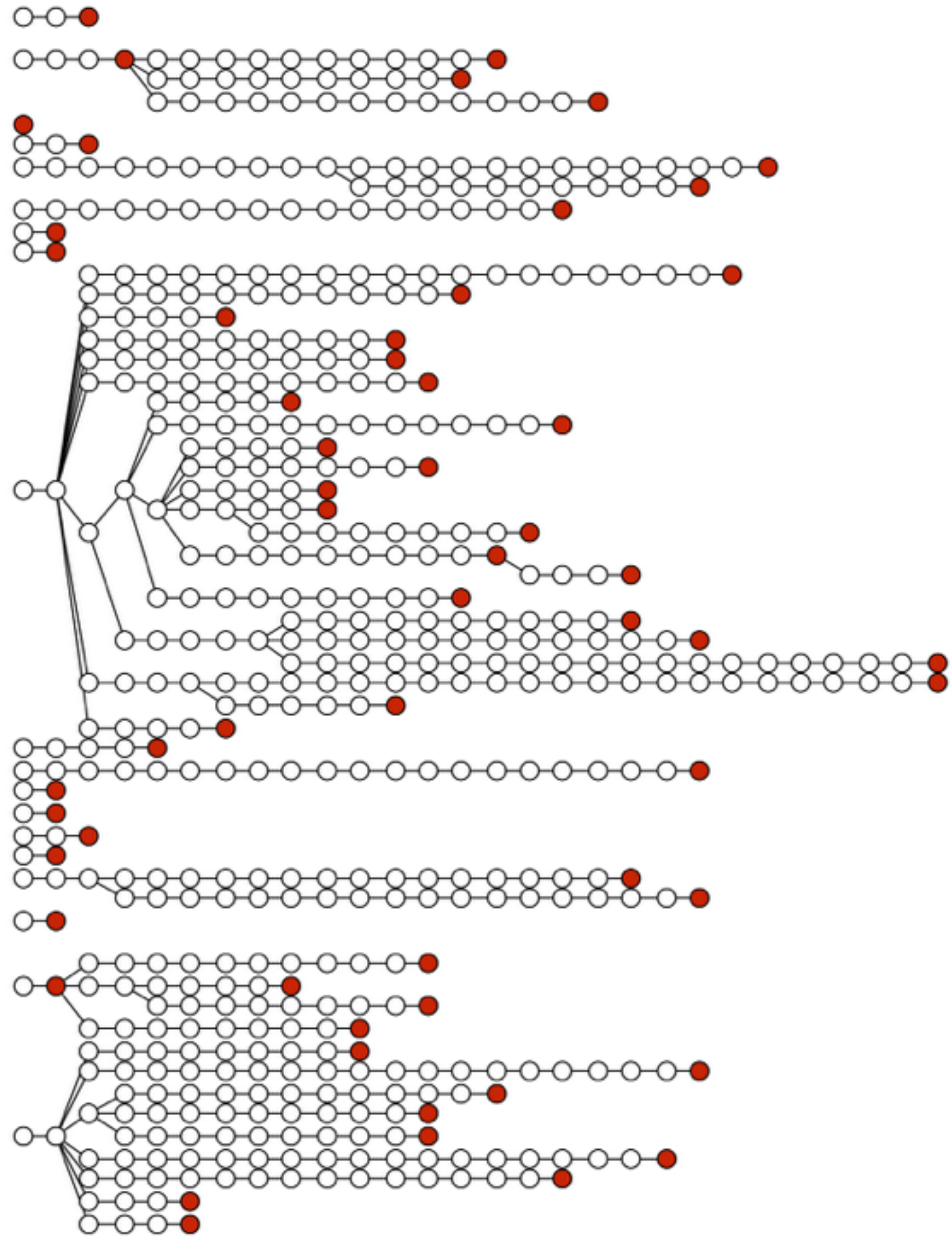


Image Data Depth

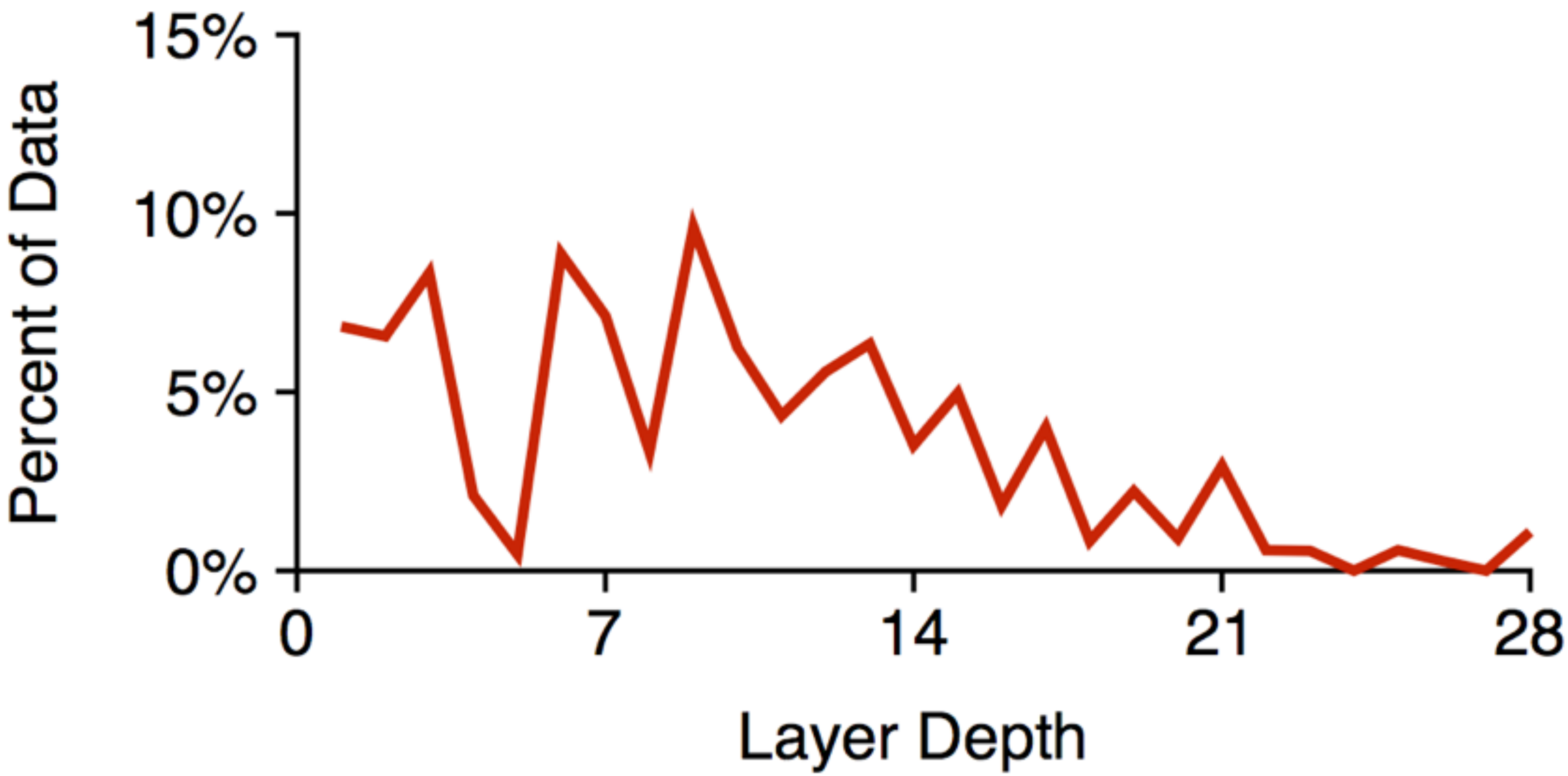
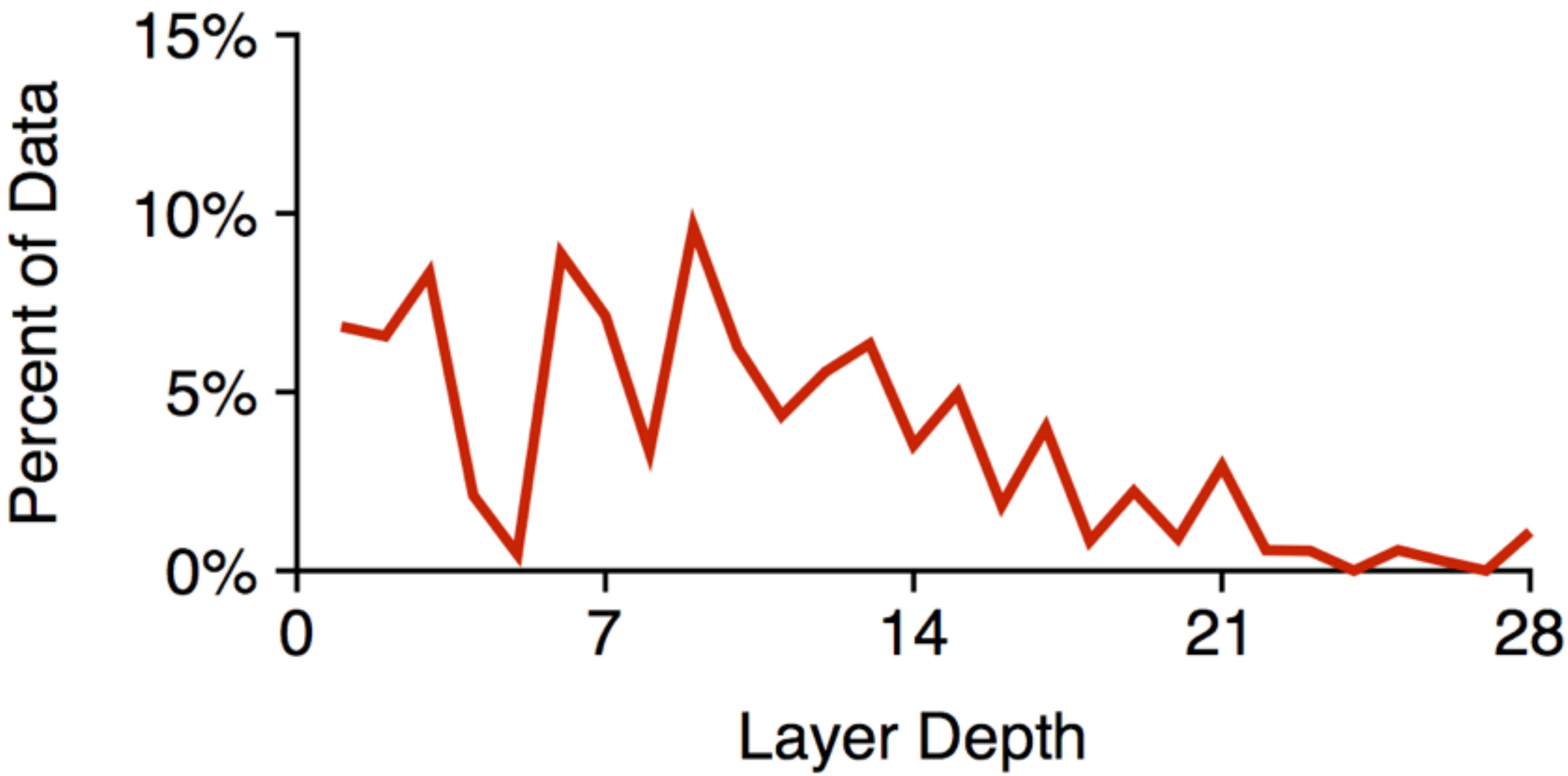


Image Data Depth



half of data is at depth 9+

Questions

How is data distributed across Docker layers?

- half of data is at depth 9+
- **design implication**: flatten layers at runtime

How much image data is needed for container startup?

How similar are reads between runs?

Questions

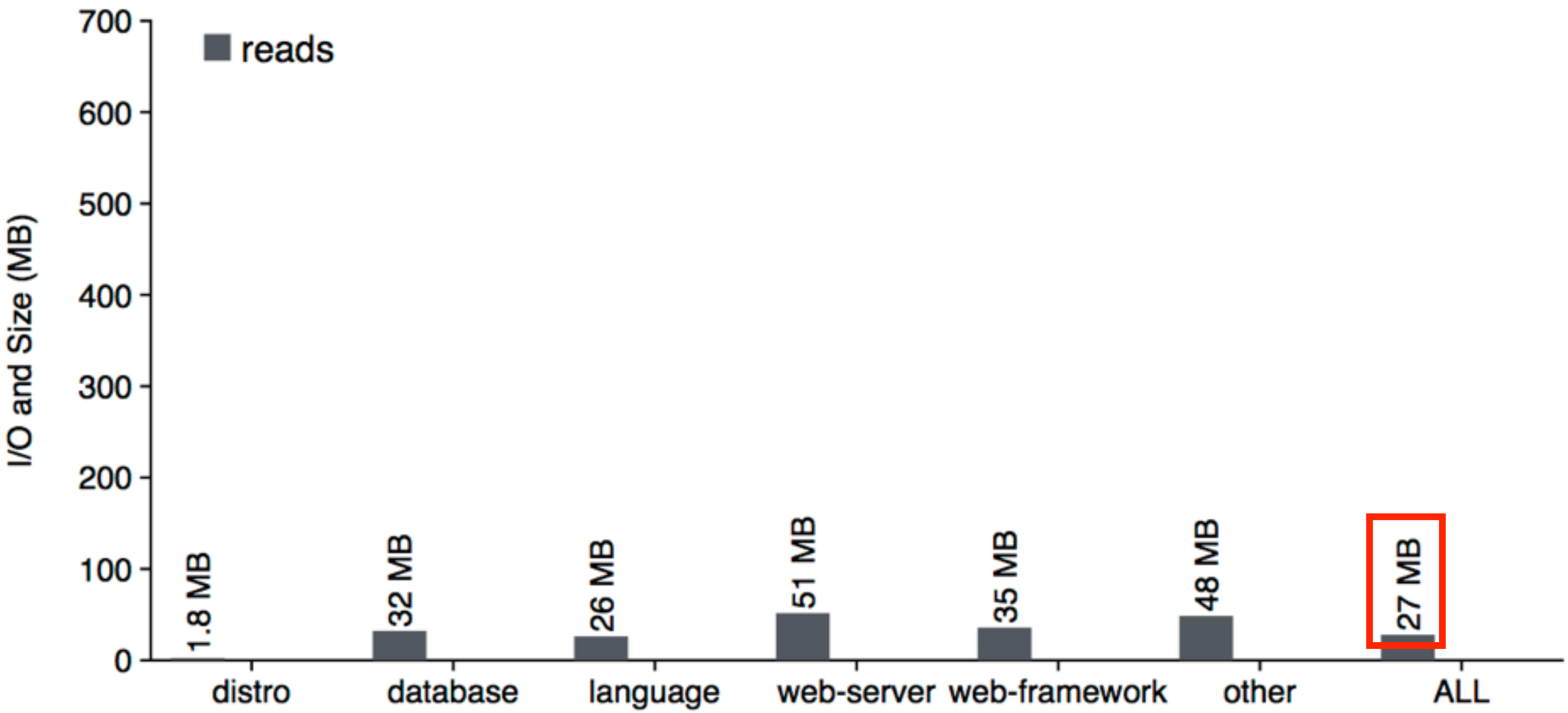
How is data distributed across Docker layers?

- half of data is at depth 9+
- **design implication**: flatten layers at runtime

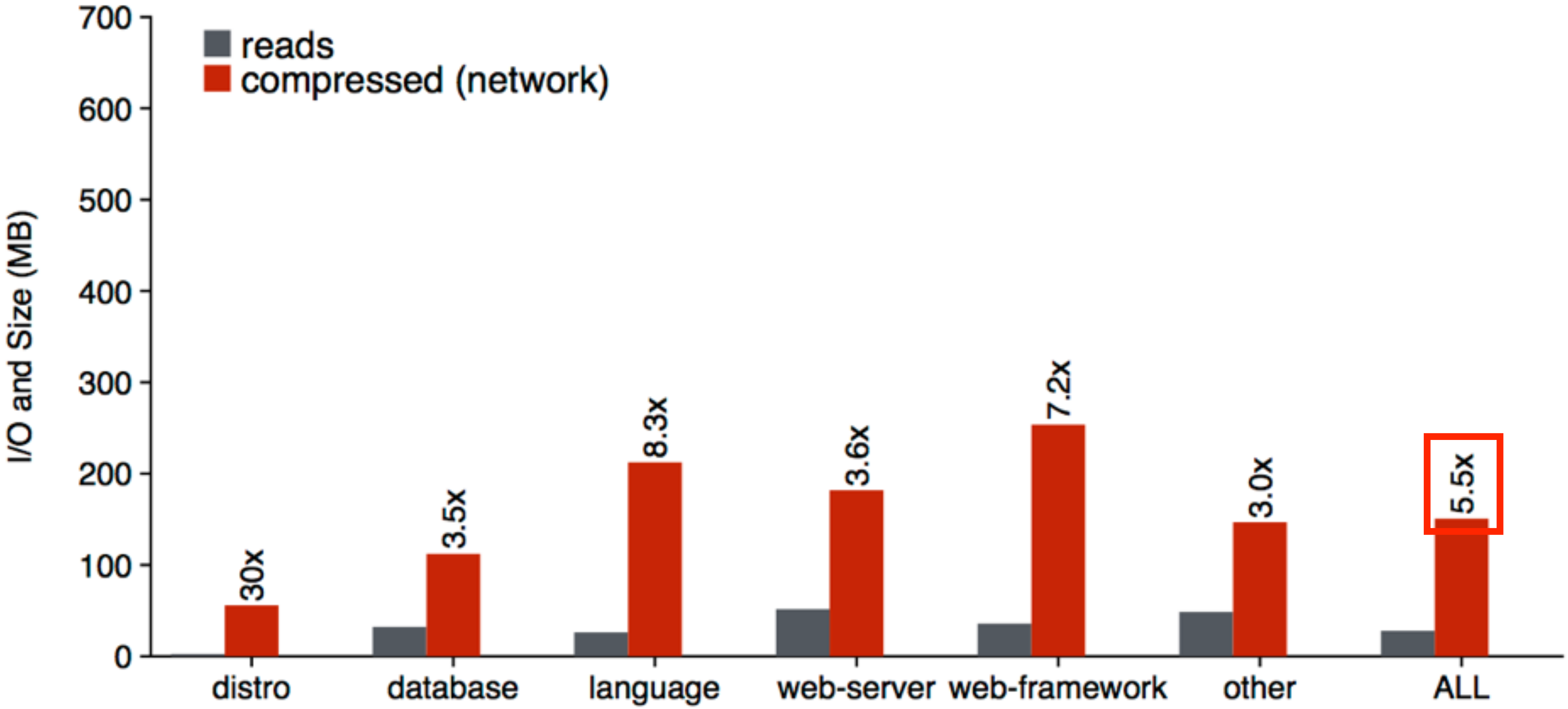
How much image data is needed for container startup?

How similar are reads between runs?

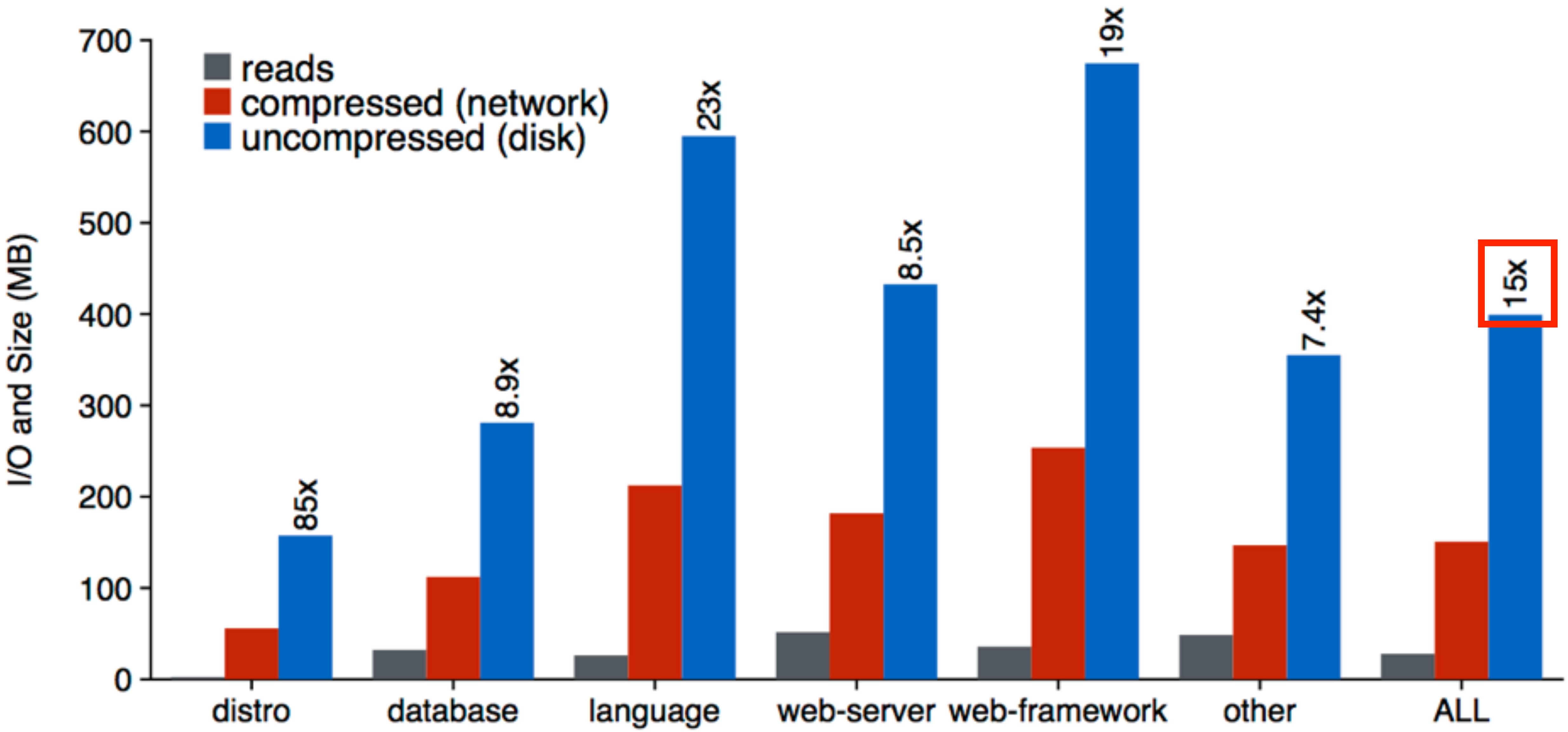
Container Amplification



Container Amplification



Container Amplification



only 6.4% of data needed during startup

Questions

How is data distributed across Docker layers?

- half of data is at depth 9+
- **design implication**: flatten layers at runtime

How much image data is needed for container startup?

- 6.4% of data is needed
- **design implication**: lazily fetch data

How similar are reads between runs?

Questions

How is data distributed across Docker layers?

- half of data is at depth 9+
- **design implication**: flatten layers at runtime

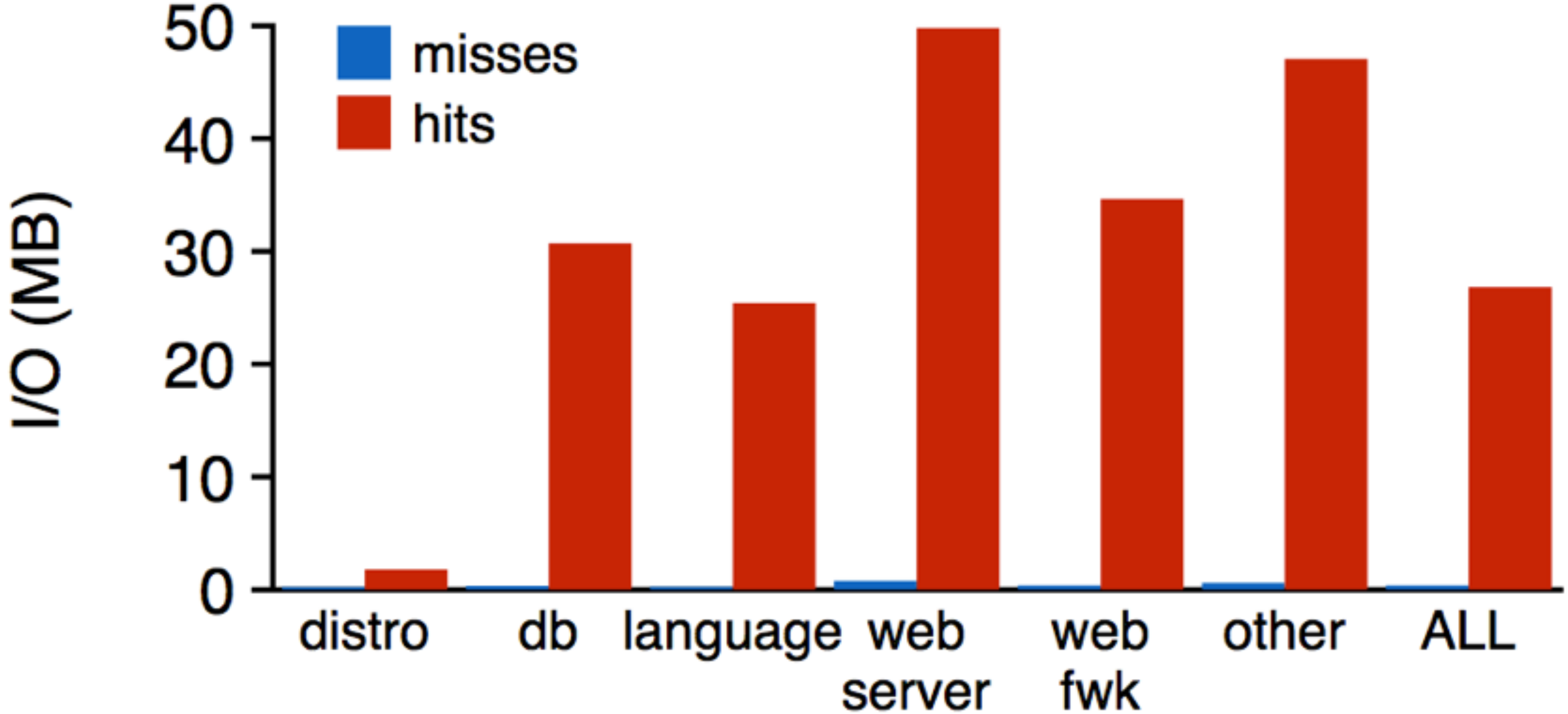
How much image data is needed for container startup?

- 6.4% of data is needed
- **design implication**: lazily fetch data

How similar are reads between runs?

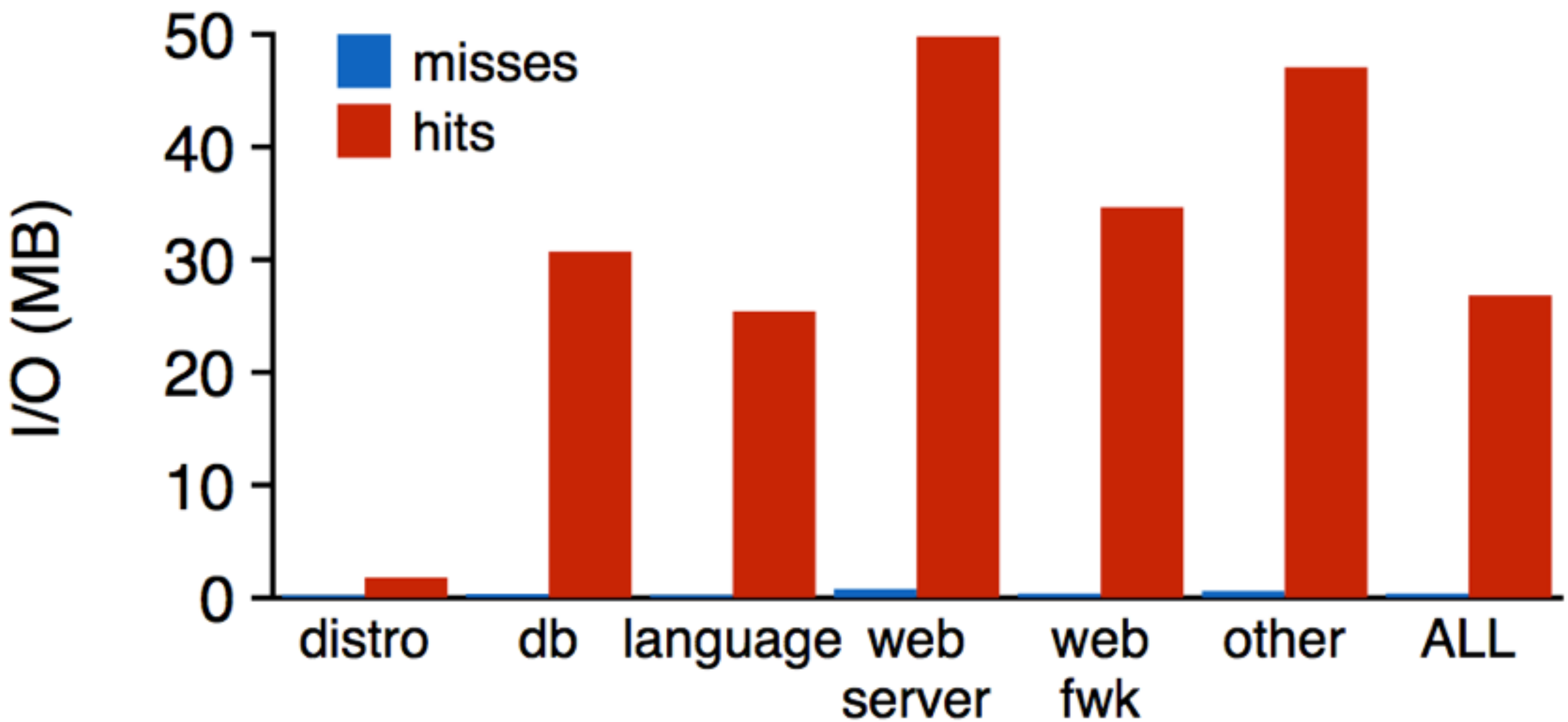
Repeat Runs

measure hits/misses for second of two runs



Repeat Runs

measure hits/misses for second of two runs



up to 99% of reads could be serviced by a cache

Questions

How is data distributed across Docker layers?

- half of data is at depth 9+
- **design implication**: flatten layers at runtime

How much image data is needed for container startup?

- 6.4% of data is needed
- **design implication**: lazily fetch data

How similar are reads between runs?

