Trends and Practices for Pulling HPC Containers in Cloud

Vanessa Sochat

Principal Computer Scientist
Lawrence Livermore National Laboratory

LLNL-PRES-872719

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

PLOS | ONE

RESEARCH ARTICLE

Singularity: Scientific containers for mobility of compute

Gregory M. Kurtzer[1], **Vanessa Sochat**[2]*, **Michael W. Bauer**[1,3,4]

1 High Performance Computing Services, Lawrence Berkeley National Lab, Berkeley, CA, United States of America, **2** Stanford Research Computing Center and School of Medicine, Stanford University, Stanford, CA, United States of America, **3** Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, United States of America, **4** Experimental Systems, GSI Helmholtzzentrum für Schwerionenforschung, Darmstadt, Germany

* vsochat@stanford.edu

Citation: Kurtzer GM, Sochat V, Bauer MW (2017) Singularity: Scientific containers for mobility of compute. PLoS ONE 12(5): e0177459. https://doi.org/10.1371/journal.pone.0177459

Editor: Attila Gursoy, Koc Universitesi, TURKEY

Received: December 20, 2016

Accepted: April 27, 2017

Published: May 11, 2017

Data Availability Statement: The source code for Singularity is available at https://github.com/singularityware/singularity, and complete documentation at http://singularity.lbl.gov/.

Funding: Author VS is supported by Stanford Research Computing (IT) and the Stanford School of Medicine, and author MWB is supported by the Frankfurt Institute of Advanced Studies (FIAS). Author GMK is an employee of Lawrence Berkeley National Lab, the Department of Energy, and UC Regents. This manuscript has been authored by an author (GMK) at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy.

Abstract

Here we present Singularity, software developed to bring containers and reproducibility to scientific computing. Using Singularity containers, developers can work in reproducible environments of their choosing and design, and these complete environments can easily be copied and executed on other platforms. Singularity is an open source initiative that harnesses the expertise of system and software engineers and researchers alike, and integrates seamlessly into common workflows for both of these groups. As its primary use case, Singularity brings mobility of computing to both users and HPC centers, providing a secure means to capture and distribute software and compute environments. This ability to create and deploy reproducible environments across these centers, a previously unmet need, makes Singularity a game changing development for computational science.

Introduction

The landscape of scientific computing is fluid. Over the past decade and a half, virtualization has gone from an engineering toy to a global infrastructure necessity, and the evolution of related technologies has thus flourished. The currency of files and folders has changed to applications and operating systems. The business of Supercomputing Centers has been to offer scalable computational resources to a set of users associated with an institution or group [1]. With this scale came the challenge of version control to provide users with not just up-to-date software, but multiple versions of it. Software modules [2, 3], virtual environments [4, 5], along with intelligently organized file systems [6] and permissions [7] were essential developments to give users control and reproducibility of work. On the administrative side, automated builds and server configuration [8, 9] have made maintenance of these large high-performance computing (HPC) clusters possible. Job schedulers such as SLURM [10] or SGE [11] are the metaphorical governors to control these custom analyses at scale, and are the primary means of relay between administrators and users. The user requires access to consume resources, and the administrator wants to make sure that the user has the tools and support to make the most efficient use of them.

PLOS ONE

RESEARCH ARTICLE

Enhancing reproducibility in scientific computing: Metrics and registry for Singularity containers

Vanessa V. Sochat[1]*, **Cameron J. Prybol**[2], **Gregory M. Kurtzer**[3]

1 Stanford Research Computing Center and School of Medicine, Stanford University, Stanford, CA, United States of America, **2** Stanford University Department of Genetics, Stanford University, Stanford, CA, United States of America, **3** High Performance Computing Services, Lawrence Berkeley National Lab, Berkeley, CA, United States of America

* vsochat@stanford.edu

Check for updates

Citation: Sochat VV, Prybol CJ, Kurtzer GM (2017) Enhancing reproducibility in scientific computing: Metrics and registry for Singularity containers. PLoS ONE 12(11): e0188511. https://doi.org/10.1371/journal.pone.0188511

Editor: Christophe Antoniewski, CNRS UMR7622 & University Paris 6 Pierre-et-Marie-Curie, FRANCE

Received: July 4, 2017

Accepted: November 8, 2017

Published: November 29, 2017

Data Availability Statement: The Singularity Python software is available at https://www.github.com/singularityware/singularity-python and Singularity Hub is available for use at https://www.singularity-hub.org.

Funding: Vanessa Sochat is supported by Stanford Research Computing and the School of Medicine. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing interests: The authors have declared that no competing interests exist.

Abstract

Here we present Singularity Hub, a framework to build and deploy Singularity containers for mobility of compute, and the singularity-python software with novel metrics for assessing reproducibility of such containers. Singularity containers make it possible for scientists and developers to package reproducible software, and Singularity Hub adds automation to this workflow by building, capturing metadata for, visualizing, and serving containers programmatically. Our novel metrics, based on custom filters of content hashes of container contents, allow for comparison of an entire container, including operating system, custom software, and metadata. First we will review Singularity Hub's primary use cases and how the infrastructure has been designed to support modern, common workflows. Next, we conduct three analyses to demonstrate build consistency, reproducibility metric and performance and interpretability, and potential for discovery. This is the first effort to demonstrate a rigorous assessment of measurable similarity between containers and operating systems. We provide these capabilities within Singularity Hub, as well as the source software singularity-python that provides the underlying functionality. Singularity Hub is available at https://singularity-hub.org, and we are excited to provide it as an openly available platform for building, and deploying scientific containers.

1 Introduction

The modern scientist is challenged with the responsibilities of having expertise in a field, procuring funding, teaching, and publishing to maintain a career. The publication that these scientists produce are implicitly expected to be "reproducible", meaning that they document and make available the methods, data, and tools necessary to repeat experiments and reliably produce similar or identical results. The "reproducibility crisis" [1–3] revealed that a large proportion of publications were not reproducible by other trained scientists. What followed was powerful, proactive action: an investigation and discussion about standards for publication, data sharing, and dissemination of code and tools for reproducible science [1–6].

What is the best way to pull a container?

What are we going to be talking about today?

1. How has the container ecosystem changed since 2014?

What are we going to be talking about today?

1. How has the container ecosystem changed since 2014?
2. What about best practices?

What are we going to be talking about today?

1. How has the container ecosystem changed since 2014?
2. What about best practices?
3. Can I simulate the pulling part of a container study (and build a tool for others)?

What are we going to be talking about today?

1. How has the container ecosystem changed since 2014?
2. What about best practices?
3. Can I simulate the pulling part of a container study (and build a tool for others)?
4. What are "best practices" for pulling strategies, and how do they hold up?

What are we going to be talking about today?

1. How has the container ecosystem changed since 2014?
2. What about best practices?
3. Can I simulate the pulling part of a container study (and build a tool for others)?
4. What are "best practices" for pulling strategies, and how do they hold up?
5. Why should I care?

How has the container ecosystem changed since 2014?

How has the container ecosystem changed since 2014?

1. What we are interested in (that we can derive from registries):

Size of entire containers?
Size of layers?
Number of layers?
Image similarity?

How has the container ecosystem changed since 2014?

1. What we are interested in (that we can derive from registries):

Size of entire containers?
Size of layers?
Number of layers?
Image similarity?

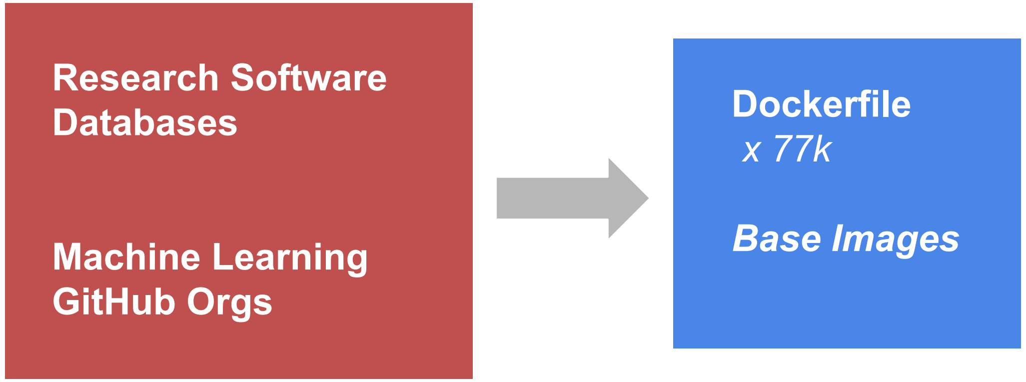
Research Software Databases

Machine Learning GitHub Orgs

How has the container ecosystem changed since 2014?

1. What we are interested in (that we can derive from registries):

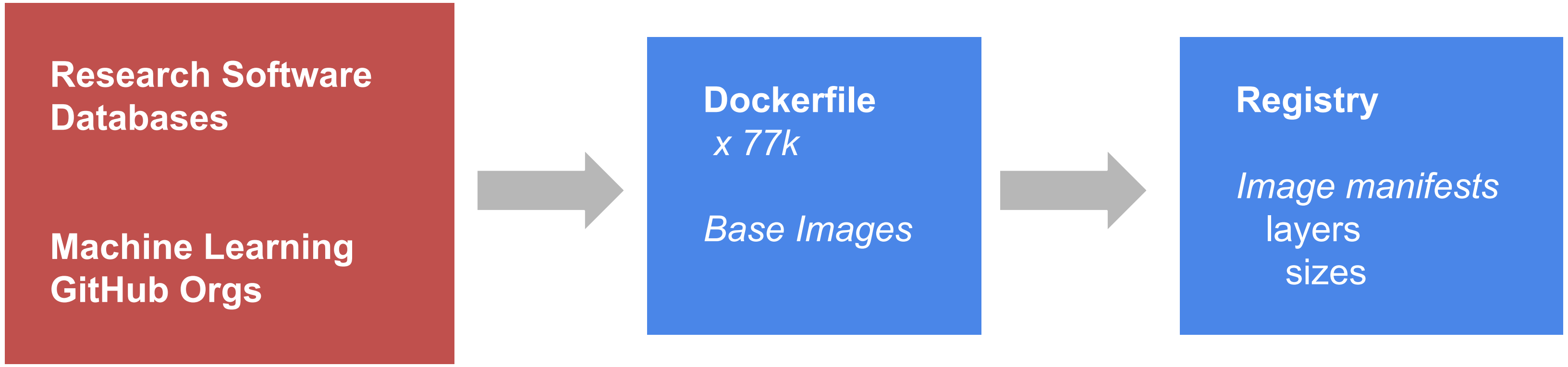
Size of entire containers?
Size of layers?
Number of layers?
Image similarity?



How has the container ecosystem changed since 2014?

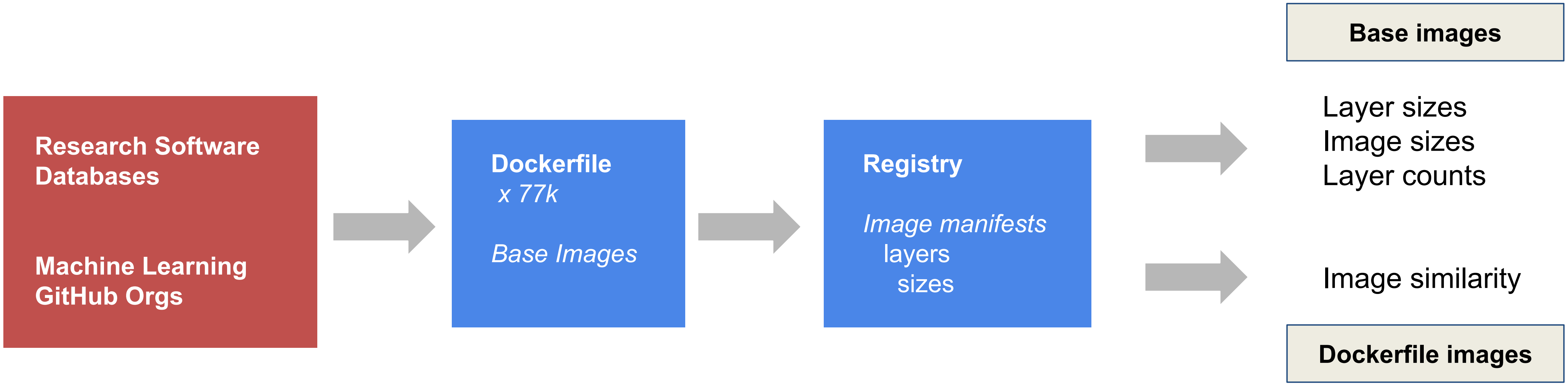
1. What we are interested in (that we can derive from registries):

Size of entire containers?
Size of layers?
Number of layers?
Image similarity?

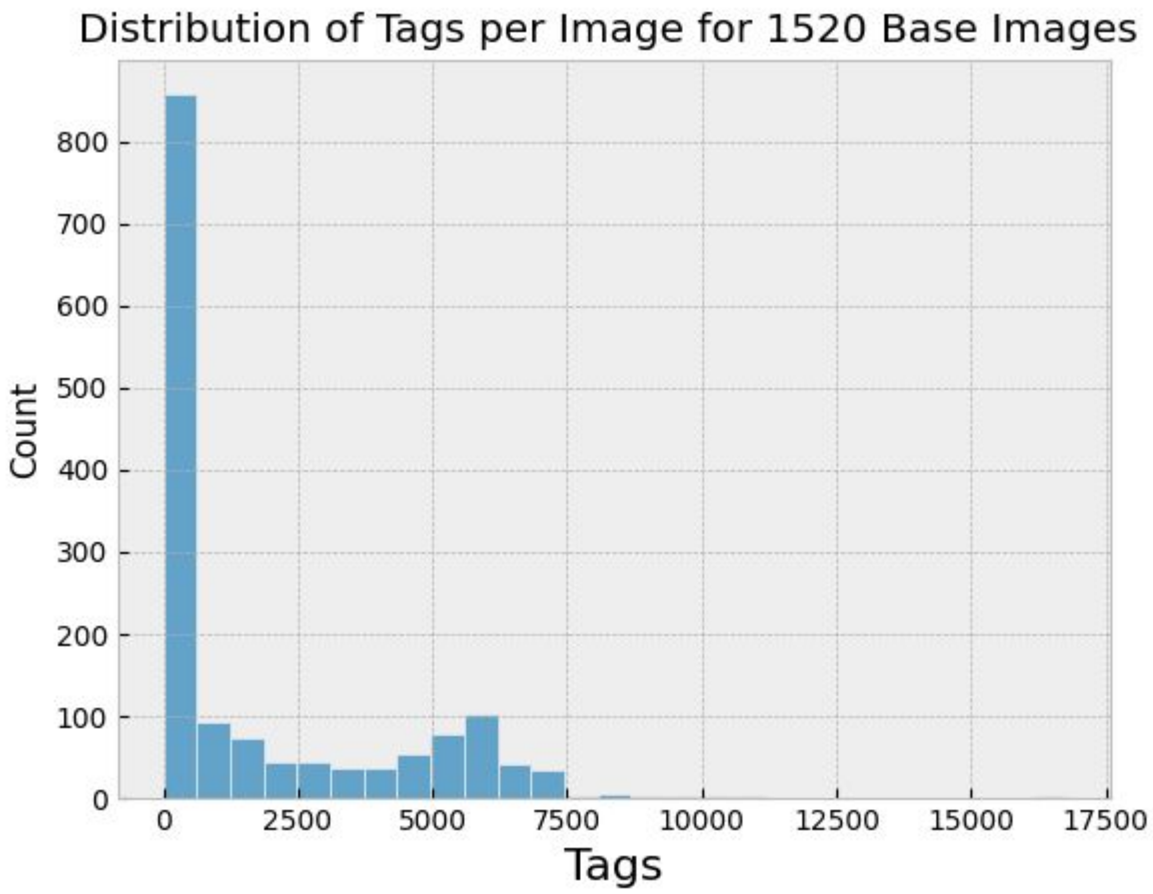


How has the container ecosystem changed since 2014?

1. What we are interested in (that we can derive from registries):

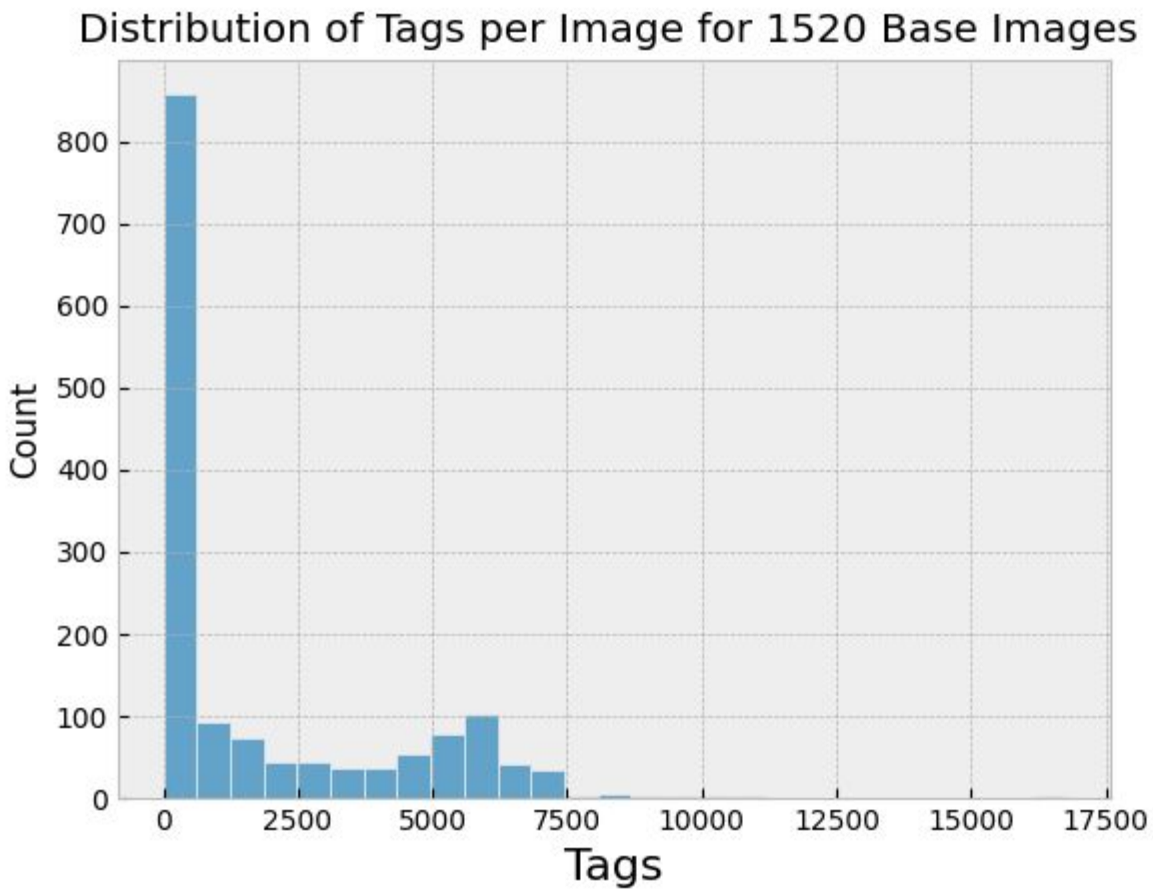


How many tags does each *base image* have?



- Ranges from 1 to ~17k tags
- Mean 1842 tags, std 2,531 tags
- One outlier removed (nix/nixos) ~47k tags

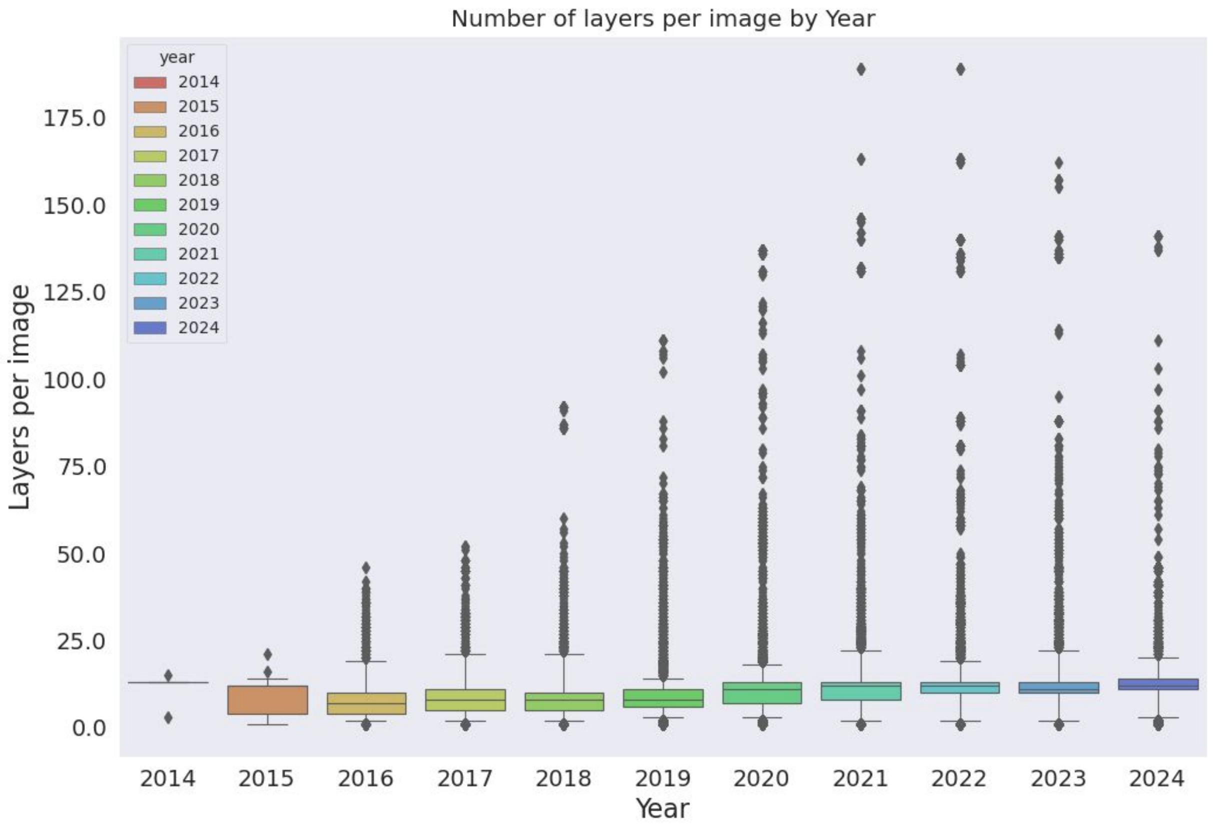
How many tags does each *base image* have?



