Trends for Pulling HPC Containers in Cloud

Vanessa Sochat
Lawrence Livermore National Laboratory
e-mail: sochat1@llnl.gov

***Abstract*—Container technologies are foundational for cloud orchestration and have captured the interest of the High Performance Computing (HPC) community. While much work has been done to demonstrate that there is no additional overhead when using a container technology, strategies for building and pulling scientific containers to cloud environments and the cost implications of those choices have not been fully studied. Due to the importance and predominance in the ecosystem, considerations that minimize the time of operations, such as pulling and staging, are essential. This understanding and innovation in the space is becoming more important as more scientific applications are ported to cloud environments. In this study, we first aim to understand the landscape of containerized scientific applications, assembling a sample of more than 77K Dockerfile recipes discovered from repositories in a research software engineering database and across a set of well-known machine learning organizations. We assess these data for trends in building strategy and resulting containers, and show that applying best practices to a set of 10 application containers can lead to improvements in layer redundancy and thus lower time and cost to use the set. Finally, we develop a simulation tool that creates containers for controlled experiments that vary the total size, and the number and size of layers. With this tool, we run an experimental study that varies layer size and count across several scales to better understand the trade-off between layer count and size and the subsequent cost. In this experimental work, we find that total image size is a dominating variable during provisioning, and that strategies to improve I/O and enable lazy loading of images can lead to improvements of 3-15x. This work is valuable to inform the HPC community moving to the cloud about best practices for building and pulling containers.**

Keywords-cloud; containers; Kubernetes; HPC; trends.

I. INTRODUCTION

Capturing application logic in containers is a strategy to ensure reproducibility and automation of modern workflows [1]. The dominant force of cloud, and specifically container orchestration in cloud [2], has further incentivized the High Performance Computing (HPC) community to investigate and pursue strategies for optimally running HPC applications there, requiring a different mindset to consider not just optimizing performance [3]–[5] but also minimizing monetary cost. One component of this cost is the action of moving a container from a registry [6] to the cloud environment, an action often referred to as "pulling" that can add additional time to a workload, and thus incur additional monetary costs.

The process of pulling a container from a registry is governed by the Open Containers Distribution Specification [7], where first a container runtime makes a request to a registry Application Programming Interface (API) for a specific image identifier and tag, and a successful response returns an image manifest list [8]. The manifest list is parsed until a matching image platform is found, and then the container runtime tool can download the final image manifest, which includes a list of layers for download. Each layer is typically a compressed archive of a piece of the image filesystem, created as one command line in the build file called a Dockerfile [9]. While layers are downloaded in parallel and validated, the extraction is sequential due to the need to assemble the filesystem in the order mandated by the image configuration [10]. This entire process involves multiple interactions with the registry, and the download and extraction steps that encompass the pull can take upward of 76% of the container startup time [11].

Running a containerized workload using Kubernetes [12] starts with the pull of an application container. If all containers need to be started at the same time, node coordination is important. As container sizes grow larger and require more time to pull, this step could incur larger monetary costs. The design of the container and strategy for pulling can contribute to the efficiency of this step. If layers are assembled in a way to clean up unused files or take advantage of multi-stage builds, image size can be minimized [13]. The choice of filesystem and content retrieval and extraction strategy can further influence the time from initial pull to application start. This calls for an assessment of best practices when building and pulling containers, and the extent to which they are followed. If there is little time and incentive in the scientific community to optimize building and pulling strategies, the result can be a setup that is more monetarily costly.

While work has been done to assess all images in a registry [14] and pulling for common service containers [15], to our knowledge, no work has focused on images built in the scientific community. In this work, we do a temporal and quantitative analysis on the scientific community container ecosystem from 2014 until present day. In Section II we assess trends and practices for building containers, and look at changes in size, number of layers, base images used, and reuse. We develop an open-source tool to run simulations of container pulling across sizes and number of layers. We perform experiments across different cluster and container sizes with simulated and real application containers to identify ideal pulling strategies (Section III). We show that the total image size is the dominant variable related to pulling time, make suggestions based on our experimental results for effective pulling strategies, and develop two new pieces of software to support pulling experimentation and cluster deployment [16], [17]. Finally, in Section IV we review interesting findings, limitations, and follow-up work. Starting on the premise that the HPC community desires to move application containers from on-premises to the cloud and pulling is a required step that incurs monetary costs, this work is a starting point to

define good practices and areas of future work.

II. METHODS

A. Analysis of Container Images

Deployment of container images to HPC systems or cloud requires pulling the container images from registries, a task that can increase in time and thus be more costly for a workflow. Thus, understanding the sizes of images and change over time can provide insight into current practices and suggestions for improvement.