- containers from same image have similar read patterns
- **design implication**: share cache state between containers

Slacker Outline

Background

Container Workloads

Default Driver: AUFS
- Design
- Performance

Our Driver: Slacker

Evaluation

Conclusion

AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

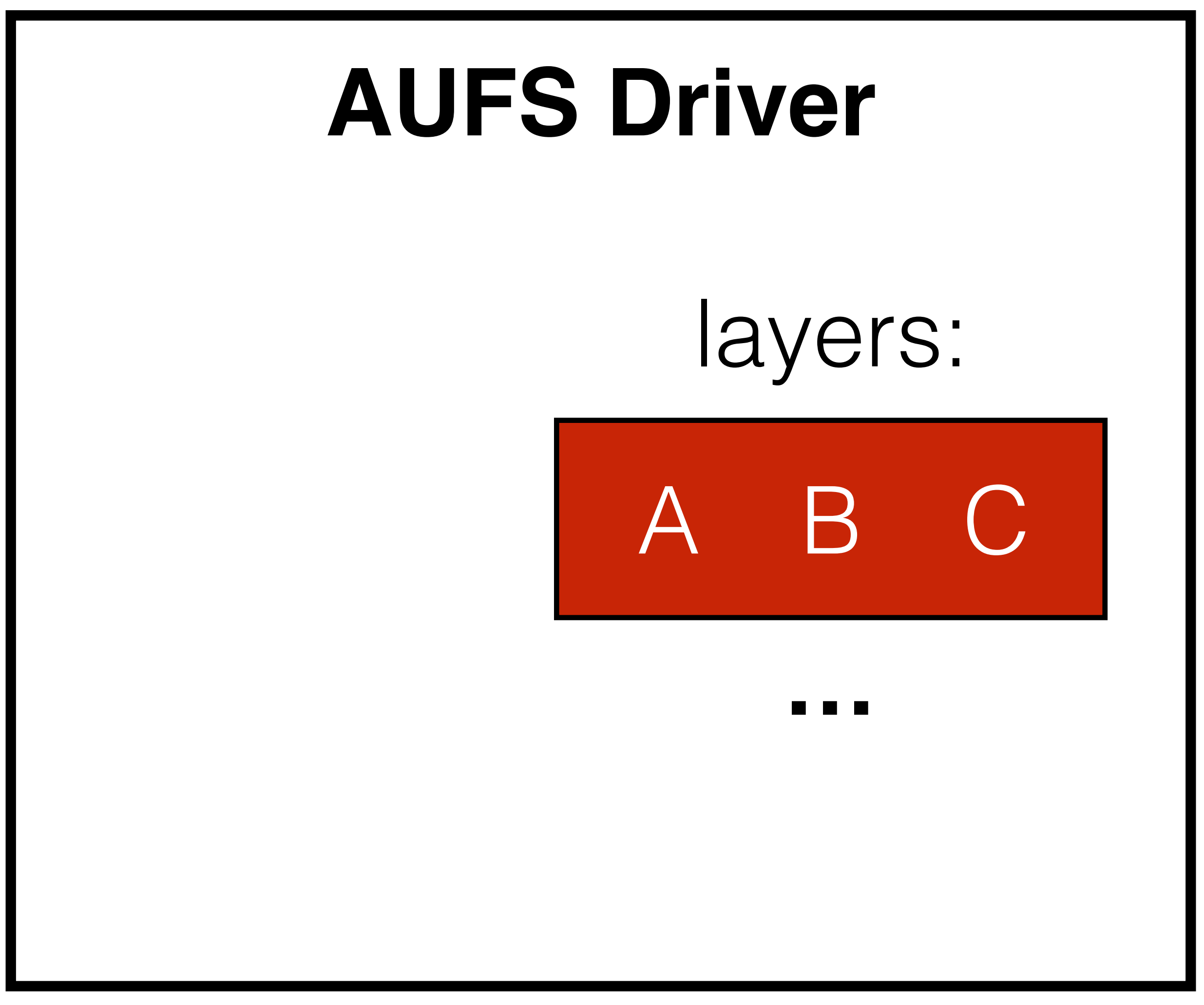
Operations

- push
- pull
- run

AUFS Storage Driver

Uses AUFS file system (Another Union FS)

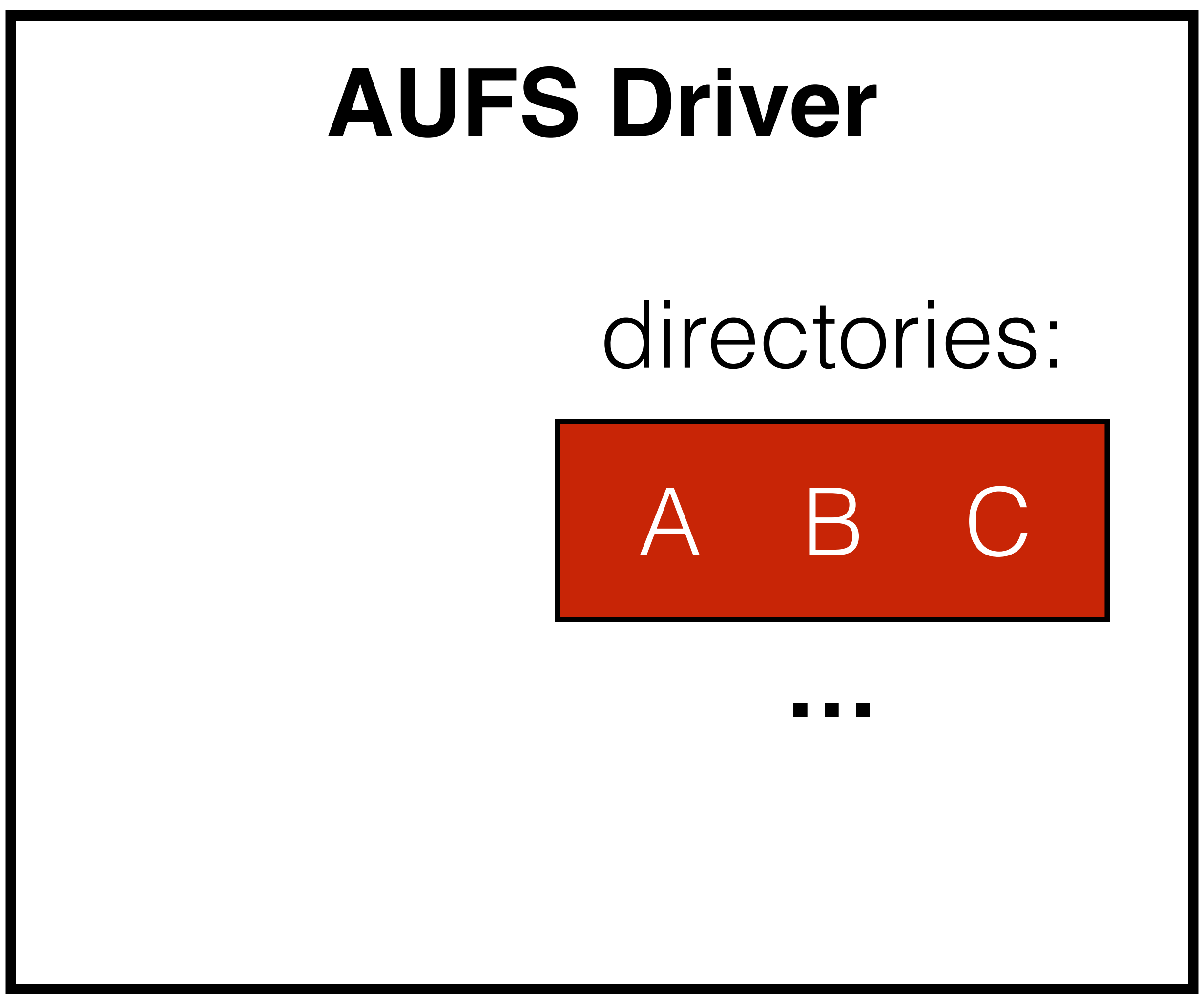
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