- Ranges from 1 to ~17k tags
- Mean 1842 tags, std 2,531 tags
- One outlier removed (nix/nixos) ~47k tags

Tag counts reflects release frequency
(and often automation)

How has number of layers changed over time?

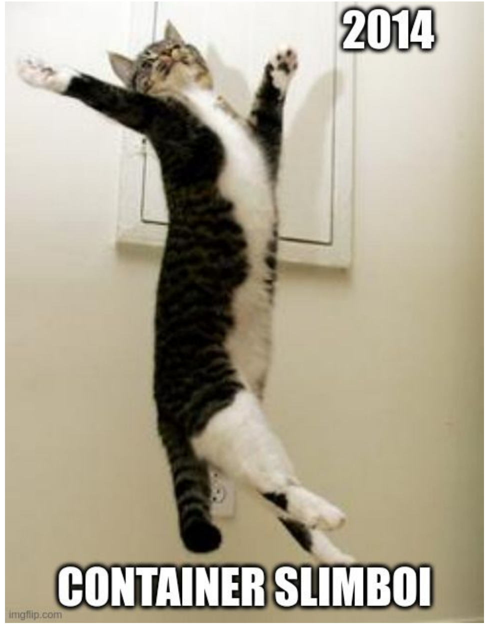


- Mean 16.58 +/- 23.66
- More outliers over the years
- Yes, people are building >> 127 layers

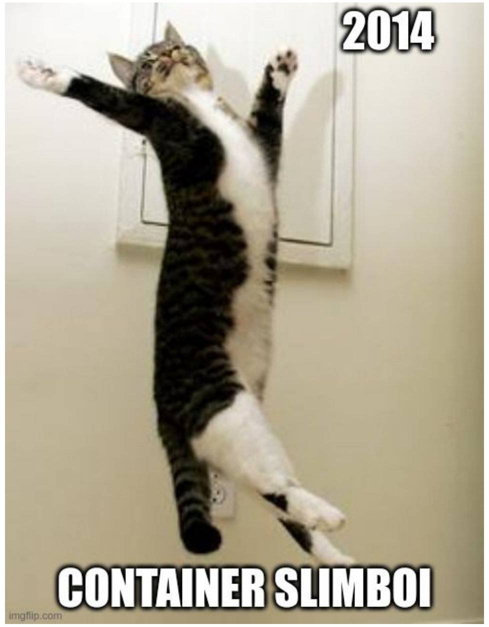
How has image size changed over time?

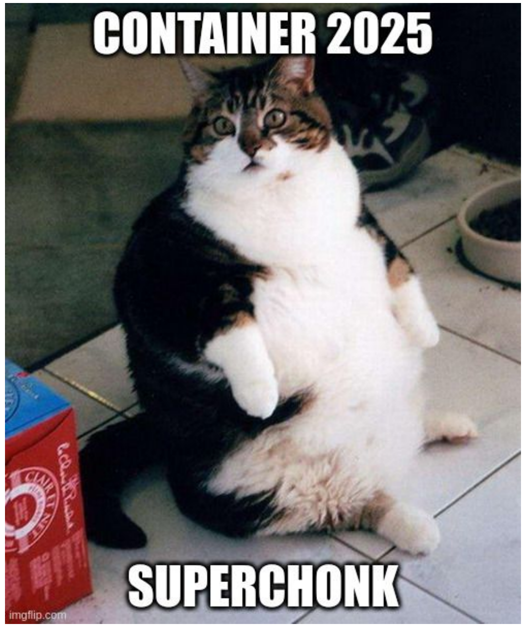
- Total size can be calculated - sum of layers
- Number of layers is relatively consistent…
- But size is trending larger

How has image size changed over time?



How has image size changed over time?





"Just build smaller containers!"

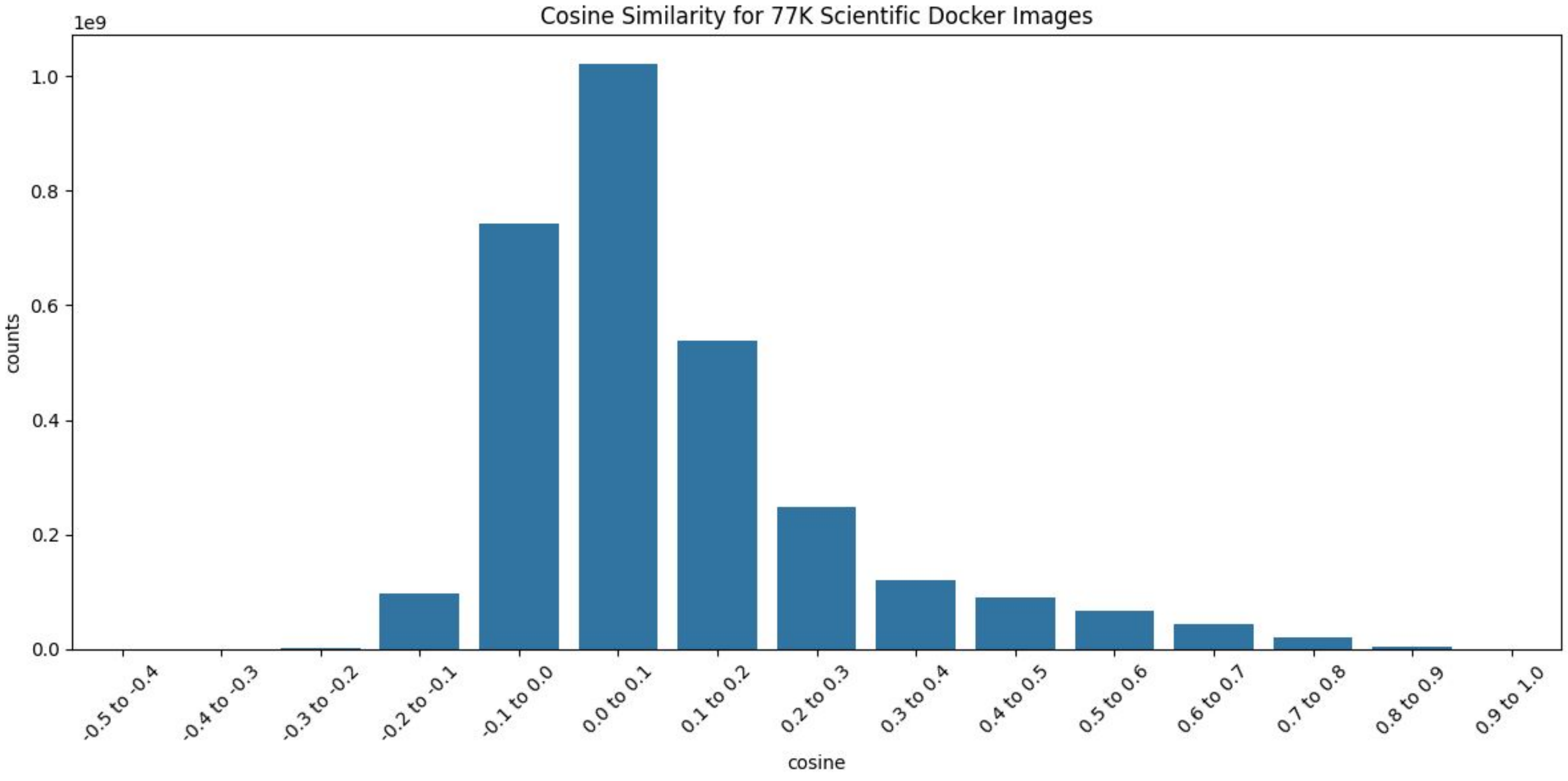
How has the container ecosystem changed since 2014?

Containers are getting larger
Layer size is relatively constant

How similar are containers since 2014?

How similar are scientific Dockerfile based on layers?

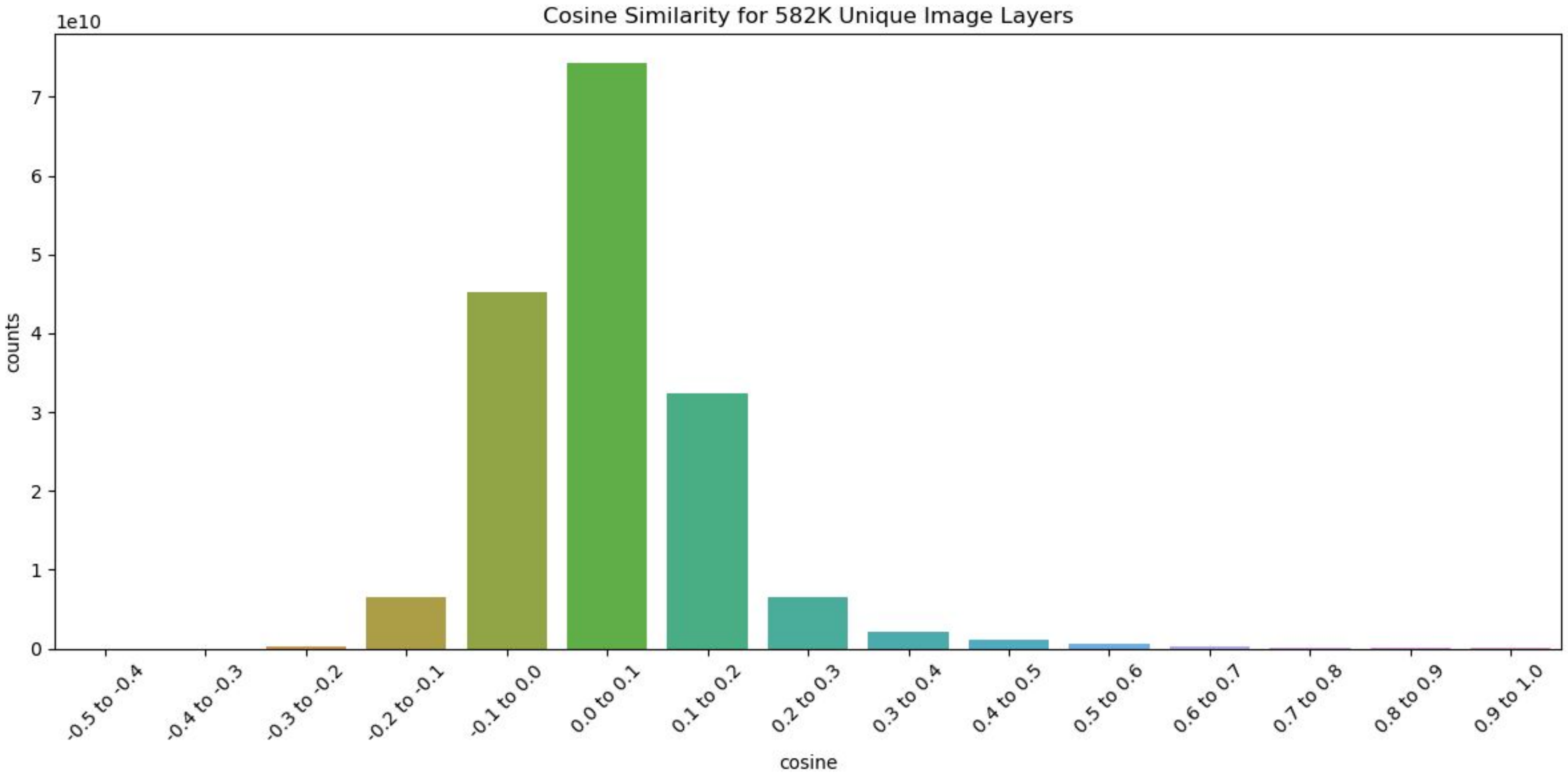
These are layers from the Dockerfile images



528K layers
Treat layers as sentences in a document
word2vec embeddings
cosine similarity

How similar are Dockerfile based on layer digests?

These are explicit layer digests (determining need to pull or not)



528K layers
Treat layers as sentences in a document
word2vec embeddings
cosine similarity

What is the most commonly used base image?

TABLE III
BASE IMAGE CLASSIFICATION

Count	Base Image
debian	393
alpine	95
ubuntu	74
centos	64
fedora	15
rockylinux	11
busybox	4

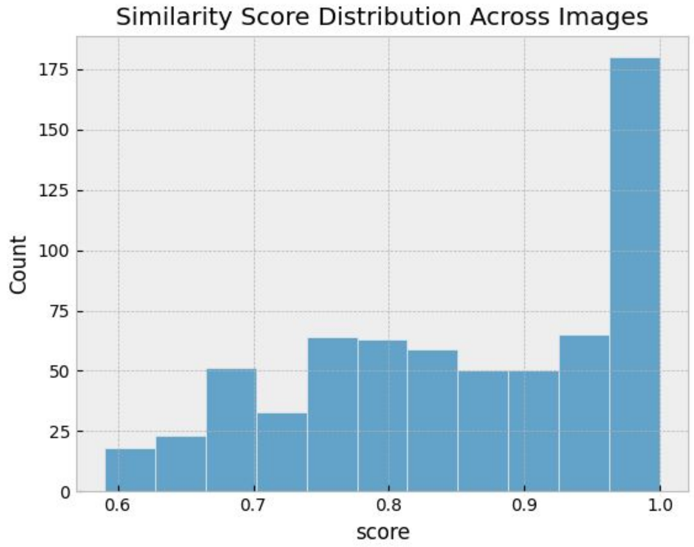
- Algorithm provided by "guts" software
- Compares each image against database of common bases
- Similarity is based on similarity of paths (Jaccaard)

What is the most commonly used base image?

TABLE III
BASE IMAGE CLASSIFICATION

Count	Base Image
debian	393
alpine	95
ubuntu	74
centos	64
fedora	15
rockylinux	11
busybox	4

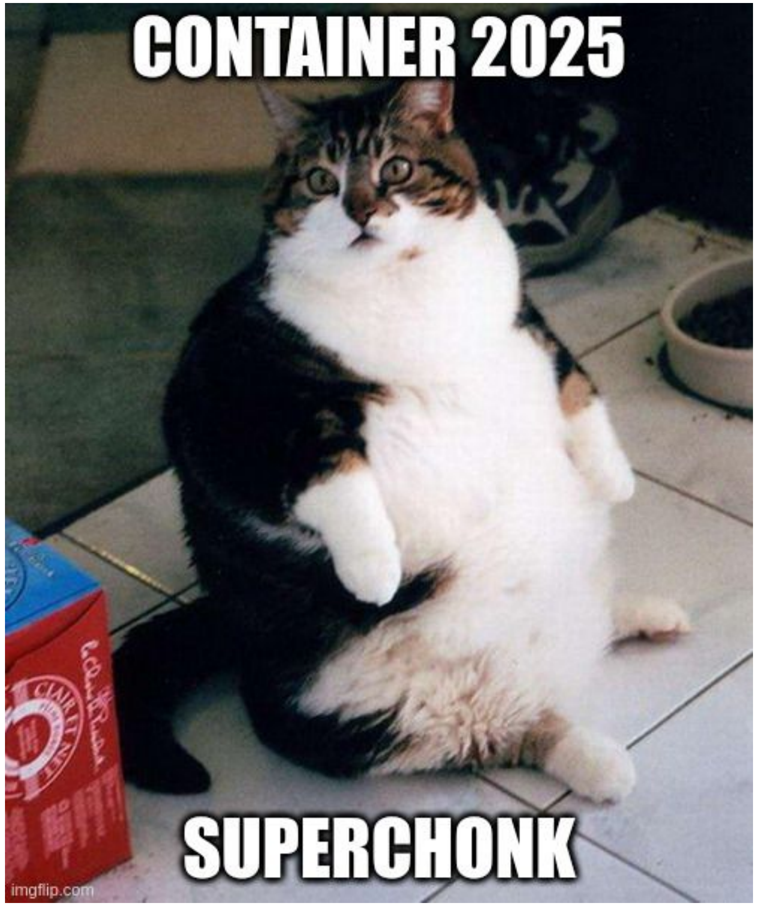
- Algorithm provided by "guts" software
- Compares each image against database of common bases
- Similarity is based on similarity of paths (Jaccaard)



How does container build strategy impact similarity?

How does container build strategy impact similarity?

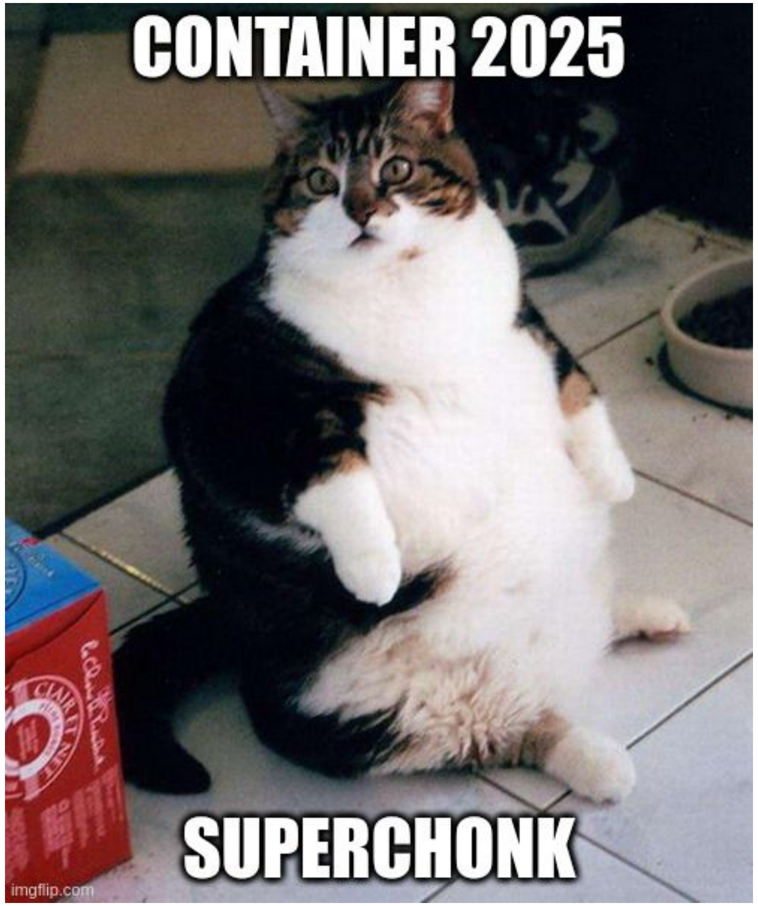
1. More similar containers mean redundancy of layers, and less space used on the filesystem and pull time



How does container build strategy impact similarity?

1. More similar containers mean redundancy of layers, and less space used on the filesystem and pull time

1) Reasonable effort to create redundancy
   - Real performance study containers

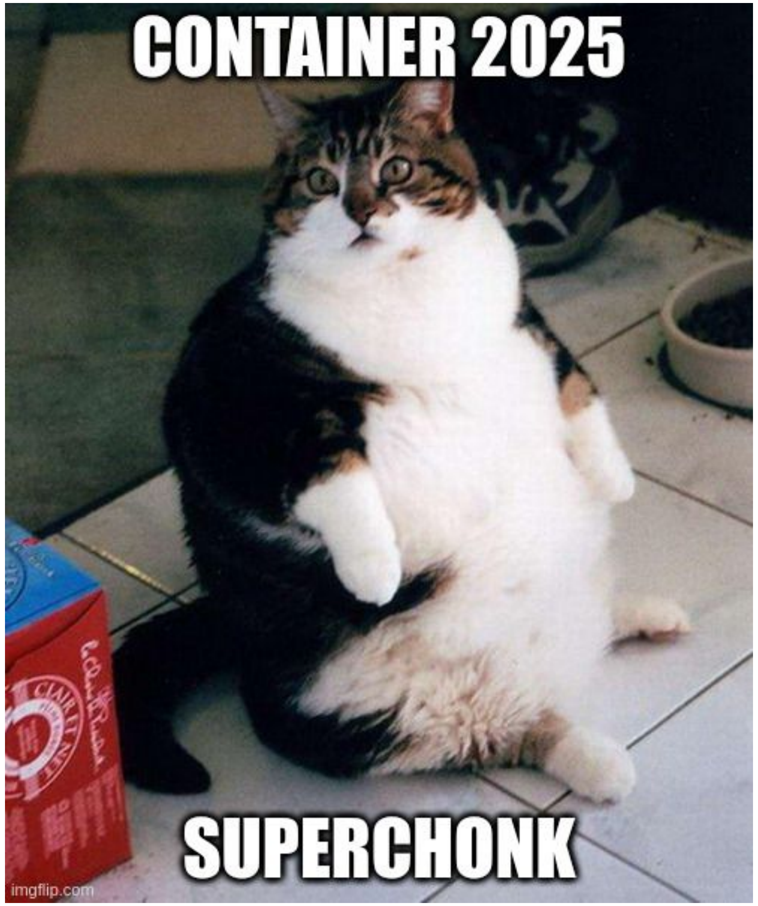


How does container build strategy impact similarity?