B. Dockerfile Ecosystem

We aim to use build recipes for containers to assess image and layer sizes and content. Larger sizes will take longer to pull, incurring higher monetary costs, and this can be associated with good and bad practices in the build recipes themselves [18]. The first task was to assemble a database of container image references, often called unique resource identifiers (URIs). A URI is a unique identifier that includes the registry, repository, and tag associated with a specific digest that indicates a version (e.g., docker.io/library/ubuntu:latest). While registries can expose a catalog endpoint to retrieve a catalog of all containers, most do not as it would increase load on already busy registries. Thus, we opt for a programmatic approach using the Research Software Encyclopedia, a meta-software database of over 5.6K curated research software projects [19], to discover Dockerfile recipes from established research software and machine learning projects. We can use this set to identify common underlying base images, and do an analysis of container and layer sizes.

C. Image and Layer Sizes

The container registry can provide manifests – JSON documents that detail the contents of the images, namely layers and digests, and the configurations. We are first interested in using this metadata to better understand the number of layers across images and tags, and how this has changed across time. Seeing that the number of layers has changed over time might reflect a change in build practices. We then can assess similarity of images by way of pure digests of images and content of the layers themselves.

D. Image Similarity

1) Content Similarity

A single Dockerfile provides three ways to assess similarity – the similarity of the base images used to build the container, the similarity of the Dockerfile build instructions themselves, and the similarity of the exact digests of the layers. The choice of a base image reflects a user preference, as different bases bring different package managers and potential needs for building software. The content similarity reflects layers having similar functionality, and the exact matching of digests is directly related to reuse, as a digest match indicates a cache hit and not needing to pull a new layer [14].

Similarity of container images by way of content can be done for both our Dockerfile database and the base images they are built from. For each, we treat the image build instructions as a document, where each line that builds a layer is parsed as a single sentence. To derive the build instructions for the base images, we parse the FROM directives of the Dockerfiles, and retrieve build instructions from the "history" section of the image manifest in the registry. For our Dockerfile database, we simply need to parse the RUN directives directly in the Dockerfile. For each of these sets of build instructions, we apply the following approach. If we consider a grouping of layers that encompasses a Dockerfile (and image) to be akin to paragraphs or sentences that make up a document, we can use these build instructions as a corpus. For each document, we pre-process the instruction lines to remove a subset of punctuation, and replace other punctuation with a space (e.g., underscores and dashes) and tokenize the result. We can then derive word2vec embeddings [20] each of length 300 for each image or Dockerfile, and do a pairwise similarity of these vectors using cosine similarity to derive a similarity matrix. These similarity values reflect the degree to which build instructions (from base images or Dockerfiles) are built with common logic.

2) Digest Similarity

While our similarity analysis primarily aimed to reflect on similarity of content, a different goal is to better understand the impact of build strategies on resulting digest similarity. The exact digest is the unique identifier for a layer, and the decision point about whether a container runtime needs to pull a layer. Since layers are saved to a cache [21], it follows that a strategy that minimizes redundancy of layer digests requires fewer pulls, both taking up less space on the filesystem and time to do the pulls.

Toward the goal of understanding digest similarity "in the wild" and similarity when care is taken to ensure redundancy, we can first assess the similarity of digests across our set of unique resource identifiers for base images. We will calculate the Jaccard coefficient between pairwise images, which is the ratio of the number of intersecting digests divided by the union of all digests. A Jaccard coefficient of 1 indicates most similar, and 0 most dissimilar. This value is expected to be low, as each digest reflects not only the content within it, but the previous layers [22]. We can then compare these scores against equivalently calculated values for different sets of images that are intentionally built with different redundancy strategies in mind:

1) A reasonable effort to create redundancy
2) A best effort to create redundancy
3) Little effort to create redundancy.

For each of these sets, we aim to compare a set of $\tilde{1}0$ containerized HPC proxy applications (and new builds of the same applications with an improved strategy) based on a real-world performance study [23]. For (1) we will use a derivative of the containers from the study where a reasonable effort was taken to ensure redundancy, however some images used an entirely different build strategy to achieve more ideal performance. Best-effort builds of the same applications (2),

and a highly redundant set of the same applications (3) using spack [24] "containerize" to generate multi-stage builds with one large layer that included a main application and all dependencies. This exercise will demonstrate the impact to building strategy on overall image similarity, and consequently, redundancy of layers that can influence cost. All container Dockerfiles are available [25].

E. Image Bases

For our next analysis, we want to classify our images for the underlying base image, which typically falls in the set of operating systems including debian, alpine, ubuntu, centos, fedora, rockylinux, and busybox. This task requires the unique resource identifier that can be used to pull the actual image layers for analysis. To do this assessment, we first reduce our entire set of images to inspect just one tag for each, choosing either "latest" or (if not available), the newest dated tag. We do this because different tags belonging to the same URI do not typically vary with respect to the base operating system, and we can estimate the unique resource identifier of the image from one single tag. We also have to be selective due to the need to pull the entire image and extract the contents to the filesystem, which can be both expensive in time and cost for internet bandwidth.