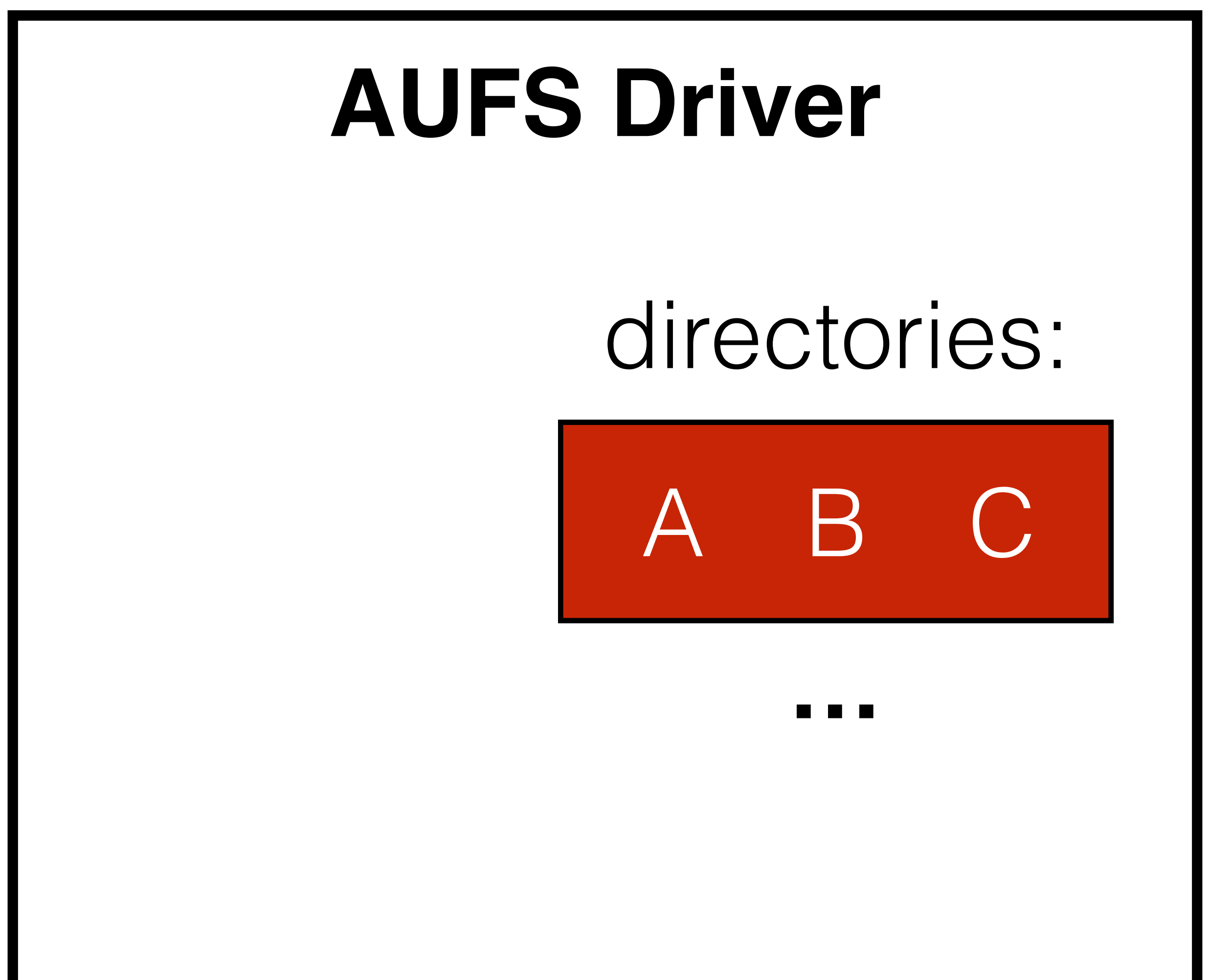


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

PUSH

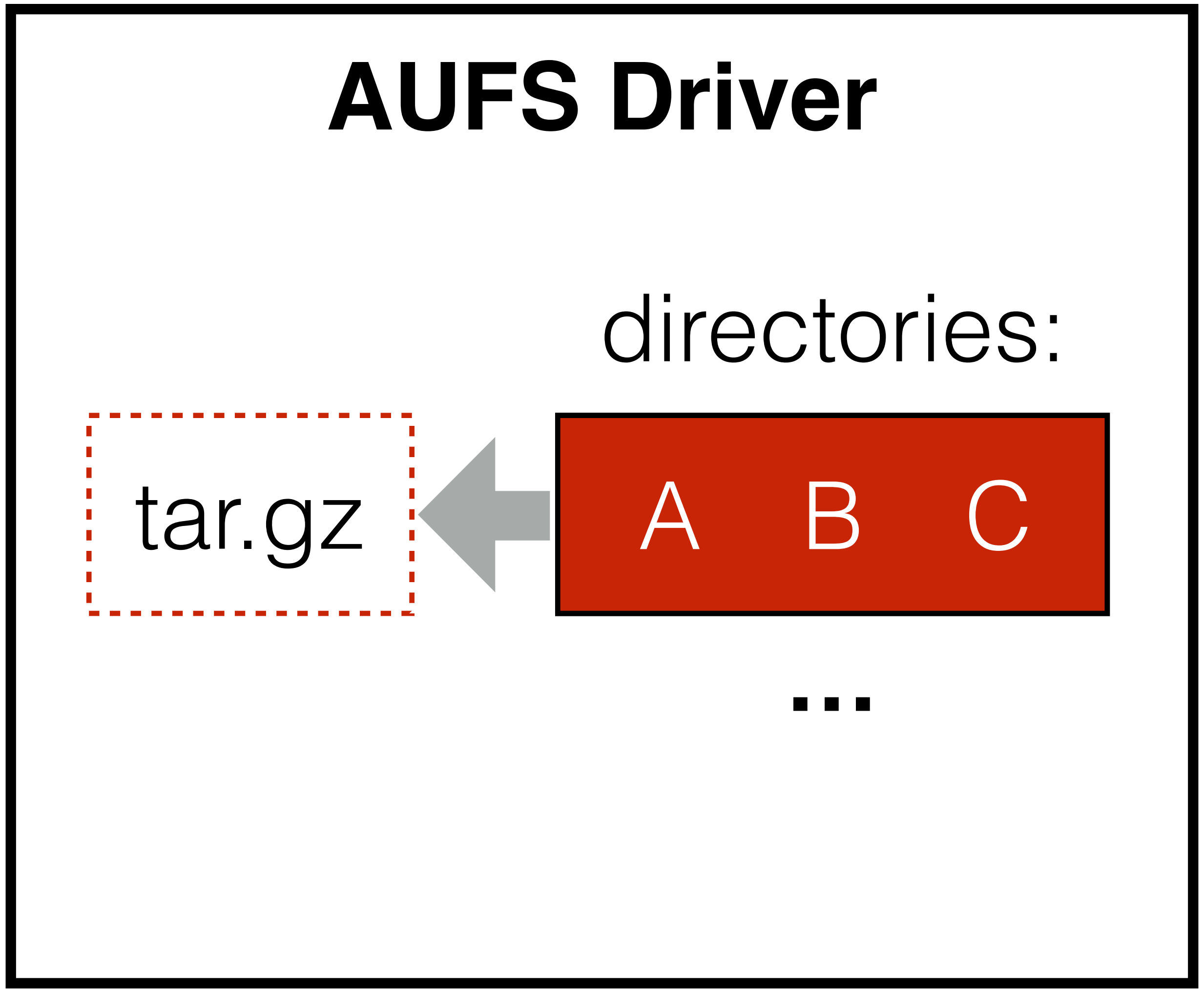


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

PUSH

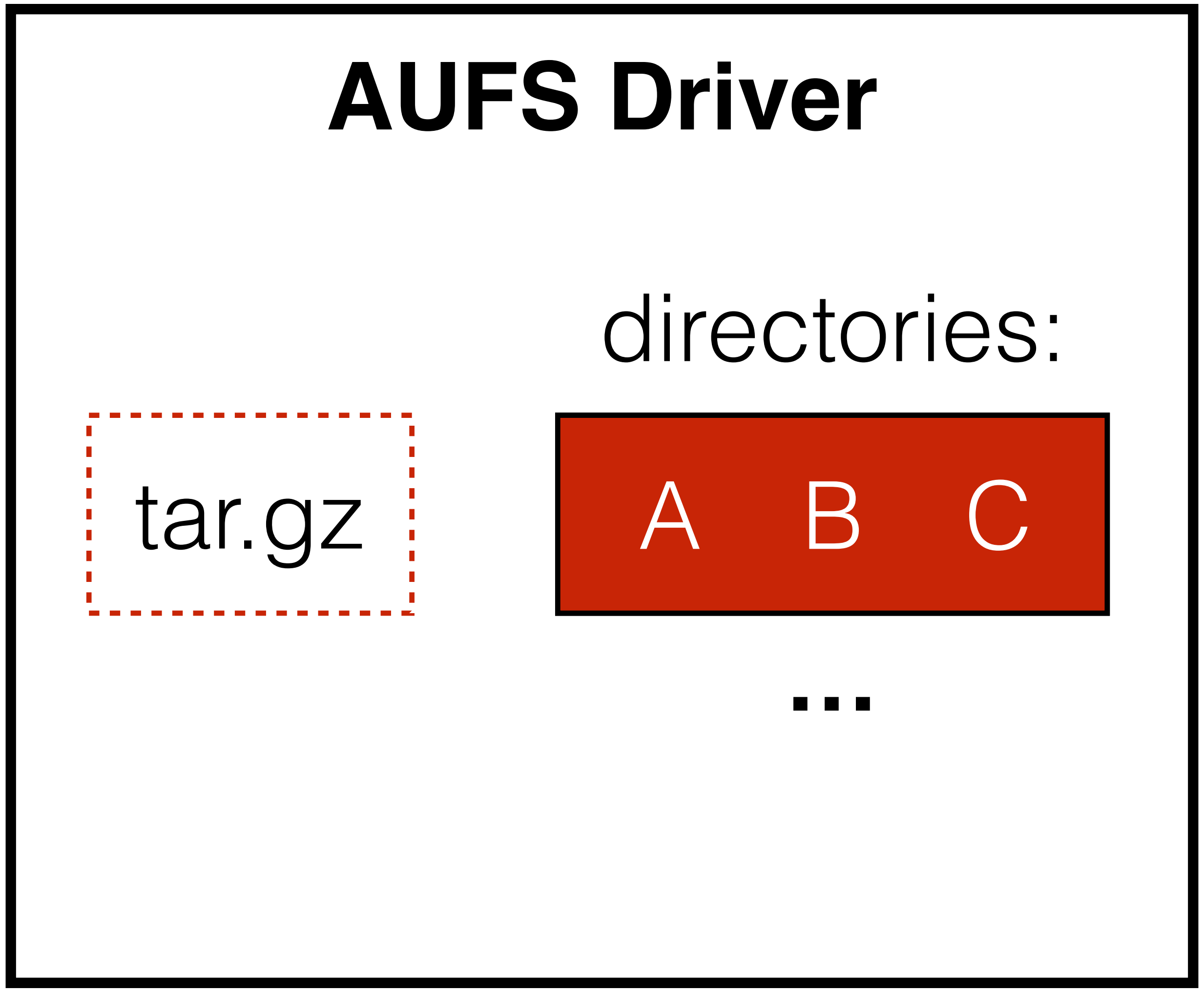


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

PUSH

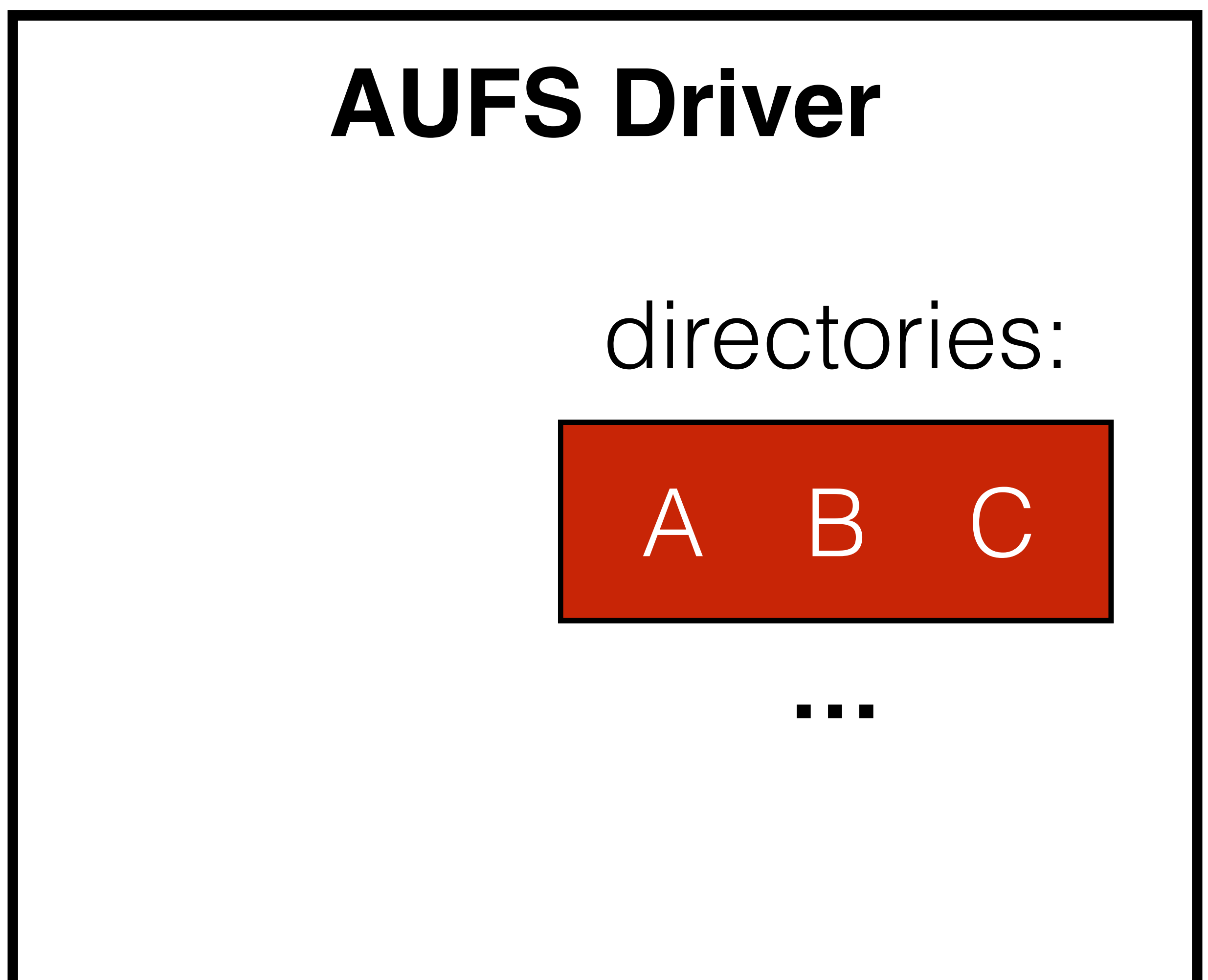


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

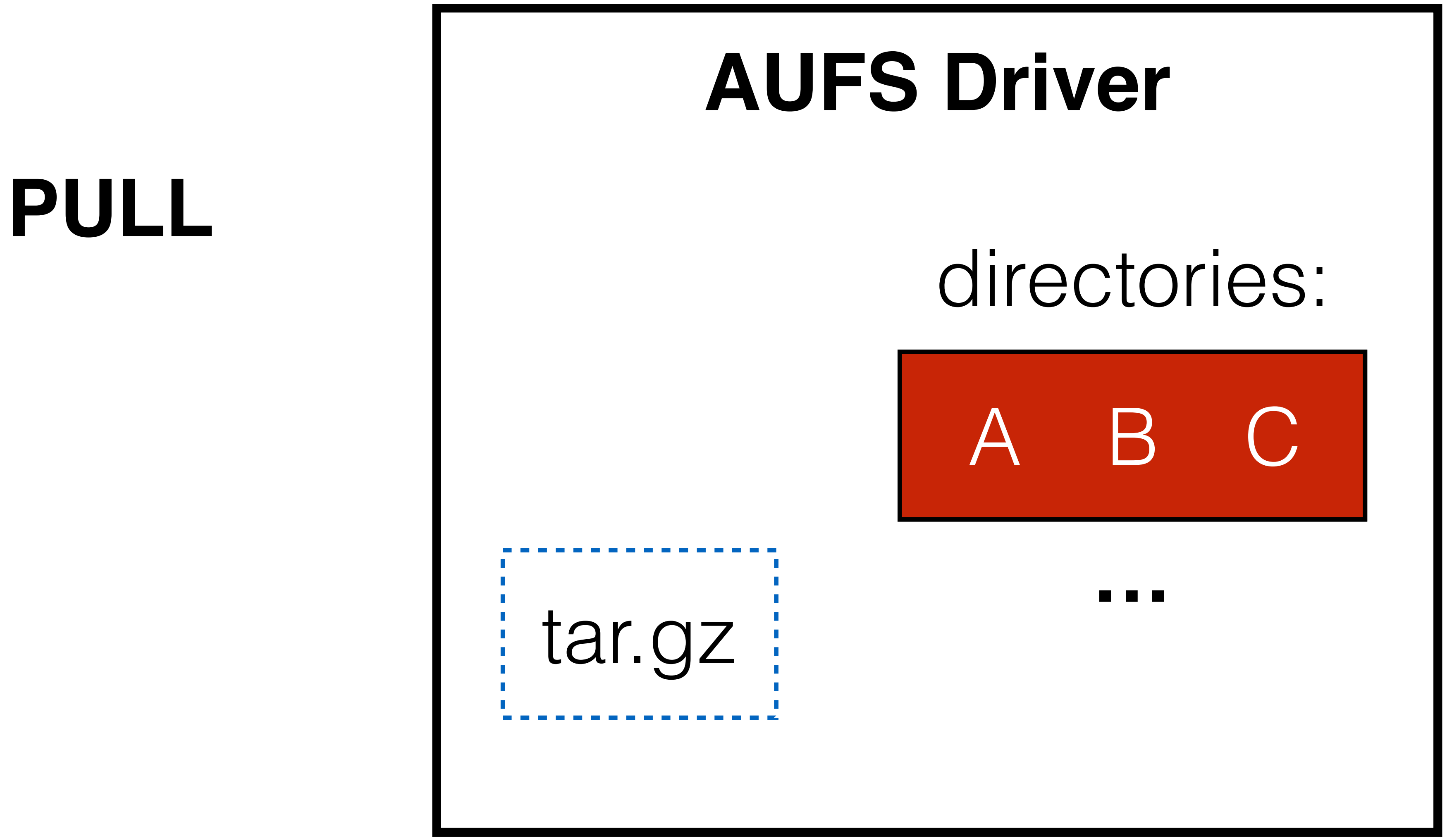
PULL



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

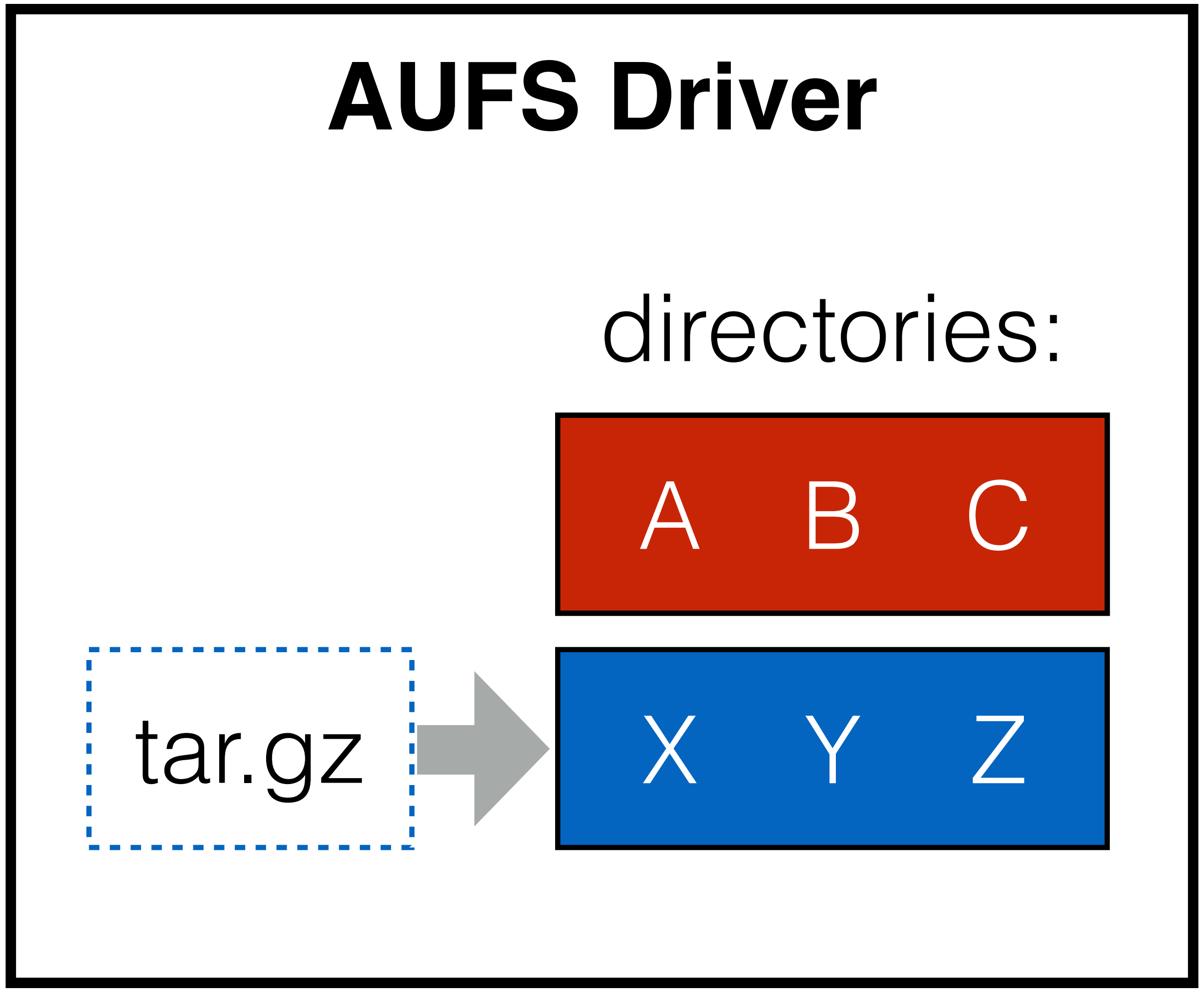


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

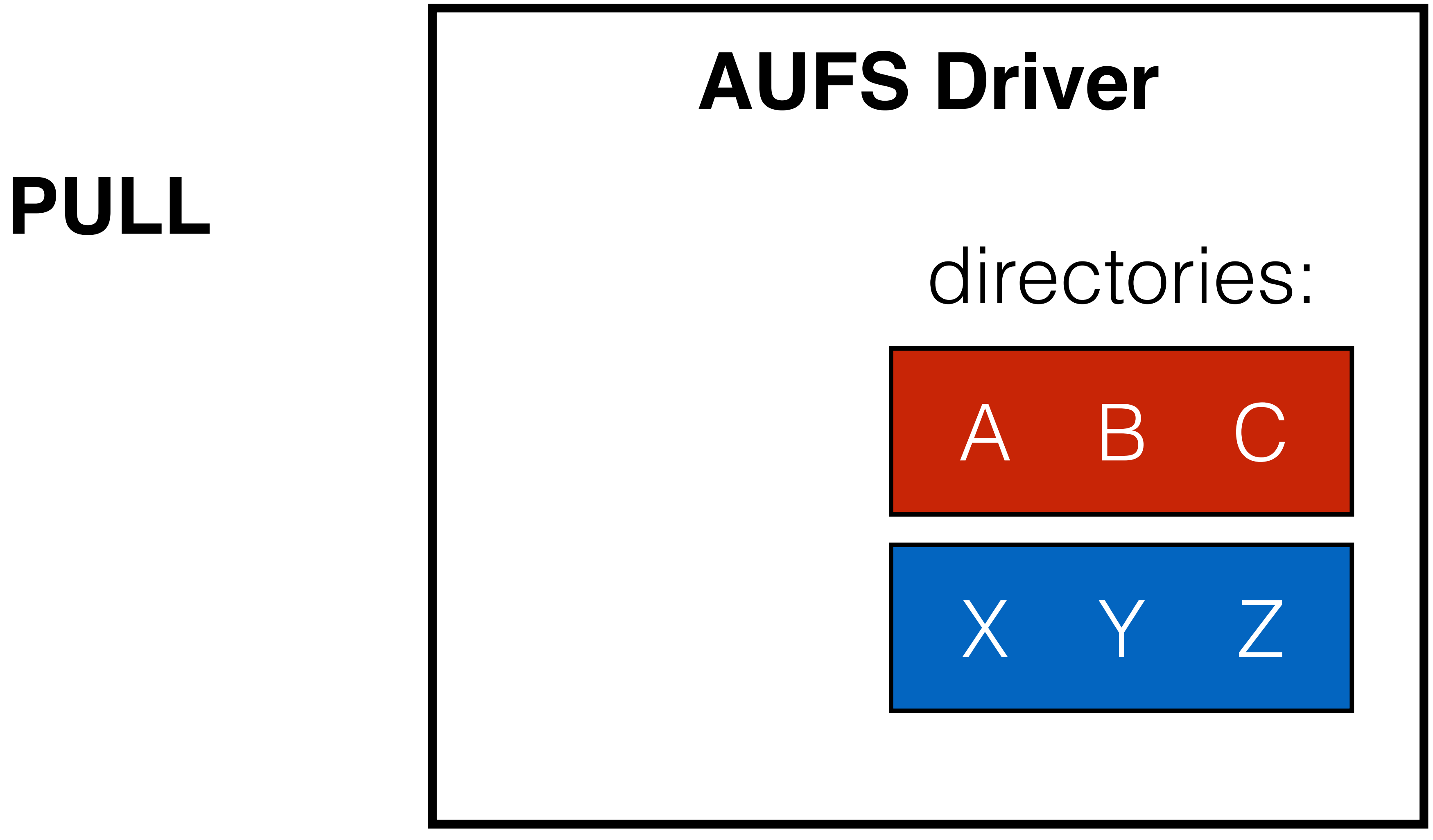
PULL



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

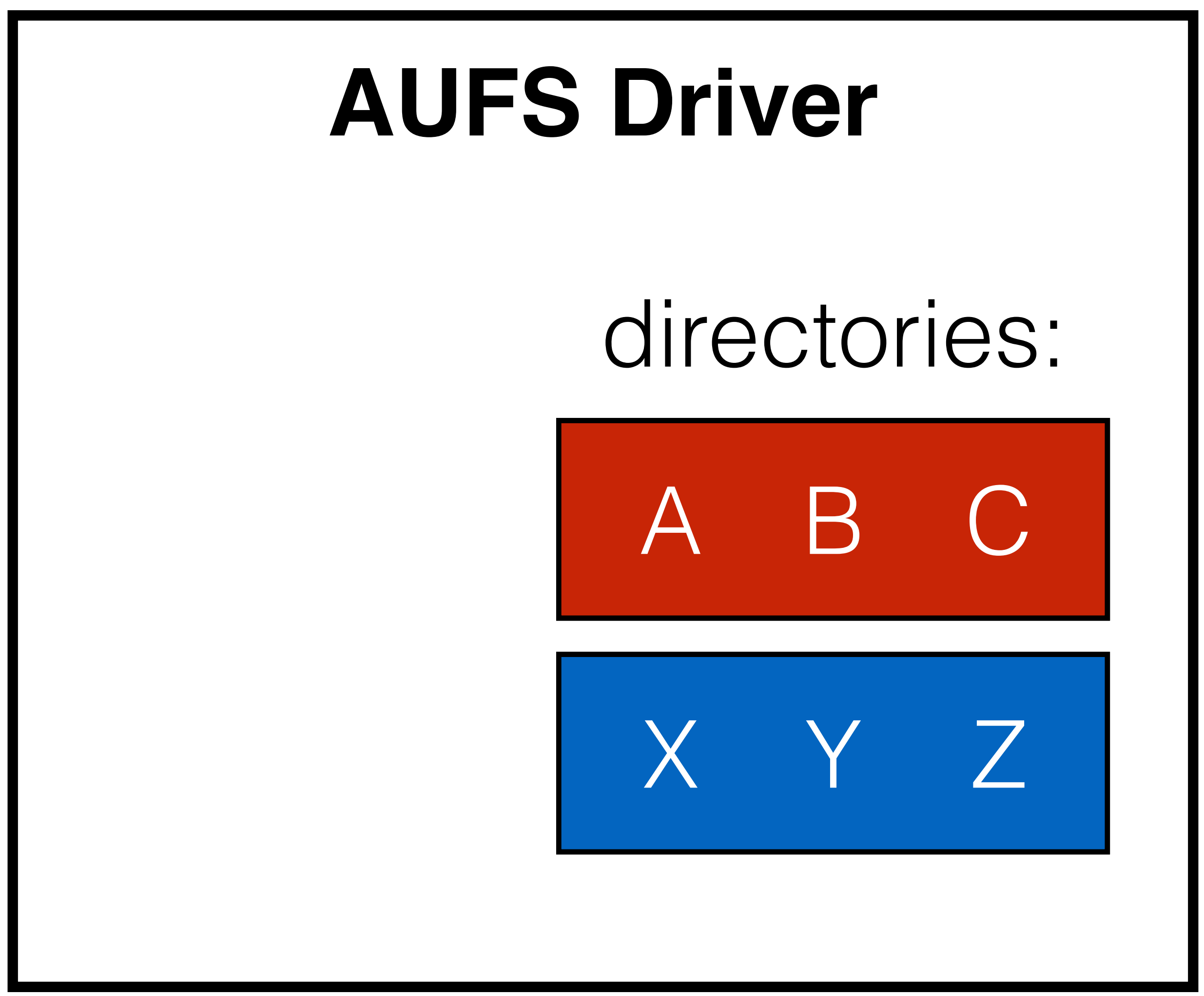


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

RUN

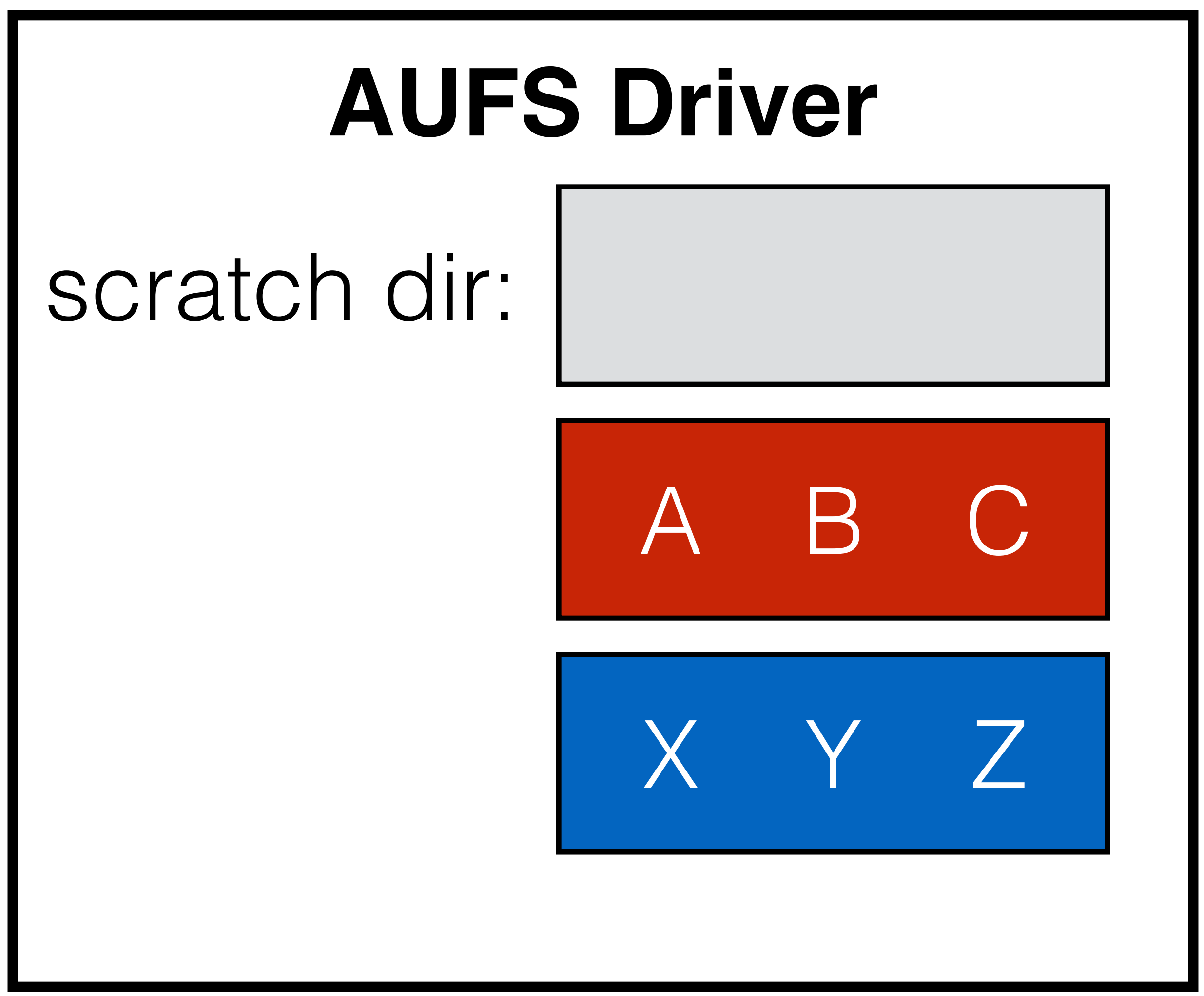


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

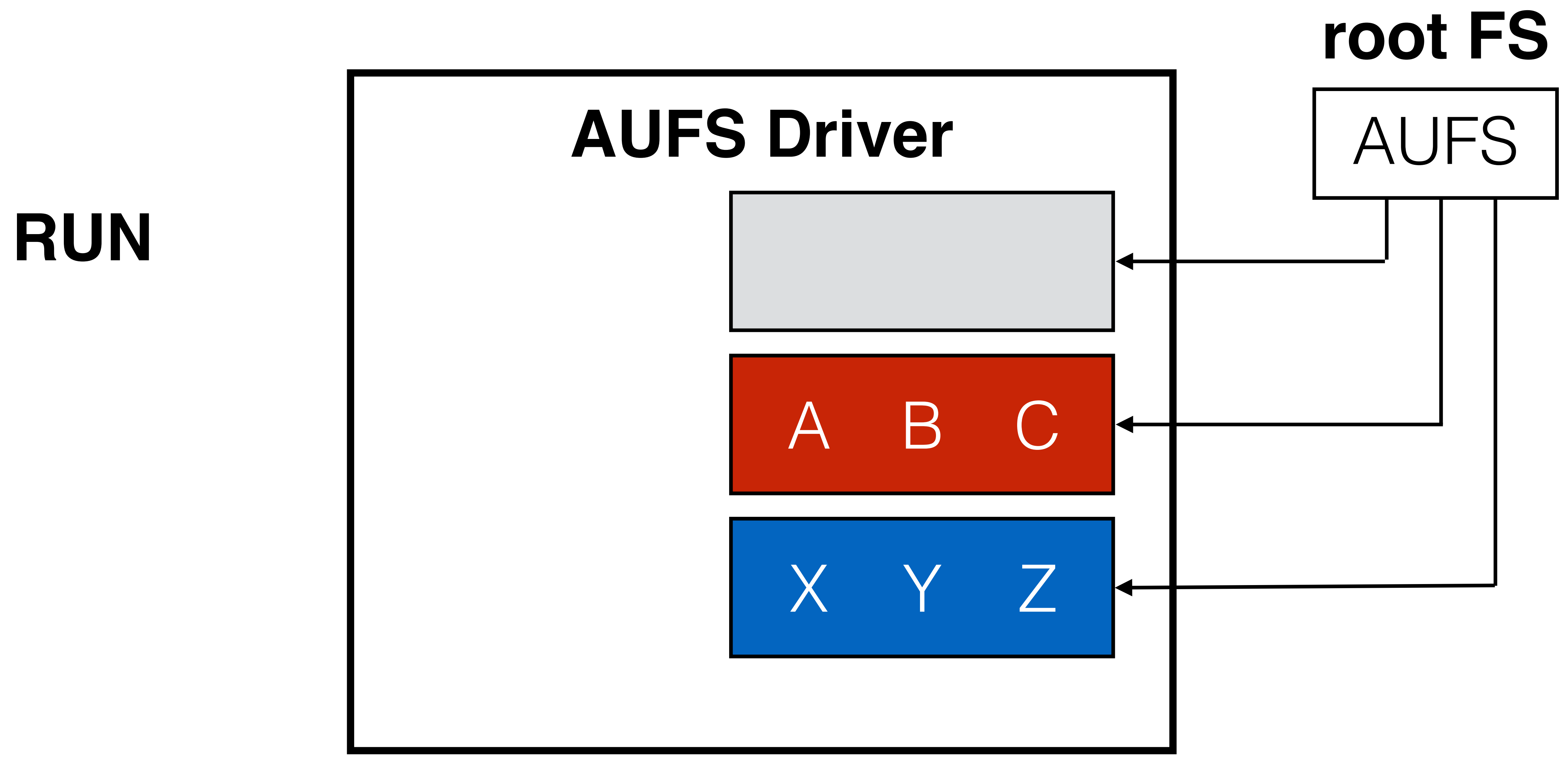
RUN



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

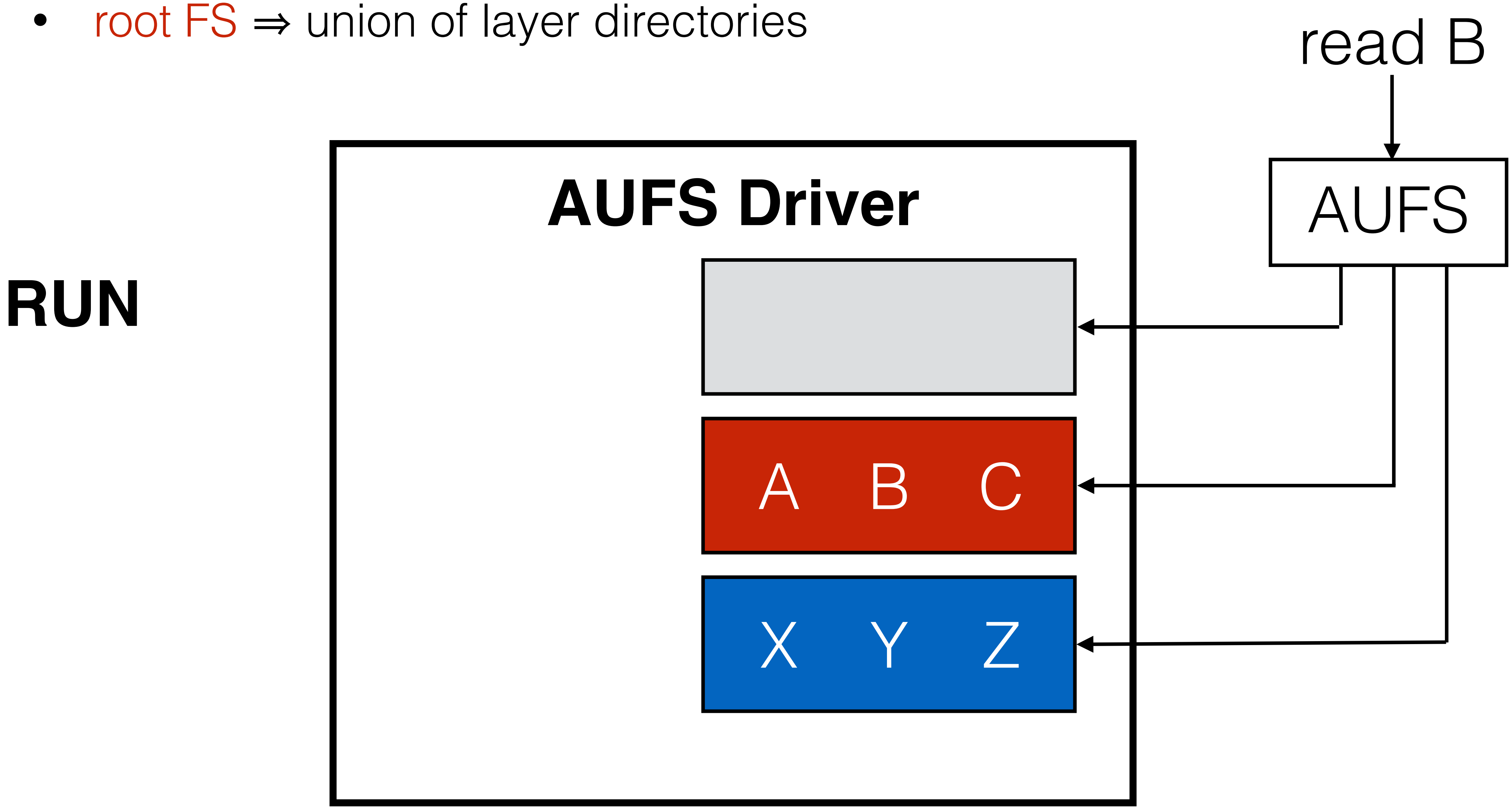
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