1. More similar containers mean redundancy of layers, and less space used on the filesystem and pull time

1) Reasonable effort to create redundancy
   - Real performance study containers

2) Best effort to create redundancy
   - Best effort builds of the same



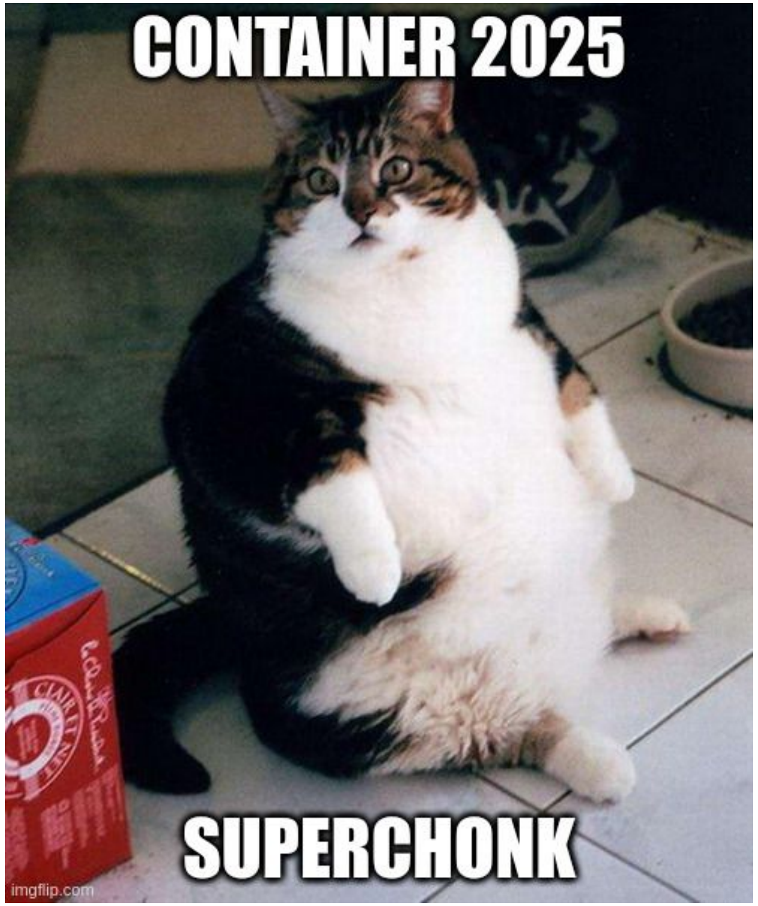
How does container build strategy impact similarity?

1. More similar containers mean redundancy of layers, and less space used on the filesystem and pull time

1) Reasonable effort to create redundancy
   - Real performance study containers

2) Best effort to create redundancy
   - Best effort builds of the same

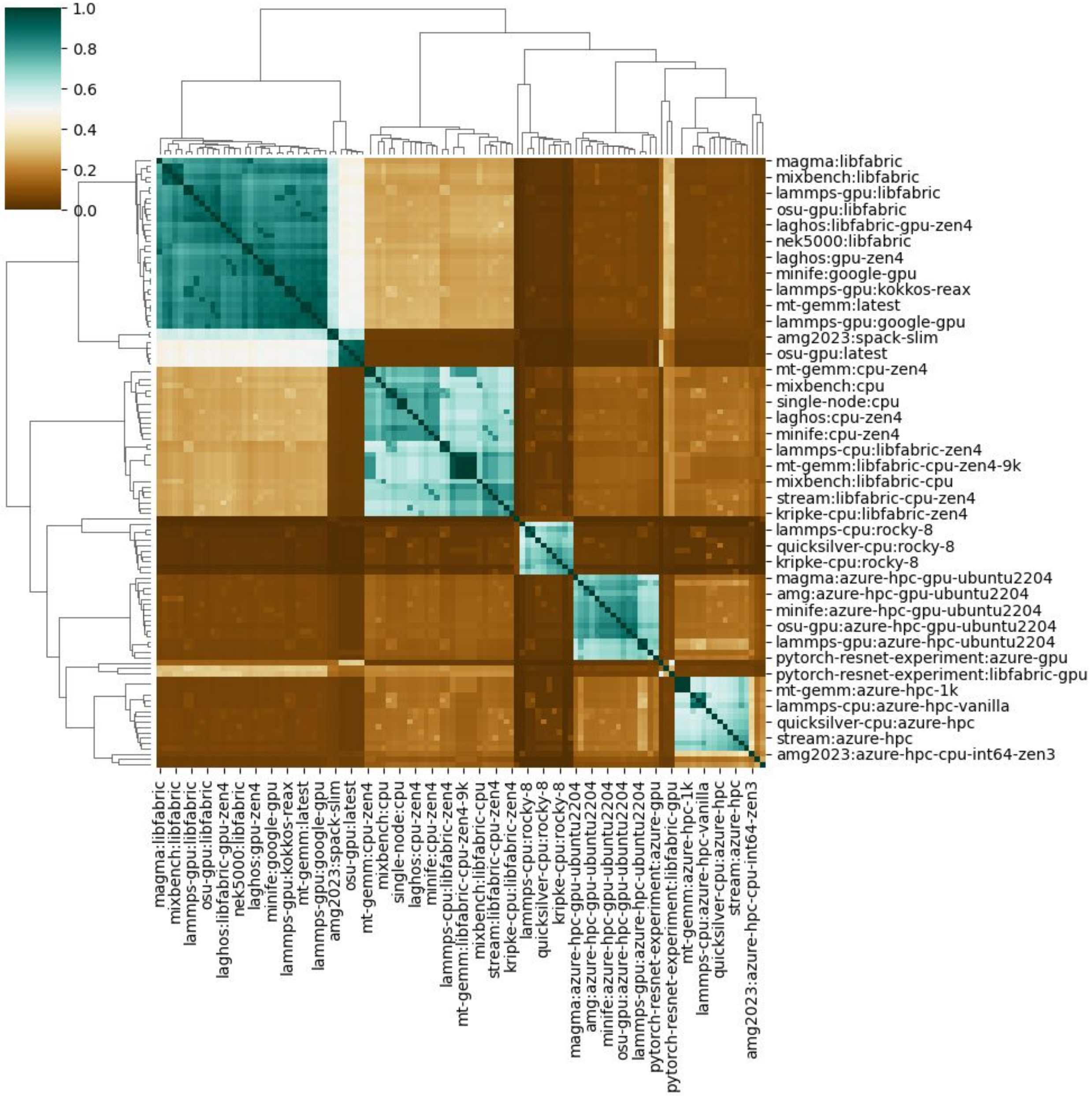
3) Little effort to create redundancy
   - High redundancy (spack)



How does container build strategy impact similarity?

Let's first look at containers from a real performance study

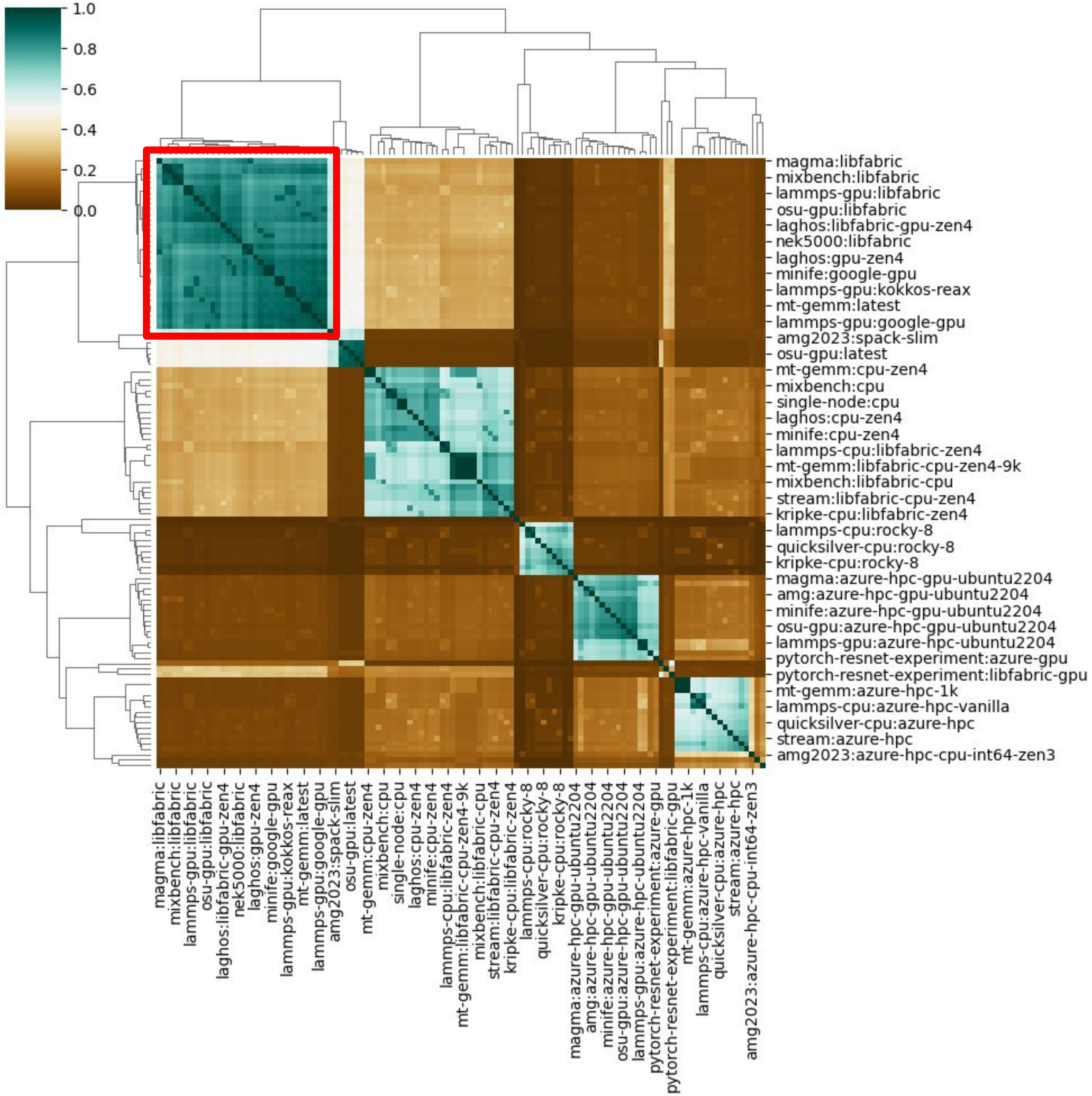
Performance Study Containers



Performance Study Containers

Which container set?

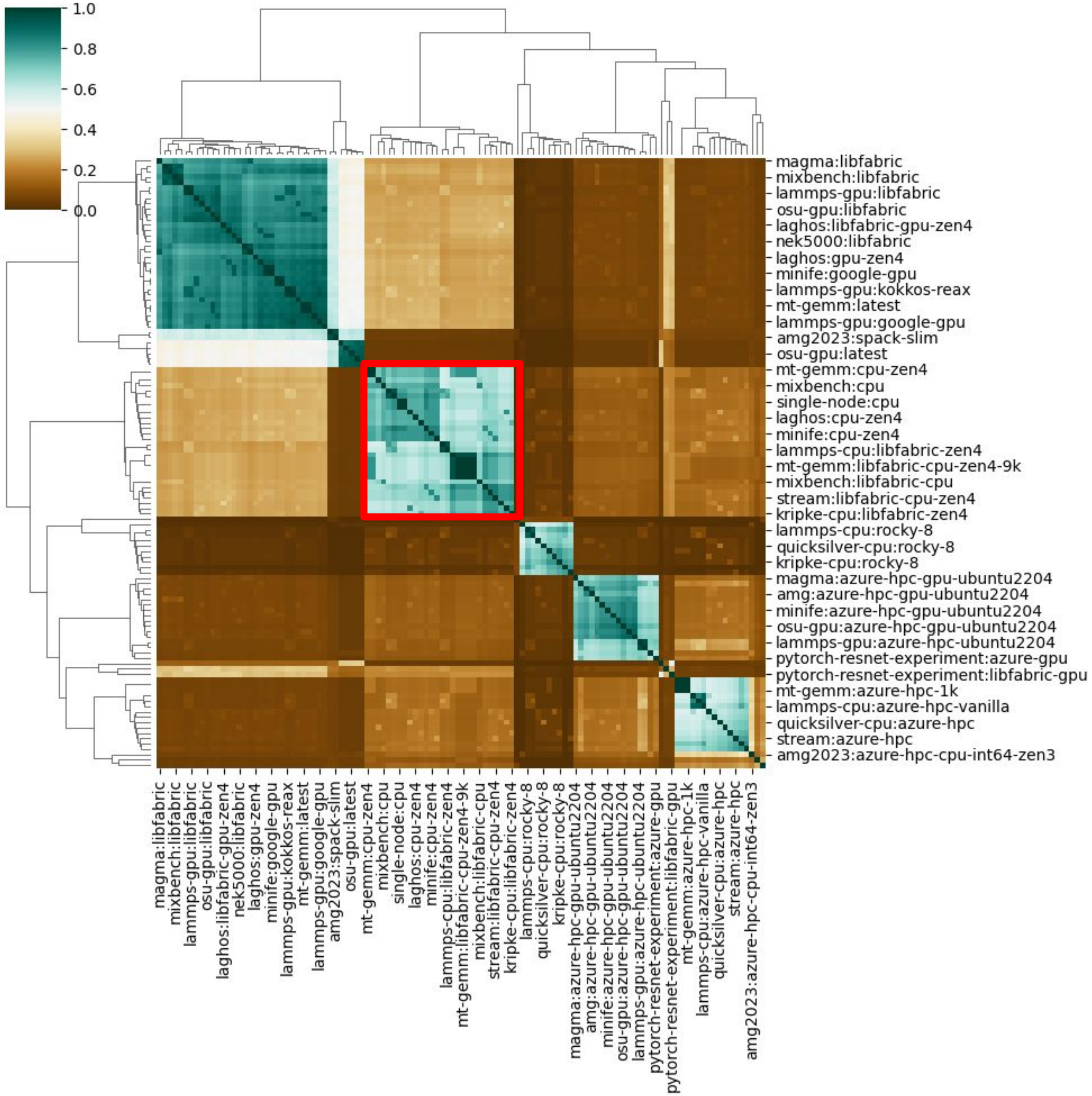
- AWS and Google GPU containers



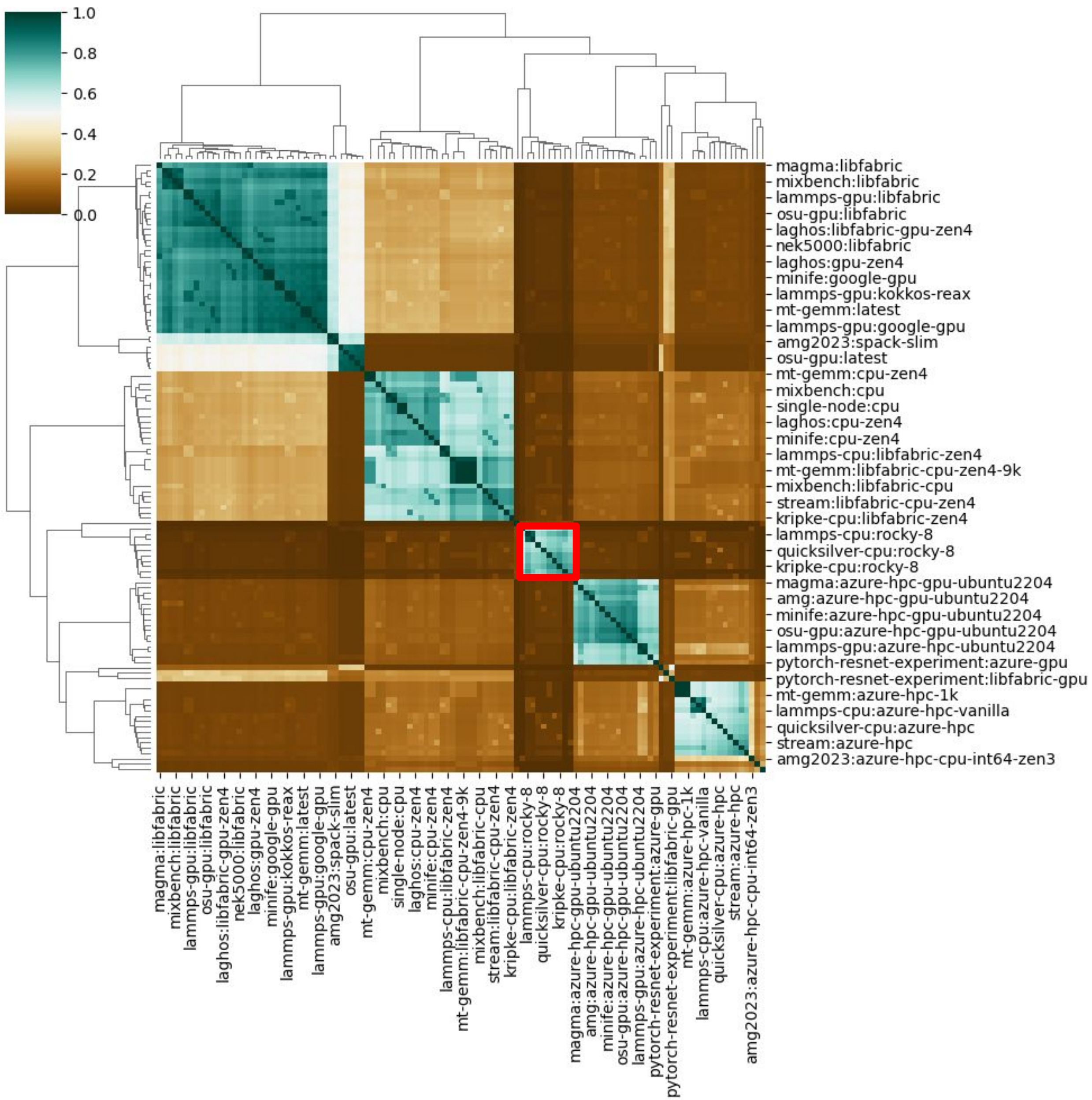
Performance Study Containers

Which container set?

- AWS and Google GPU containers
- AWS and Google CPU containers



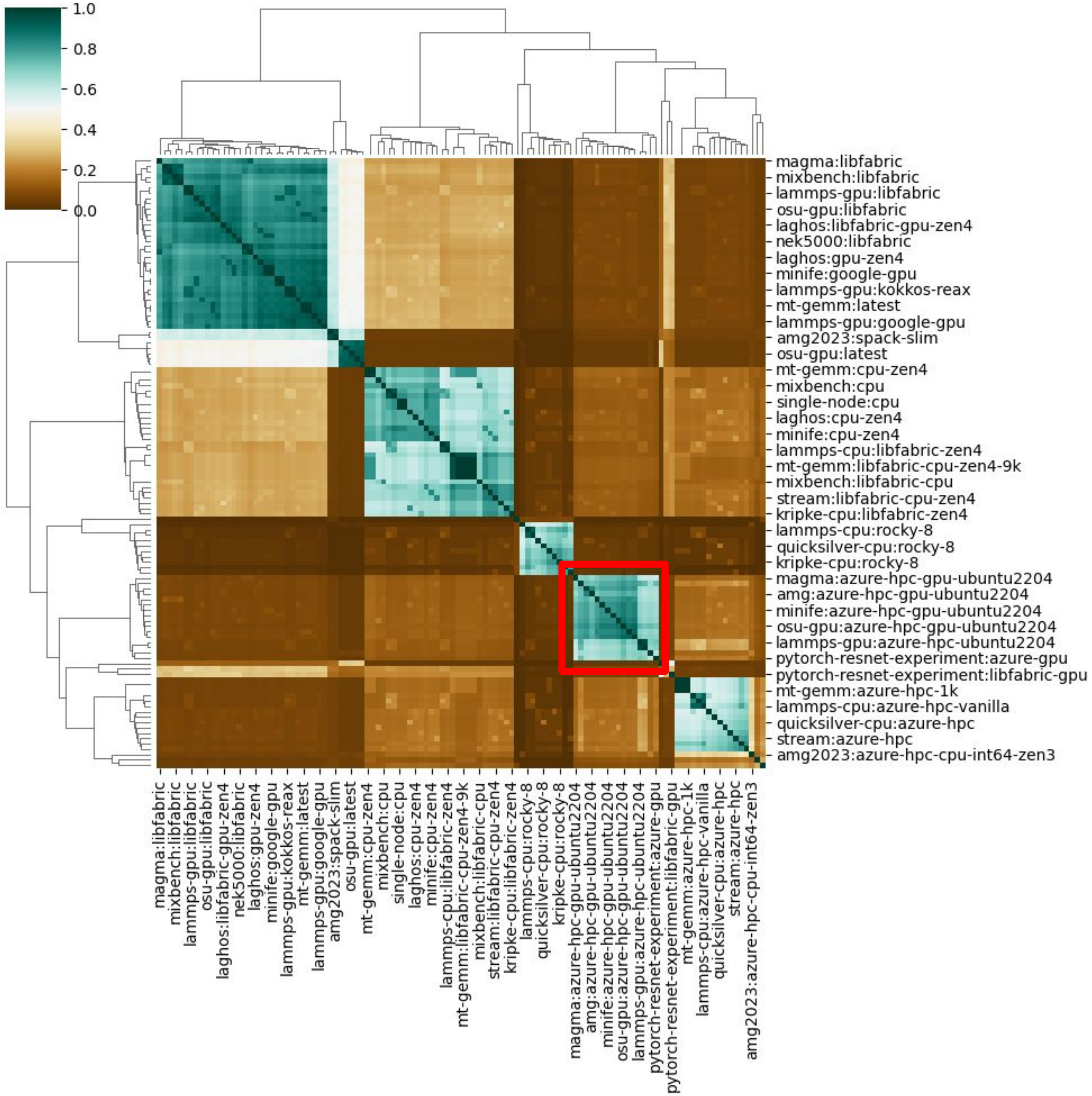
Performance Study Containers



Which container set?