Since there is no single, reliable way to derive a base operating system, we will use a simple strategy to compare the extracted filesystem paths in the image to a known database of operating system paths. This approach is enabled by the "guts" software [26] to extract the container image to the filesystem, and generate a complete manifest of file paths and environment paths. Using this manifest, we can compare each extracted image fileystem against a database of 46 base extractions across tags of those named base images [27]. This means that, given a contender image A that needs classification and a set of base images B:

- Extract paths from A (PA)
- Generate a set (intersection) of paths across B. This represents shared paths in the base images B that could not be used to distinguish them (PB)
- Subtract this set of paths from A.
- This provides distinct paths in A. (PA - PB = AP)
- AP is a combination of application-specific additions to the base image, and non-shared base image paths.
- Compare each base image paths Bi with AP to generate a score Si (Si = intersection(AP, Bi) / len(AP))

By scoping the denominator to the set of paths in AP, paths in Bi shared with other base images are again discarded. (Si = intersection(AP, Bi) / len(AP)). The score calculated above represents the percentage of extracted filesystem paths in A that are also present in Bi, a given base image set of filepaths after removing any shared paths between base images. While some of these paths will be added software installs, the remainder will be paths that identify a base image family. Given no additional software installs, all of these paths will be derived from the base image, and the maximum similarity score is 1. Given no overlapping paths between AP and Bi, the minimum score is 0. When we calculate similarity of our contender image A with all base images Bi, the maximum score is declared the matching base image.

F. Building Best Practices

In addition to word2vec embeddings generation for similarity calculations, the Dockerfile database can be used to make observations about image building best practices.

G. Image Pulling Strategy

a) Number of Layers

While the maximum number of layers for a Docker image is known to be 127, the implemented maximum set by the Docker client is in fact 125 [28]. In practice this limit is determined by the kernel version, and so different container building tools can vary in allowances. For the purposes of this work, we will test an upper limit of 125 layers, as a majority of scientific container developers build with docker [29]. The question remains for build practices, given a constant size, whether it is a better strategy to choose few large layers, or more smaller layers. And secondly, given some number of layers, how does pull time vary with image size and choice of network or registry? Toward this goal, we developed a docker building tool to generate images with a controlled total size, and number of unique layers [17]. We used the Dockerfile database to calculate a range of image sizes based on percentiles between the 25th and 100th (Table I) as a strategy to reflect images being used by the community. For each of these sizes, we will perform a container pulling study that generates the respective size at a range of layer counts that are equivalently derived from the data. The sizes were chosen at percentile increments of 5, with the exception of the range between the 95th and 100th percentile, which was broken into an additional set of three ranges due to the larger span between the values.

For the study, we chose two values for the number of layers – the median (9) of the dataset, along with the upper max of 125. For each pair of image size and layer count, the size of the layer is calculated as the total size stated above divided by the number of layers. We will only build images for which the minimum size is within the allowances of a standard registry (10MB or smaller).

The experiment will use Kubernetes, the de facto standard container orchestration framework for cloud and Fortune 500 companies [2] and be run on Google Kubernetes Engine (GKE). As each container is assembled from layers with a different generation command, there is no chance for overlap of file names so the cache can never be used when pulling images. The study will be run on each of 4, 8, 16, 32, 64, 128, and 256 n1-standard-16 nodes on Google Cloud, with 16 vCPU and 60GB RAM per node. Initial testing was done to pull containers on the n1-standard-16 and a larger instance, n1-standard-64 (64 vCPU and 240GB memory), and n1-standard-16 was chosen as the n1-standard-64 was only

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

1.028x faster, but 3.87x more expensive. For each container, a Job [30] will be created that requires pulling the container to all nodes, and the Kubernetes Event Exporter [31] will be used to capture all events from which pull times and errors can be derived. Importantly, this tool requires setting the max age of events to a large value (1200 seconds) or else events can be dropped. The experiment will be conducted multiple times, each time optimizing a different part of the setup to give actionable advice about pulling practices. At the end of this first experiment, there will be data for deciding on one or more image sizes and number of layers for subsequent experiments to assess pulling strategies across nodes [32], discussed next.

b) Local vs. Remote Registry

The location of the registry relative to the final destination of the pull can be a salient factor to pulling latency, where sources that are physically closer to their destination might see improvements in latency and pulling time. To test this approach, we will pull the application container set from GitHub packages (ghcr.io) and then directly from the registry provided by the cloud where the experiments are run, Google Artifact Registry (gcr.io).

c) Filesystem Latency

Input/Output operations per second (IOPS) and throughout can be hugely influenced by the filesystem available to the Kubelet, resulting in a 3x improved throughput [32] and thus, faster image pulls as the layers are streamed to the filesystem and then extracted. With this knowledge, we aim to test adding a single 375GB LocalSSD to each node in the cluster to which the images will be pulled.