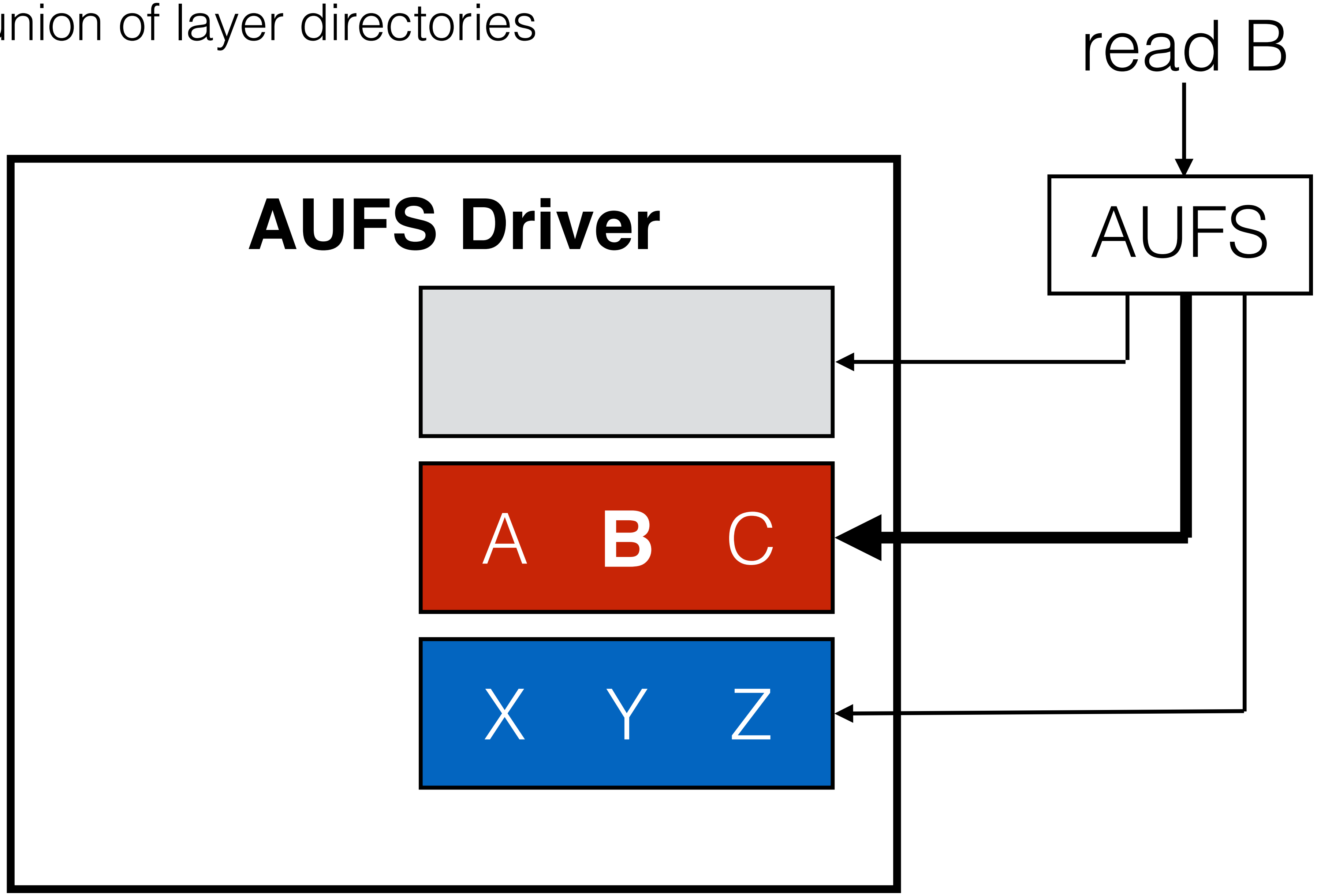


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

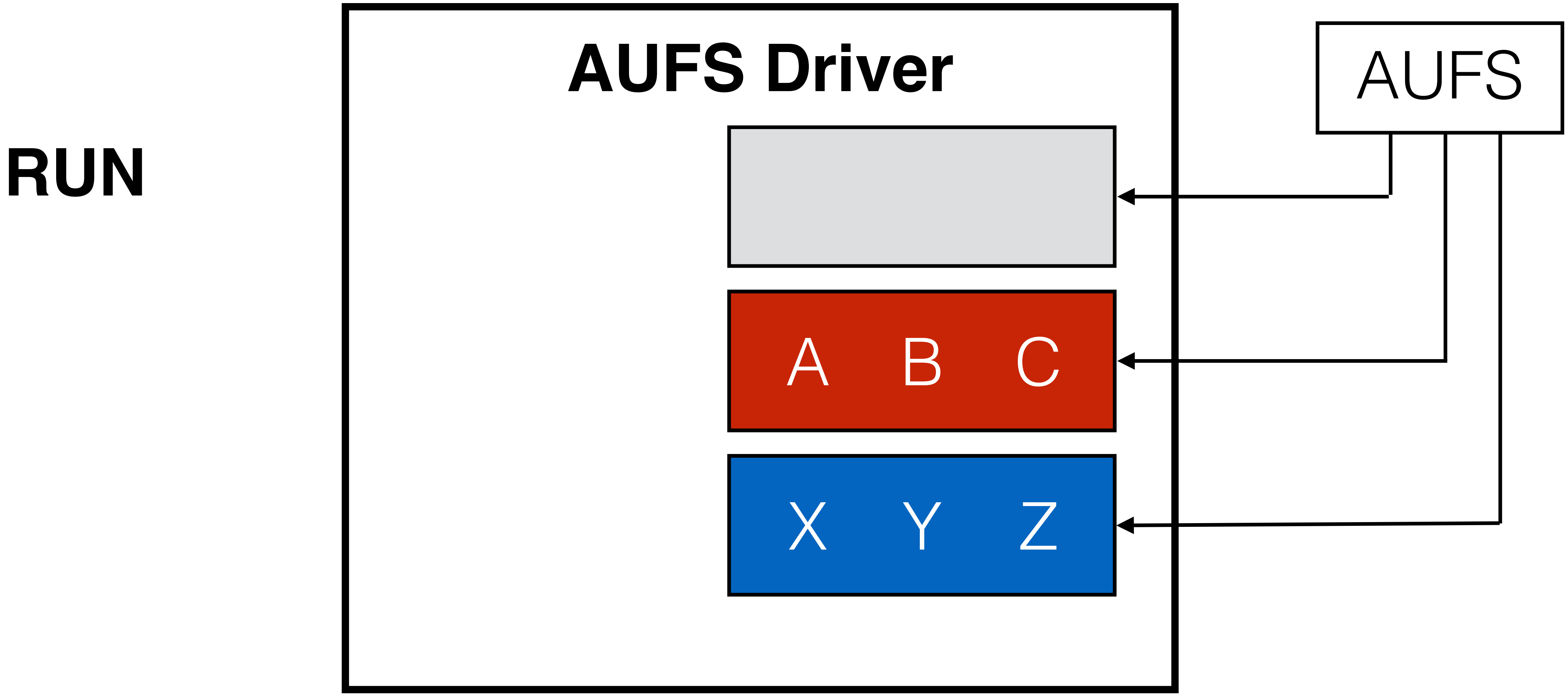
RUN



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

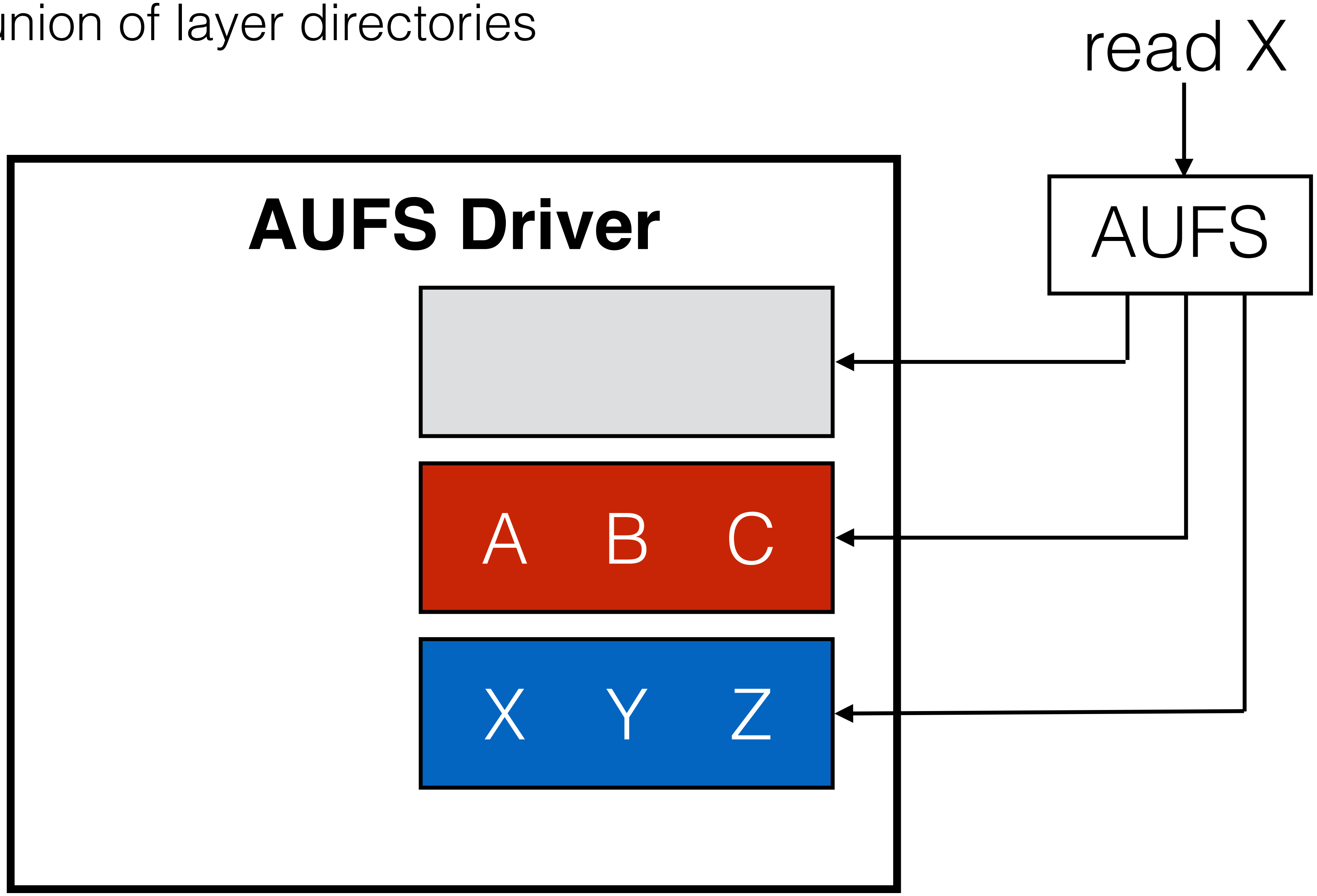


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

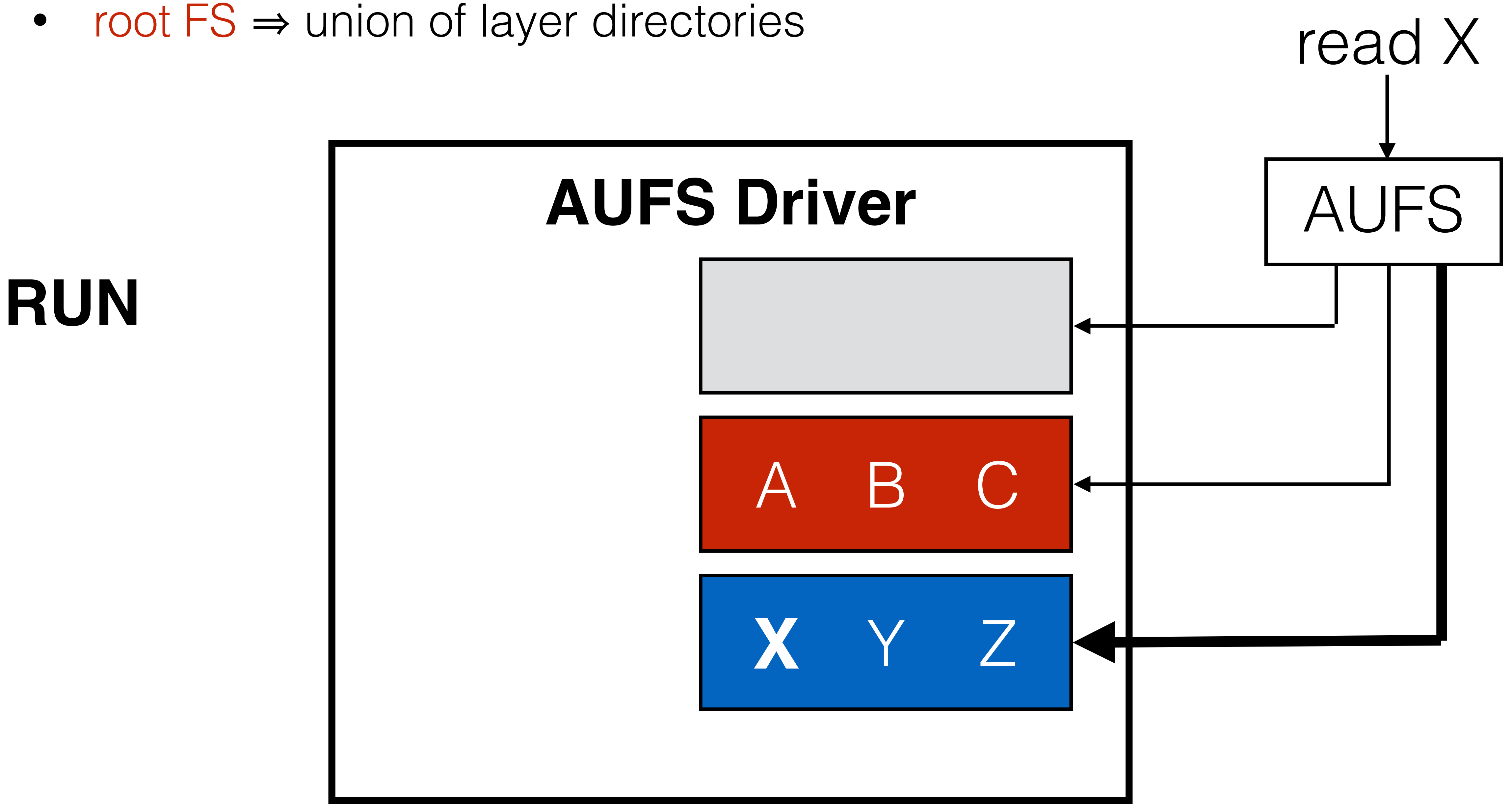
RUN



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

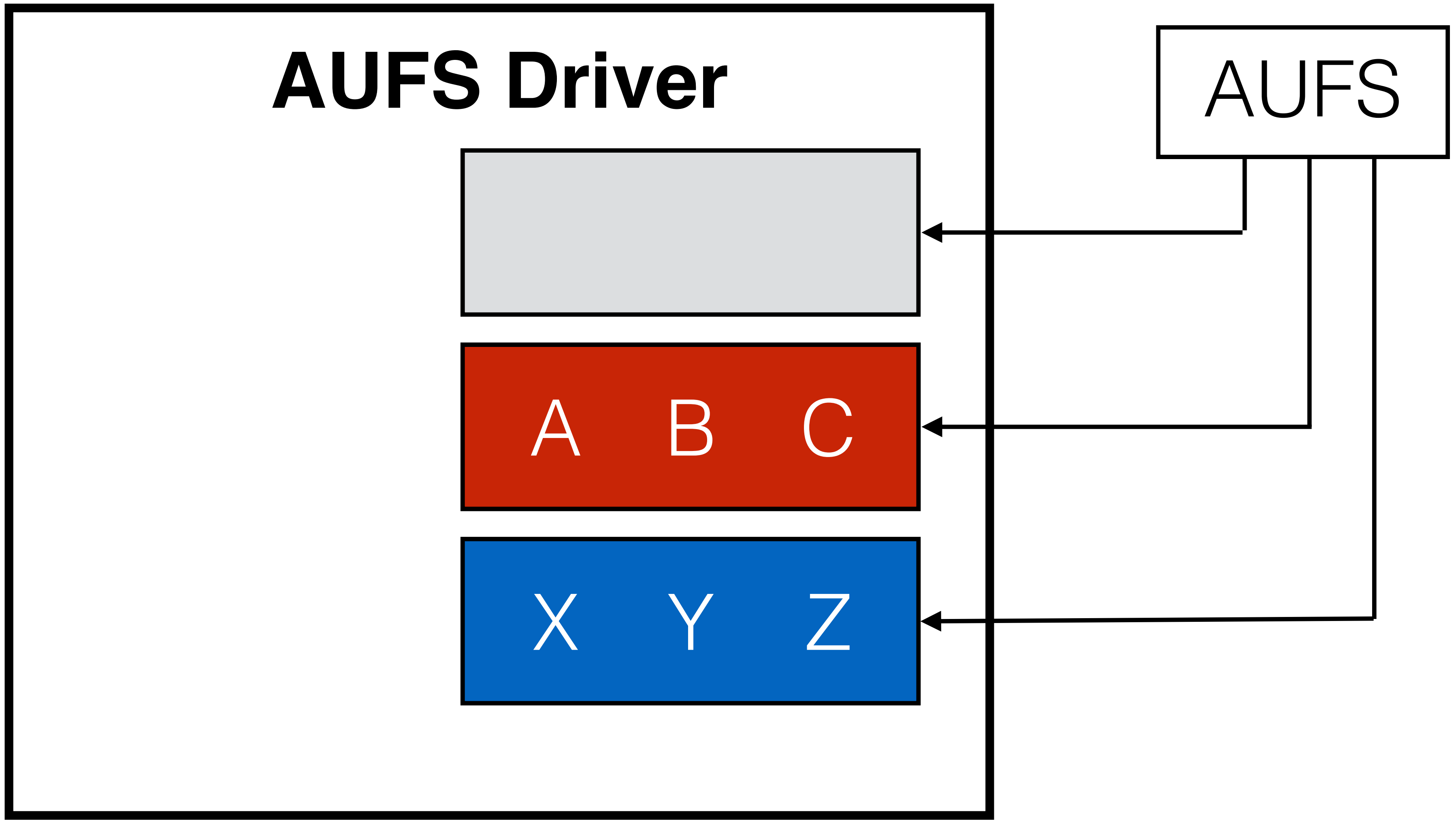


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

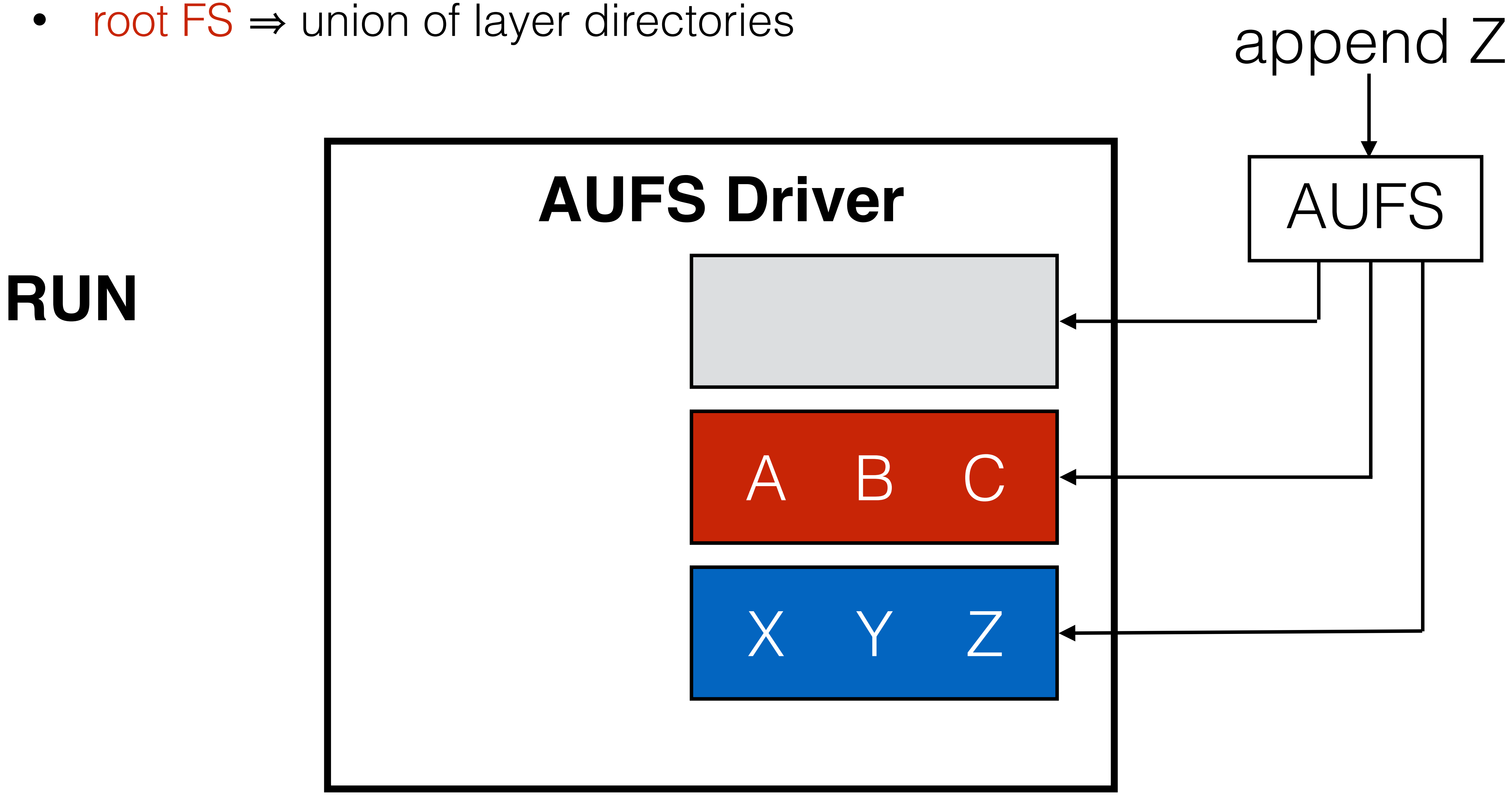
RUN



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

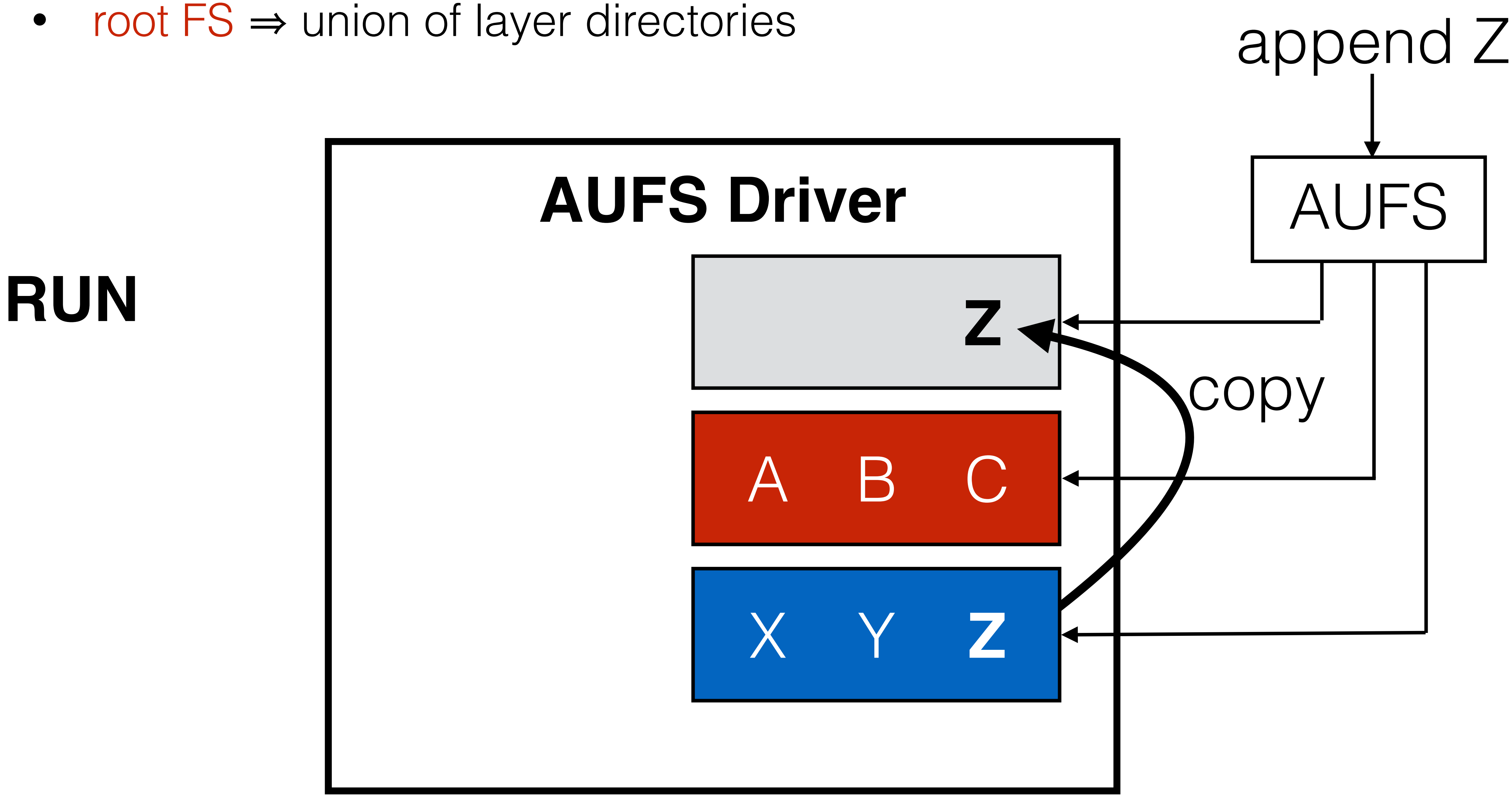
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

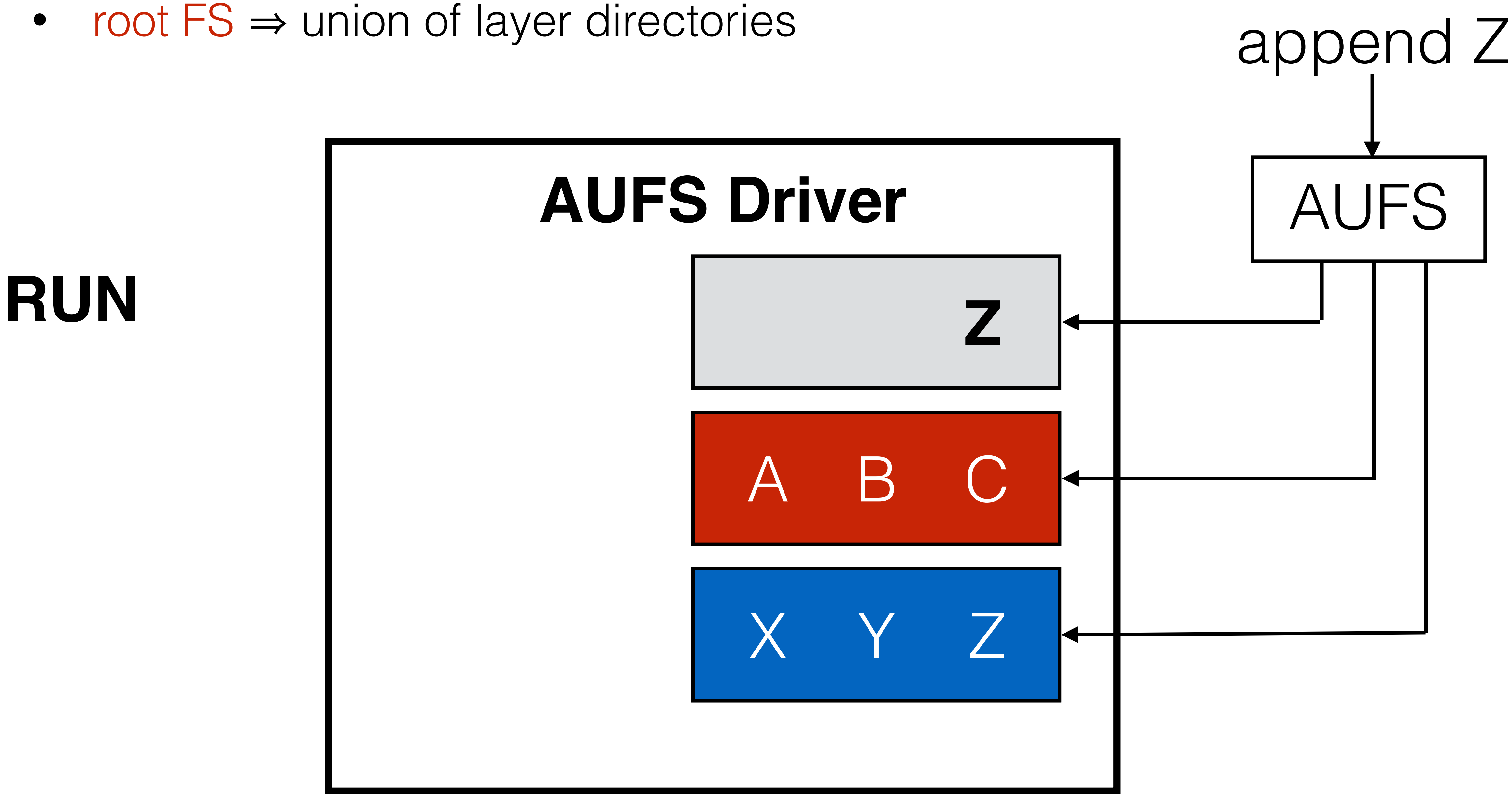
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

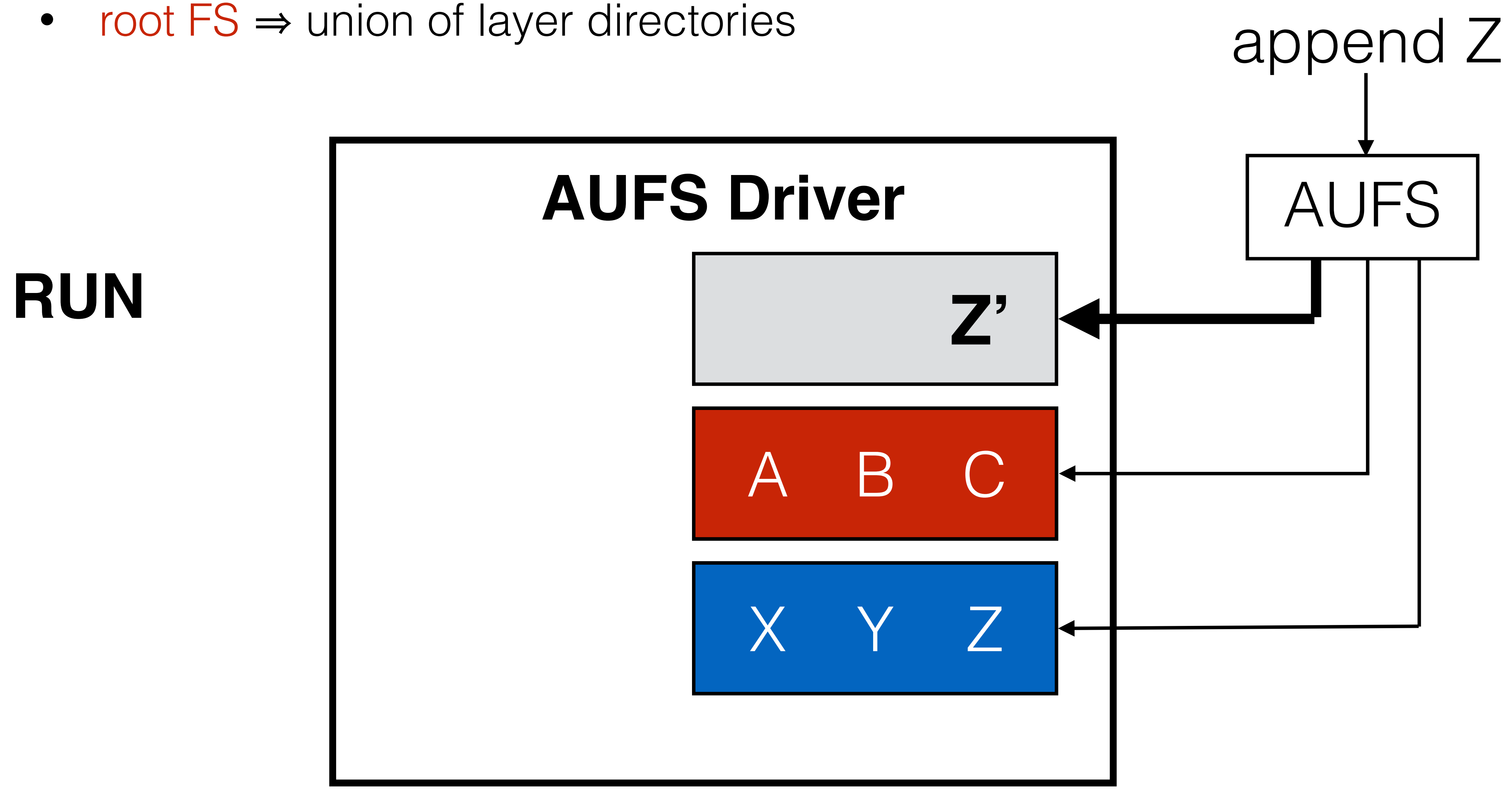
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

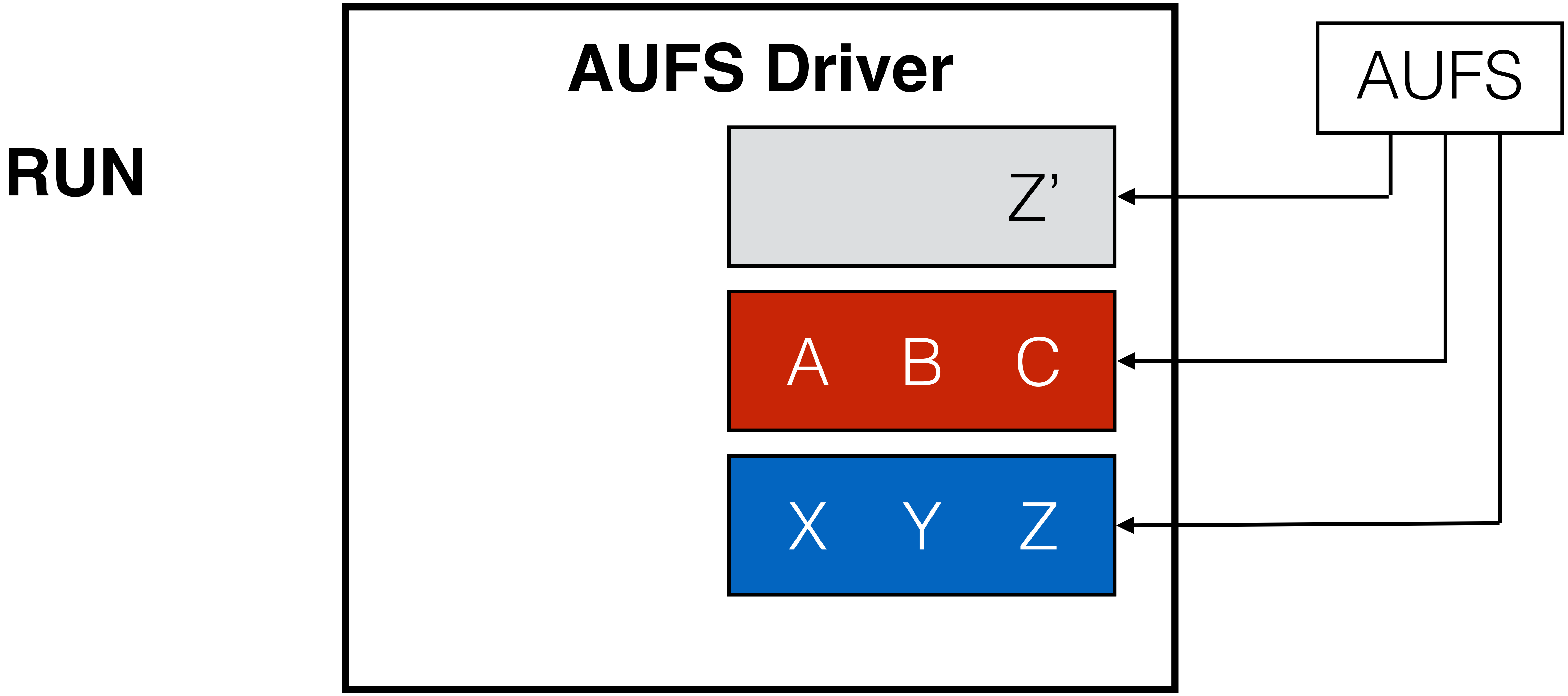
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