- AWS and Google GPU containers
- AWS and Google CPU containers
- Rocky bases for Compute Engine

Performance Study Containers



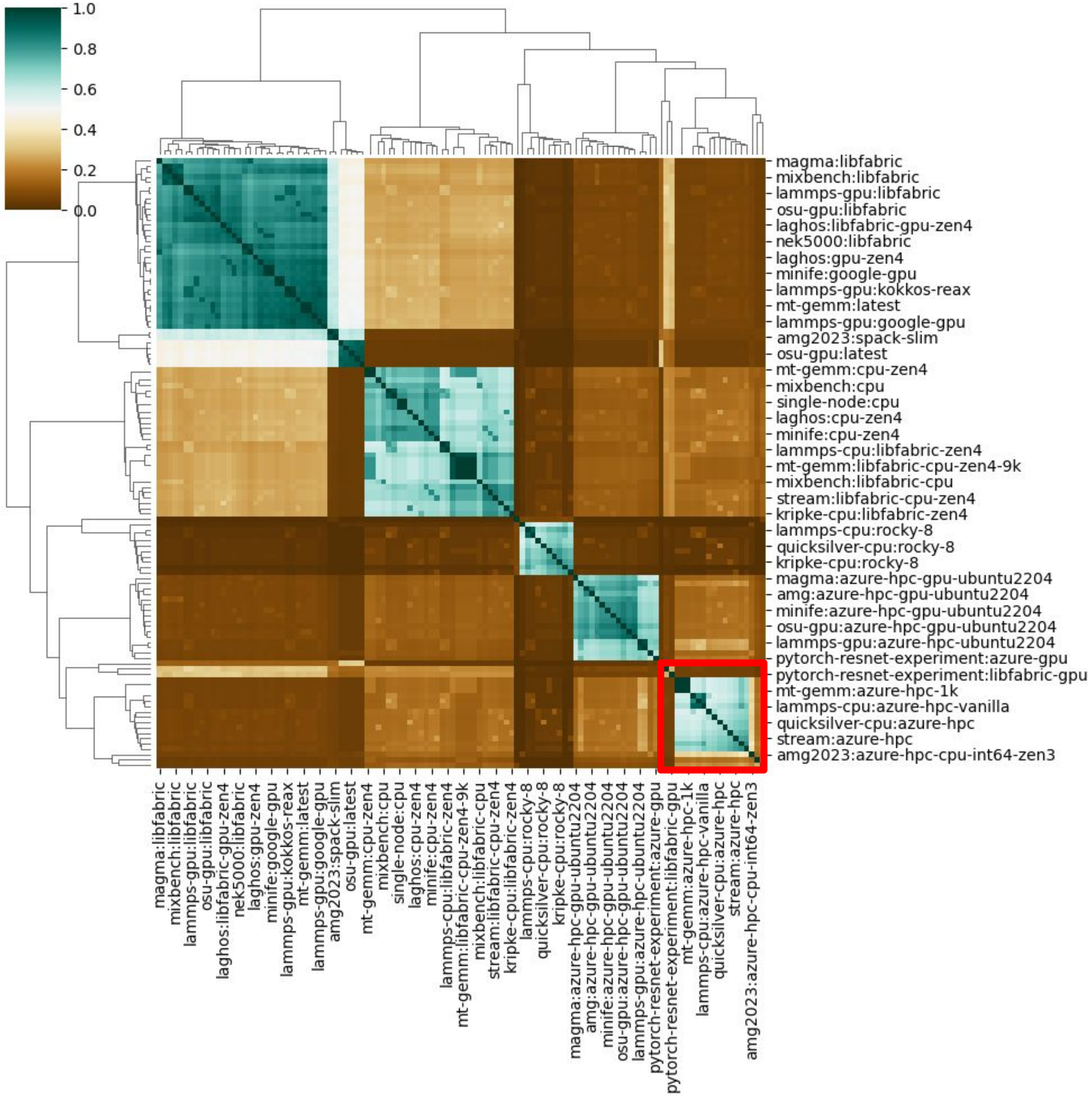
Which container set?

- AWS and Google GPU containers
- AWS and Google CPU containers
- Rocky bases for Compute Engine
- Azure GPU

Performance Study Containers

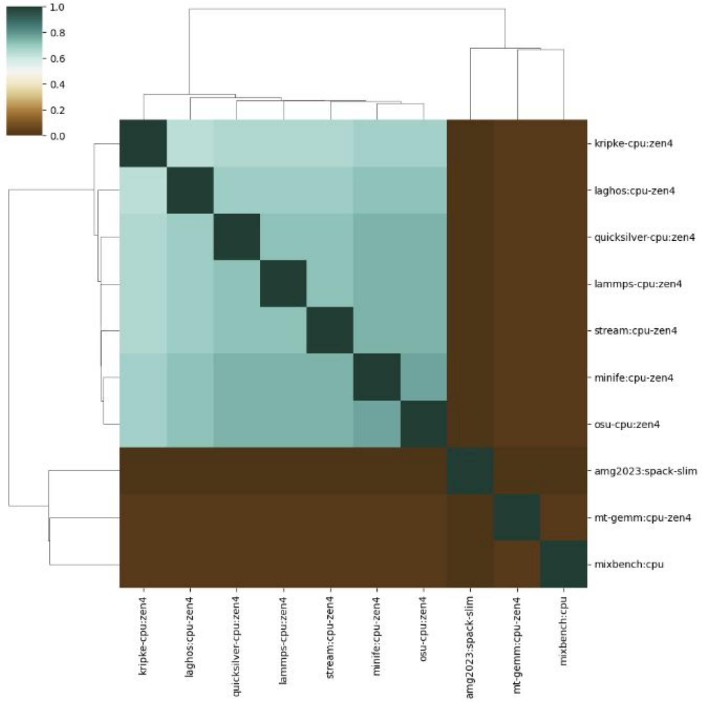
Which container set?

- AWS and Google GPU containers
- AWS and Google CPU containers
- Rocky bases for Compute Engine
- Azure GPU
- Azure CPU

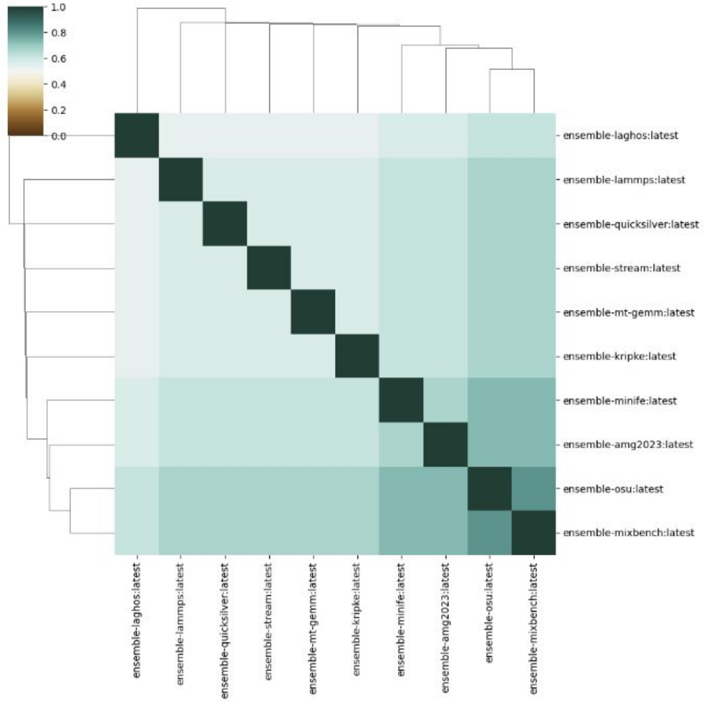


How does container build strategy impact similarity?
Now let's take a slice of that set (from one cloud)

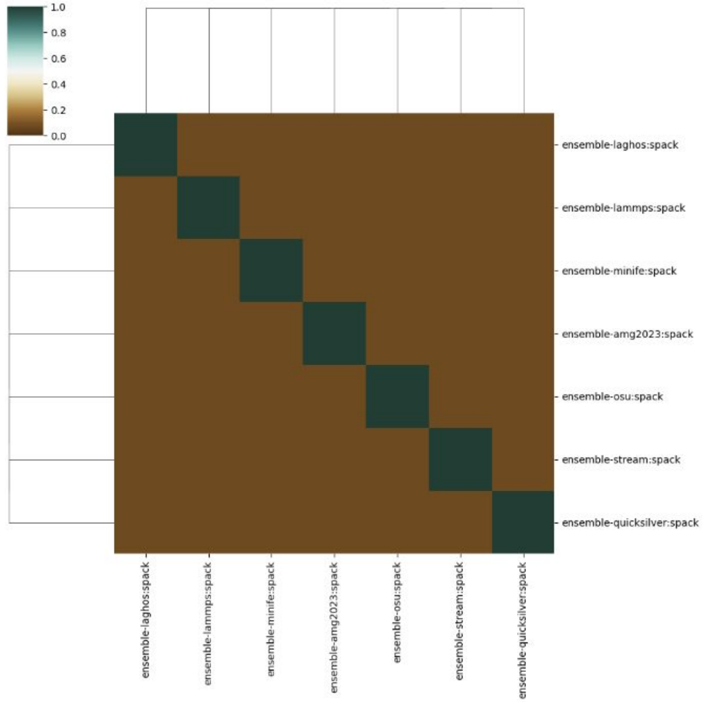
Build strategy influences container similarity



Real performance study containers



Best effort of same containers

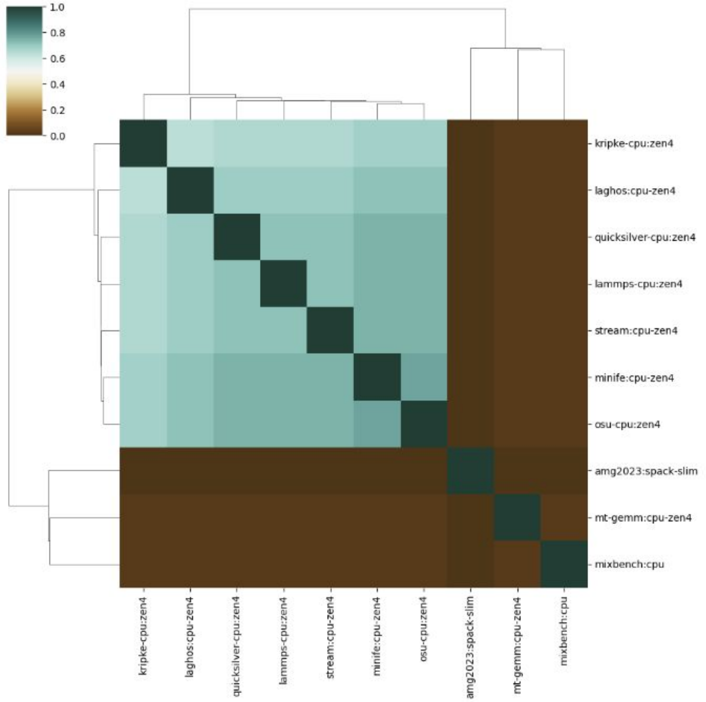


Spack

SIMILARITY OF CONTAINER SETS BASED ON BUILD STRATEGY

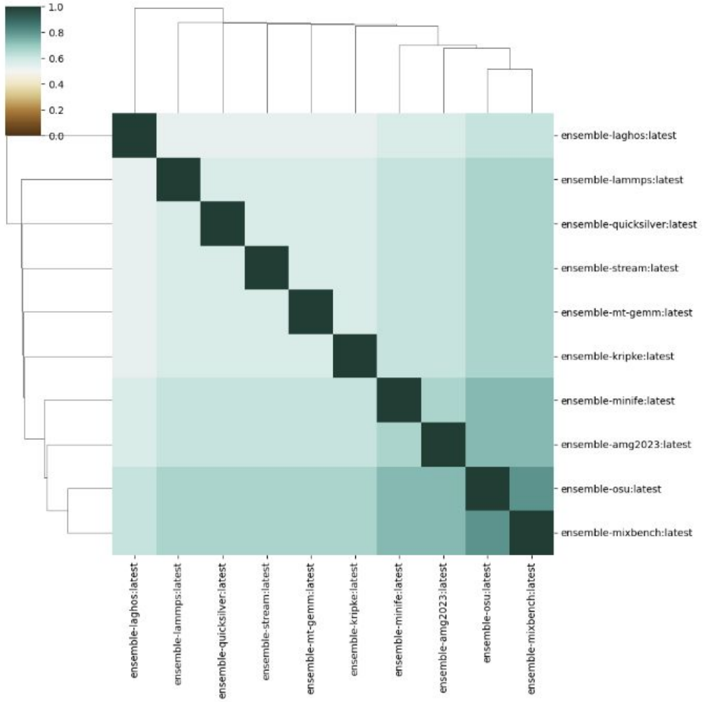
Container Set	Total Layers	Unique URIs	Unique Containers	Unique Layer Digests	Jacaard Similarity (mean and s.d)
Performance Study	258	10	10	115	0.40 (0.38)
Best Effort for Redundancy	128	10	10	33	0.66 (0.128)
Low Redundancy Builds (spack)	56	7	7	50	0.2 (0.33)

The number of unique layer pulls per strategy:



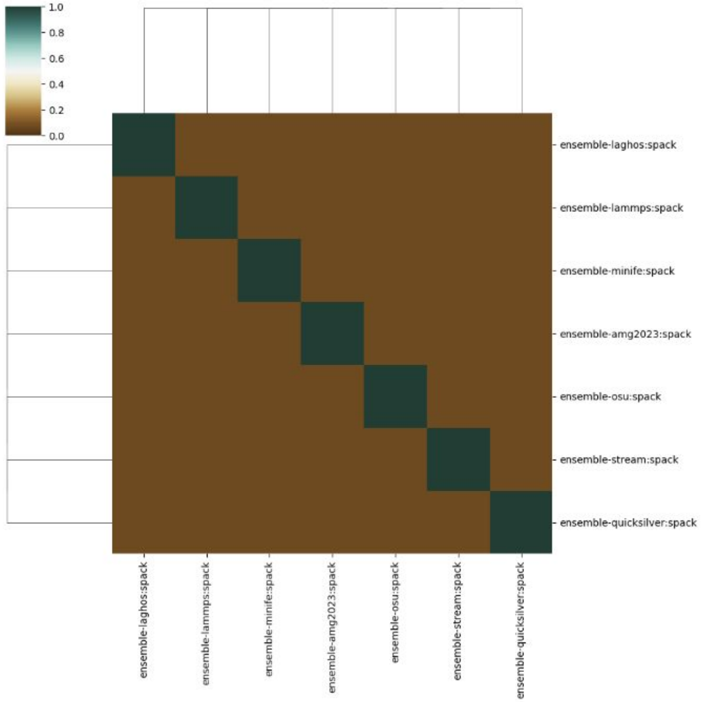
Real performance study containers

45% of layers are unique pulls



Best effort of same containers

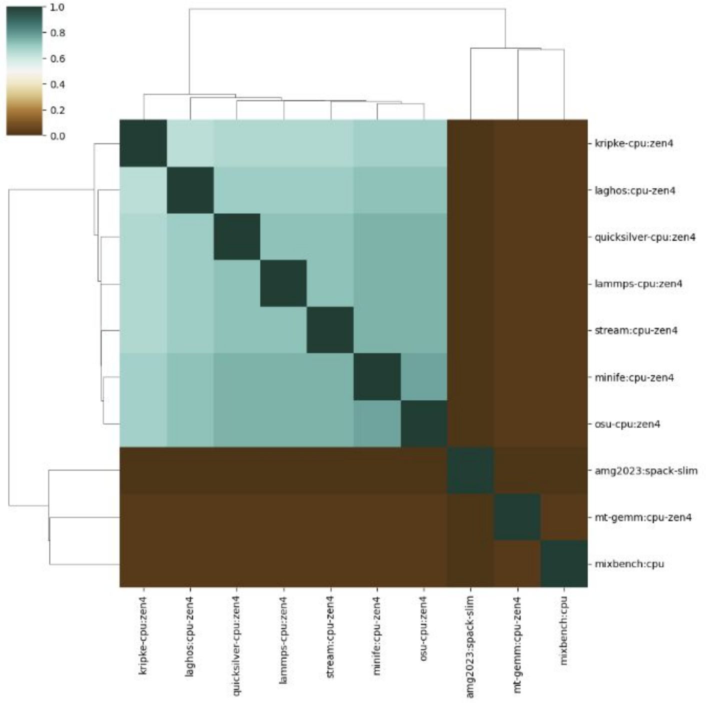
28% of layers are unique pulls



Spack

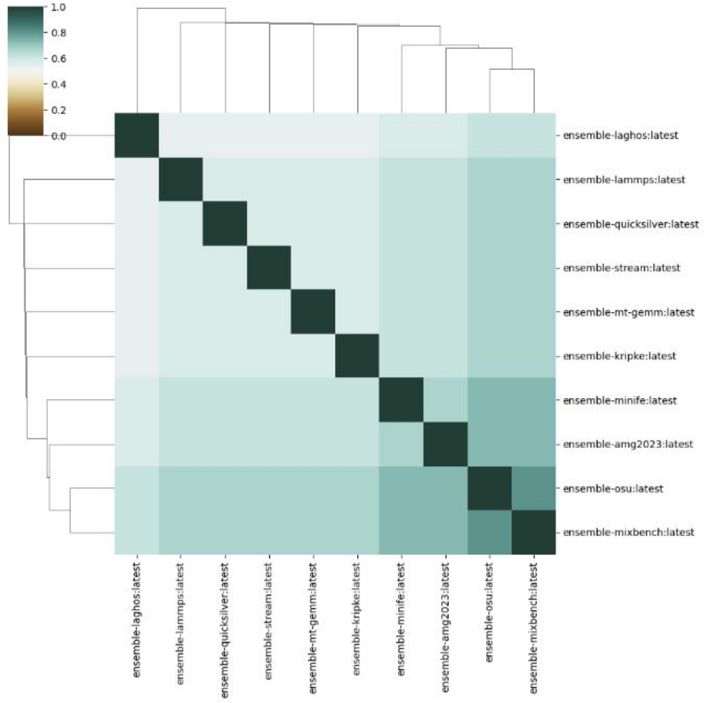
89% of layers are unique pulls

The number of unique layer pulls per strategy:



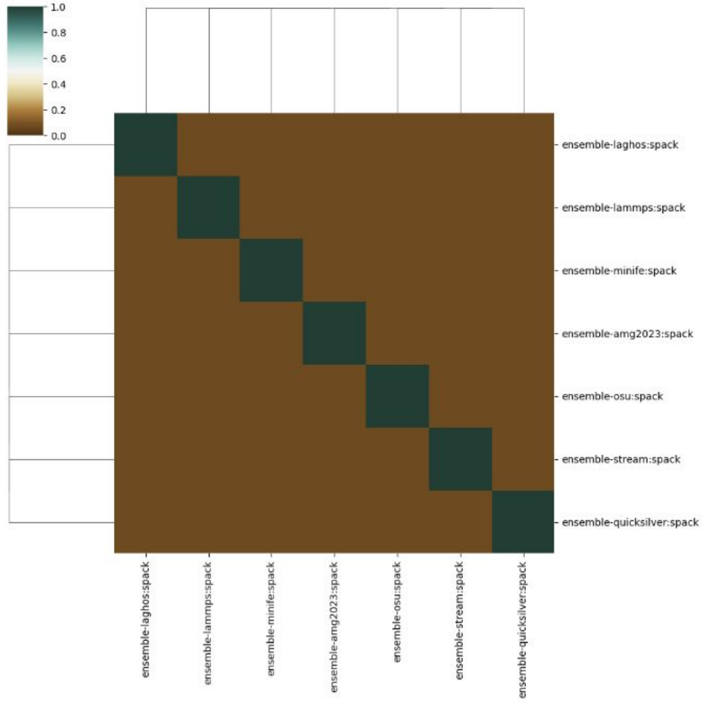
Real performance study containers

45% of layers are unique pulls



Best effort of same containers

28% of layers are unique pulls



Spack

89% of layers are unique pulls

How does container build strategy impact similarity?

Redundancy of layers increases similarity

What about best practices?

Are people using multi-stage builds?

```
# Dockerfile
# build stage
FROM buildbase as build
...
...
...

# production ready stage
FROM runbase
...
COPY --from=build
/artifact /app
```

- Look for more than one FROM in our database
- We find 2.56% of image builds use multi-build strategy

Are people using docker "official" images?