d) Streaming Images

While our experiment containers did not contain a real application (they start and complete) it is worth testing image streaming, where images are allowed to enter a running state before the entire image is downloaded. This is done by way of starting containers with content that is recorded to be accessed at the onset of the container running, and then loading content that is needed on demand – a strategy called lazy loading [33]. While it cannot be known exactly how Google Cloud has implemented this approach, an open source tool to perform this task is the SOCI (Seekable OCI) snapshotter [33], a containerd plugin that creates an index of image contents, and then is able to start the container before download finishes. This is possible because, as Harter et. al showed [11], only 6.4% of container data is needed for this step. In a real-world application scenario, this would hugely reduce costs because, although the image pull still needs to complete, the pull itself does not slow down the starting time of the workload. While we cannot determine if Google Cloud is using this exact snapshotting, the project that SOCI derives from, the stargz snapshotter [34] came from a project developed by Google engineers. The lazy loading approach has been documented to speed up a workload start time by 6.3x [32]. On Google Cloud, image streaming requires using Google's Artifact Registry, which does incur additional costs.

e) Real-World Application Test

Given a pulling strategy that is shown to be optimal in the previously stated experiments, we would finally want to test the approach with real applications. The reason is, especially for the image streaming strategy previously mentioned, the ability for the container to start depends on the logic of the entrypoint. As our simulated containers do not have real entrypoints or applications, the times for the streaming images could be unusually or unrealistically fast. Toward this goal, we can use a subset of the spack containers described in Section II-D1 (LAMMPS, OSU All Reduce, AMG, and Minife) that can guarantee that the application and dependencies are contained within one layer, and the experiment is testing actual applications that can be validated to run and return a result.

f) Node Coordination

As a final investigation in the study, we want to investigate the extent to which nodes in the cluster are coordinated for events, including a pod being scheduled, a container pulling, pulled, created, and started. If these events are not orchestrated in unison, given a workload that requires all containers running at the same time, it could further delay the application start and incur additional cost.

III. RESULTS

A. Dockerfile Ecosystem

We used the Research Software Encyclopedia to identify 4,621 associated GitHub repositories. Of that set, 694 had at least one Dockerfile. In total, we find 77,449 Dockerfile across research software engineering and machine learning projects to further explore.

B. Image and Layer Sizes

For each of our 77,449 Dockerfile, we retrieve complete metadata about tags available and configurations from the

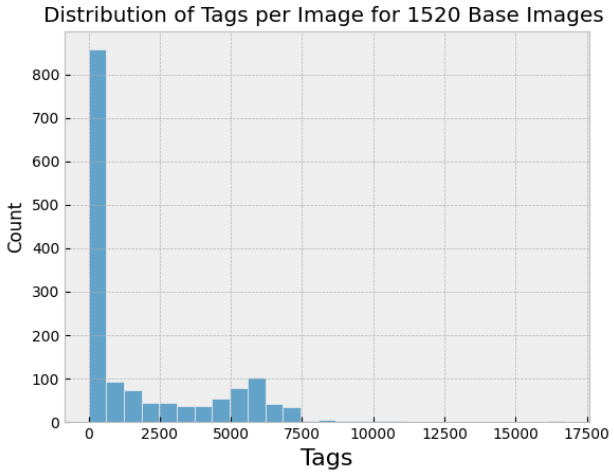

Figure 1. Tags per image with one outlier removed.

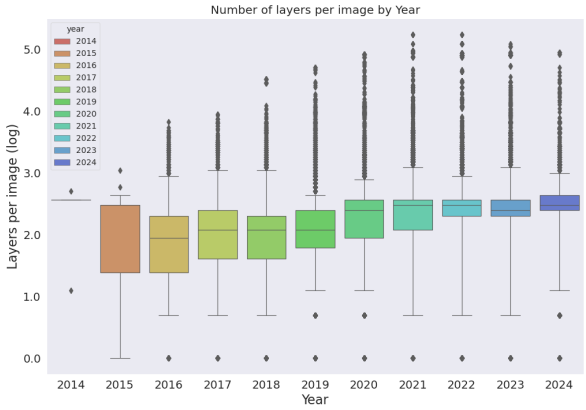

Figure 2. Layers per image by year shows a fairly consistent mean trend with smaller variance and an increase in outliers.

respective registry endpoints. Despite the large number of Dockerfile, the set of unique base images (each with some number of underlying tags, image configurations, and manifests) was much smaller, (2,132) and with huge variance with respect to the number of tags. With one outlier removed (47,428 tags for nix/nixos) (Figure 1), the number of tags ranges from 1 to 16,748, with a mean of 1842 and standard deviation of 2,531 tags.

This tells us that there is quite a bit of variation with respect to release frequency across our set, as each tag is typically indicative of a version or release. We are first interested in the number of layers across images and tags, and how this has changed across time. Seeing that the number of layers has changed over time might reflect a change in build practices. In Figure 2 we see this result for the decade between 2014 and 2024, and while the variation has increased slightly (meaning some images have many more layers) the general means are the same (16.58 +/- 23.66) across time, visually suggesting that people are not building images with significantly more layers. In this exercise, we also found several images from the RedHat registry (8 repositories with a total of 205 tags) with greater than 127 layers. This was an unexpected finding that challenged our “common” knowledge that images could not exceed 127 layers.