Slacker Outline

Background

Container Workloads

Default Driver: AUFS

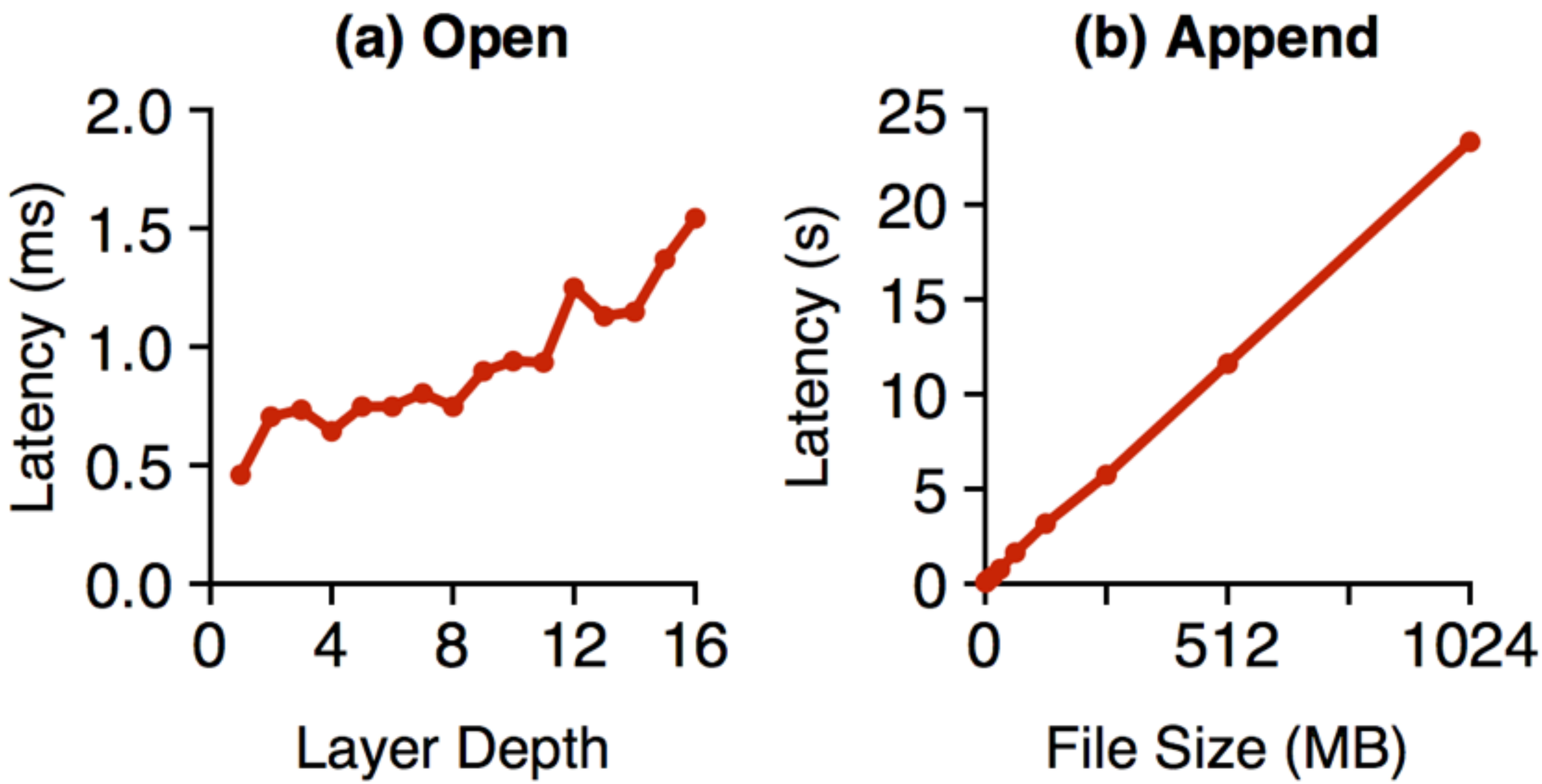
- Design
- Performance

Our Driver: Slacker

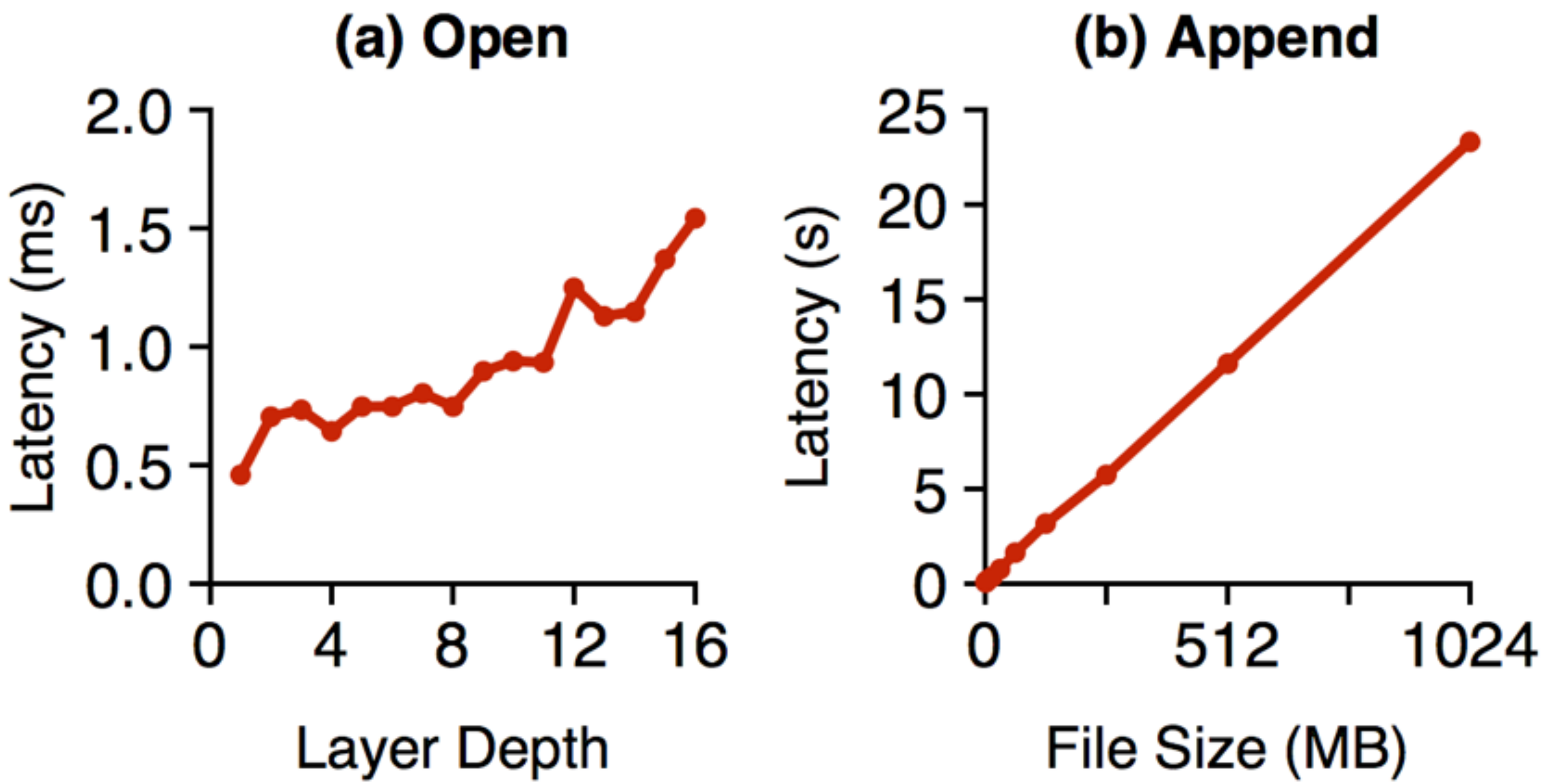
Evaluation

Conclusion

AUFS File System

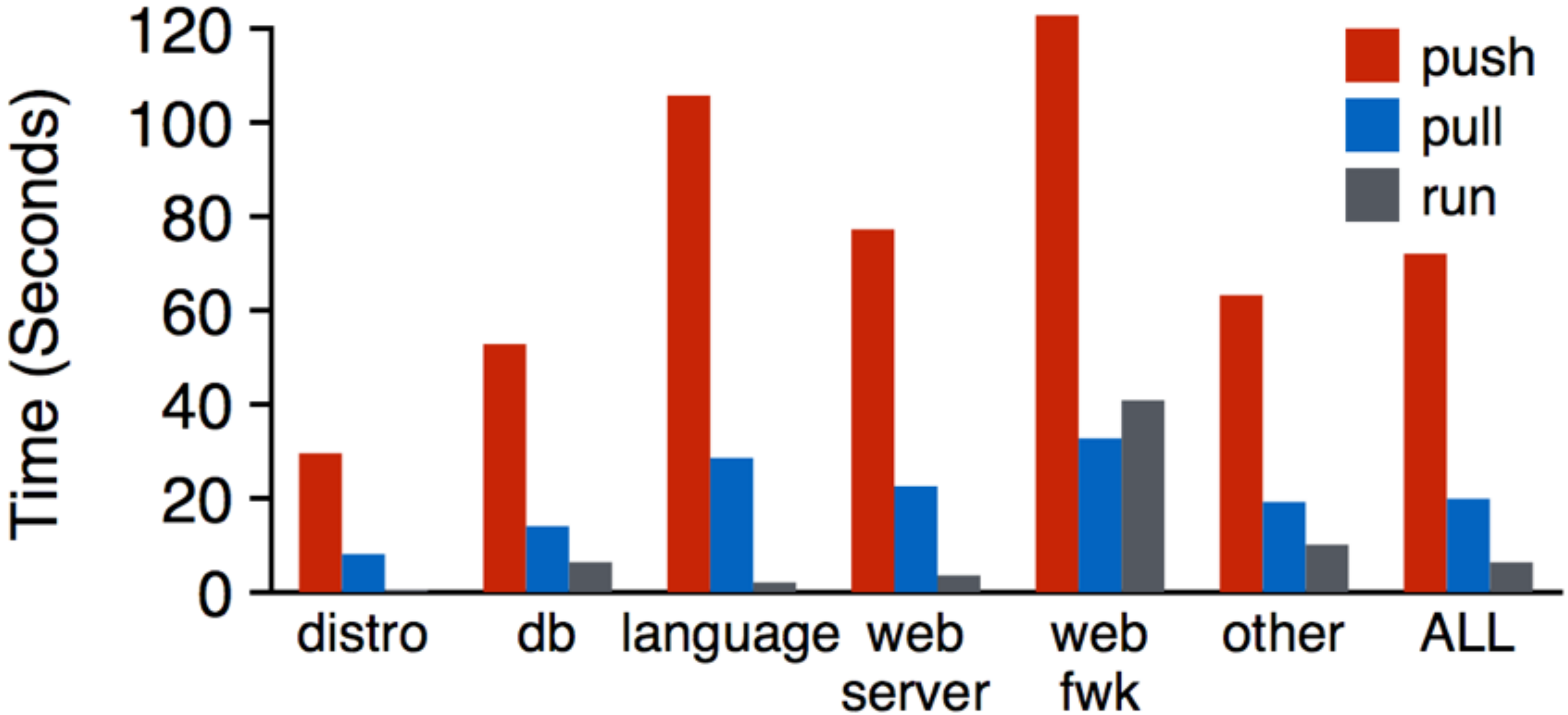


AUFS File System

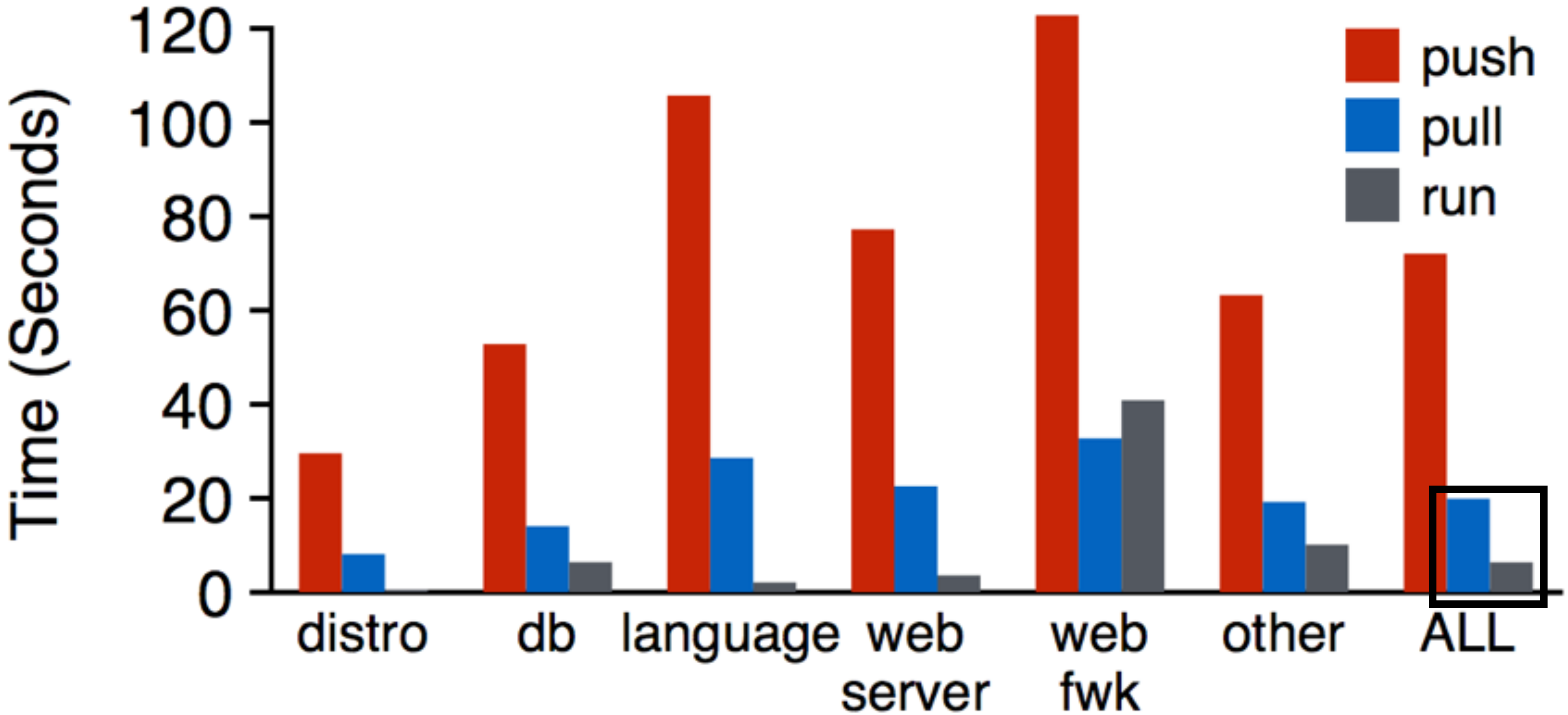


Deep data is slow

AUFS Storage Driver



AUFS Storage Driver



76% of deployment cycle spent on pull

Slacker Outline

Background

Container Workloads

Default Driver: AUFS

Our Driver: Slacker

Evaluation

Conclusion

Slacker Driver

Goals

- make push+pull very fast
- utilize powerful primitives of a modern storage server (Tintri VMstore)
- create drop-in replacement; don't change Docker framework itself

Design

- lazy pull
- layer flattening
- cache sharing

Slacker Driver

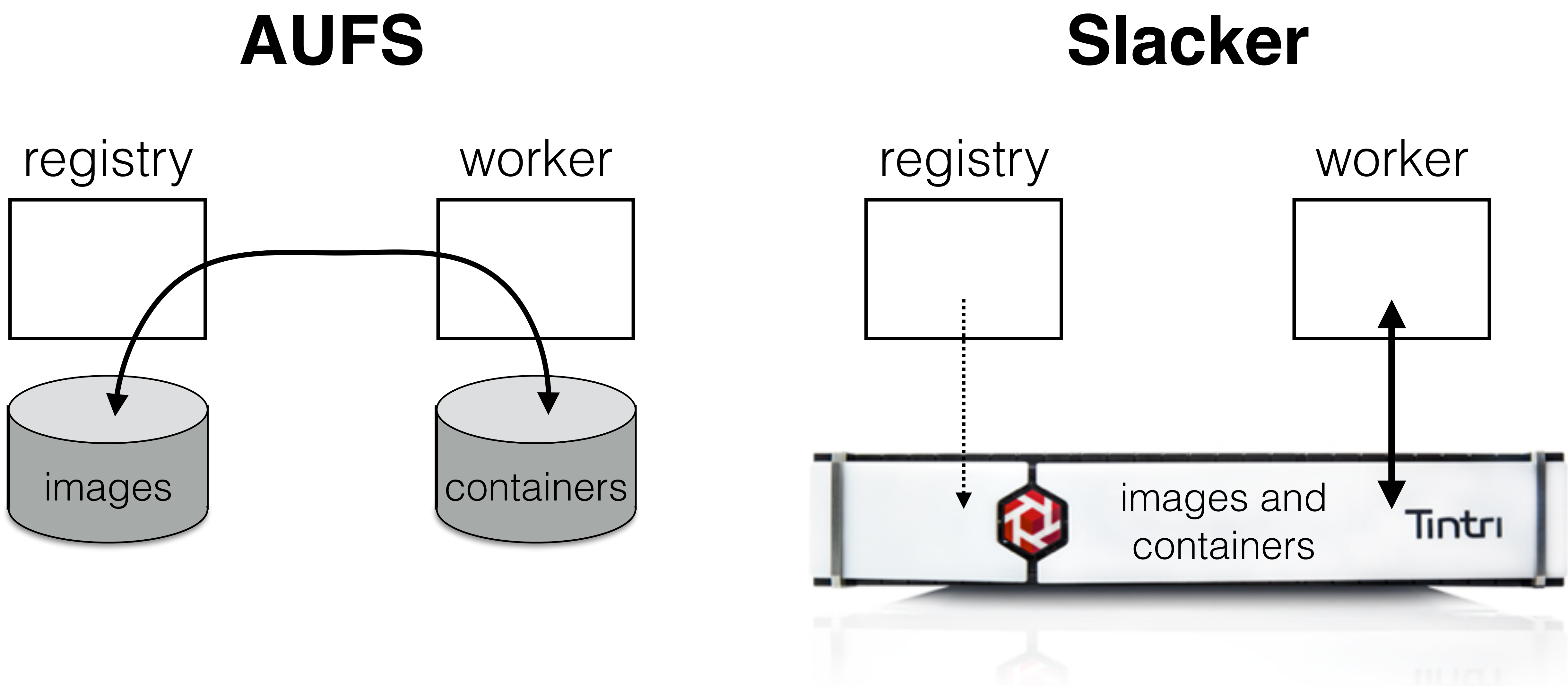
Goals

- make push+pull very fast
- utilize powerful primitives of a modern storage server (Tintri VMstore)
- create drop-in replacement; don't change Docker framework itself

Design

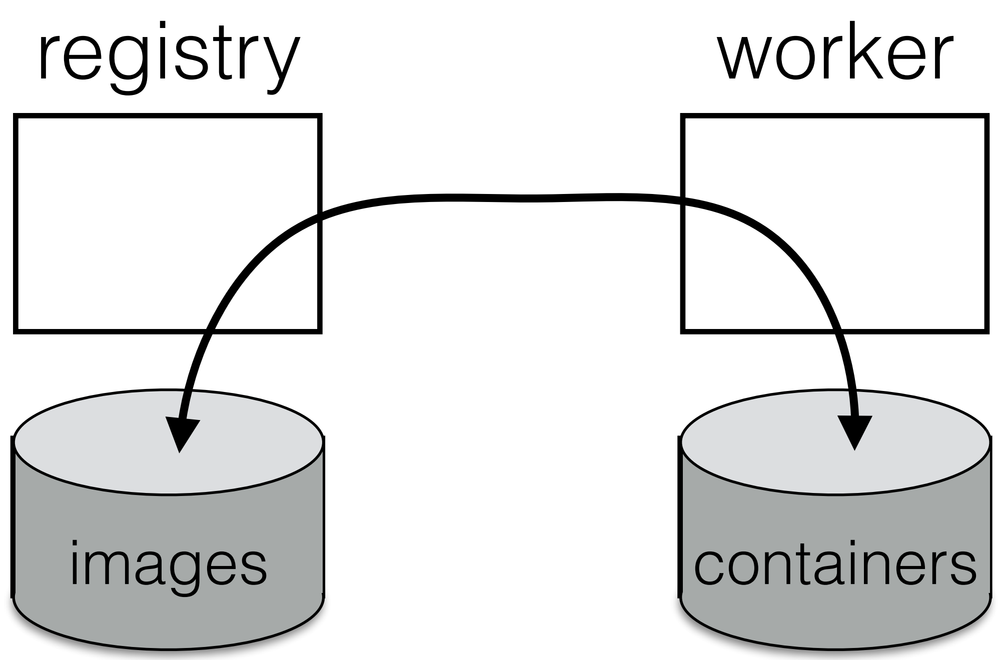
- lazy pull
- layer flattening
- cache sharing

Prefetch vs. Lazy Fetch



Prefetch vs. Lazy Fetch

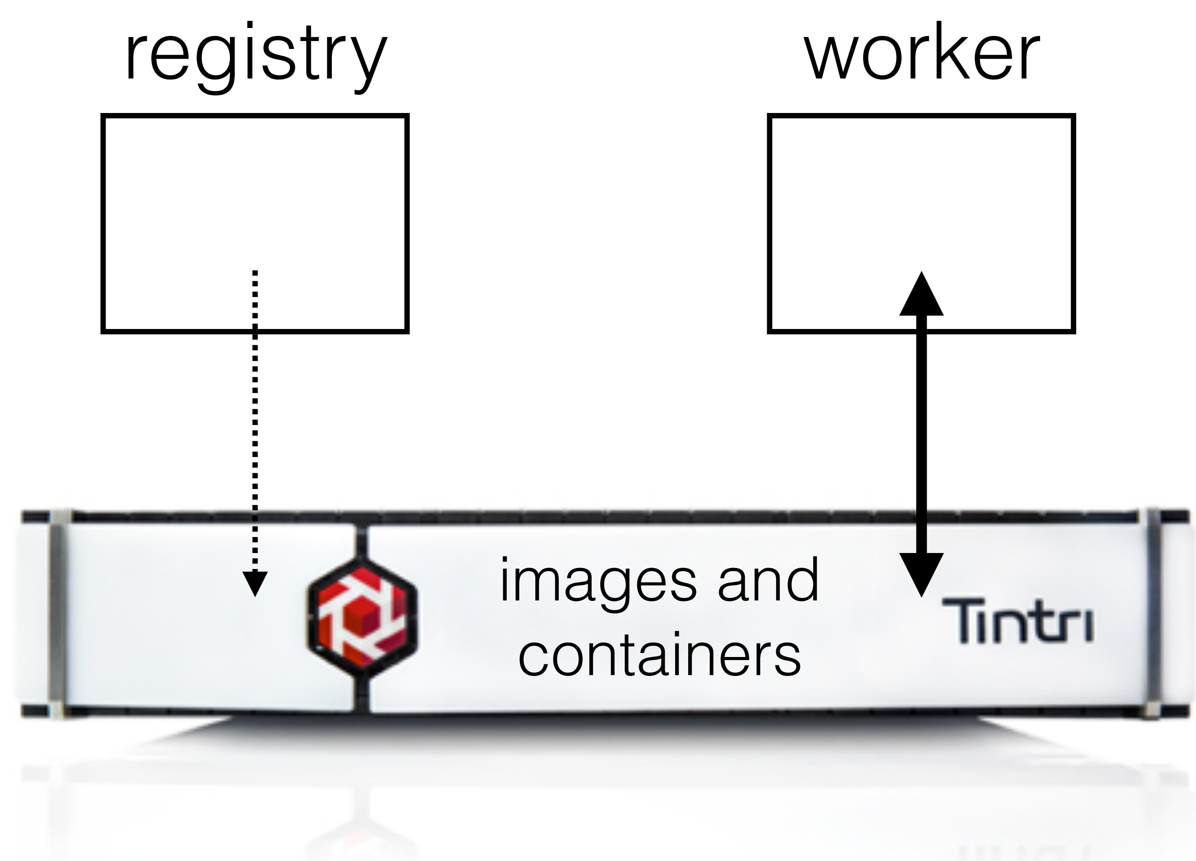
AUFS



significant copying

- over network
- to/from disk

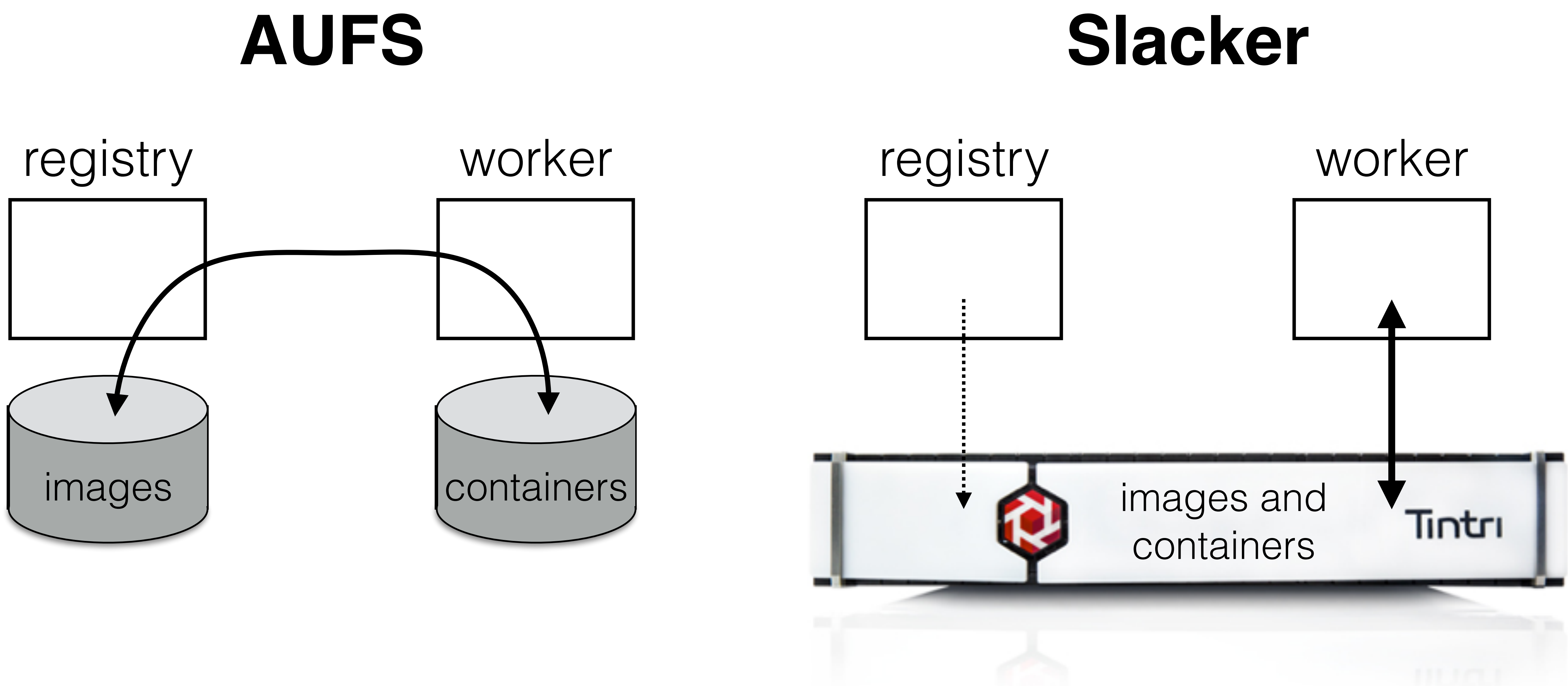
Slacker



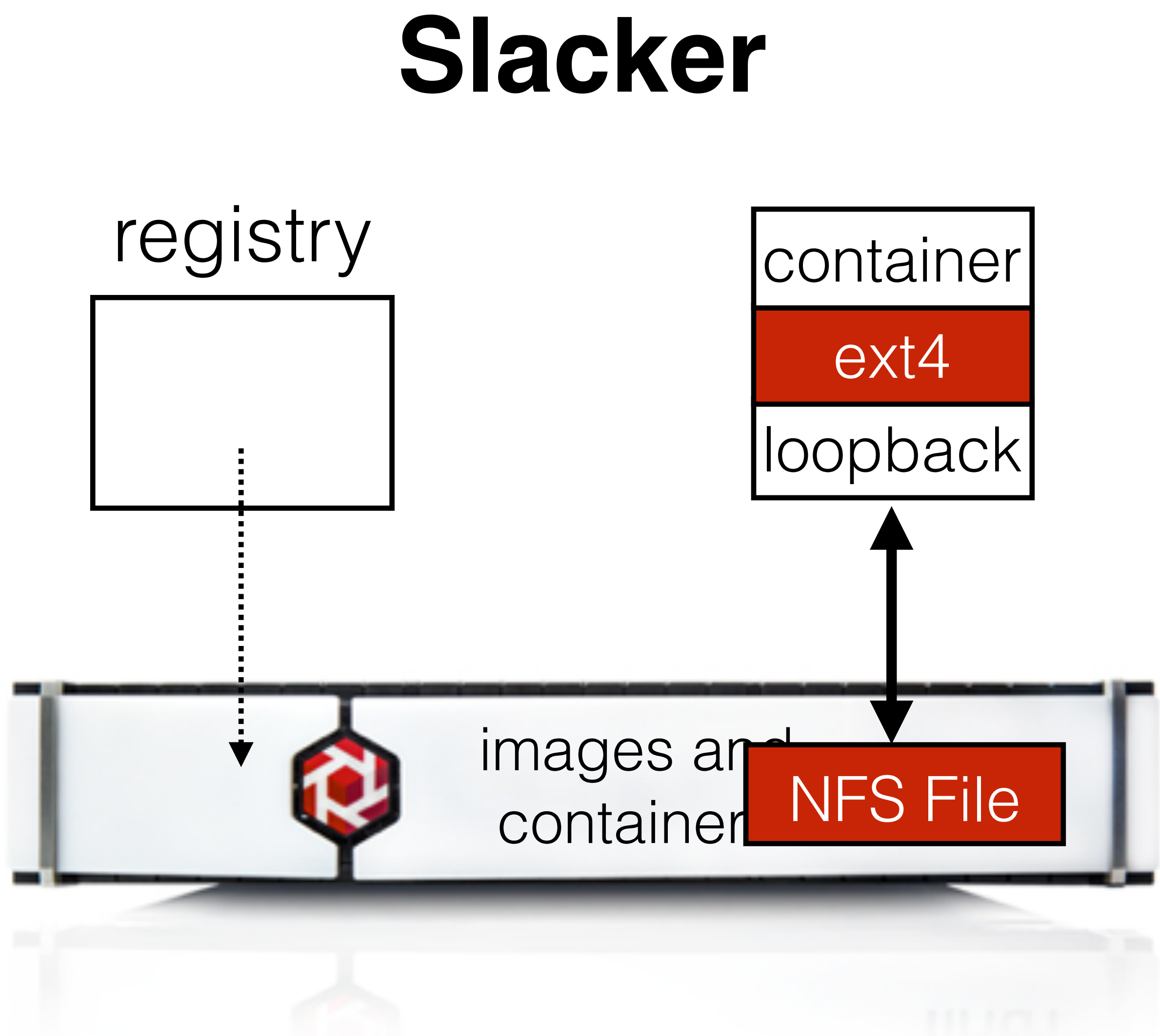
centralized storage

- easy sharing

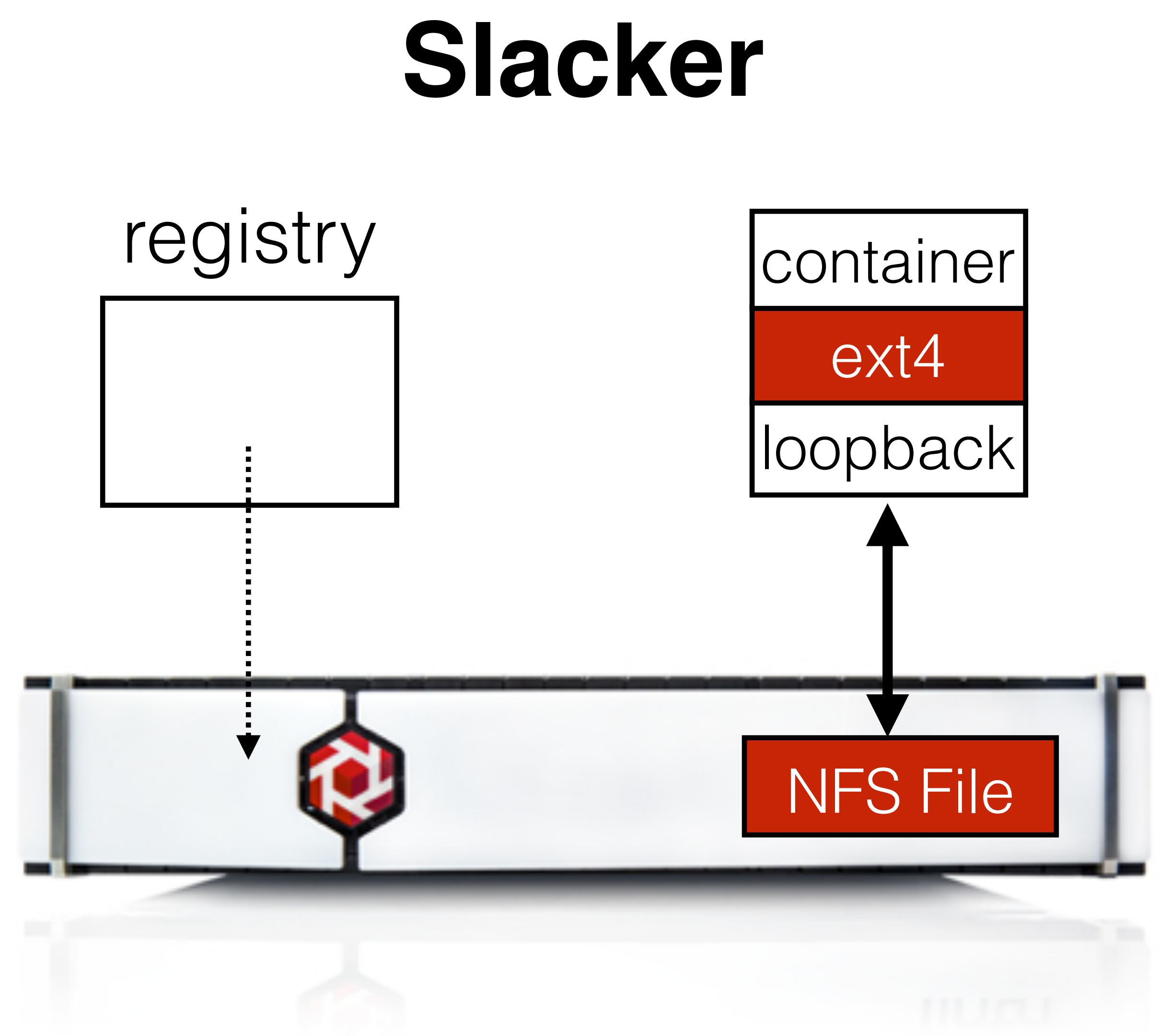
Prefetch vs. Lazy Fetch



Prefetch vs. Lazy Fetch

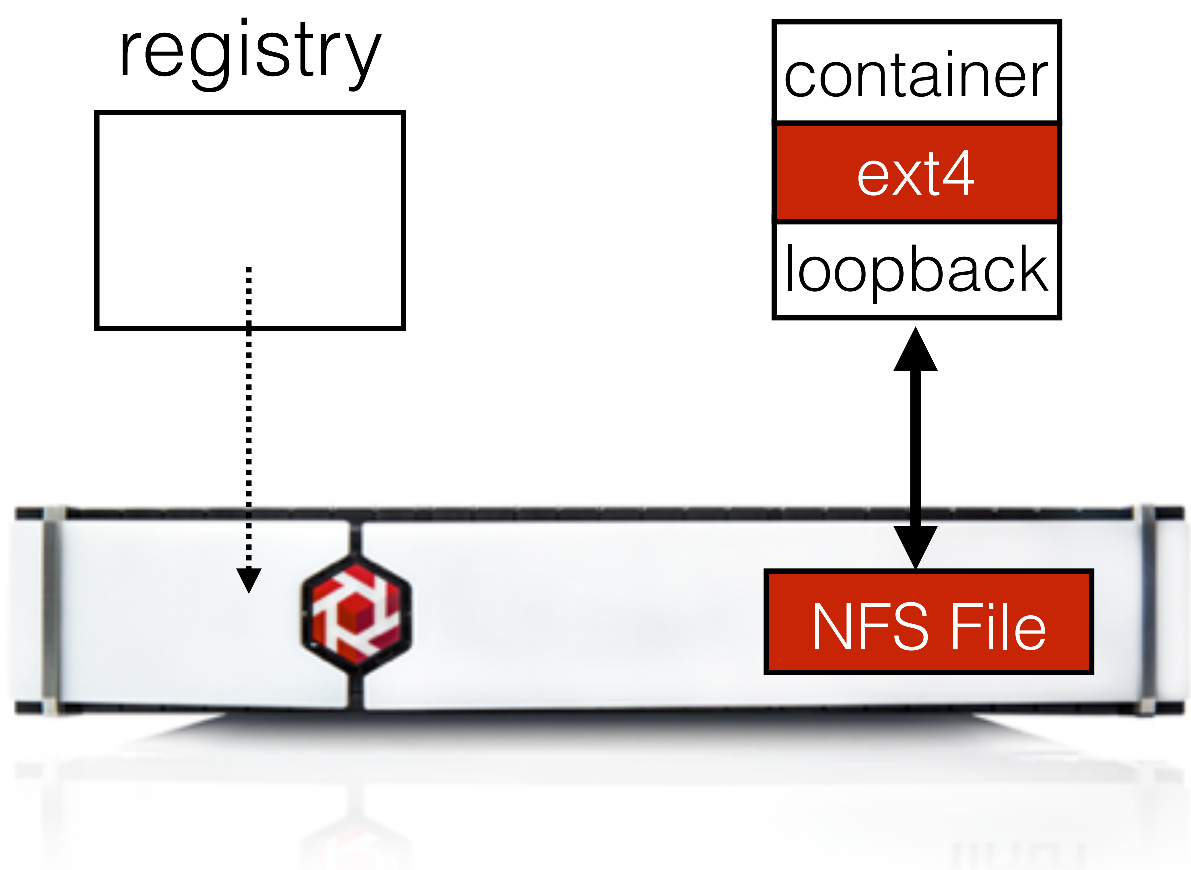


Prefetch vs. Lazy Fetch



Prefetch vs. Lazy Fetch

Slacker



VMstore abstractions…

VMstore Abstractions

Copy-on-Write

- VMstore provides `snapshot()` and `clone()`
- block granularity avoids AUFS's problems with file granularity