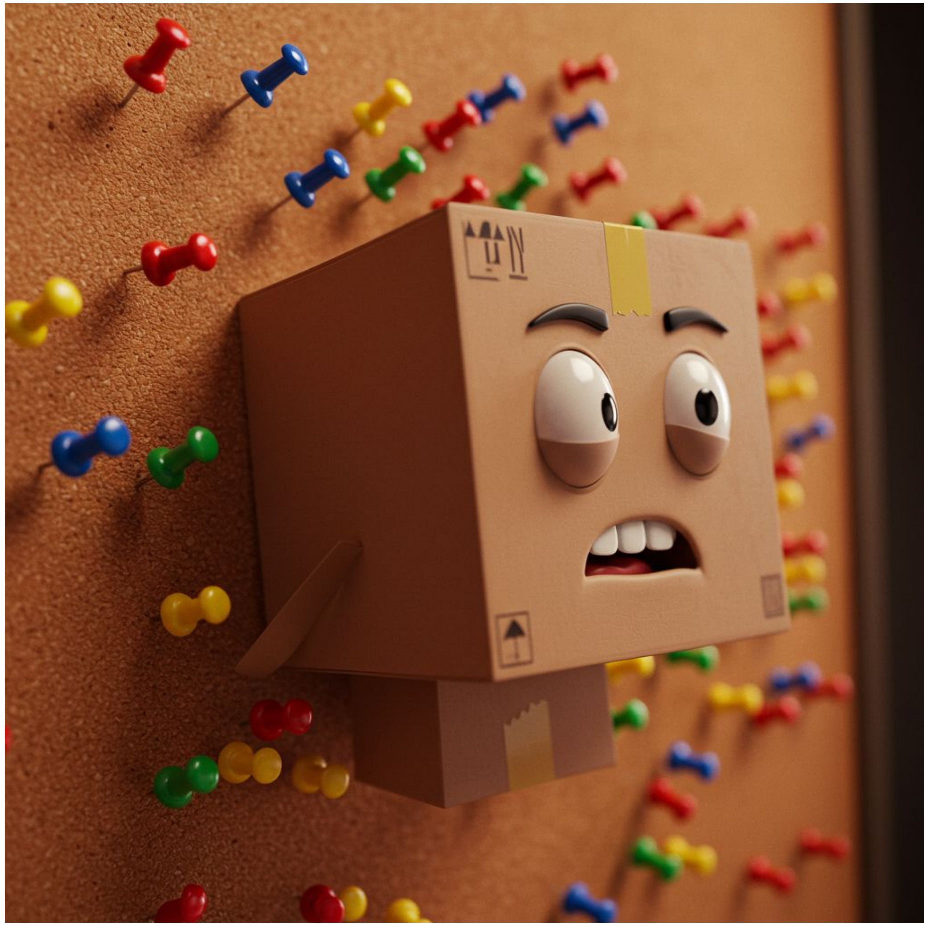
- Look at FROM directive
- 14.77% of image base are from Docker Hub

Are people using the "latest" tags?

- This is considered a bad practice (moving target)
- We can look at the FROM directive tag
- 5.3% of images use latest

Are people using pinned image digests?

- This guarantees an exact build (version)
- Comes at the cost of security updates
- We can look for a sha256 instead of a tag
- Only 0.09 (less than 1%) found



apt-get and install in the same line?

- 507,695 layers use apt-get
- Of that set, 94.3% also have apt-get install
- Of that set, 67.8% do a clean too

apt-get and install in the same line?

```
	if [ -d "$rootfsDir/etc/apt/apt.conf.d" ]; then
		# _keep_ us lean by effectively running "apt-get clean" after every install
		aptGetClean='"rm -f /var/cache/apt/archives/*.deb /var/cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true";'
		echo >&2 "+ cat > '$rootfsDir/etc/apt/apt.conf.d/docker-clean'"
		cat > "$rootfsDir/etc/apt/apt.conf.d/docker-clean" <<-EOF
			# Since for most Docker users, package installs happen in "docker build" steps,
			# they essentially become individual layers due to the way Docker handles
			# layering, especially using CoW filesystems.  What this means for us is that
			# the caches that APT keeps end up just wasting space in those layers, making
			# our layers unnecessarily large (especially since we'll normally never use
			# these caches again and will instead just "docker build" again and make a brand
			# new image).

```

What about best practices?

People often don't follow them, but best that the tooling implements them.

What is more important, image size or number of layers?

Does the number of layers *matter at all?*

I built a simulation tool "container-crafter" for pulling studies

```yaml
# URI is the base or root to build
uri: ghcr.io/converged-computing/container-chonks-run1

# Sizes are in bytes, the total size for the container
sizes:
  - total: 53702097      # 25th
  - total: 58049507.8    # 30th
  - total: 71460665.0    # 35th
  - total: 91388866.2    # 40th
  - total: 108513992.4   # 45th
  - total: 132399102     # 50th
  - total: 163049655.0   # 55th
  - total: 218665412.8   # 60th
  - total: 271728773.4   # 65th
  - total: 320018606.2   # 70th
  - total: 392602448     # 75th
  - total: 496514346.8   # 80th
  - total: 687439577.6   # 85th
  - total: 1181249324.6  # 90th
  - total: 2775722493.4  # 95th
  - total: 6841726027.3
  - total: 10907729561.2 # range between the two
  - total: 14973733095.1
  - total: 19039736629   # 100th

# Layers are the number of layers to do for each size
# We will do the median and the extreme (max)
# https://github.com/moby/moby/blob/4001d0704ba38a82e1dbc26f0593fca66db1cb98/layer/layer_store.go#L28
layers:
  - exact: 9
  - exact: 125
```

- A config file is used to build mock containers.
- We control the layer count; total image size
- Each layer is guaranteed to be unique
- The tool will build to a specific URI
- Each layer only allowed up to 10GB

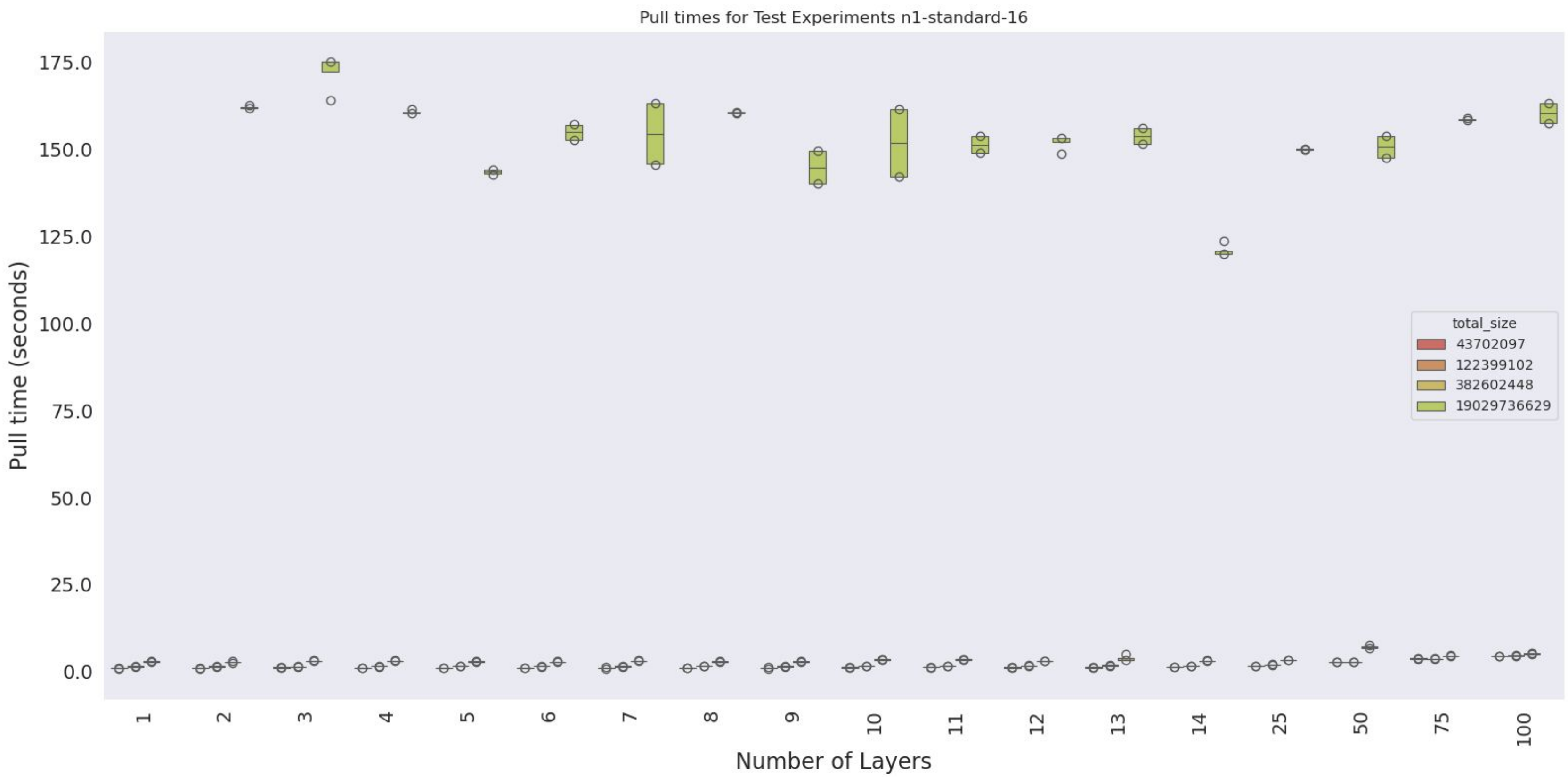
I built a simulation tool "container-crafter" for pulling studies

TABLE I
IMAGE SIZES CHOSEN FOR PULLING STUDY

Image Size (bytes)	Human readable	Percentile from Database
53702097.0	(53.7 MB)	25th
58049507.8	(58.05 MB)	30th
71460665.0	(71.46 MB)	35th
91388866.2	(91.39 MB)	40th
108513992.4	(108.51 MB)	45th
132399102.0	(132.4 MB)	50th
163049655.0	(163.05 MB)	55th
218665412.8	(218.67 MB)	60th
271728773.4	(271.73 MB)	65th
320018606.2	(320.02 MB)	70th
392602448.0	(392.60 MB)	75th
496514346.8	(496.51 MB)	80th
687439577.6	(687.44 MB)	85th
1181249324.6	(1.18 GB)	90th
2775722493.4	(2.78 GB)	95th
6841726027.3	(6.84 GB)	96.25th
10907729561.2	(10.91 GB)	97.5th
14973733095.1	(14.97 GB)	98.75th
19039736629.0	(19.04 GB)	100th

Sizes chosen at percentile increments of 5 derived from the real data, with the exception of the 95th-100th percentile that was broken into an additional set of three ranges.

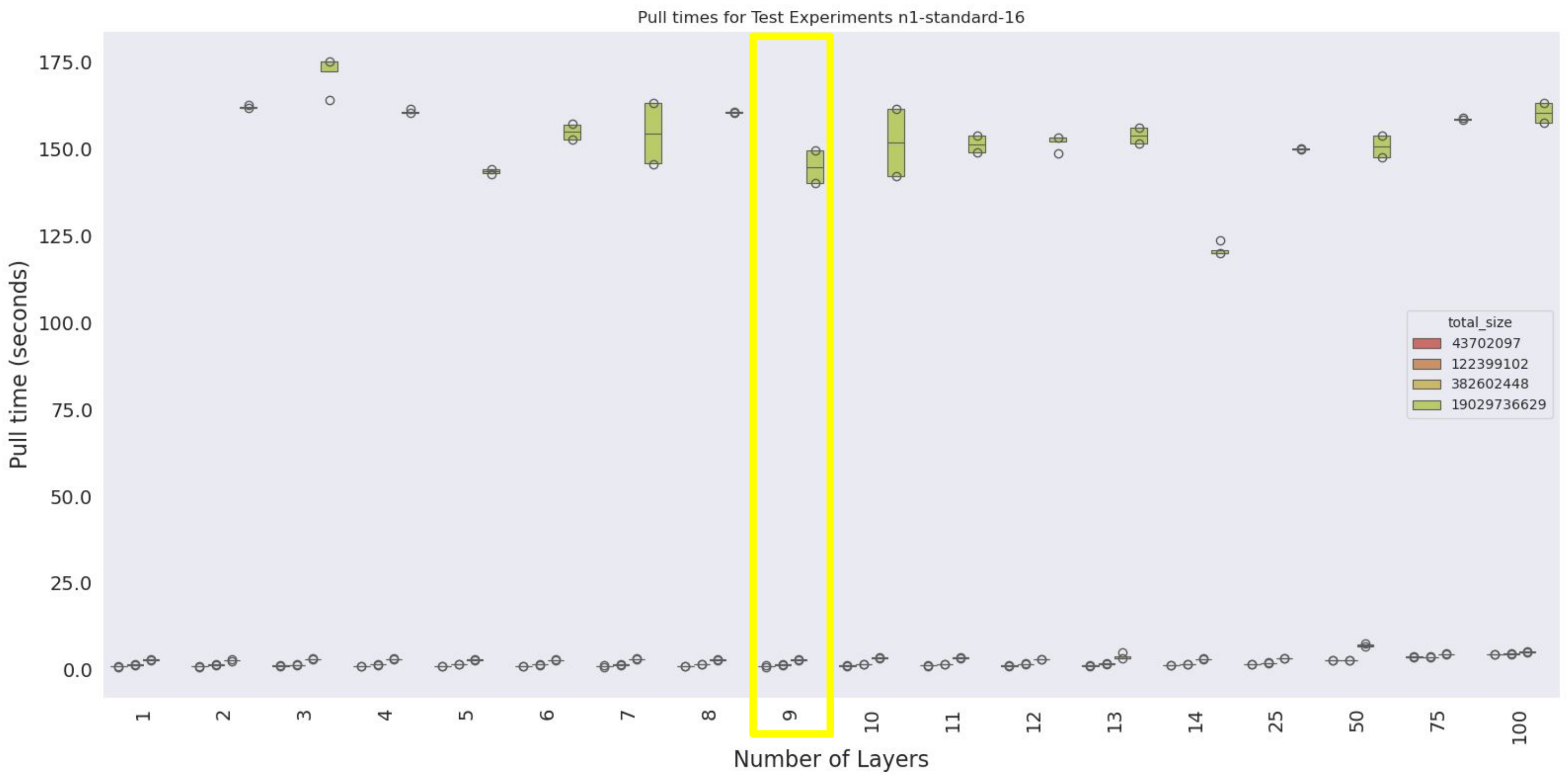
What matter is total image size, not number of layers



Sizes between 14MB-19GB

The same total size split across 1-100 layers takes the same amount of time.
What explodes pulling time is just the total size of the image.
The number of layers largely doesn't matter.

For the study, use a value that reflects actual practice



For further study, I chose sizes 9 (median of the dataset) and max 125

The same total size split across 1-100 layers takes the same amount of time.
What explodes pulling time is just the total size of the image.
The number of layers largely doesn't matter.

What is more important, image size or number of layers?

Image size!

What is the best strategy for container pulling?

Cloud Pulling Study

- Google Kubernetes Engine (GKE)
- 16 vCPU, 60GB RAM / node
- Node (cluster) sizes 4, 8, 32, 64, 128, and 256

Cloud Pulling Study

- Google Kubernetes Engine (GKE)
- 16 vCPU, 60GB RAM / node
- Node (cluster) sizes 4, 8, 32, 64, 128, and 256

n1-standard-64 was only 1.028x faster, but 3.87x more expensive

For each container (size and layers):

A Job will be created to pull the container

Kubernetes Event Exporter used to collect all events

Cloud Pulling Study

- Google Kubernetes Engine (GKE)
- 16 vCPU, 60GB RAM / node
- Node (cluster) sizes 4, 8, 32, 64, 128, and 256

n1-standard-64 was only 1.028x faster, but 3.87x more expensive

For each container (size and layers):

A Job will be created to pull the container

Kubernetes Event Exporter used to collect all events

Each experiment will be conducted several times to assess a setup

Strategies for optimized container pulling in Kubernetes

- Use a local (cloud provided) registry
- Use a solid state drive (SSD) instead of persistent disk (HDD)
- Use image streaming (SOCI Snapshotter and similar)
- Use zstandard compression (greater than 3x faster than gzip)
- Preload images onto nodes (using a Daemonset)?

Strategies for optimized container pulling in Kubernetes

- Use a local (cloud provided) registry (pulling latency)
- Use a solid state drive (SSD) instead of persistent disk (HDD) (FS latency)
- Use image streaming (SOCI Snapshotter and similar)
- Use zstandard compression (greater than 3x faster than gzip)
- Preload images onto nodes (using a Daemonset)?

Strategies for optimized container pulling in Kubernetes

- Use a local (cloud provided) registry (pulling latency)
- Use a solid state drive (SSD) instead of persistent disk (HDD) (FS latency)
- Use image streaming (SOCI Snapshotter and similar)

1. First test with containers generated from simulation tool.
2. Then use real-world application containers.

Strategies for optimized container pulling in Kubernetes

- Use a local (cloud provided) registry (pulling latency)
- Use a solid state drive (SSD) instead of persistent disk (HDD) (FS latency)
- Use image streaming (SOCI Snapshotter and similar)

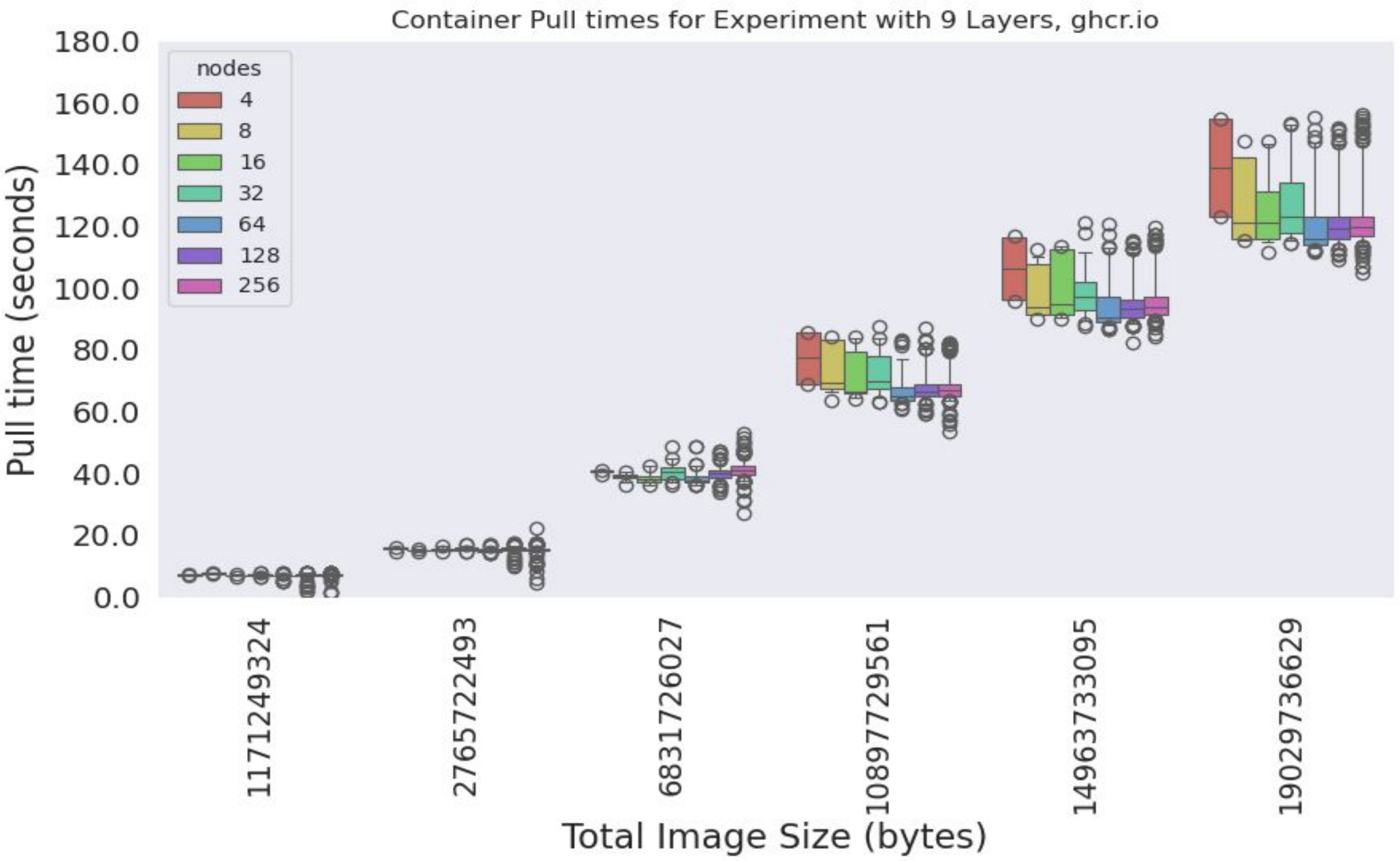
1. First test with containers generated from simulation tool.
2. Then use real-world application containers.
3. Test node coordination

What is the best strategy for container pulling?

Let's look at the results!

Does pulling from a local registry improve pull times?

No, not really



ghcr.io

Does pulling from a local registry improve pull times?