We can then use the manifests, which contain layer sizes in bytes, to look at the change in size over this same period (Figure 3). In this figure we see a different pattern – that images are indeed getting larger.

Finally, we might ask how often layers are repeated. This depiction is biased to our dataset, which would more likely have common layers between different tags from the same image. Even still, for a set of 528,449 unique digests (layers) the count of replicated layers drops off quickly, with only 120 instances of a layer being repeated more than 500 times, 52 instances of greater than 1000 times, and only 4 repeated more than 4 times. This data is presented in Table II.

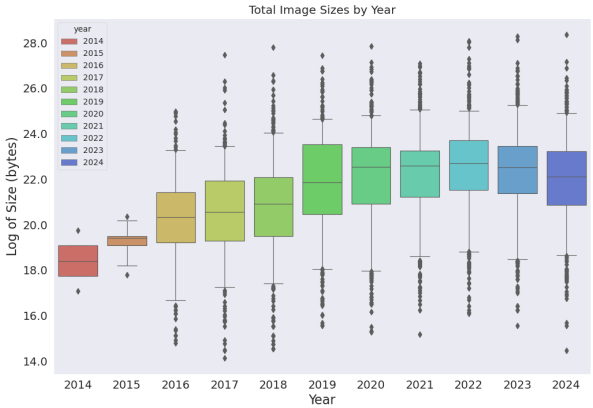

Figure 3. Total image sizes (sum of layers) by year. Outliers with more than 127 layers, the declared maximum, do in fact exist in the RedHat registry.

Interestingly, we discovered an outlier in this set - a layer that appeared to be repeated 67,897 times:

sha256:4f4fb700ef54461cfa02571ae0db9a0dc1e0cdb55
77484a6d75e68dc38e8acc1

Further investigation revealed this was an empty set of 32 bytes that was often associated with a WORKDIR directive in the Dockerfile, but only for cases where the directory already existed. Discussion with OCI maintainers revealed that there is an “empty layer” flag in the image configuration. If the tooling decides not to set the flag, the tool must ship a valid tar+gzip, and that, even without any files being packaged, takes up some space for the tar and gzip headers. This is the empty layer we discovered that when extracted results in the digest that we found. This was implemented before it was realized that /dev/null is an actual valid empty tar file.

TABLE II. REPEATED LAYERS ACROSS DATABASE

Threshold	Count
= 1	312095
>1	216354
>2	136054
>50	9255
>100	3333
>500	120
>1000	52
>3000	4

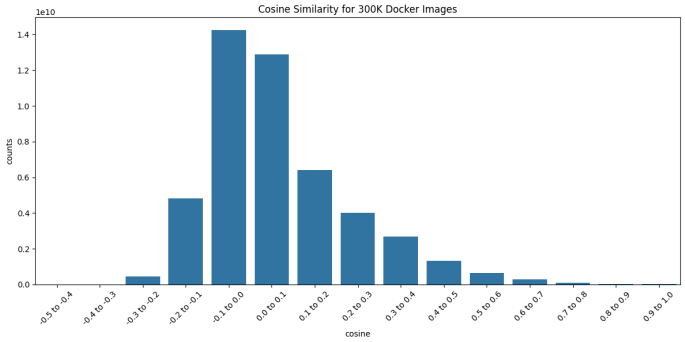

Figure 4. Distribution of image similarities across 300K Docker images, where each is calculated from image build history from the manifest configuration.

C. Content Similarity

We next used the text of the unique layers from the base corpus (N=528,449 layers) to derive both image and layer similarity. We started with 2,132 manifests and treated the layer "history" lines as sentences that make up a document, deriving a set of 309,317 documents. The word2vec embeddings generated from the tokenized documents were then used to calculate pairwise cosine similarity (Figure 4). The cosine matrix generally shows that the bulk of images are not very similar at all, with cosine scores under 0.2.

When we apply the same processing technique to the text from the original scientific Dockerfile (N=77,449) RUN statements we see a similar pattern (Figure 5).

We next want to assess layer similarity. This calculation was more challenging, as we have a total set of 6,535,425

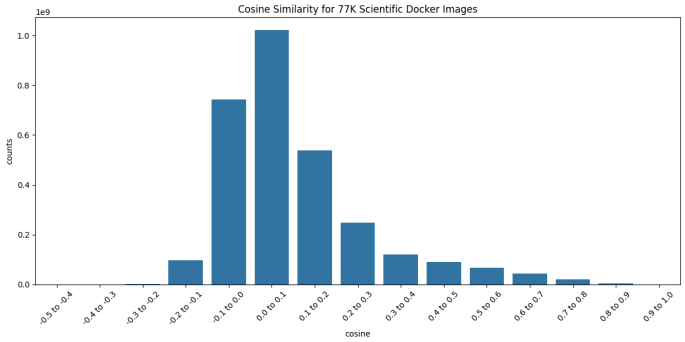

Figure 5. Distribution of image similarities across 77K scientific Dockerfile.