`snapshot(nfs_path)`

- create read-only copy of NFS file
- return snapshot ID

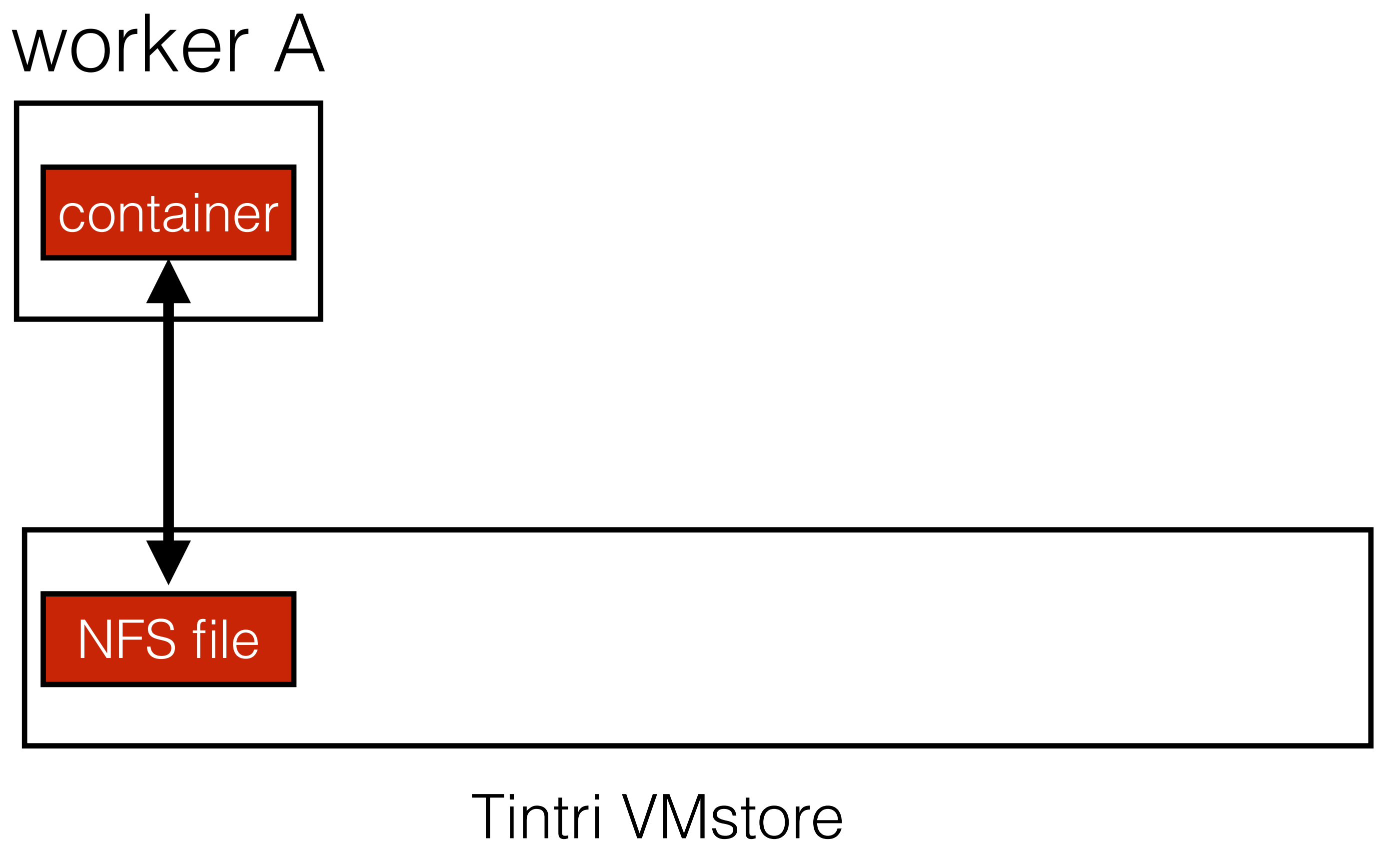
`clone(snapshot_id)`

- create r/w NFS file from snapshot

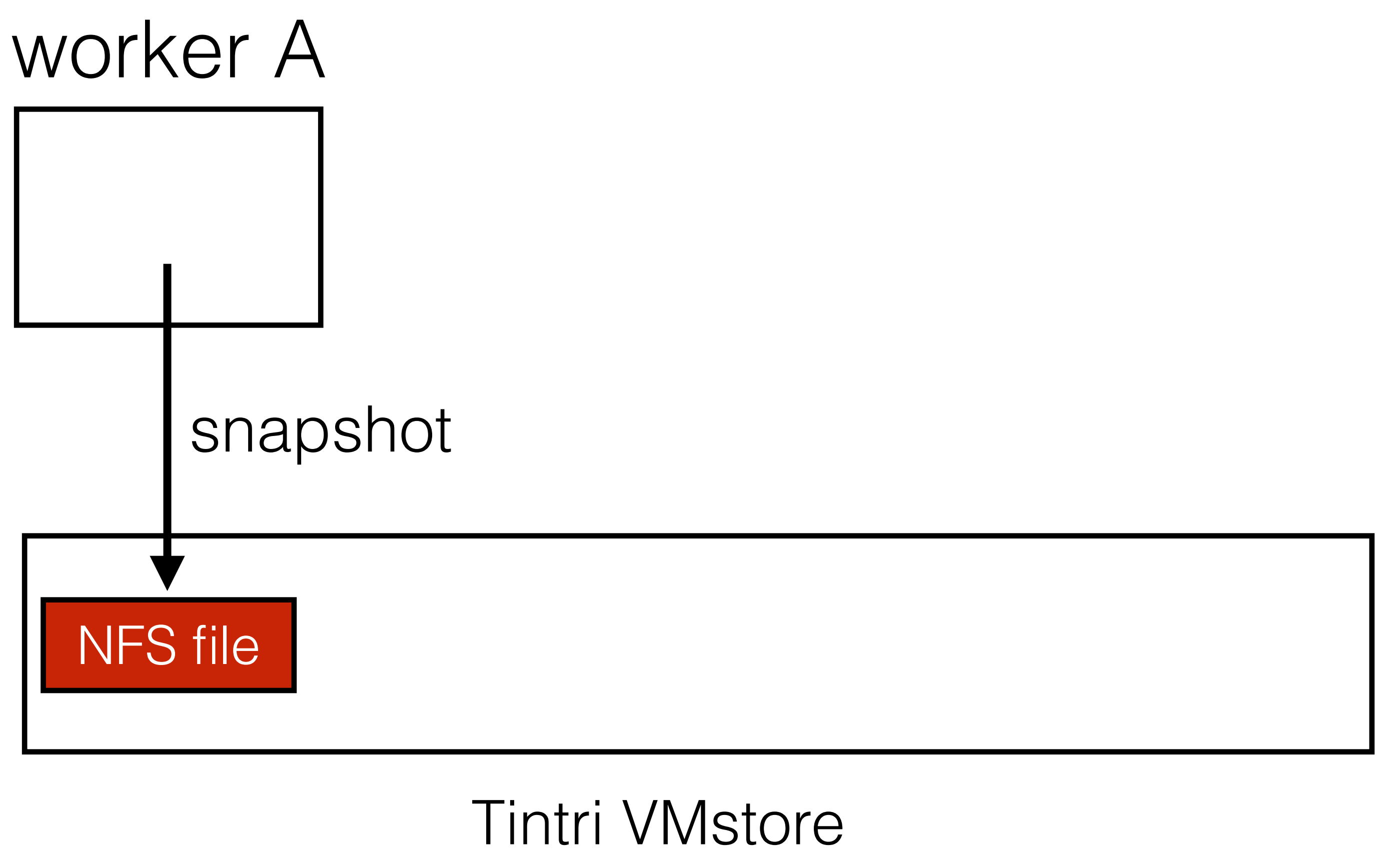
Slacker Usage

- NFS files ⇒ container storage
- snapshots ⇒ image storage
- `clone()` ⇒ provision container from image
- `snapshot()` ⇒ create image from container

Snapshot and Clone

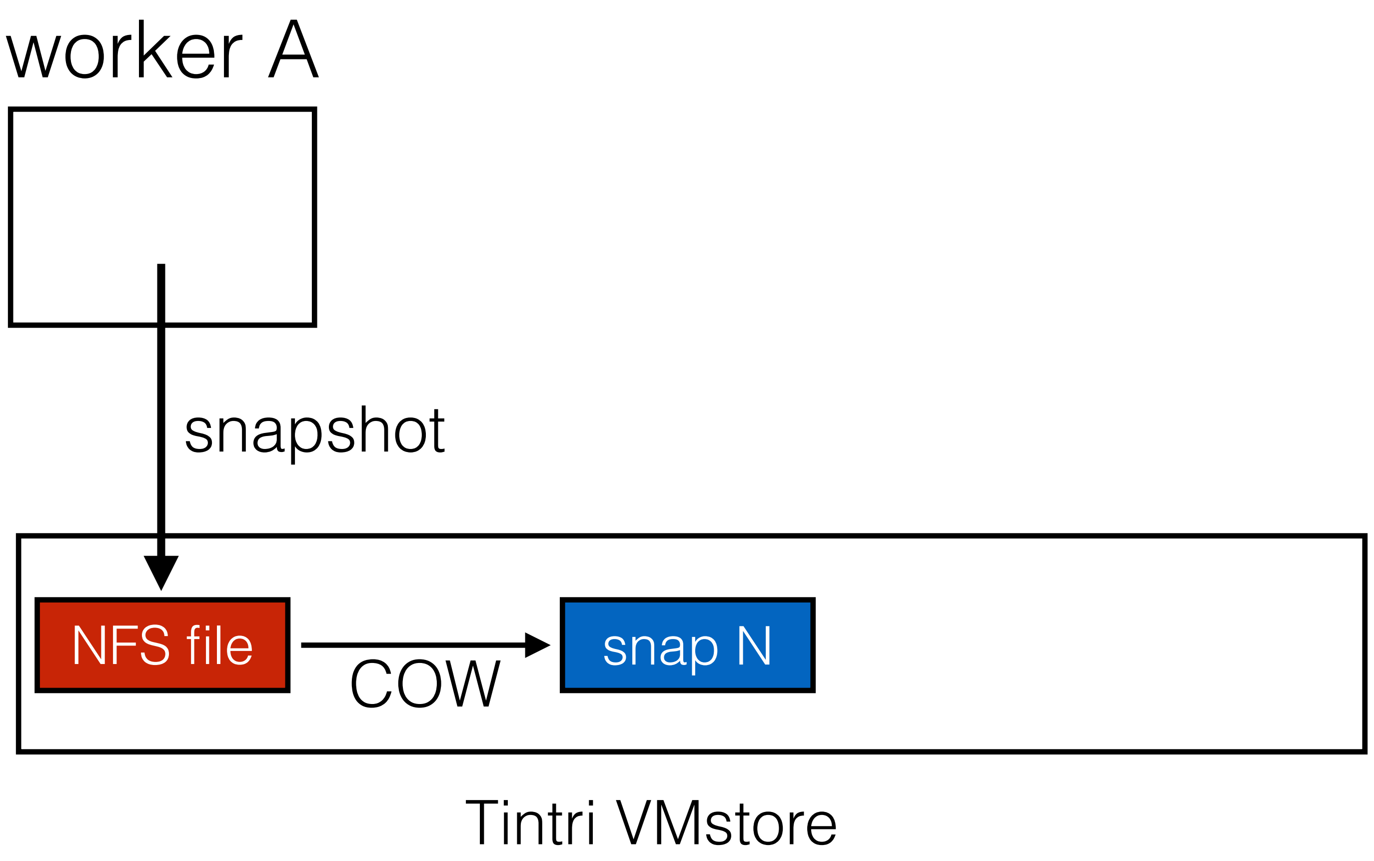


Snapshot and Clone



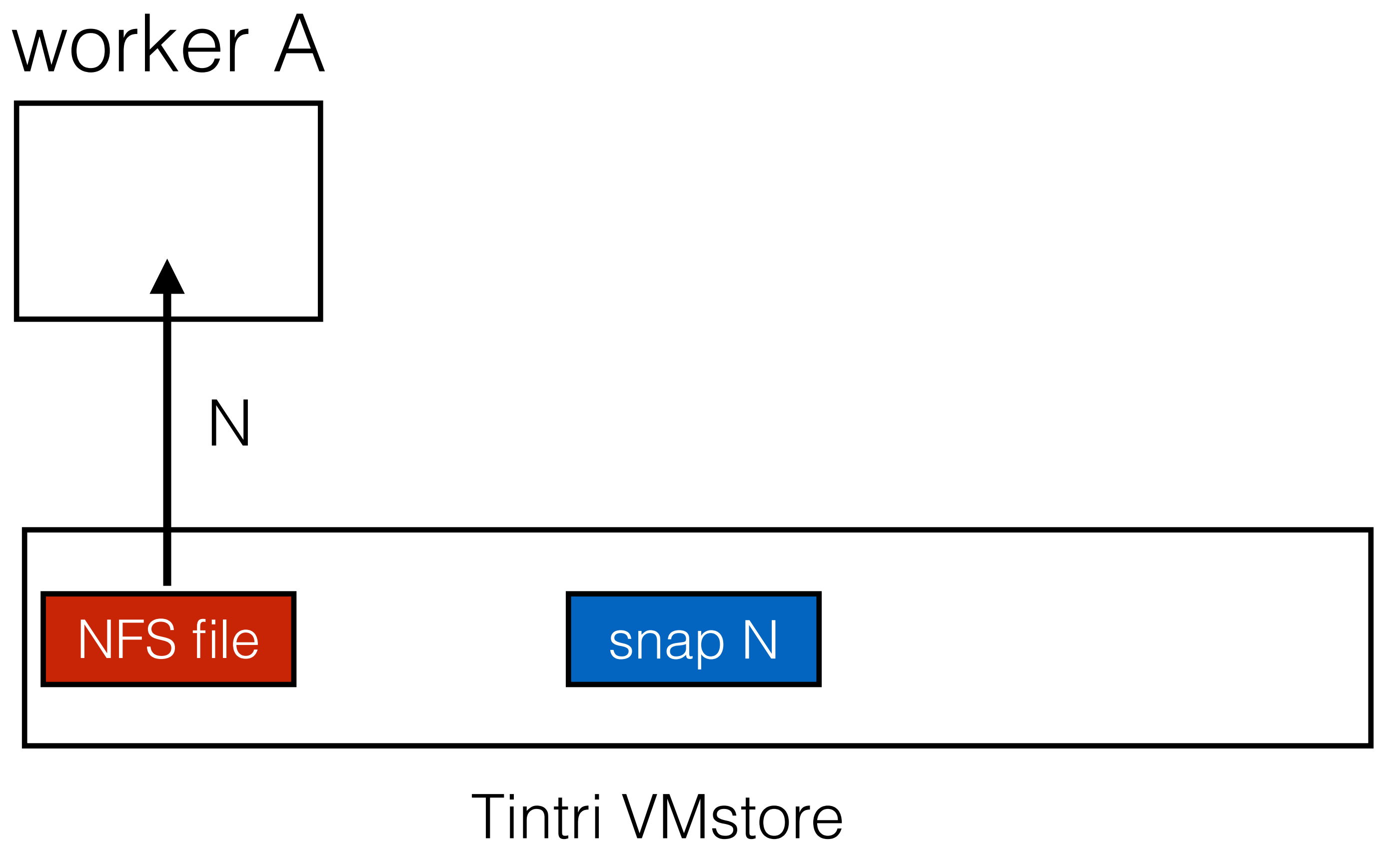
Worker A: push

Snapshot and Clone



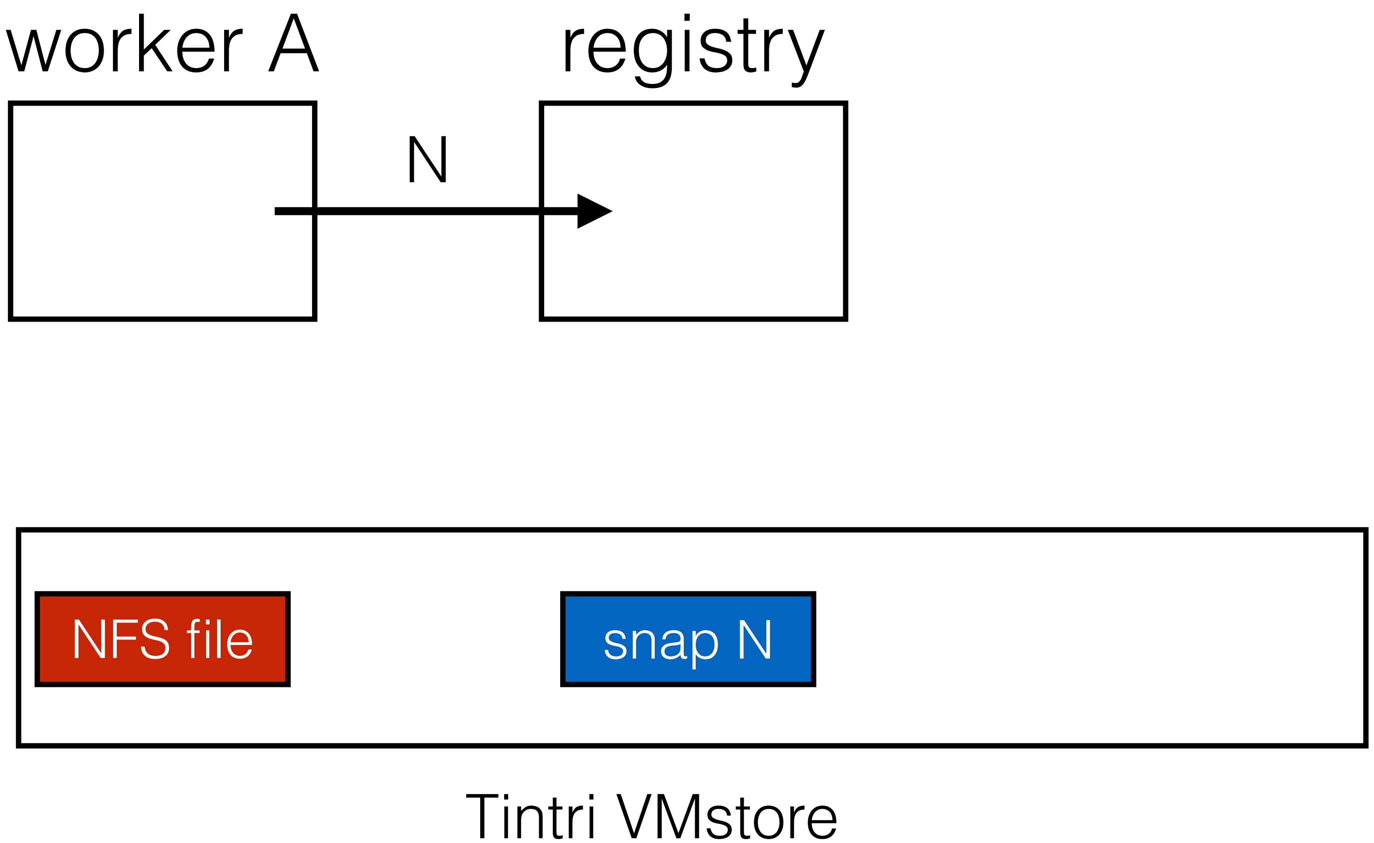
Worker A: push

Snapshot and Clone



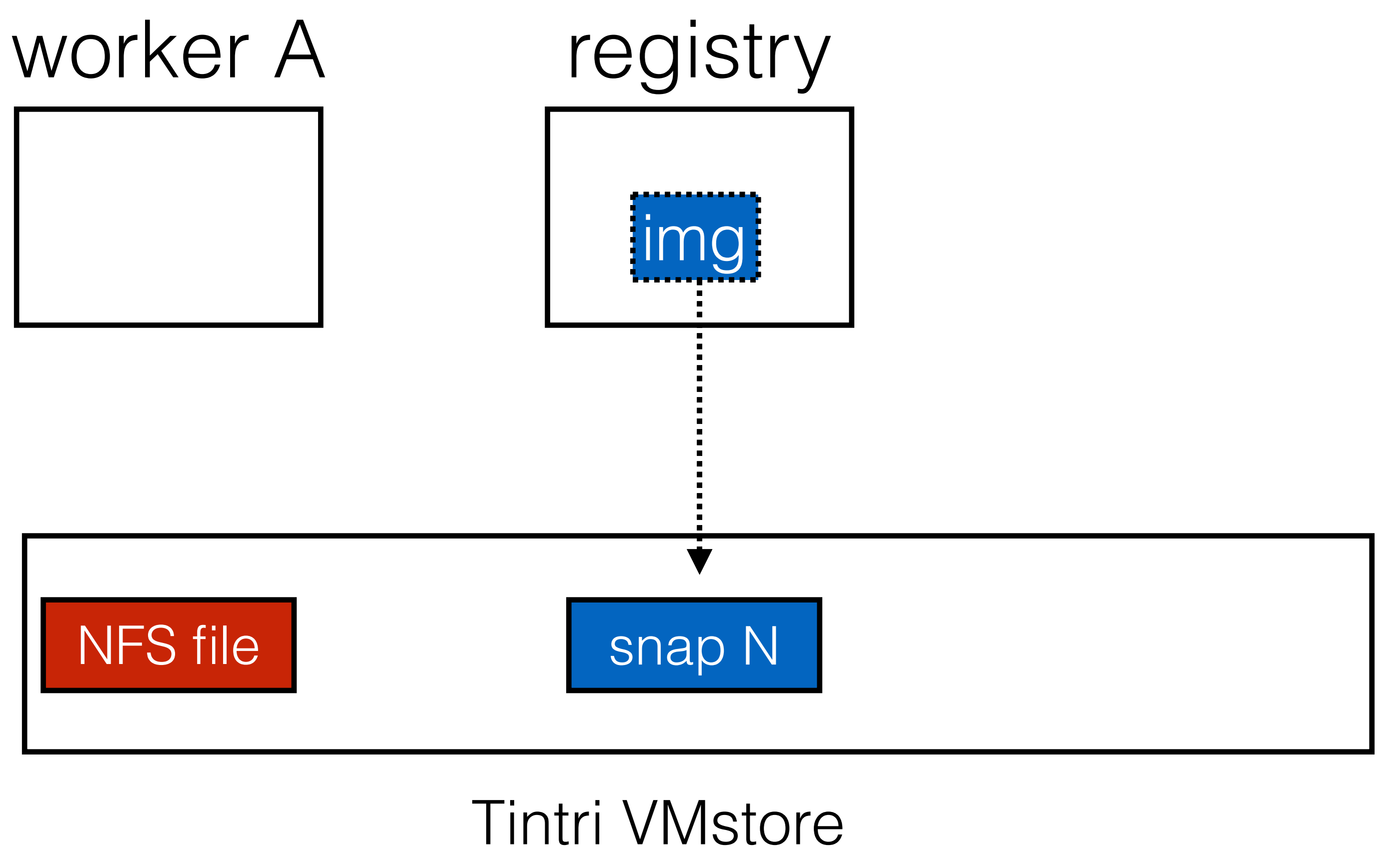
Worker A: push

Snapshot and Clone



Worker A: push

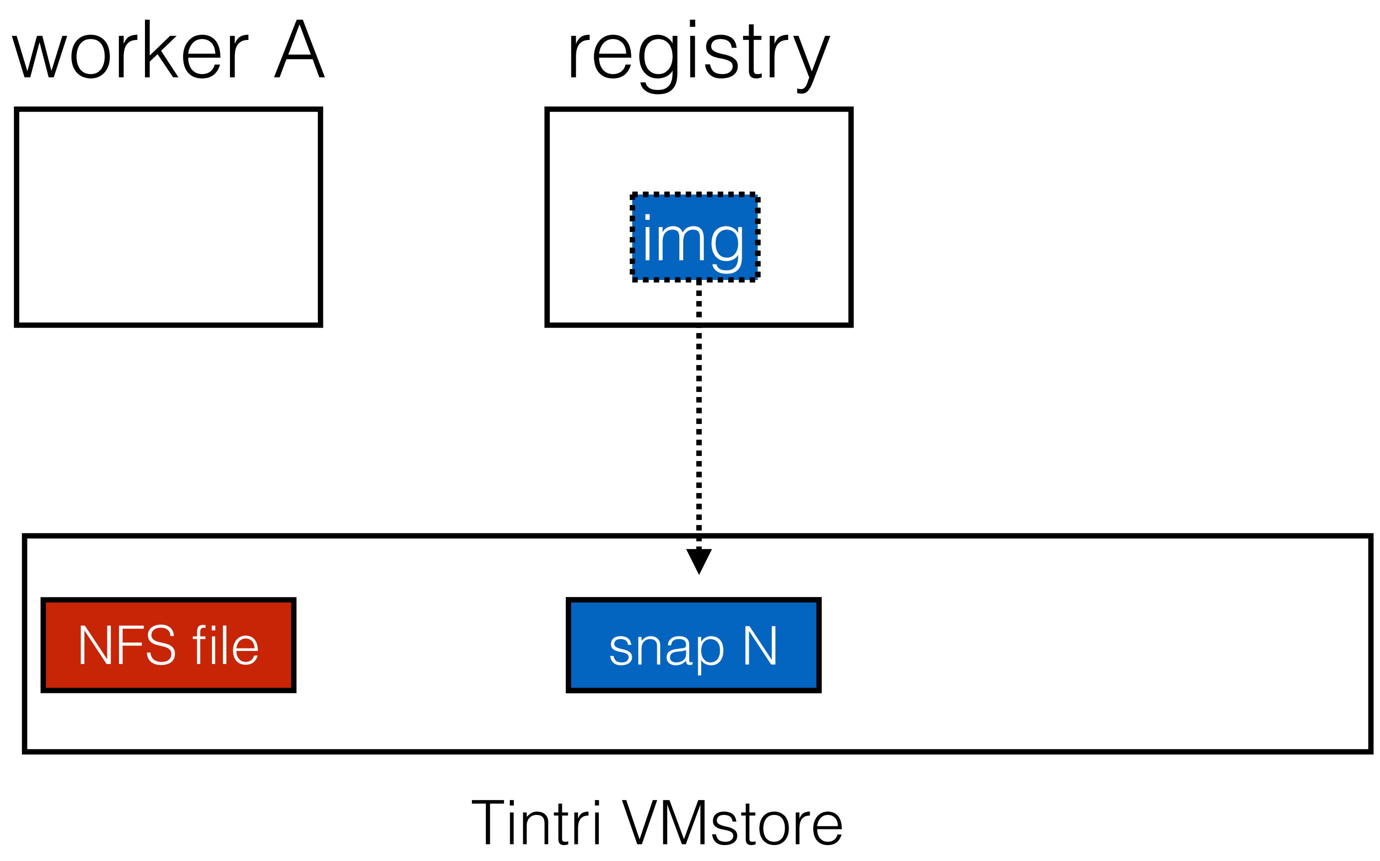
Snapshot and Clone



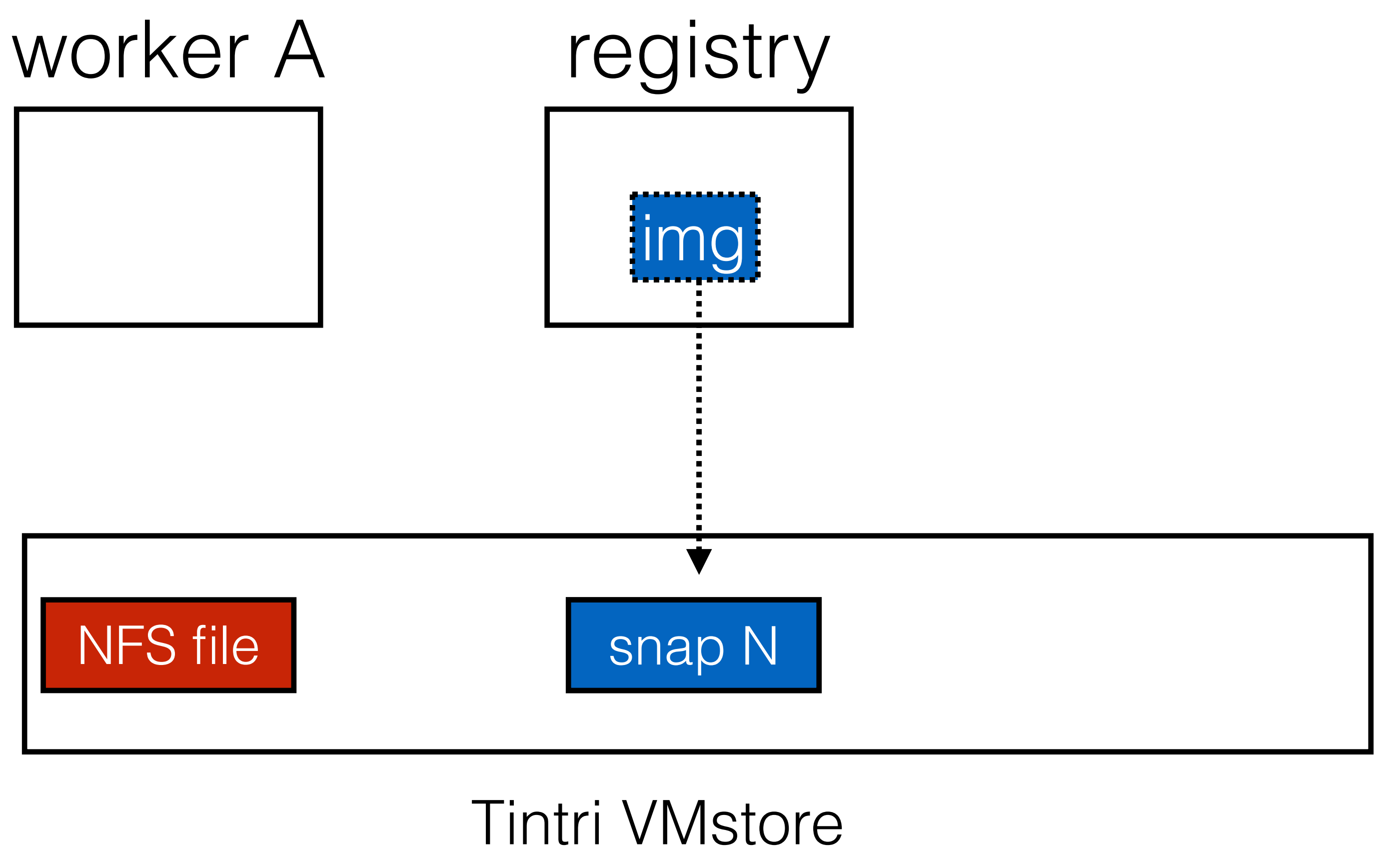
Worker A: push

Snapshot and Clone

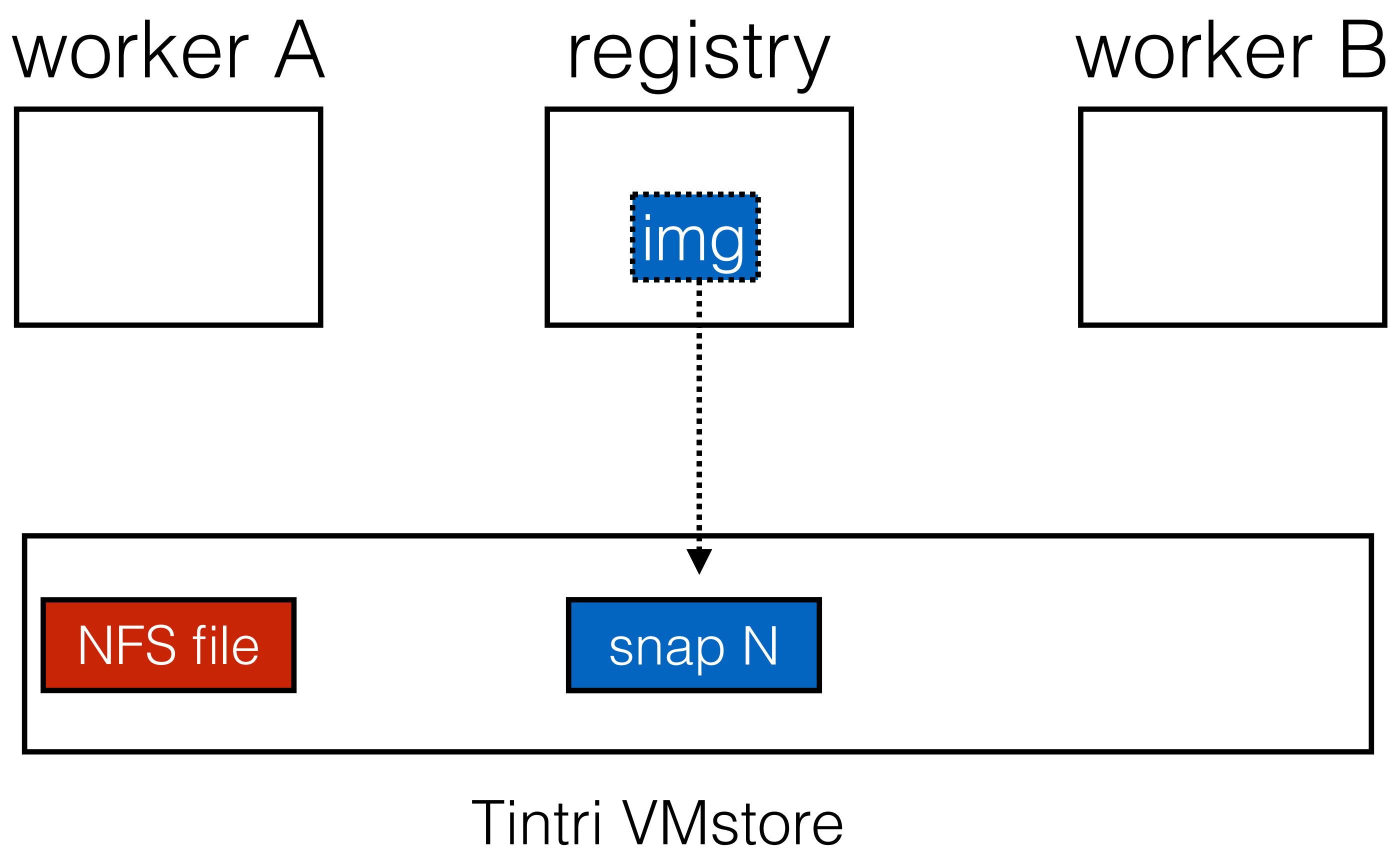
Note: registry is only a name server.
Maps layer metadata ⇒ snapshot ID