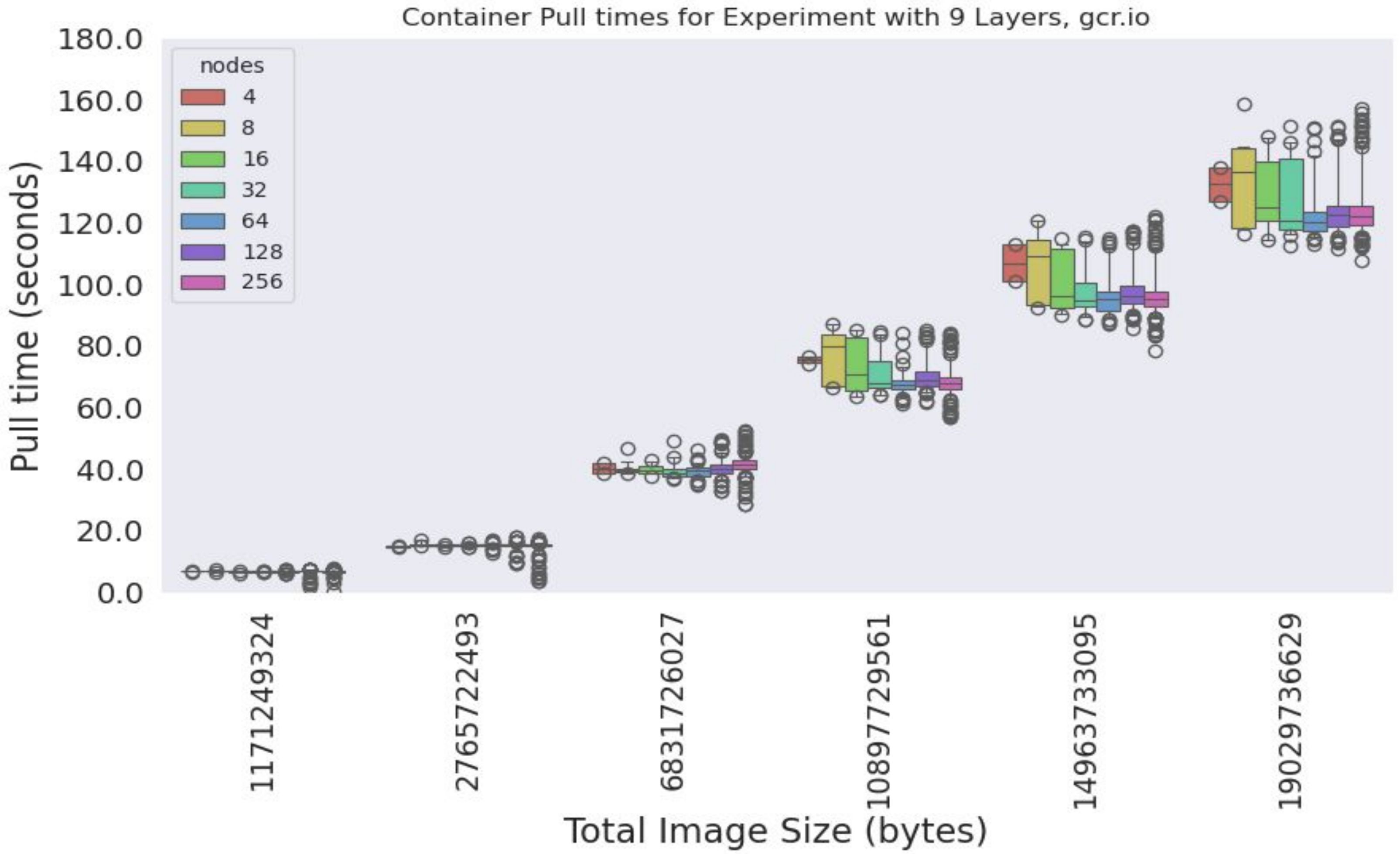
No, not really

ghcr.io

Pull time does not increase for larger clusters!

Does pulling from a local registry improve pull times?

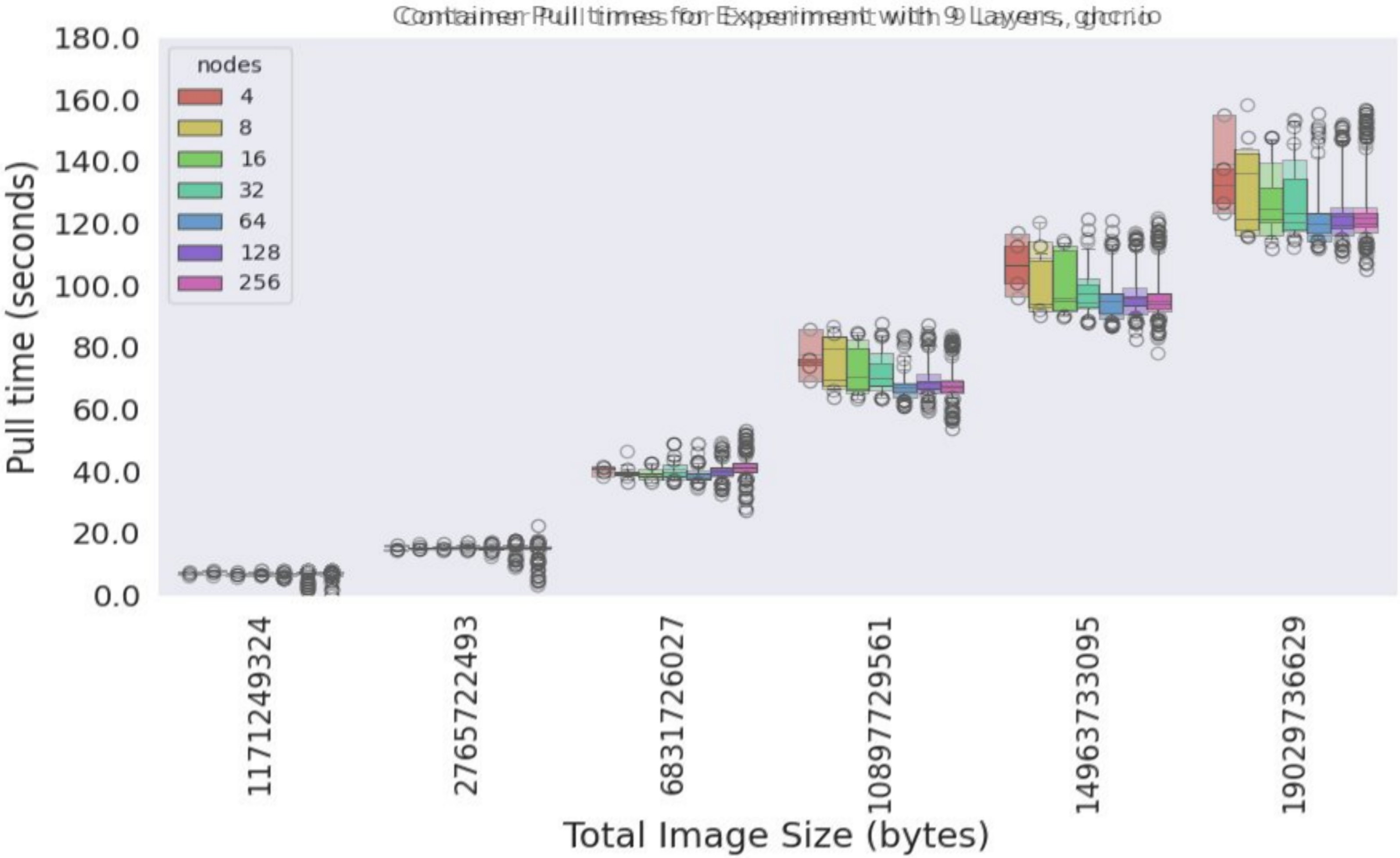
No, not really



gcr.io

Does pulling from a local registry improve pull times?

No, not really

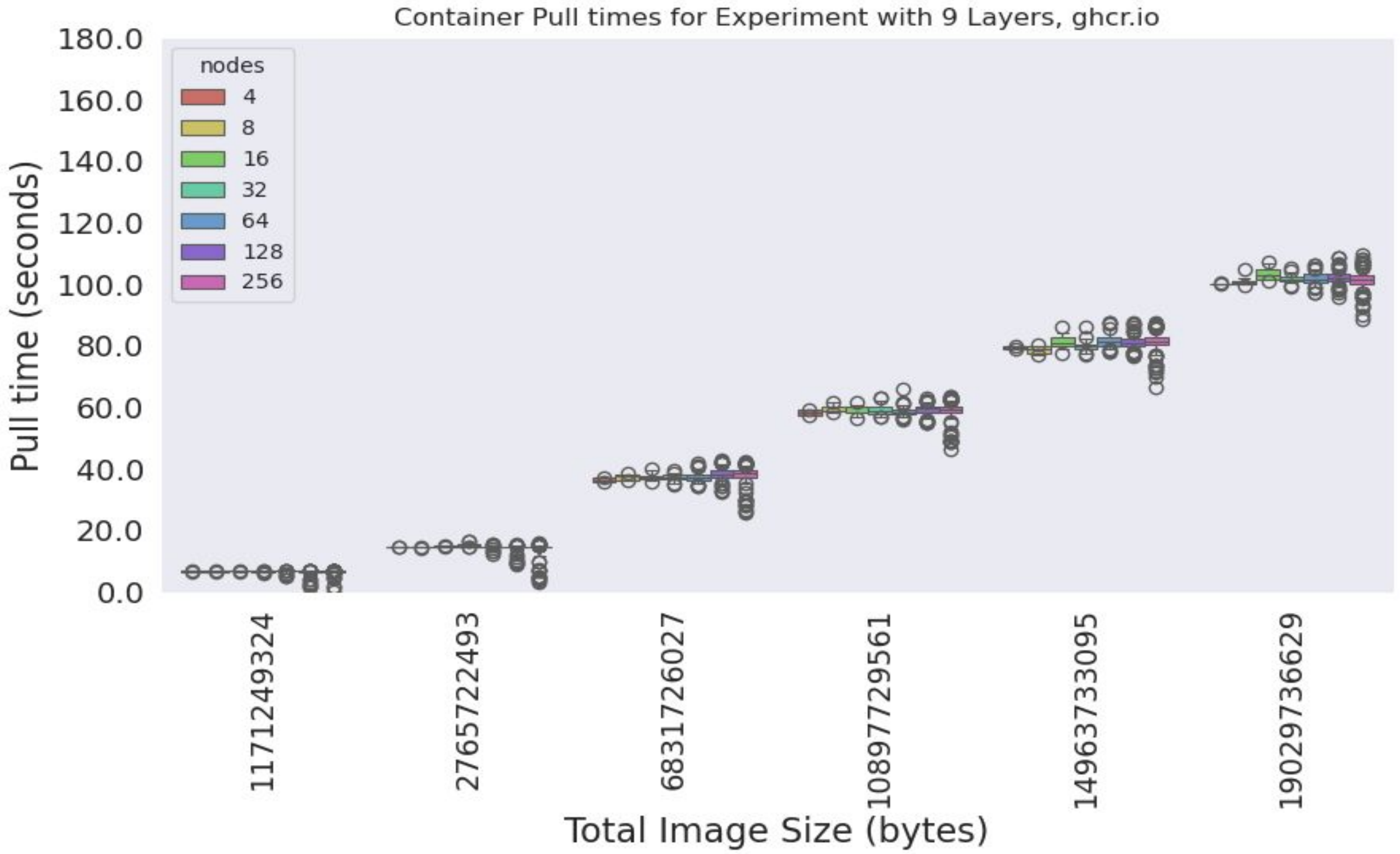


ghcr.io

gcr.io

Does pulling with local SSD improve pull times?

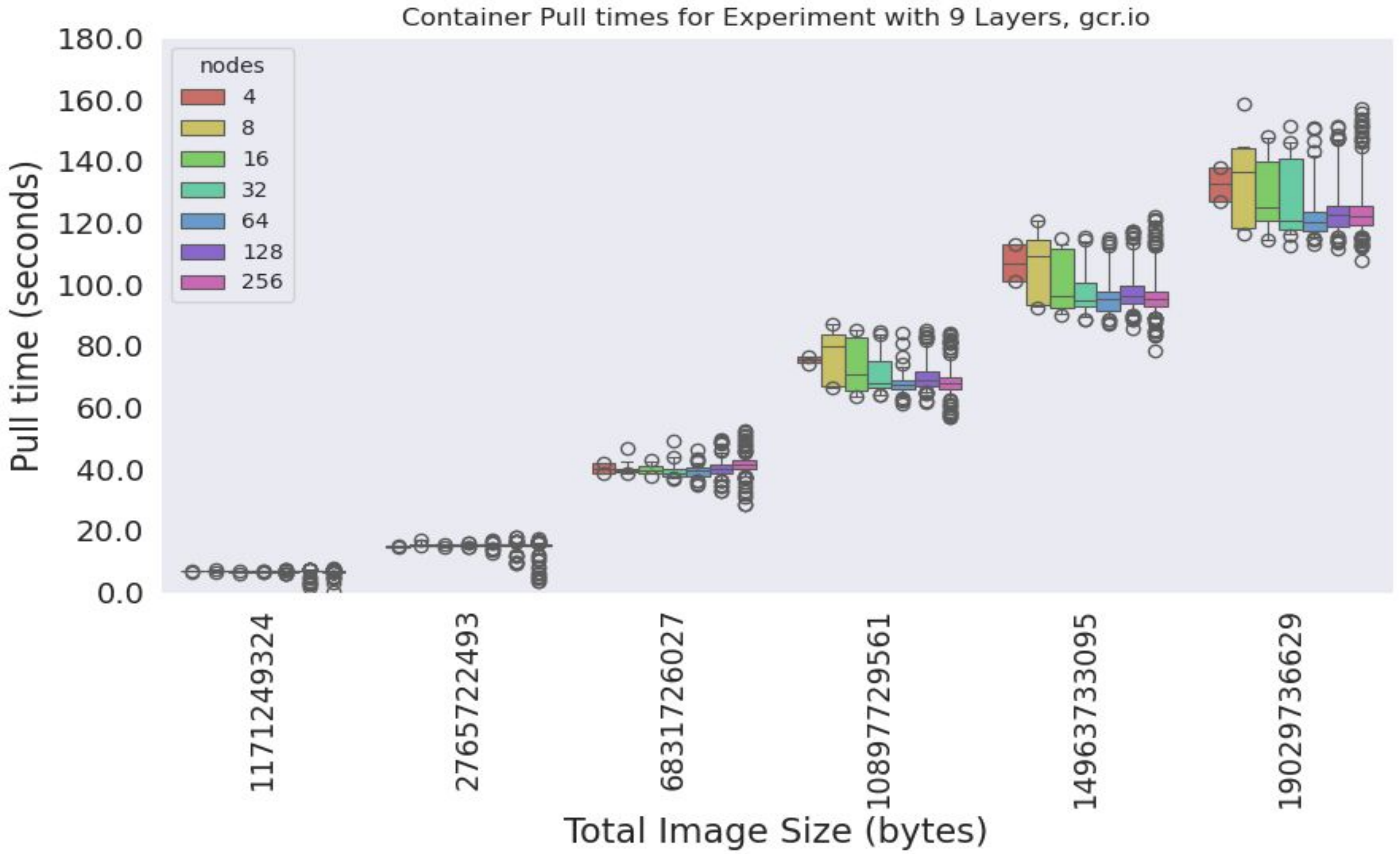
Yes! Often 1.25x



gcr.io

Does pulling with local SSD improve pull times?

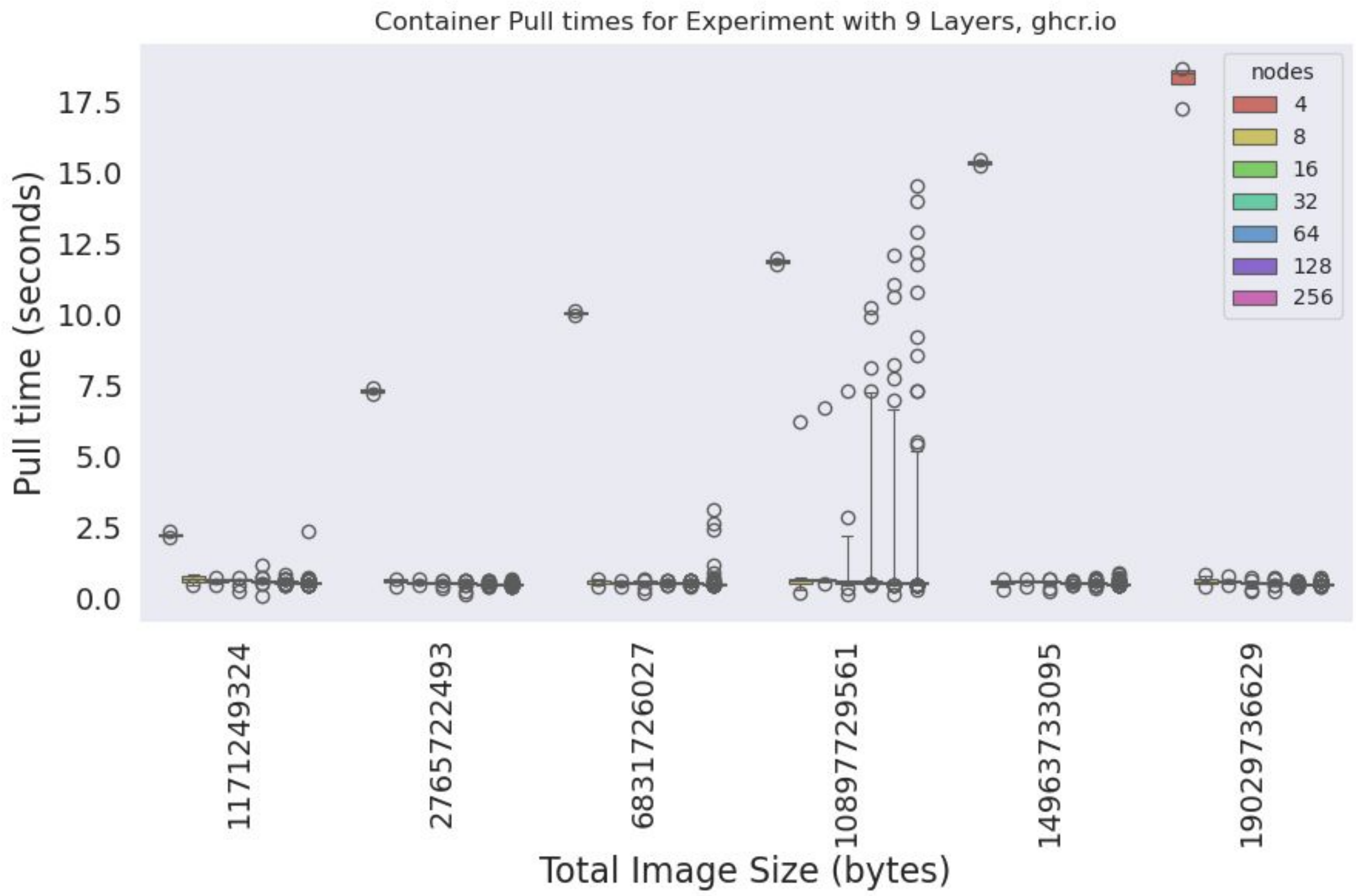
Yes! Often 1.25x



gcr.io

Does pulling with image streaming improve pull times?

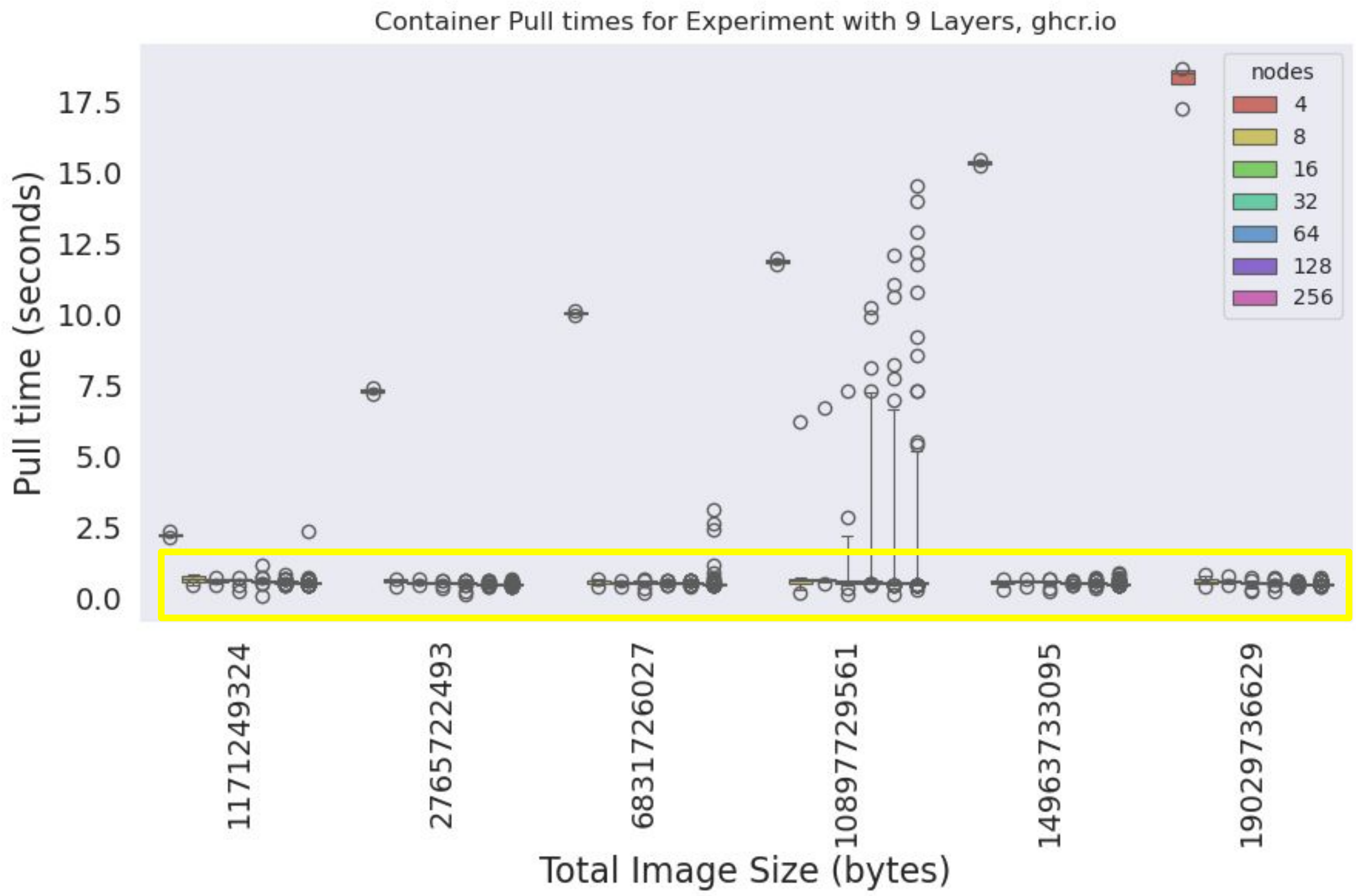
Impossibility, yes.



gcr.io

Does pulling with image streaming improve pull times?