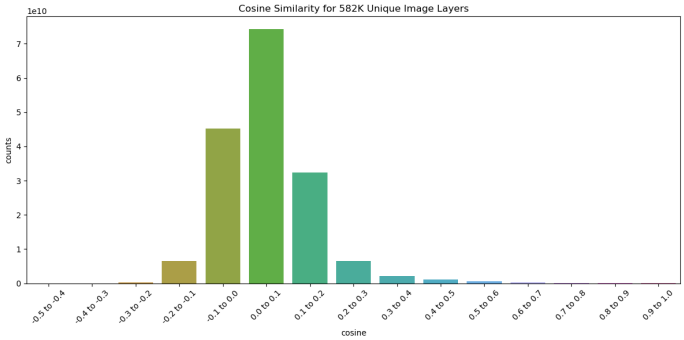

Figure 6. Distribution of layer similarities across 582K unique layers

(non-unique) layers across images. We chose a strategy that removes exact duplicates (typically equivalent layers between temporally close tags of the same image), and calculate from the reduced set. Since we are explicitly removing exact duplicates, our goal would not be to say something globally about the ecosystem, but say something about similarity of layers that are not exactly the same. When we tokenize and process and filter down to unique, ensuring that layers from images from the same tag are removed, we have 597,591 layers from base images. When we calculate similarity scores across these layers, we see a similar pattern (Figure 6) where most layers are largely not similar.

At a high level, what we can see from this small analysis is that most layers are not re-used across images.

D. Image Bases

When we classify a subset of our images (Table III), removing the version of the image, since we are biased to select for newer images, we find the majority have a debian base, followed by alpine and ubuntu. We also see that values in the similarity score distribution are generally high (Figure 7), indicative of shared paths and thus confidence in the classifications. The minimum score in the above is 0.59, and the maximum is 1. We see that debian is by far the most frequently used, at least for this sample of images we are looking at.

TABLE III. BASE IMAGE CLASSIFICATION

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

E. Building Best Practices

In addition to word2vec embeddings generation for similarity calculations, we can use our database of 77K Dockerfile to make observations about image building best practices.

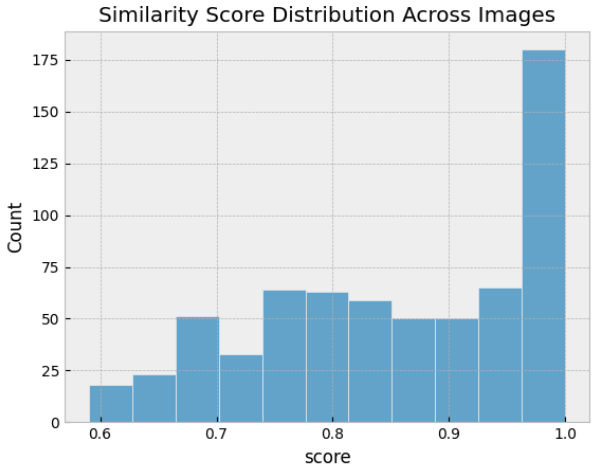

Figure 7. Distribution of image similarity scores used by the "guts" software to derive the base image labels.

1) Limit layers amount

While the maximum number of layers allowed in most registries is 127 and there is no strong guidance on how many layers are good for a single image, we can see from Figure 2 that the layer count has remained relatively stable over time with large variance (16.58 +/- 23.66). We might guess that in practice, people do not explicitly attempt to build images with the fewest layers, but rather build exactly what they need or is easiest.

2) Multi stage builds

Multi-stage builds are useful for separating builds into stages, such as compiling an application and then providing the final binary and libraries in the final image. They are indicated by way of finding more than one FROM statement in the Dockerfile, and considered best practice in that they can reduce the size of the final production image. When we parse our repository of 77K Dockerfile build recipes we can look for greater than 1 FROM statement to indicate such a build. In this set, we find a total of 1984 Dockerfile, which represents 2.56% of image builds.

3) Docker official images

While a Docker "verified" image can come from the docker official images, sponsored open source, or verified publisher, we chose to look explicitly for Docker official images (e.g., ubuntu) as these are provisioned directly by Docker Hub with provided scanning and security checks. We can detect which images are in this set by way of looking at the FROM unique resource identifier. If it has docker.io or library or is missing the registry name (which then will default to the Docker Hub registry) we have found a docker official image. In our database, we found a total of 11,439 images that use a Docker official image, representing 14.77% of the entire set.