Snapshot and Clone

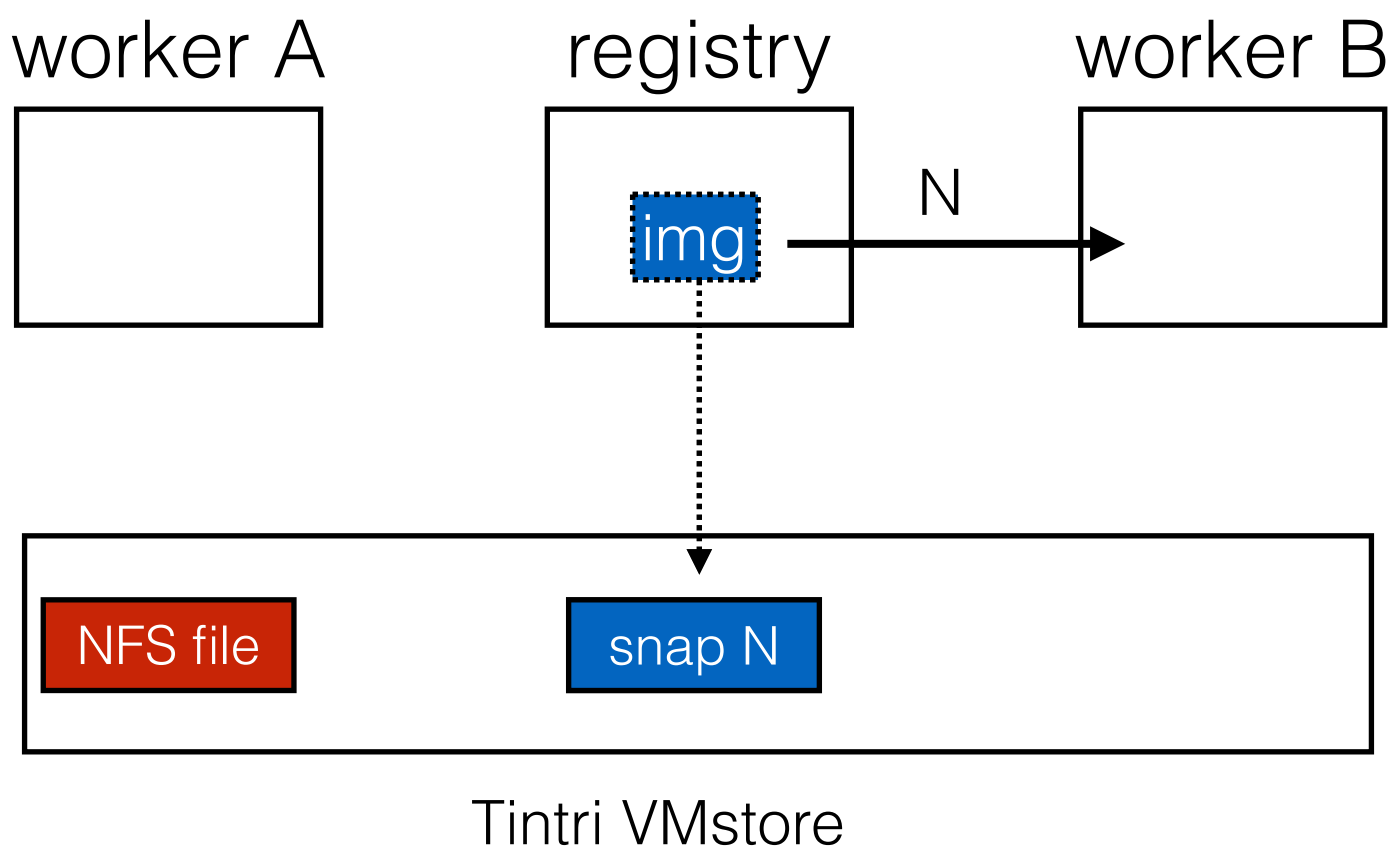


Snapshot and Clone



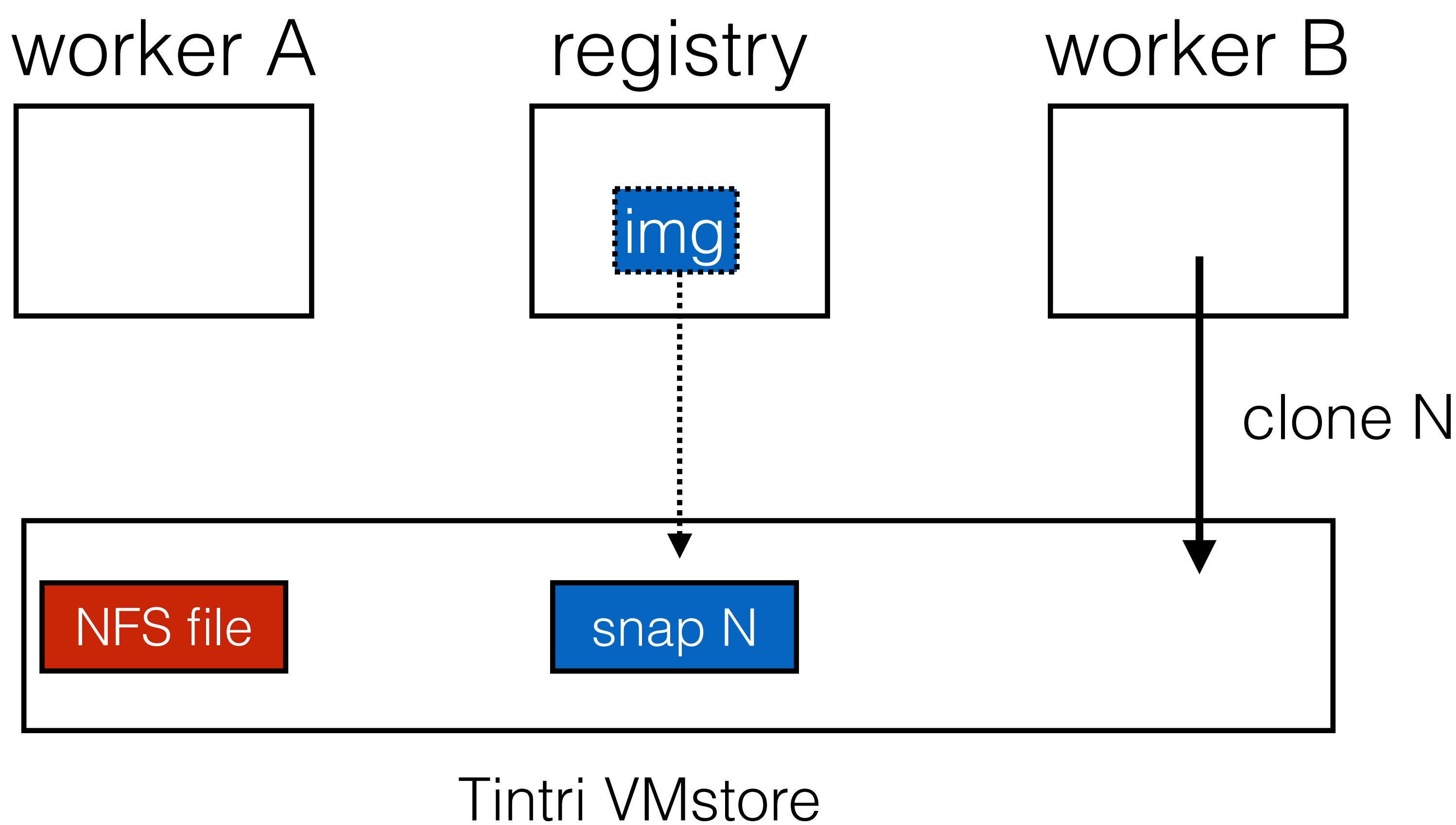
Worker B: pull and run

Snapshot and Clone



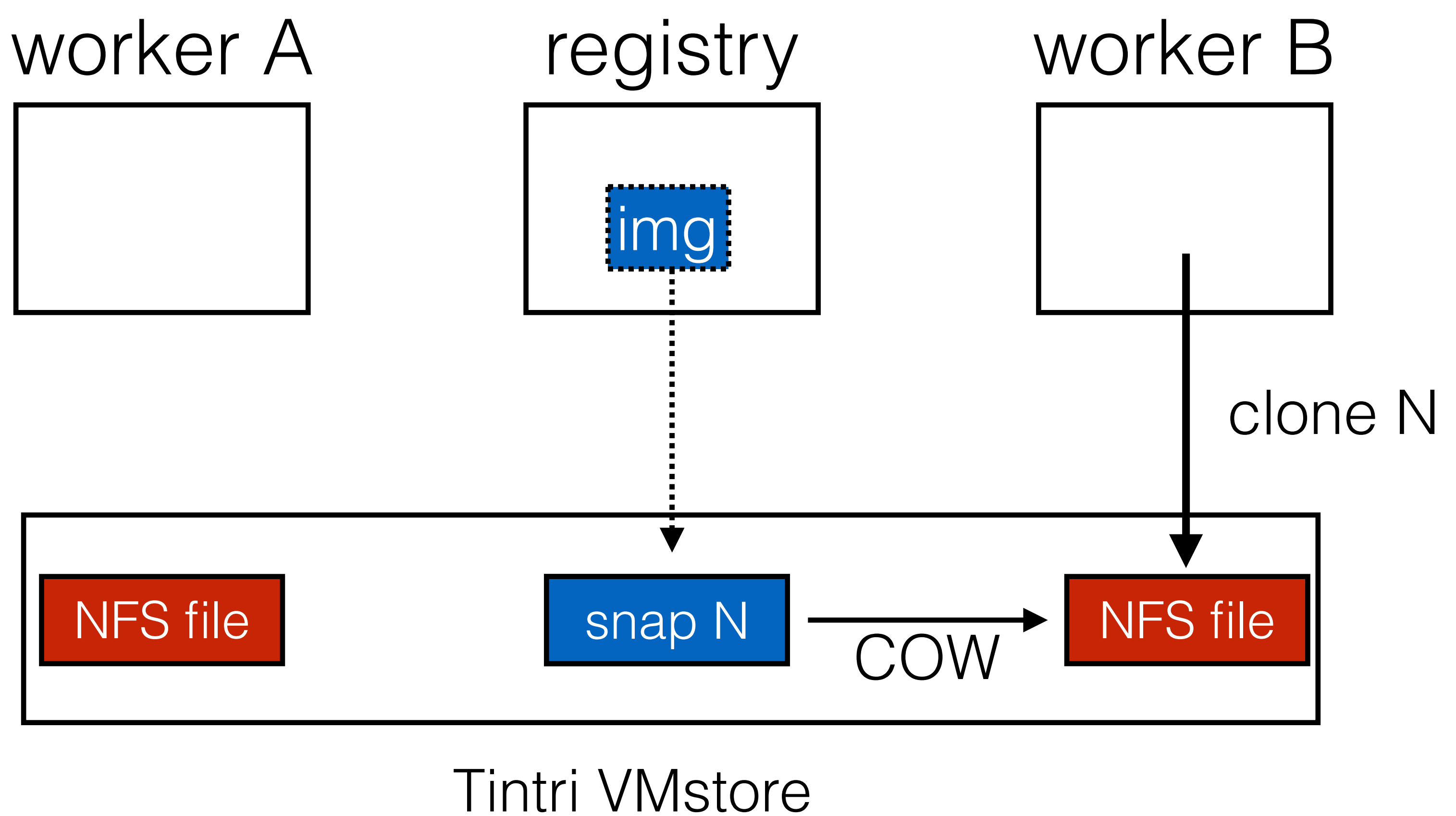
Worker B: pull and run

Snapshot and Clone



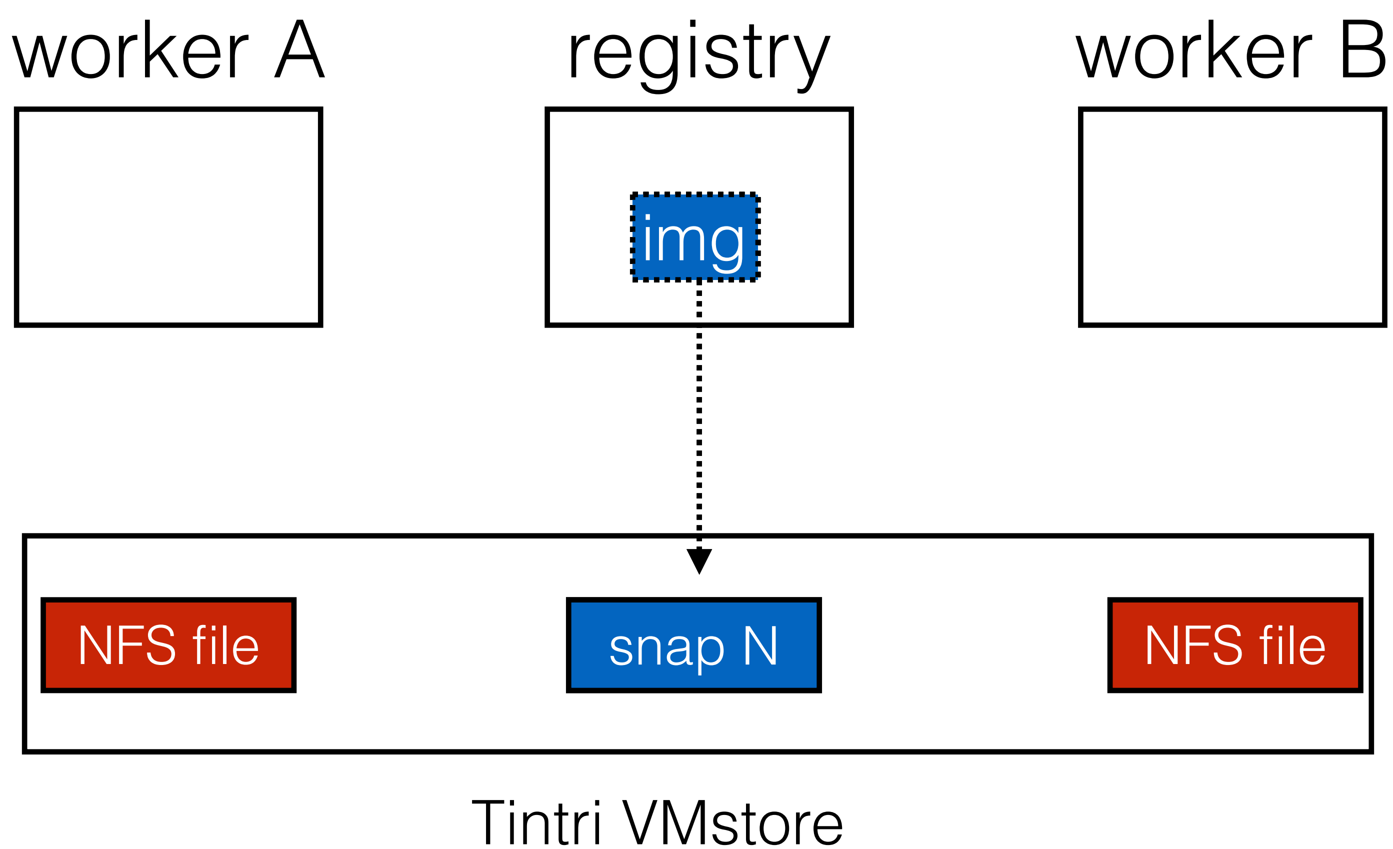
Worker B: pull and run

Snapshot and Clone



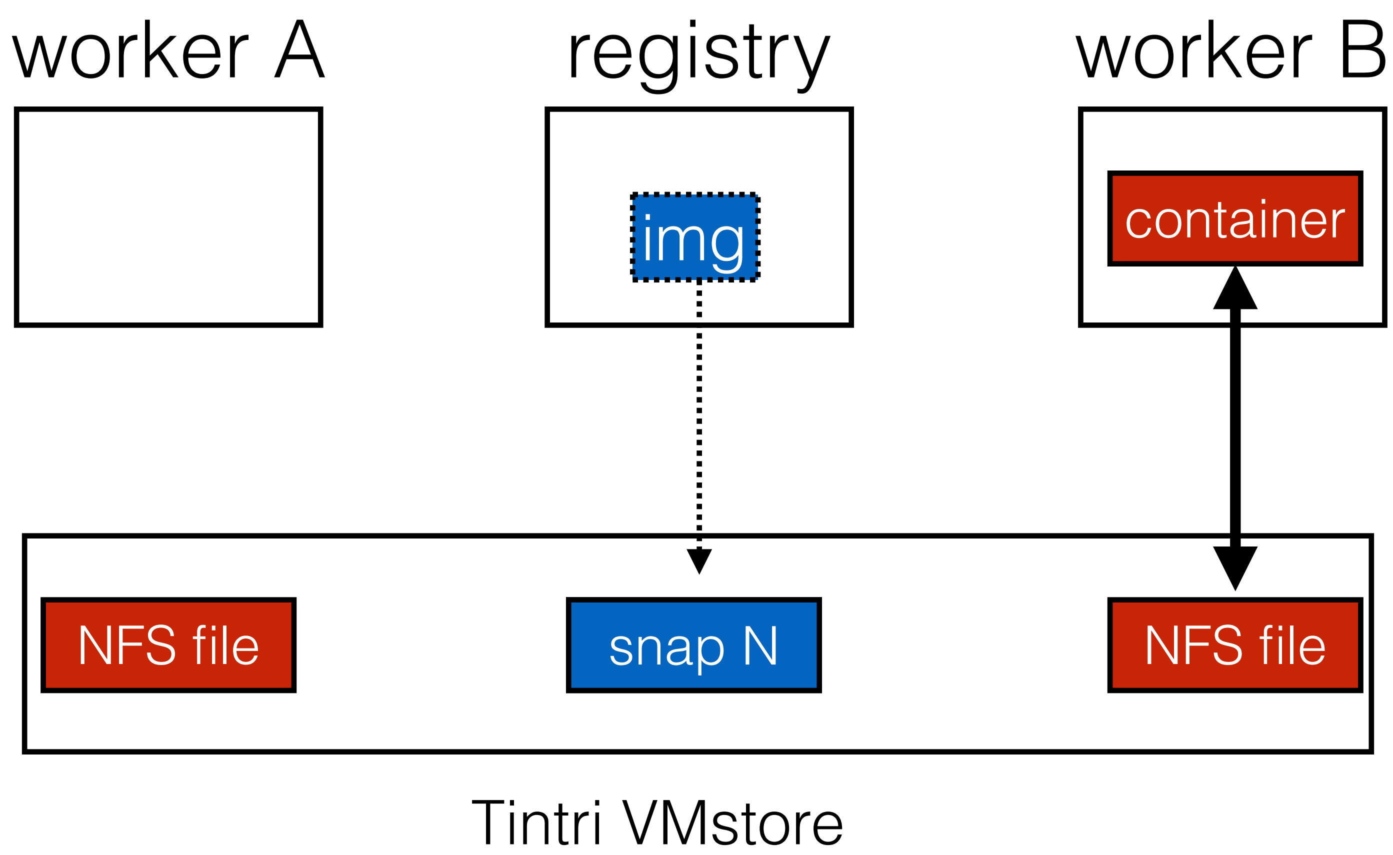
Worker B: pull and run

Snapshot and Clone



Worker B: pull and run

Snapshot and Clone



Worker B: pull and run

Slacker Driver

Goals

- make push+pull very fast
- utilize powerful primitives of a modern storage server (Tintri VMstore)
- create drop-in replacement; don't change Docker framework itself

Design

- lazy pull
- layer flattening
- cache sharing

Slacker Driver

Goals

- make push+pull very fast
- utilize powerful primitives of a modern storage server (Tintri VMstore)
- create drop-in replacement; don't change Docker framework itself

Design

- lazy pull
- layer flattening
- cache sharing

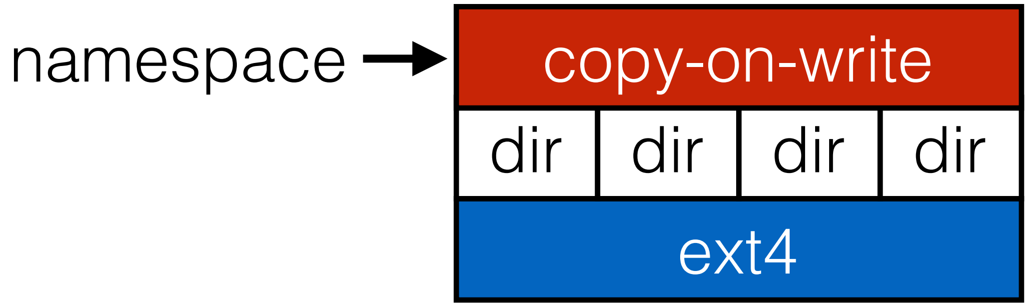
Slacker Flattening

File Namespace Level

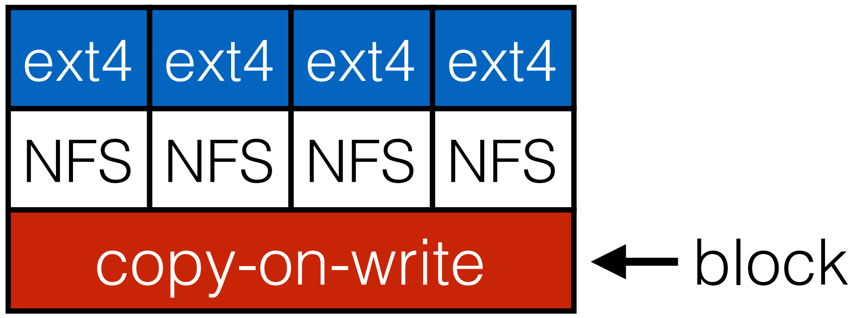
- flatten layers
- if B is child of A, then “copy” A to B to start. Don’t make B empty

Block Level

- do COW+dedup beneath NFS files, inside VMstore



AUFS



Slacker

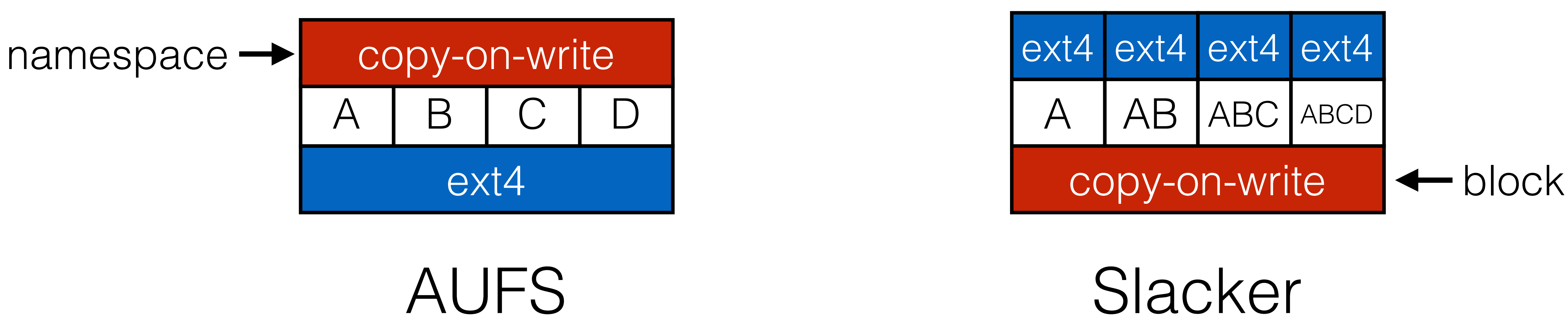
Slacker Flattening

File Namespace Level

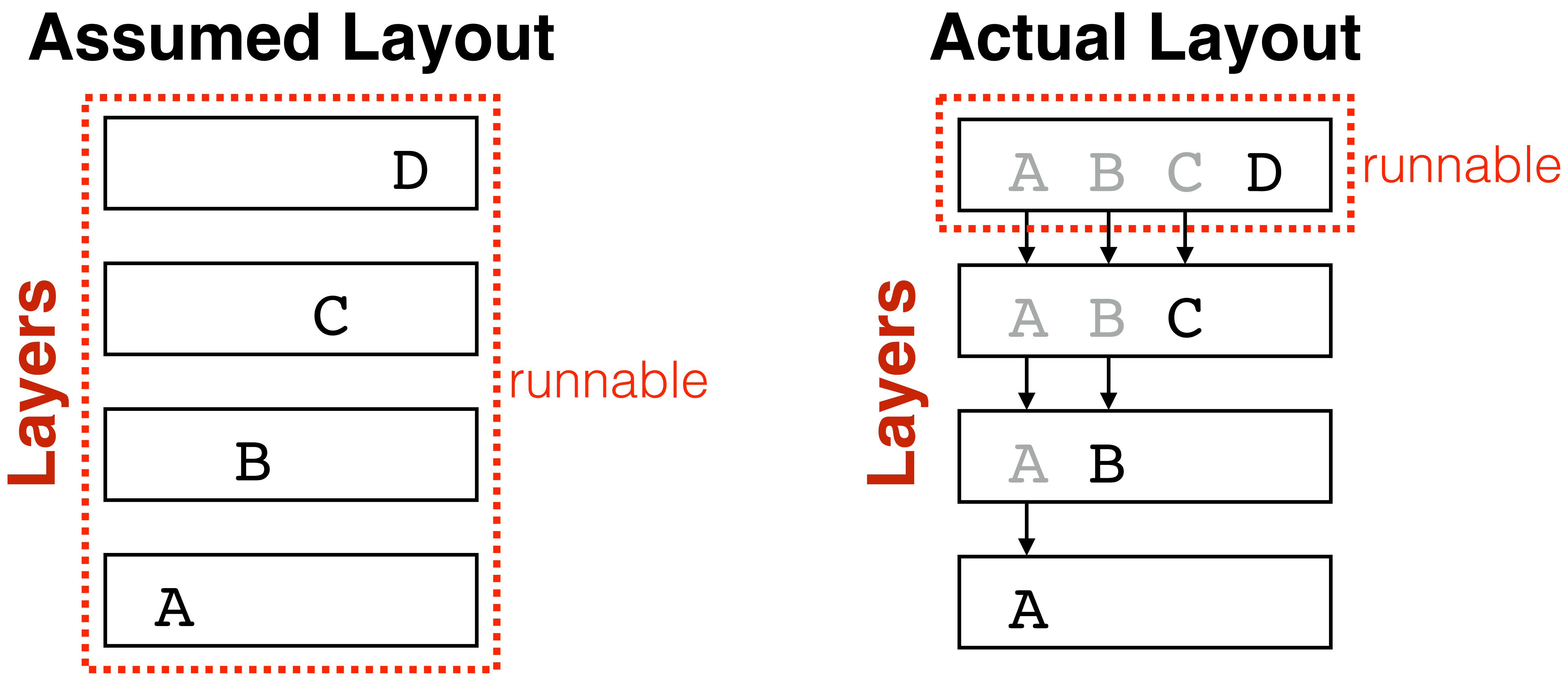
- flatten layers
- if B is child of A, then “copy” A to B to start. Don’t make B empty