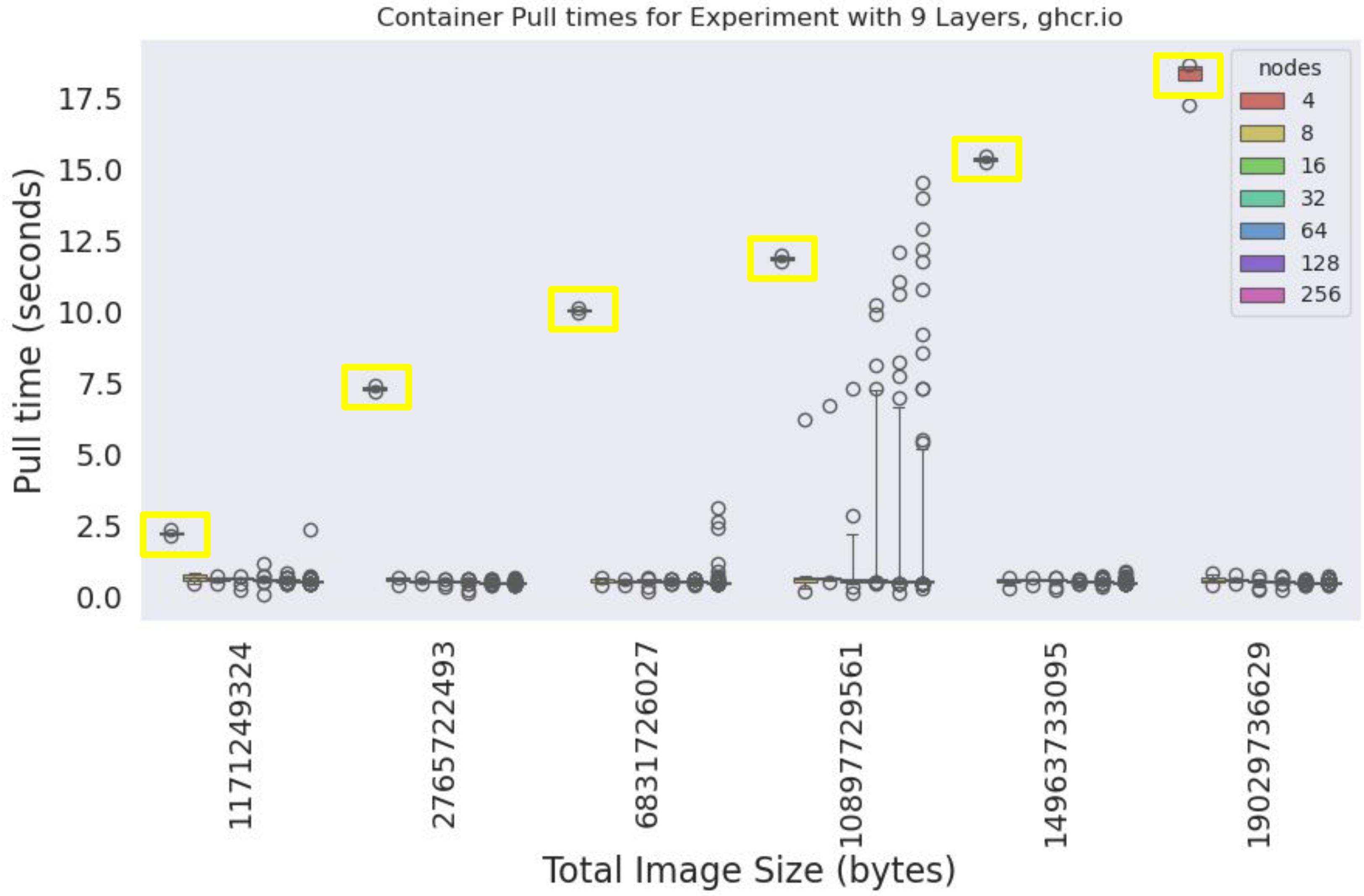
Impossibility, yes.



gcr.io

Does pulling with image streaming improve pull times?

Impossibility, yes.



gcr.io

Does pulling with image streaming improve pull times?

Real application containers for AMG, LAMMPS, OSU, Minife built with spack

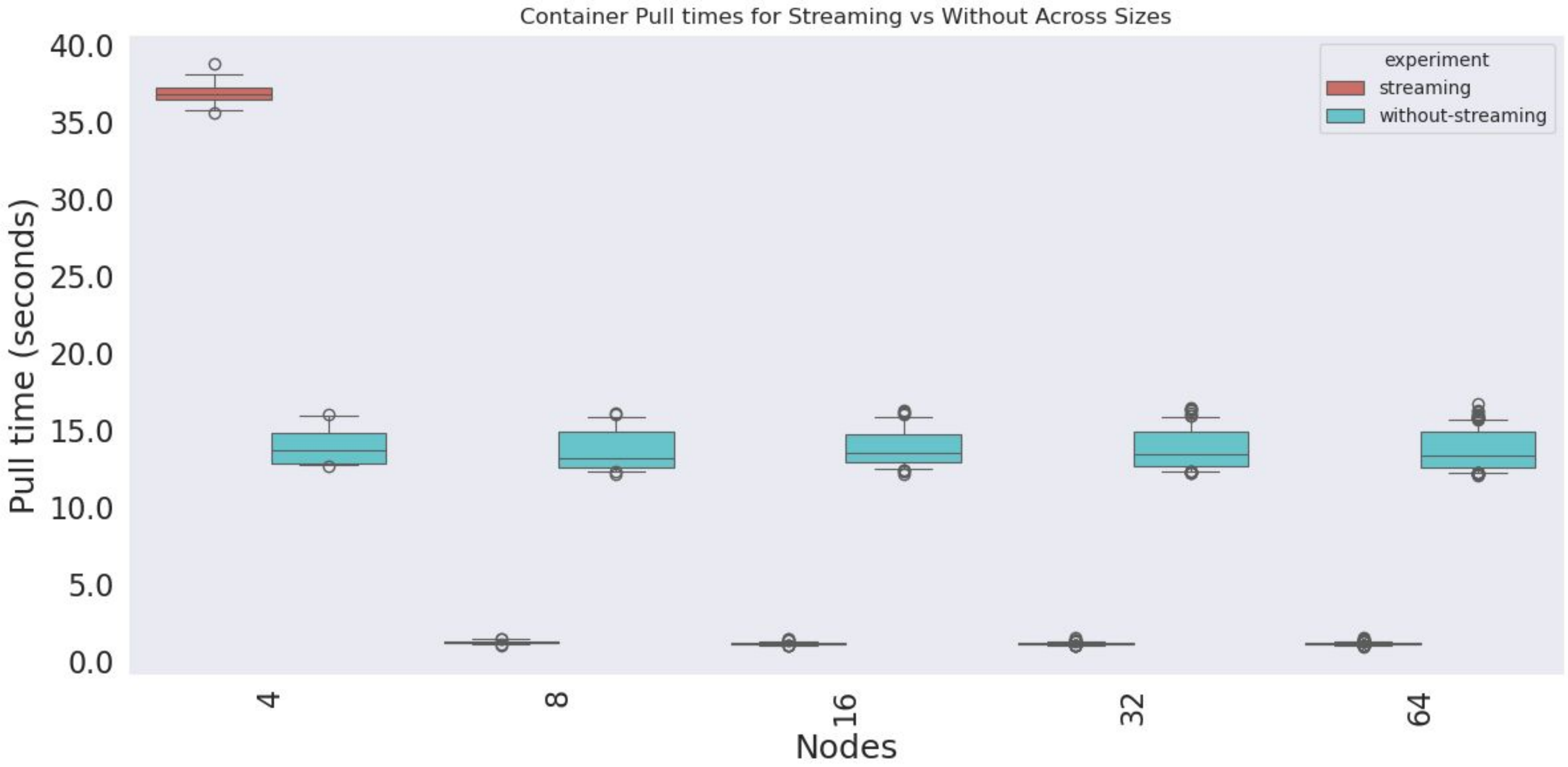


Image Streaming - why was it investigated in the first place?

"Image download accounts for 76% of container startup time, but on average only 6.4% of the fetched data is actually needed for the container to start doing useful work."

Harter et al FAST '16

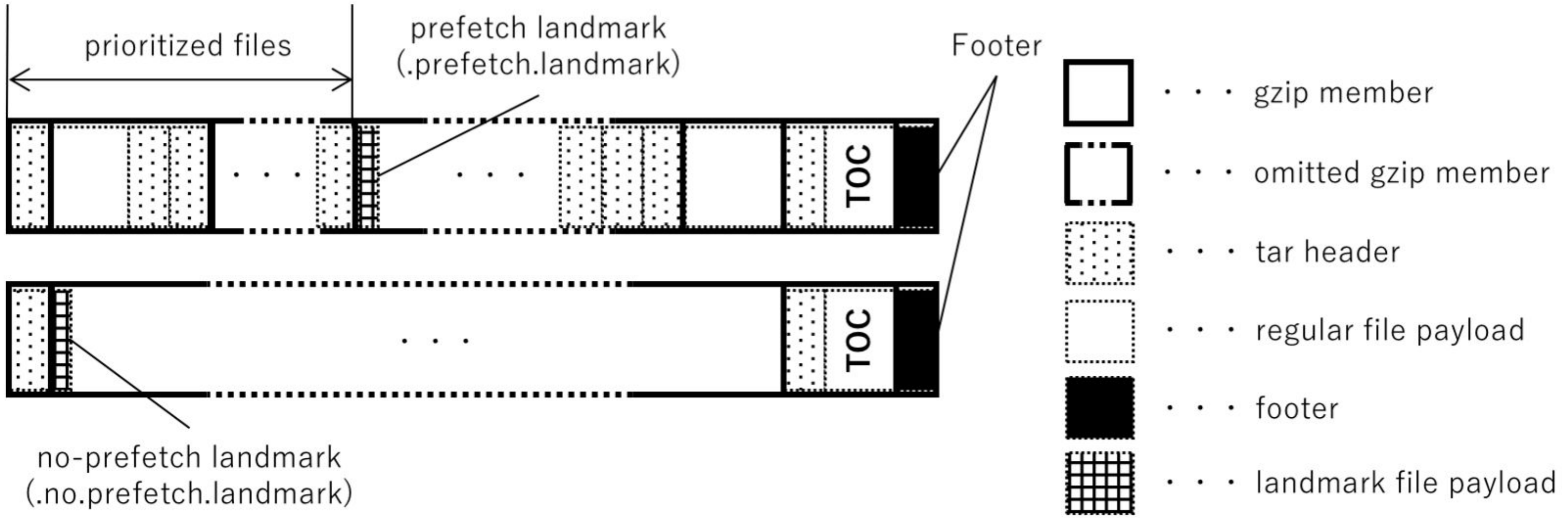
Image Streaming - why was it investigated in the first place?

Faster Container Pulling in Kubernetes
The SOCI "Seekable OCI" Snapshotter

https://youtu.be/ZXM1gP4goP8

Image Streaming - how does it work?

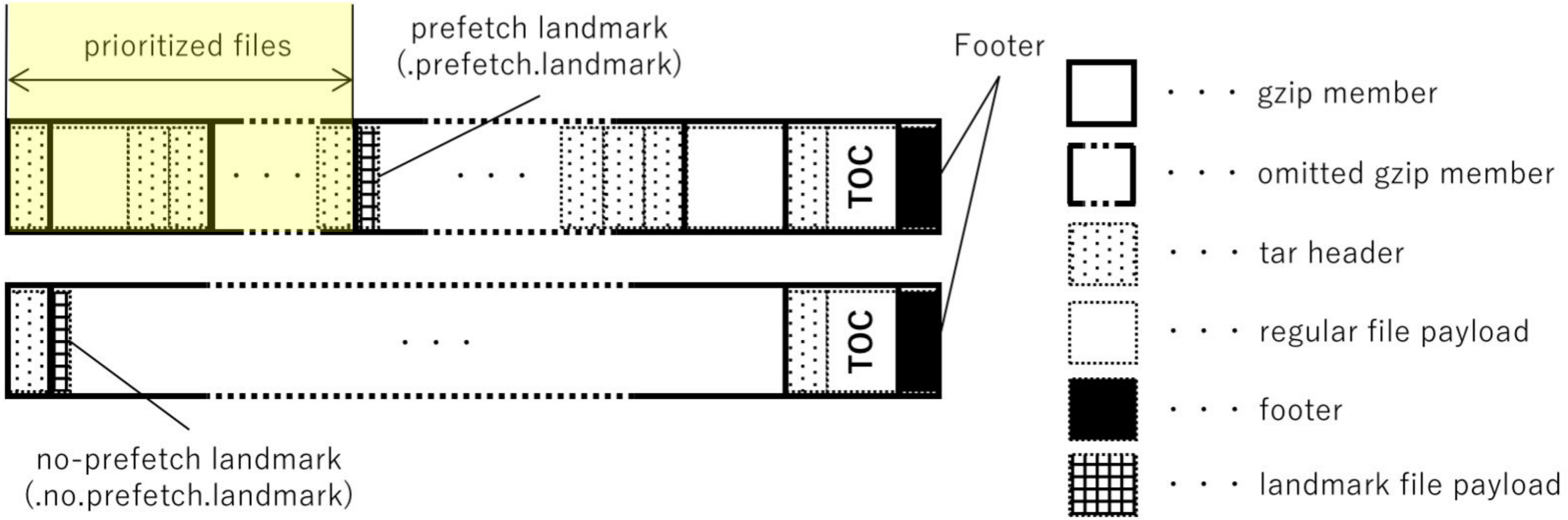
Step 1: We record the entrypoint to find "prioritized files"



https://github.com/containerd/stargz-snapshotter/blob/main/docs/estargz.md

Image Streaming - how does it work?

Step 1: We record the entrypoint to find "prioritized files"



https://github.com/containerd/stargz-snapshotter/blob/main/docs/estargz.md

Image Streaming - how does it work?

Step 2: Image and table of contents (artifact) pushed to registry

docker pull

"My host is amd64"

Image Manifest List

Image Manifest

Referrers Manifest List

SBOM (software bill of materials)
SOCI Index
Squirrel Snacks?

SOCI Index

ztoc artifact


```
{
    "annotations": {
        "com.amazon.soci.build-tool-identifier": "AWS SOCI CLI v0.1"
    },
    "config": {
        "digest": "sha256:44136fa355b3678a1146ad16f7e8649e94fb4fc21fe77e8310c060f61caaff8a",
        "mediaType": "application/vnd.amazon.soci.index.v1+json",
        "size": 2
    },
    "layers": [
        {
            "annotations": {
                "com.amazon.soci.image-layer-digest": "sha256:aece8493d3972efa43bfd4ee3cdba659c0f787f8f59c82fb3e48c87cbb22a12e",
                "com.amazon.soci.image-layer-mediaType": "application/vnd.oci.image.layer.v1.tar+gzip"
            },
            "digest": "sha256:cb011209b93fde5a7a698302ca2e2dd4928278da32a31995c5df6d16cb27c638",
            "mediaType": "application/octet-stream",
            "size": 1229704
        },
        {
            "annotations": {
                "com.amazon.soci.image-layer-digest": "sha256:e8b79c92d68de7b64f81149d112547255d8edecba2f64054a533954ee3c106fe",
                "com.amazon.soci.image-layer-mediaType": "application/vnd.oci.image.layer.v1.tar+gzip"
            },
            "digest": "sha256:7ae9f4713a78a4895a7a522117c552a519d8289eecc446820b1b528cf9257b82",
            "mediaType": "application/octet-stream",
            "size": 841192
        },
        {
            "annotations": {
                "com.amazon.soci.image-layer-digest": "sha256:bb157f88c818b56b154696493271c0d7a2704ed253fc2111d0a96fa89bf44984",
                "com.amazon.soci.image-layer-mediaType": "application/vnd.oci.image.layer.v1.tar+gzip"
            },
            "digest": "sha256:715ee958c9cff9bca859e940cb05a3937bd2a6d2570fec910602df095339b067",
            "mediaType": "application/octet-stream",
            "size": 1086672
        }
    ],
    "mediaType": "application/vnd.oci.image.manifest.v1+json",
    "schemaVersion": 2,
    "subject": {
        "digest": "sha256:ab5a62720aa71c964aae400825284ca3e8229934928394e68a06e395a225faae",
        "mediaType": "application/vnd.oci.image.manifest.v1+json",
        "size": 2752
    }
}
```



```
{
  "version": "0.9",
  "build_tool": "AWS SOCI CLI v0.1",
  "size": 1086672,
  "span_size": 4194304,
  "num_spans": 9,
  "num_files": 4102,
  "num_multi_span_files": 8,
  "files": [
    {
      "filename": "etc/",
      "offset": 512,
      "size": 0,
      "type": "dir",
      "start_span": 0,
      "end_span": 0
    },
    {
      "filename": "etc/ca-certificates/",
      "offset": 1024,
      "size": 0,
      "type": "dir",
      "start_span": 0,
      "end_span": 0
    },
    ... ,
    {
      "filename": "var/log/dpkg.log",
      "offset": 34542592,
      "size": 202359,
      "type": "reg",
      "start_span": 8,
      "end_span": 8
    }
  ]
}
```

Image Streaming - how does it work?
Putting it all together!

A snapshot is a view of the container filesystem, prepared from a layer

1. We start with a registry that supports artifacts, and has a pushed image and associated table of contents

Image Streaming - how does it work?
Putting it all together!

A snapshot is a view of the container filesystem, prepared from a layer

1. We start with a registry that supports artifacts, and has a pushed image and associated table of contents
2. The snapshotter plugin knows how to use that table of contents to download just the prioritized files

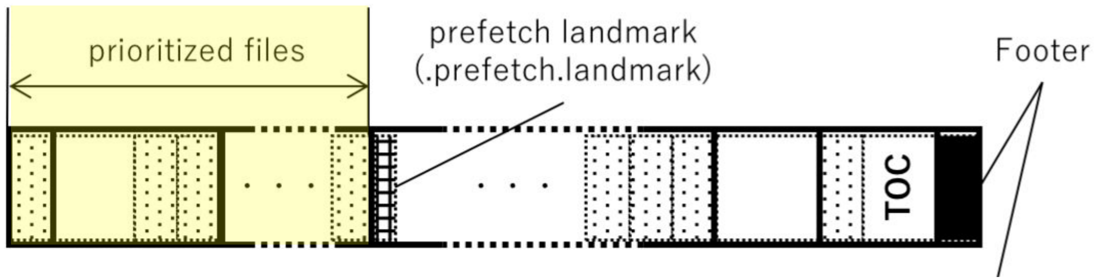


Image Streaming - how does it work?
Putting it all together!

A snapshot is a view of the container filesystem, prepared from a layer

1. We start with a registry that supports artifacts, and has a pushed image and associated table of contents
2. The snapshotter plugin knows how to use that table of contents to download just the prioritized files
3. After the prioritized files are downloaded, we mark the container as ready (and it's ready much faster)

Image Streaming - how does it work?
Putting it all together!

A snapshot is a view of the container filesystem, prepared from a layer

1. We start with a registry that supports artifacts, and has a pushed image and associated table of contents
2. The snapshotter plugin knows how to use that table of contents to download just the prioritized files
3. After the prioritized files are downloaded, we mark the container as ready (and it's ready much faster)
4. Additional content needed is loaded on demand using the distribution spec "resumable pull"

Resumable Pull

Company X is having more connectivity problems but this time in their deployment datacenter. When downloading a blob, the connection is interrupted before completion. The client keeps the partial data and uses http `Range` requests to avoid downloading repeated data.

https://github.com/opencontainers/distribution-spec/blob/main/spec.md

Image Streaming - how does it work?

Putting it all together!

A snapshot is a view of the container filesystem, prepared from a layer

1. We start with a registry that supports artifacts, and has a pushed image and associated table of contents
2. The snapshotter plugin knows how to use that table of contents to download just the prioritized files
3. After the prioritized files are downloaded, we mark the container as ready (and it's ready much faster)
4. Additional content needed is loaded on demand using the distribution spec "resumable pull"
5. From the user perspective, the container pulled a lot faster because we only *actually* pulled a small subset of files.

Image Streaming - how does it work?
Putting it all together!

A snapshot is a view of the container filesystem, prepared from a layer

1. We start with a registry that supports artifacts, and has a pushed image and associated table of contents
2. The snapshotter plugin knows how to use that table of contents to download just the prioritized files
3. After the prioritized files are downloaded, we mark the container as ready (and it's ready much faster)
4. Additional content needed is loaded on demand using the distribution spec "resumable pull"
5. From the user perspective, the container pulled a lot faster because we only *actually* pulled a small subset of files.

14. Range Requests

Clients often encounter interrupted data transfers as a result of canceled requests or dropped connections. When a client has stored a partial representation, it is desirable to request the remainder of that representation in a subsequent request rather than transfer the entire representation. Likewise, devices with limited local storage might benefit from being able to request only a subset of a larger representation, such as a single page of a very large document, or the dimensions of an embedded image.

Range requests are an OPTIONAL feature of HTTP, designed so that recipients not implementing this feature (or not supporting it for the target resource) can respond as if it is a normal GET request without impacting interoperability. Partial responses are indicated by a distinct status code to not be mistaken for full responses by caches that might not implement the feature.

https://www.rfc-editor.org/rfc/rfc9110.html#name-range-requests

Image Streaming - how does it work?
Putting it all together!