4) Latest image

It is conventional wisdom to not use a "latest" tag, the reason being that it is a moving target and can hinder reproducibility. When an image "latest" updates the operating system version, image builds can break as library names or availability can change over time. For our set of 77K Dockerfile, we looked for images that would pull a "latest" tag by way of providing it directly in the unique resource identifier, or leaving out the tag entirely (which defaults to latest). Of the set, we found 4,114 Dockerfile that use a latest image, representing 5.3% of the entire set.

5) Pinned image digest

It is considered better practice to pin an image digest directly, which is more granular than a tag in that it represents an exact build of a base image for a point in time. In our set, we looked for these digests in the FROM statement by searching for the string "sha256," which is the hashing algorithm used for this purpose, and the correct way to specify using a digest. In our set we found only 74 Dockerfile (0.09% of the set) used a pinned digest, reflecting that the practice is not common.

6) Using apt get with apt install in same line

For debian or ubuntu images, it is recommended to use apt get with apt install in the same line to properly use the apt cache. Across our Dockerfile database, for the subset of layers that use apt get (507,695 across Dockerfile images), the large majority (478,742 layers, or 94.3%) take this approach.

7) Using apt get with a clean / autoremove

Since each layer is an isolated unit, and files that are added (and not removed) between layers can lead to bloated layers even if they are cleaned up, it is advisable to remove lists and clean. While debian and ubuntu images automatically run apt-get clean [35], this is arguably still a good practice when applied to other package managers and methods to install software. For our Dockerfile dataset, we find that of the subset that use apt, 67.8% do a clean (clean or autoremove), 0.048% do only an autoremove, and 11.19% do both.

F. Impact of Build Strategy on Digest Similarity

To demonstrate the influence of container building strategy on resulting container layer similarity, we aimed to compare overall layer digest similarity between 10 applications that were built in multiple ways. The spack builds for three containers (low redundancy strategy) were not successful and were not used in the analysis. The Jaccard scores are shown in Figure 8 and summary metrics in Table IV.

Interestingly, the performance study set has a cluster of images that are more similar than the best effort set, likely resulting from having overall a larger number of matching layers between images.

The best effort set of builds (middle panel in Figure 8) that have fewer overall layers would require 33/118 (28%) unique layer pulls. Since we are certain that these containers were built with redundancy of layers in mind, we can state

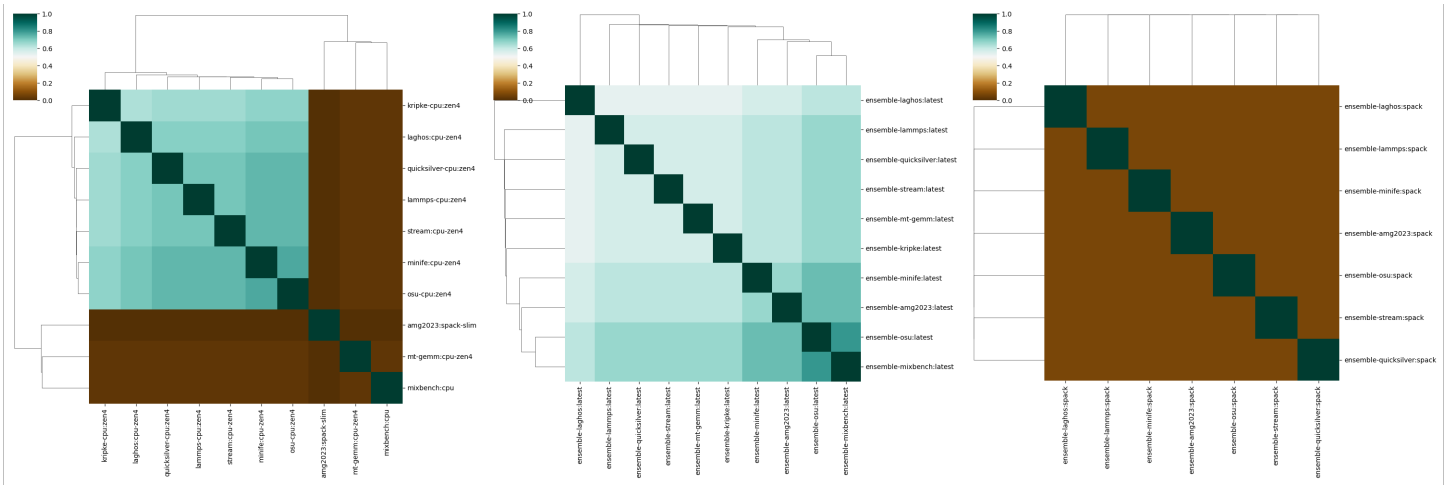

Figure 8. Jacaard scores for three build strategies. A reasonable effort to ensure redundancy (left) that produces a mixture of similar and dissimilar images, a best effort strategy (middle), and a build tool that eliminates the possibility for redundancy (right).