Block Level

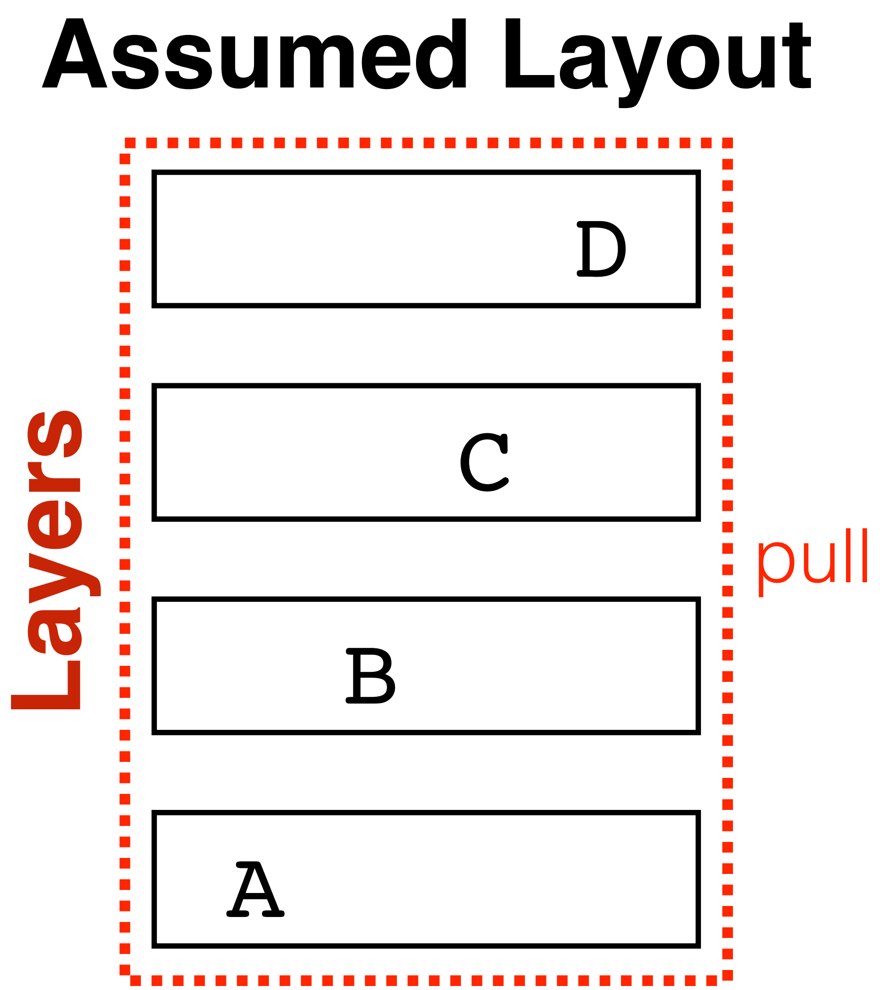
- do COW+dedup beneath NFS files, inside VMstore



Challenge: Framework Assumptions

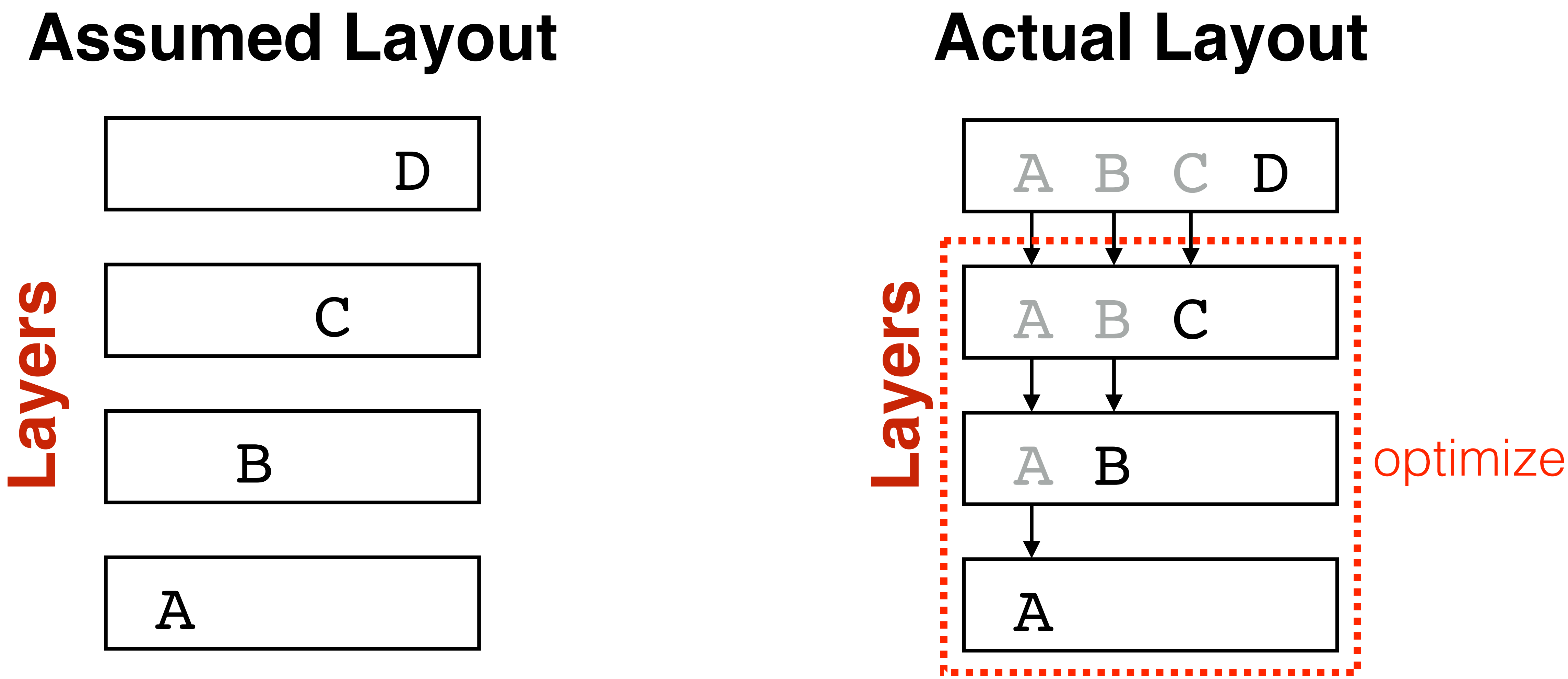


Challenge: Framework Assumptions



Challenge: Framework Assumptions

Strategy: **lazy cloning**. Don't clone non-top layers until Docker tries to mount them.



Slacker Driver

Goals

- make push+pull very fast
- utilize powerful primitives of a modern storage server (Tintri VMstore)
- create drop-in replacement; don't change Docker framework itself

Design

- lazy pull
- layer flattening ←
- cache sharing

Slacker Driver

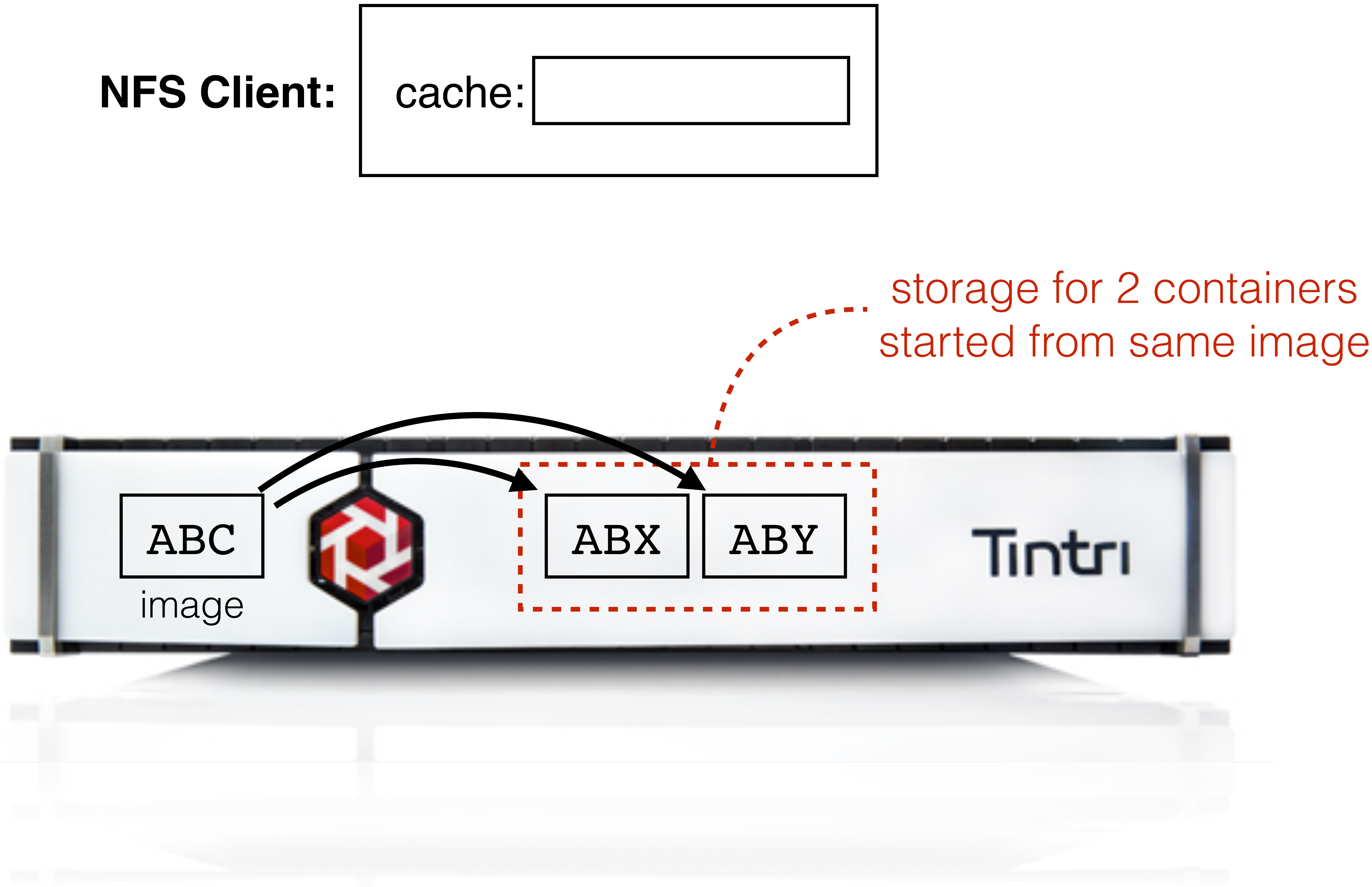
Goals

- make push+pull very fast
- utilize powerful primitives of a modern storage server (Tintri VMstore)
- create drop-in replacement; don't change Docker framework itself

Design

- lazy pull
- layer flattening
- cache sharing

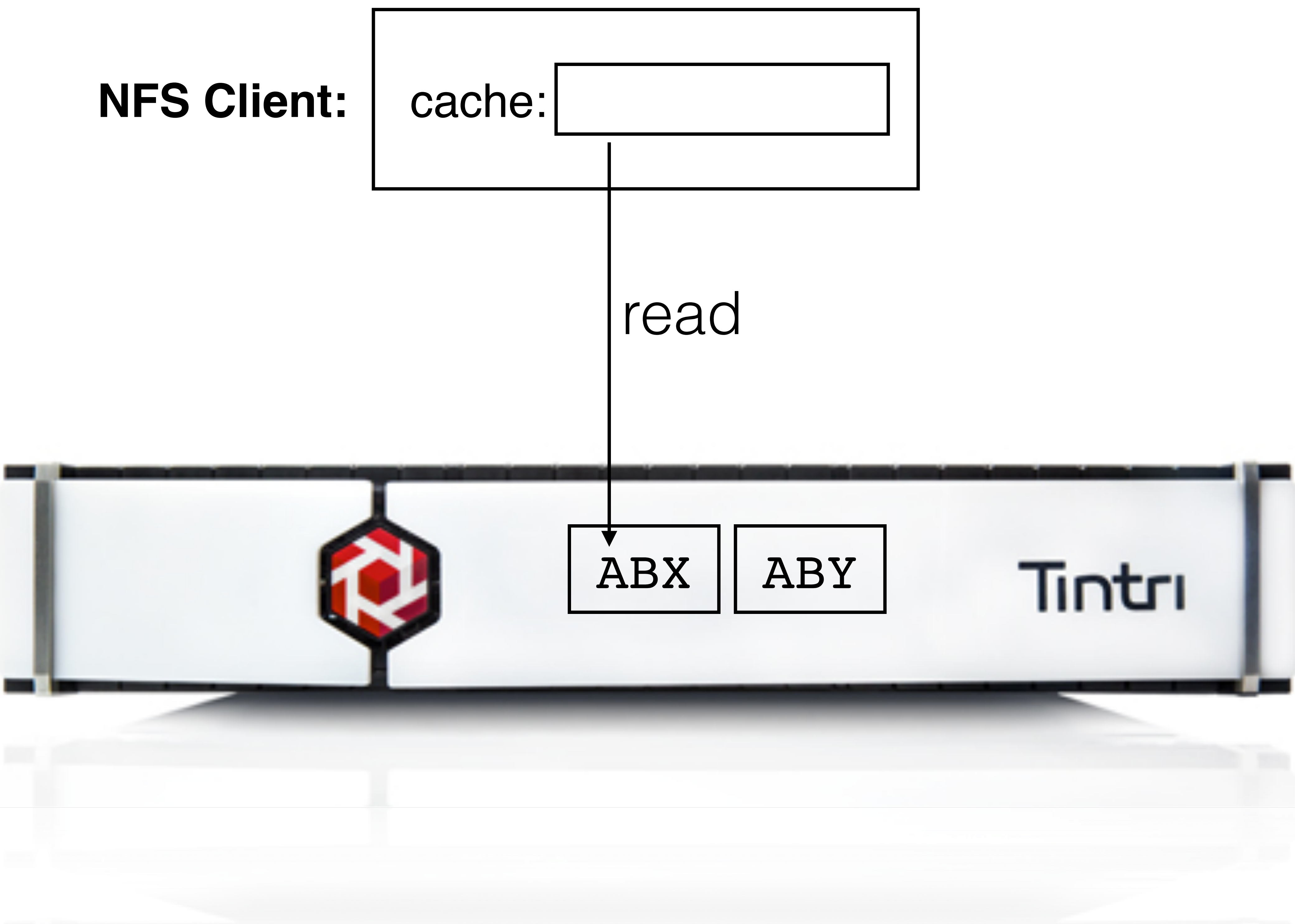
Challenge: Cache Sharing



Challenge: Cache Sharing

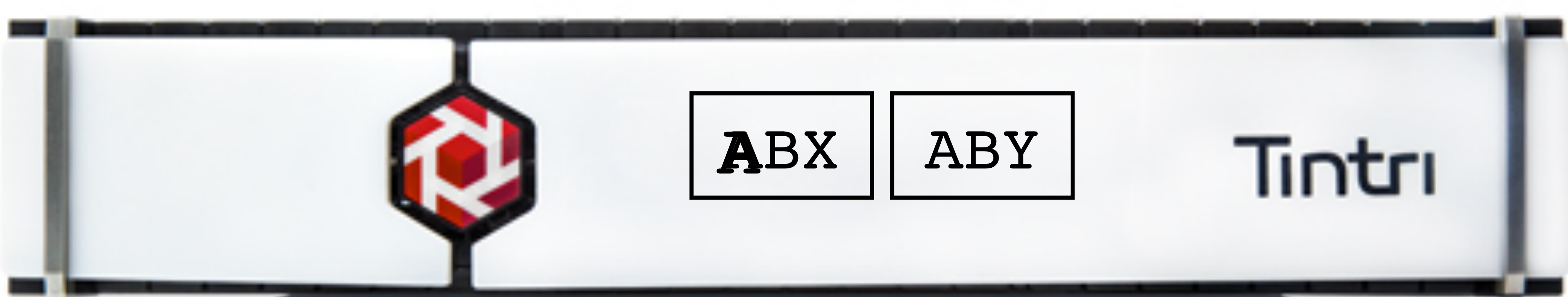
NFS Client: cache:

Challenge: Cache Sharing



Challenge: Cache Sharing

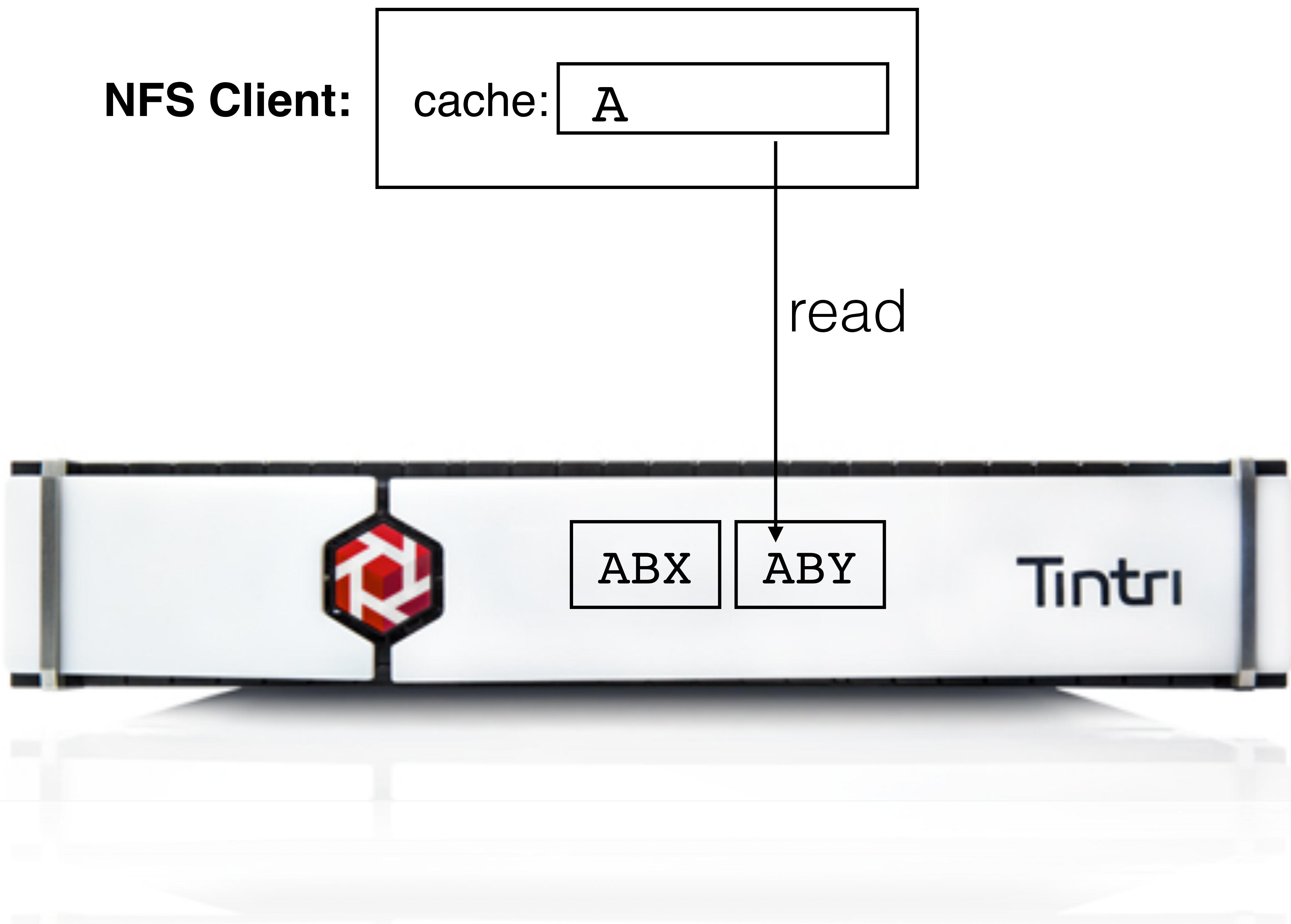
NFS Client: cache:



Challenge: Cache Sharing

NFS Client: cache: A

Challenge: Cache Sharing



Challenge: Cache Sharing

NFS Client: cache: A

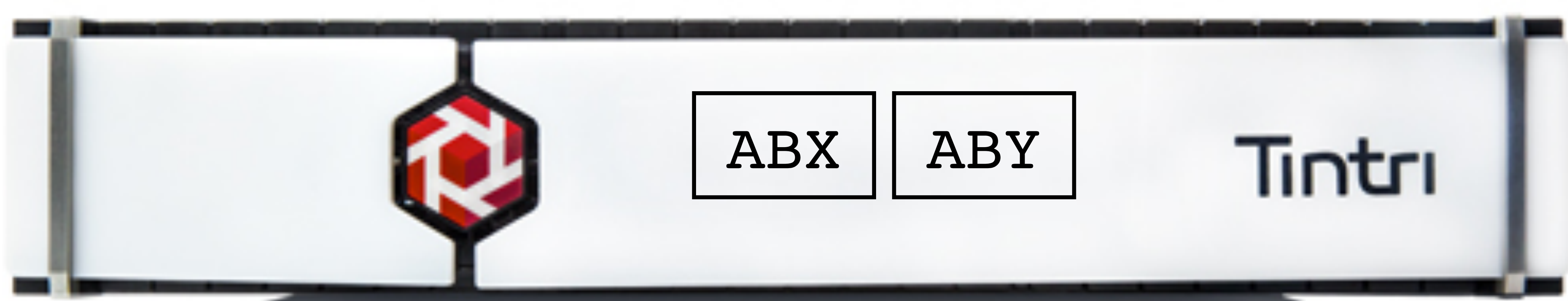
Challenge: Cache Sharing

NFS Client: cache: A A

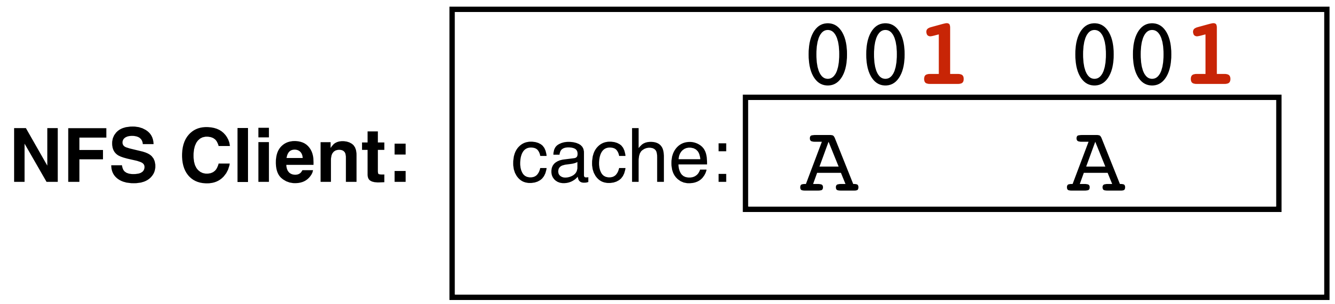
Challenge: Cache Sharing

NFS Client: cache: A A

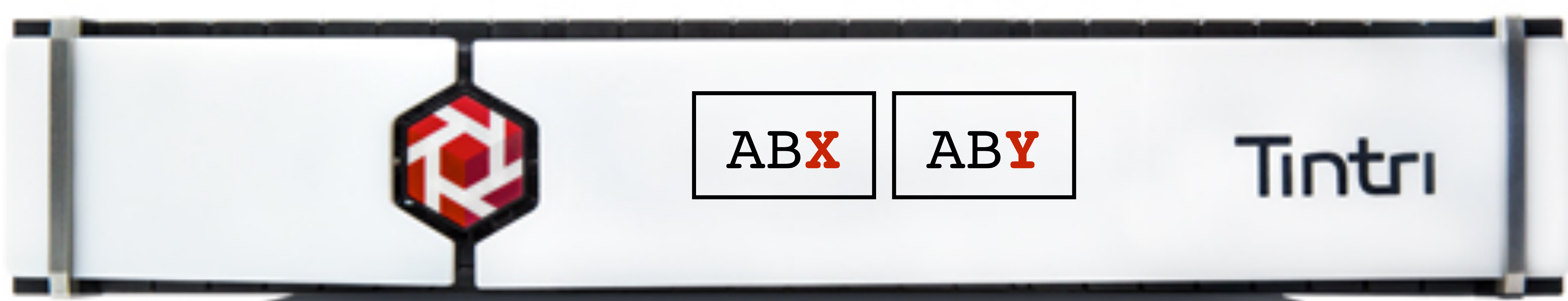
Challenge: how to avoid space and I/O waste?



Challenge: Cache Sharing



Strategy: track differences and deduplicate I/O (more in paper)



Slacker Outline

Background

Container Workloads

Default Driver: AUFS

Our Driver: Slacker

Evaluation

Conclusion

Questions

What are deployment and development speedups?

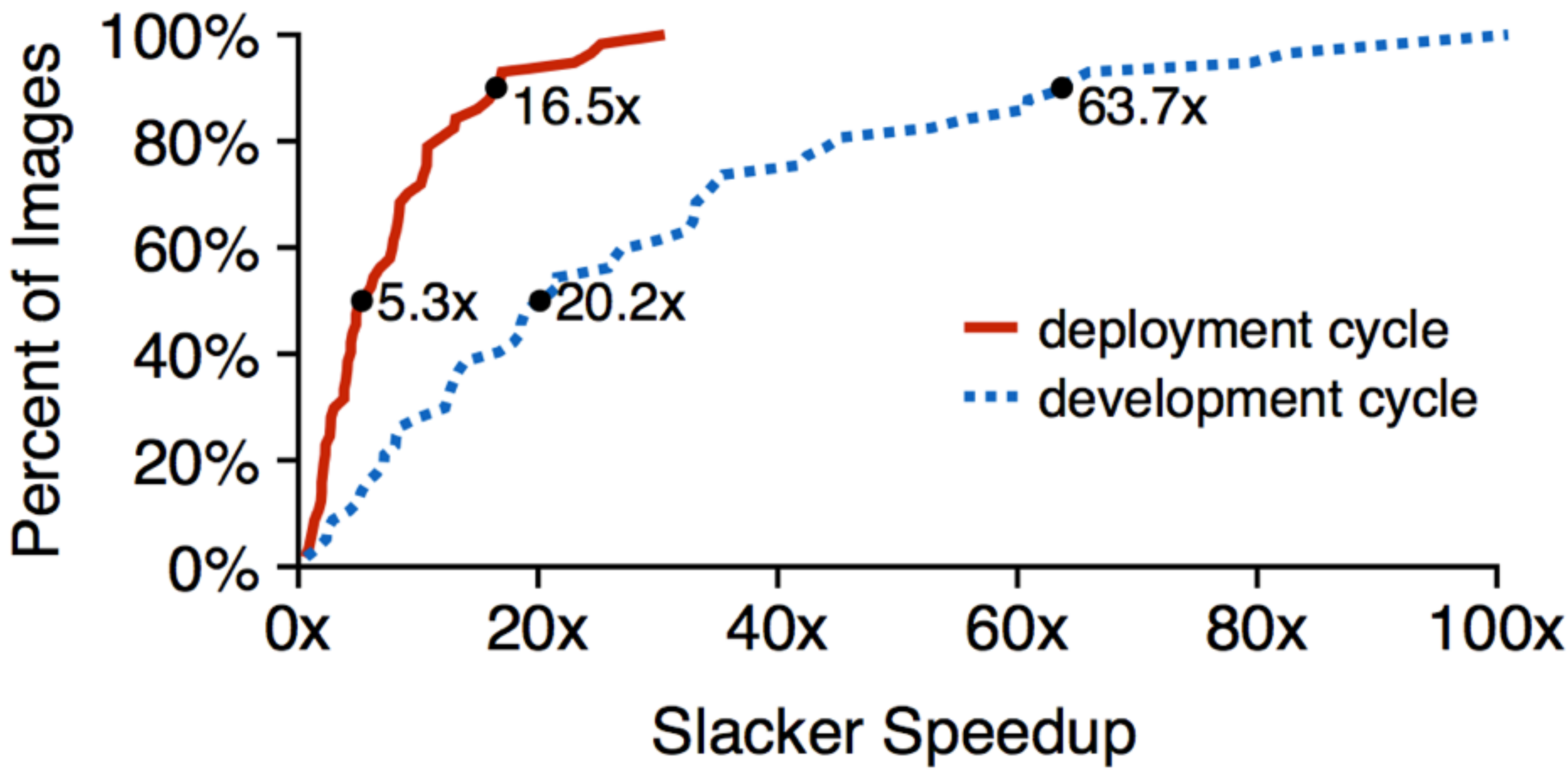
How is long-term performance?

Questions

What are deployment and development speedups?

How is long-term performance?

HelloBench Performance



deployment: pull+run
development: push+pull+run

Questions

What are deployment and development speedups?

- 5x and 20x faster respectively (median speedup)

How is long-term performance?

Questions

What are deployment and development speedups?

- 5x and 20x faster respectively (median speedup)

How is long-term performance?

Server Benchmarks

Databases and Web Servers

- PostgreSQL
- Redis
- Apache web server (static)
- io.js Javascript server (dynamic)

Experiment

- measure throughput (after startup)
- run 5 minutes

Server Benchmarks

Databases and Web Servers

- PostgreSQL
- Redis
- Apache web server (static)
- io.js Javascript server (dynamic)

Experiment

- measure throughput (after startup)
- run 5 minutes

Result: Slacker is always at least as fast as AUFS

Questions

What are deployment and development speedups?

- 5x and 20x faster respectively (median speedup)

How is long-term performance?

- there is no long-term penalty for being lazy

Slacker Outline

Background

Container Workloads

Default Driver: AUFS

Our Driver: Slacker

Evaluation

Conclusion

Strengths:
1. very nice design of HelloBench, Slacker and MultiMaker.
2. Good lverage of VMStore based on their prior work for block ID based snapshot and clone
3. Good optimization with lazy fetch and ID-based cache; Modified kernel driver further enables client side caching.
Weaknesses:
1. The presentation has too many animation and failed to convey the beauty of the design and implementation.
2. Could have discussed the implication of Slacker to large parallel file or storage systems because it is unlikely for big data centers to use NSF-based backup.

Conclusion

Containers are inherently lightweight

- but existing frameworks are not

COW between workers is necessary for fast startup

- use shared storage
- utilize VMstore snapshot and clone

Slacker driver

- **5x** deployment speedup
- **20x** development speedup

HelloBench: https://github.com/Tintri/hello-bench

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Tintri