A snapshot is a view of the container filesystem, prepared from a layer

1. We start with a registry that supports artifacts, and has a pushed image and associated table of contents
2. The snapshotter plugin knows how to use that table of contents to download just the prioritized files
3. After the prioritized files are downloaded, we mark the container as ready (and it's ready much faster)
4. Additional content needed is loaded on demand using the distribution spec "resumable pull"
5. From the user perspective, the container pulled a lot faster because we only *actually* pulled a small subset of files.

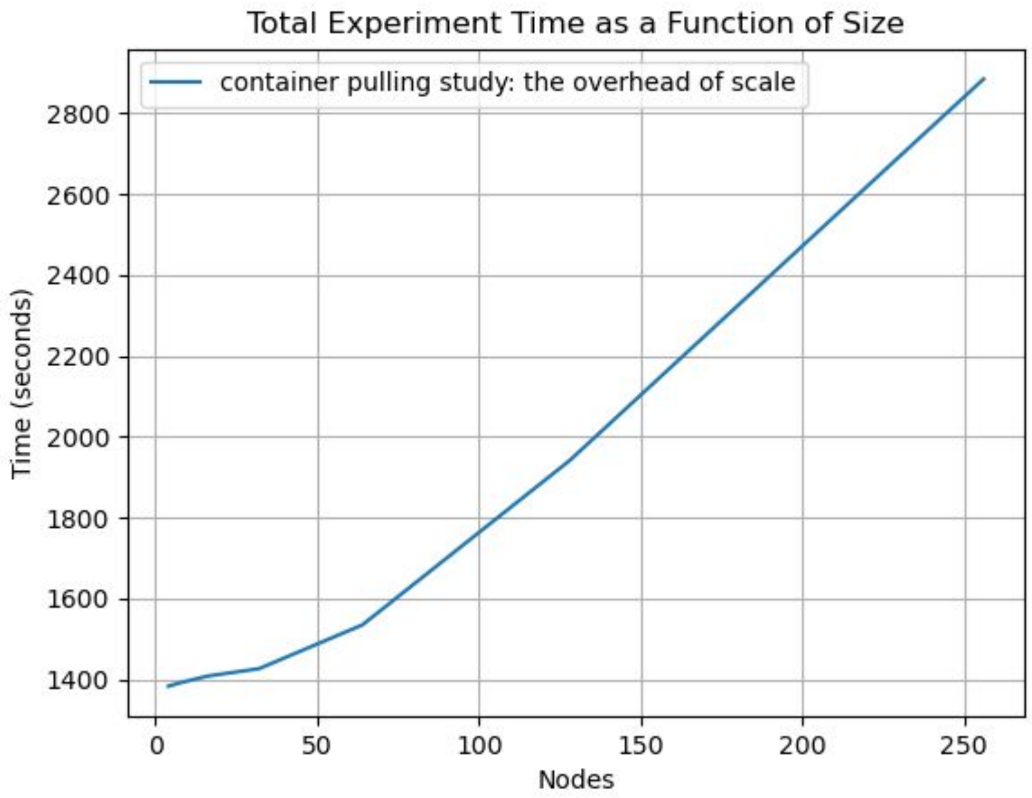
Since files needed later in execution are pulled on demand, we have to be cautious about using that plugin for apps that require loading large data later in execution!

What is the best strategy for container pulling?
SSD is a good idea always, image streaming sometimes

Node Coordination

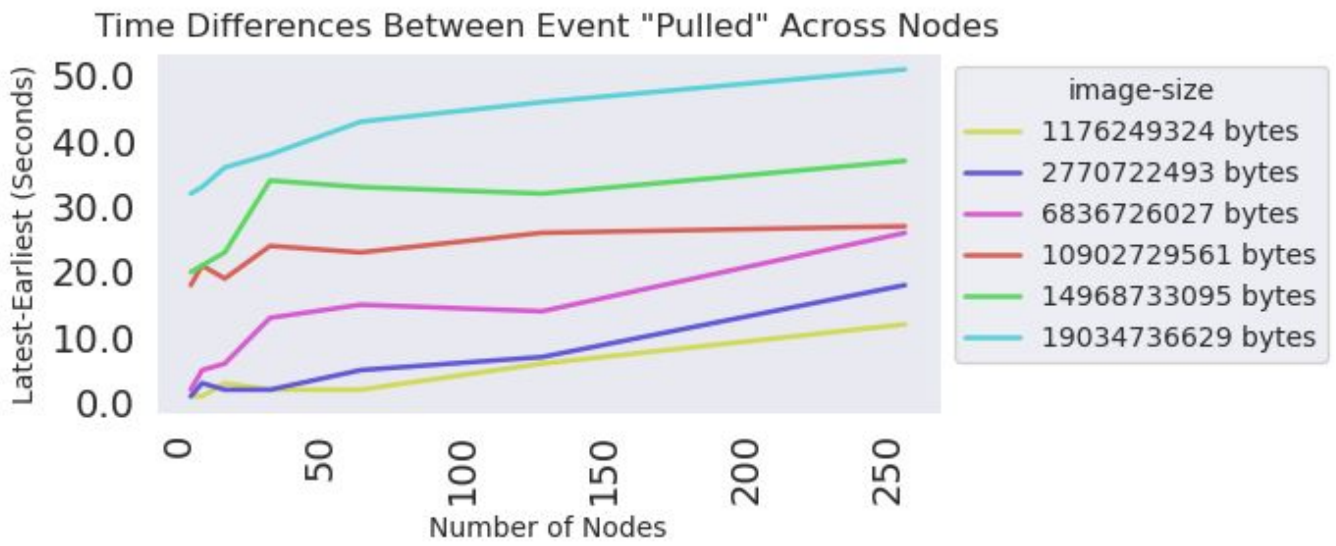
Does node coordination lead to slower pull times?
Are we limited by the slowest node?

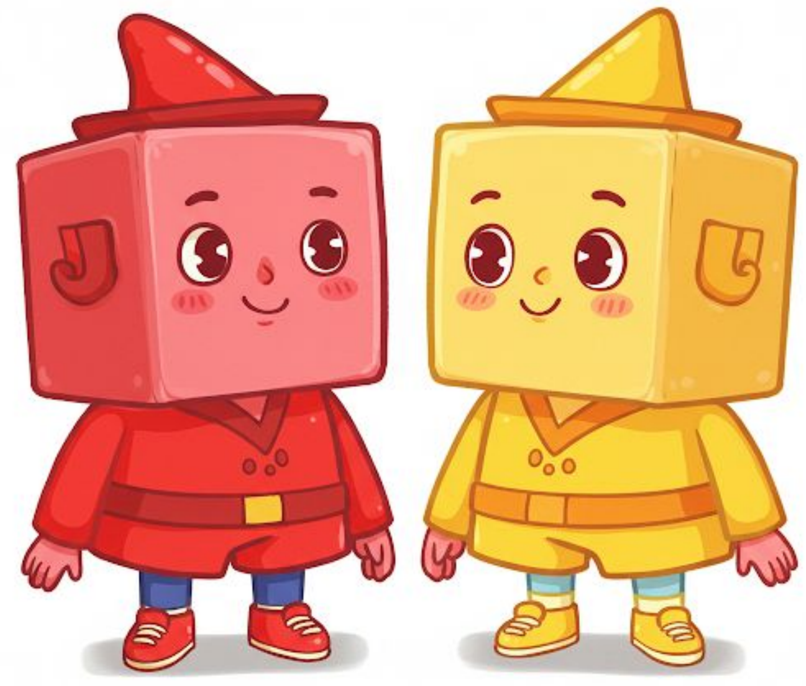
Pull times didn't increase but…
Overall experiment time increased with cluster size



Does node coordination lead to slower pull times?
Are we limited by the slowest node?

Event times are not coordinated (understandably) across nodes…
but it means we *are* limited by the slowest node





Node Coordination

Nodes are less coordinated as nodes increase, we need to better understand why.

Takeaways

What did we learn from this work?

- A container building strategy optimized for similarity in container layers, and a pulling strategy (filesystem or algorithm) to decrease pull time can decrease total cost for a study.

This improvement becomes more salient when using **expensive resources** such as GPU, or an auto-scaling strategy that provisions new nodes that don't have images cached.

Let's calculate cost with respect to unique layers

Assuming:
- We have 10 total applications (what we had for our performance study)
- Each image has mean layer size (~12MB), and mean layers (16) (means from dataset of 77K Dockerfiles)

Let's calculate cost with respect to unique layers

Assuming:
- We have 10 total applications (what we had for our performance study)
- Each image has mean layer size (~12MB), and mean layers (16) (means from dataset of 77K Dockerfiles)

Strategy: look at extremes
- In an ideal case, the first image pull (across nodes) pulls all layers (and they are cached)
- In the worst case, no layer redundancy means 16 new layers each time

Let's calculate cost with respect to unique layers

Assuming:
- We have 10 total applications (what we had for our performance study)
- Each image has mean layer size (~12MB), and mean layers (16) (means from dataset of 77K Dockerfiles)

Strategy: look at extremes
- In an ideal case, the first image pull (across nodes) pulls all layers (and they are cached)
- In the worst case, no layer redundancy means 16 new layers each time

How many layer pulls?
- In the best (hypothetical) case of all the same layers we would pull only the equivalent of 1 container, 16 layers
  - *This is not a realistic case because you can't have different apps with the exact same layers!*
- In the worst case of all different layers, we would pull 10 x 16 == 160 layers

Let's calculate cost with respect to unique layers

Assuming:
- We have 10 total applications (what we had for our performance study)
- Each image has mean layer size (~12MB), and mean layers (16) (means from dataset of 77K Dockerfiles)

Strategy: look at extremes
- In an ideal case, the first image pull (across nodes) pulls all layers (and they are cached)
- In the worst case, no layer redundancy means 16 new layers each time

How many layer pulls?
- In the best (hypothetical) case of all the same layers we would pull only the equivalent of 1 container, 16 layers
  - *This is not a realistic case because you can't have different apps with the exact same layers!*
- In the worst case of all different layers, we would pull 10 x 16 == 160 layers

Sizes:
- In the best case, mean layer size 12MB x 16 == 192 MB
- In the worst case, 12MB x 160 == 1920 MB

Let's calculate cost with respect to unique layers

Assuming:
- We have 10 total applications (what we had for our performance study)
- Each image has mean layer size (~12MB), and mean layers (16) (means from dataset of 77K Dockerfiles)

Strategy: look at extremes
- In an ideal case, the first image pull (across nodes) pulls all layers (and they are cached)
- In the worst case, no layer redundancy means 16 new layers each time

How many layer pulls?
- In the best (hypothetical) case of all the same layers we would pull only the equivalent of 1 container, 16 layers
  - *This is not a realistic case because you can't have different apps with the exact same layers!*
- In the worst case of all different layers, we would pull 10 x 16 == 160 layers

Sizes:
- In the best case, mean layer size 12MB x 16 == 192 MB
- In the worst case, 12MB x 160 == 1920 MB

Pull times (assuming 3-5MB per second)
- In the best case, 192 MB will take: 39-64 seconds
- In the worst case, 1920 MB will take: 384-640 seconds (6.4 - 11 minutes)
- Differences 345-576 seconds (5.75-9.6 minutes)

Let's calculate cost with respect to unique layers

Assuming:
- We have 10 total applications (what we had for our performance study)
- Each image has mean layer size (~12MB), and mean layers (16) (means from dataset of 77K Dockerfiles)

Strategy: look at extremes
- In an ideal case, the first image pull (across nodes) pulls all layers (and they are cached)
- In the worst case, no layer redundancy means 16 new layers each time

How many layer pulls?
- In the best (hypothetical) case of all the same layers we would pull only the equivalent of 1 container, 16 layers
  - *This is not a realistic case because you can't have different apps with the exact same layers!*
- In the worst case of all different layers, we would pull 10 x 16 == 160 layers

Sizes:
- In the best case, mean layer size 12MB x 16 == 192 MB
- In the worst case, 12MB x 160 == 1920 MB

Pull times (assuming 3-5MB per second)
- In the best case, 192 MB will take: 39-64 seconds
- In the worst case, 1920 MB will take: 384-640 seconds (6.4 - 11 minutes)
- Differences 345-576 seconds (5.75-9.6 minutes)

For this hypothetical scenario, we estimate 5.75 - 9.6 minutes more of node running time to account for pulling. Whether this amount of time is significant depends on the size of the cluster, the cost of the nodes, and the budget. E.g., the p5.48xlarge node at AWS is $98.32/hour. For a size 32 cluster (~$3146/hour), it would be an additional appox $301 - $503.

What about with a ML oriented image?

Assuming:

- 10 images like pytorch/pytorch (with nothing else)
- Layers include:

    30MB
    7.3MB
    3.6GB (3622 MB)

What about with a ML oriented image?

Assuming:

- 10 images like pytorch/pytorch (with nothing else)
- Layers include:

    30MB
    7.3MB
    3.6GB (3622 MB)

Sizes:

- In the best case, we pull 3 layers == 3659.3 MB
- In the worst case, 3659.3 3 layers x 10 == 36593 MB (36.593 GB)

What about with a ML oriented image?

Assuming:

- 10 images like pytorch/pytorch (with nothing else)
- Layers include:

    30MB
    7.3MB
    3.6GB (3622 MB)

Sizes:

- In the best case, we pull 3 layers == 3659.3 MB
- In the worst case, 3659.3 3 layers x 10 == 36593 MB (36.593 GB)

Pull times (assuming 3-5MB per second)

- In the best case, 192 MB will take: 732-1219 seconds (12-20 minutes)
- In the worst case, pulling takes: 7318-12198 seconds (122-203 minutes)
- Differences 6586-10979 seconds (110-183 minutes)

What about with a ML oriented image?

Assuming:

- 10 images like pytorch/pytorch (with nothing else)
- Layers include:

    30MB
    7.3MB
    3.6GB (3622 MB)

Sizes:

- In the best case, we pull 3 layers == 3659.3 MB
- In the worst case, 3659.3 3 layers x 10 == 36593 MB (36.593 GB)

Pull times (assuming 3-5MB per second)

- In the best case, 192 MB will take: 732-1219 seconds (12-20 minutes)
- In the worst case, pulling takes: 7318-12198 seconds (122-203 minutes)
- Differences 6586-10979 seconds (110-183 minutes)

For this hypothetical scenario with ML images, we estimate 110-183 minutes more of node running time to account for pulling. Given the p5.48xlarge node at $98.32/hour (on demand) for a size 32 cluster (~$3146/hour), it would be an additional appox $5768 - $9595.3.

Some Additional Strategies

What else can we do in these cases?
Other strategies for caching image layers…

- Use something like AWS Parallel Cluster where you can pre-pull to a head node with a shared volume, and the workers then create and bind to it.
- If you are auto-scaling, use a setup that mounts a read only volume with containers that are pre-pulled.
- Use a pull-through cache that provides a local registry cache alongside your cluster.
- For innovation, we can explore other algorithms for predicting content to pull, compression algorithms, and improved file-system latency.

What did we learn from this work?
It is our responsibility to be aware of cost savings

- A container building strategy optimized for similarity in container layers can increase layer redundancy, decreasing time needed to pull and thus decreasing total time and cost for a study.

- Extra time is accumulated as clusters get larger, a result that could be due to decreases in node coordination and needing to wait for the slowest node to finish pulling. This finding is interesting and warrants further exploration for behavior and solutions.

What did we learn from this work?
It is our responsibility to be aware of cost savings

- A container building strategy optimized for similarity in container layers can increase layer redundancy, decreasing time needed to pull and thus decreasing total time and cost for a study.

- Extra time is accumulated as clusters get larger, a result that could be due to decreases in node coordination and needing to wait for the slowest node to finish pulling. This finding is interesting and warrants further exploration for behavior and solutions.

- Container streaming is an ideal strategy for quickly starting containers that are large, but caution should be used if large amounts of new data are needed for application execution later in the run than is recorded by the snapshotter tool.

Interesting Findings

The Genghis Khan of container layers!
The outlier in the layer set - a digest that appeared 67,897 times!

- An empty set of 32 bytes associated with a WORKDIR directive

The Genghis Khan of container layers!

The outlier in the layer set - a digest that appeared 67,897 times!

- An empty set of 32 bytes associated with a WORKDIR directive
- But… only for cases when the directory already existed.

The Genghis Khan of container layers!

The outlier in the layer set - a digest that appeared 67,897 times!

- An empty set of 32 bytes associated with a WORKDIR directive
- But… only for cases when the directory already existed.
- Turns out… there is an "empty layer" flag in the image config. If a tool decides not to set that flag for some reason, the tool needs to ship a valid tar+gzip, so even without any files being packaged, this takes up space in the tar and gzip headers.

The Genghis Khan of container layers!

The outlier in the layer set - a digest that appeared 67,897 times!

- An empty set of 32 bytes associated with a WORKDIR directive
- But… only for cases when the directory already existed.
- Turns out… there is an "empty layer" flag in the image config. If a tool decides not to set that flag for some reason, the tool needs to ship a valid tar+gzip, so even without any files being packaged, this takes up space in the tar and gzip headers.

This was implemented before it was discovered that /dev/null is a valid empty file.

```go
}

var (
	emptyGZLayer = digest.Digest("sha256:4f4fb700ef54461cfa02571ae0db9a0dc1e0cdb5577484a6d75e68dc38e8acc1")
	emptyDigest  = digest.Digest("")
)
```

How many layers are we allowed to build?
"Common" wisdom is often wrong (or outdated)

- I started with an understanding that the limit is 127 layers

How many layers are we allowed to build?
"Common" wisdom is often wrong (or outdated)

- I started with an understanding that the limit is 127 layers
- In Docker source code, you'll find references for each of 125 and 128 depending on overlay driver

How many layers are we allowed to build?
"Common" wisdom is often wrong (or outdated)

- I started with an understanding that the limit is 127 layers
- In Docker source code, you'll find references for each of 125 and 128 depending on overlay driver
- But containerd doesn't set a maximum…

How many layers are we allowed to build?
"Common" wisdom is often wrong (or outdated)

- I started with an understanding that the limit is 127 layers
- In Docker source code, you'll find references for each of 125 and 128 depending on overlay driver
- But containerd doesn't set a maximum…

We built and pushed (successfully) docker.io/tianon/test:many-layers-256 with 256 layers, no problem!

How many layers are we allowed to build?
"Common" wisdom is often wrong (or outdated)

- I started with an understanding that the limit is 127 layers
- In Docker source code, you'll find references for each of 125 and 128 depending on overlay driver
- But containerd doesn't set a maximum…

We built and pushed (successfully) docker.io/tianon/test:many-layers-256 with 256 layers, no problem!

- The limit was originally enforced because of a limit with mounting layers
  - Specifically the length of an argument to a syscall that led to technical maximums

How many layers are we allowed to build?
"Common" wisdom is often wrong (or outdated)

- I started with an understanding that the limit is 127 layers
- In Docker source code, you'll find references for each of 125 and 128 depending on overlay driver
- But containerd doesn't set a maximum…

We built and pushed (successfully) docker.io/tianon/test:many-layers-256 with 256 layers, no problem!

- The limit was originally enforced because of a limit with mounting layers
  - Specifically the length of an argument to a syscall that led to technical maximums
- But this depends on the operator system, kernel version, and container runtime!

How many layers are we allowed to build?
"Common" wisdom is often wrong (or outdated)

- I started with an understanding that the limit is 127 layers
- In Docker source code, you'll find references for each of 125 and 128 depending on overlay driver
- But containerd doesn't set a maximum…

We built and pushed (successfully) docker.io/tianon/test:many-layers-256 with 256 layers, no problem!

- The limit was originally enforced because of a limit with mounting layers
  - Specifically the length of an argument to a syscall that led to technical maximums
- But this depends on the operator system, kernel version, and container runtime!
- containerd and buildkit use a practical approach that doesn't validate (and allows the error to propagate)

How many layers are we allowed to build?
"Common" wisdom is often wrong (or outdated)

- I started with an understanding that the limit is 127 layers
- In Docker source code, you'll find references for each of 125 and 128 depending on overlay driver
- But containerd doesn't set a maximum…

We built and pushed (successfully) docker.io/tianon/test:many-layers-256 with 256 layers, no problem!

- The limit was originally enforced because of a limit with mounting layers
    - Specifically the length of an argument to a syscall that led to technical maximums
- But this depends on the operator system, kernel version, and container runtime!
- containerd and buildkit use a practical approach that doesn't validate (and allows the error to propagate)
- docker hard codes manual checks so you don't get to that point

How many layers are we allowed to build?
"Common" wisdom is often wrong (or outdated)

- I started with an understanding that the limit is 127 layers
- In Docker source code, you'll find references for each of 125 and 128 depending on overlay driver
- But containerd doesn't set a maximum…

We built and pushed (successfully) docker.io/tianon/test:many-layers-256 with 256 layers, no problem!

- The limit was originally enforced because of a limit with mounting layers
  - Specifically the length of an argument to a syscall that led to technical maximums
- But this depends on the operator system, kernel version, and container runtime!
- containerd and buildkit use a practical approach that doesn't validate (and allows the error to propogate)
- docker hard codes manual checks so you don't get to that point
- docker will fail on this mount step *after* pulling the layers…


AGAIN, THIS IS INFORMATION
YOU COULD HAVE TOLD ME
BEFORE PULLING!
imgflip.com

Let's make a SOCI snapshotter daemonset!

This is much easier to install!

```
kubectl apply -f soci-installer.yaml
```

This logic can be extended:

- To support other authentication schemes
- Other clouds (that don't have flags already)

Thank you!

sochat1@llnl.gov

Lawrence Livermore National Laboratory

Lawrence Livermore National Laboratory