TABLE IV. 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)

that this 28% represents the application logic specific to each container. For the performance study where some care was taken for redundancy, 115/258 (45%) of layers would require isolated pulls. Finally, for the spack build strategy that creates a large layer that consists of a custom spack view, 50/56 (89%) of layers would require unique pulls, a strategy that does not allow for large amount of redundancy. We can see this result reflected in the Jaccard similarity cluster maps in Figure 8. The exercise demonstrates that a choice of a build tool can have "trickle down" implications for experiment costs, and often unique application logic makes the task of redundancy a challenging one.

G. Image Pulling Strategy

We assessed the trade-off between number of layers and image size. We found no discernible impact to the number of layers and image pull time (Figure 9). Instead, total image size appeared to be the most important factor to increase pull time. We proceeded with subsequent experiments to only include the median (N=9) number of layers, and a set of 6 larger sizes between the 90th and 100th percentile of the Dockerfile dataset, ranging between 148MB and 19GB.

Pulling from a registry external to the cloud provider (Figure 10) made no difference to pull times as compared to pulling from a registry provided by the cloud (Figure 11)

However, pulling with the presence of LocalSSD (Figure 12) improved times, often by 20-40 seconds (approximately 1.25x), and made pull times more consistent between nodes. This is an advisable strategy as the cost of storage ($0.1046 per GB per month) is relatively inexpensive compared to the cost of running experiments.

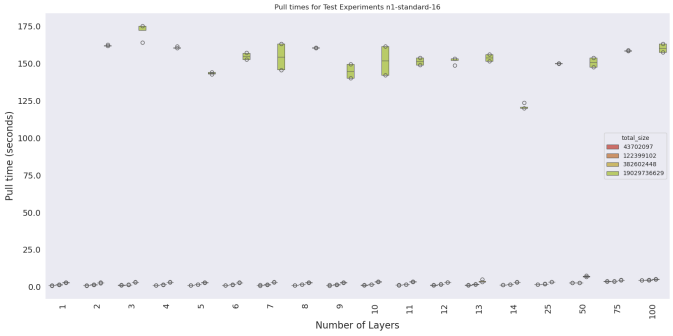

Figure 9. Testing the influence of number of layers across 4 image sizes and 18 different layer counts showed no discernible difference in pull times, but rather, suggested that size was a salient factor.

The most surprising and impressing result was using image streaming, which could reduce pull times down to close to 1 second, assuming to be assisted by a caching strategy [36]. This finding is demonstrated in (Figure 13), where the benefits of image streaming are fully realized after the initial pull of the experiment containers for the size 4 cluster. The caching strategy provided by Google Cloud that makes subsequent cluster pull times almost instantaneous persists across different clusters.

Extending images to a real-world set of applications, the improved pulling times when using image streaming as compared to not using it was still substantial, an approximate 15x improvement (Figure 14).

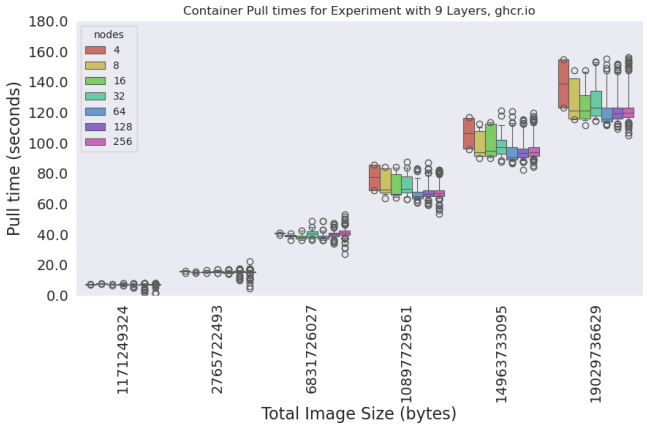

Figure 10. Pull times from a remote registry (GitHub packages) to Google Cloud showed an increase as the size of the container increased.

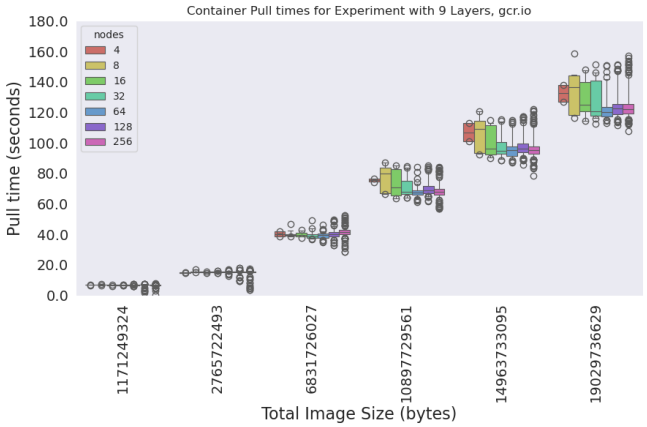

Figure 11. Pull times from a local registry (Google Artifact Registry) to Google Cloud in the same region did not improve pull times.

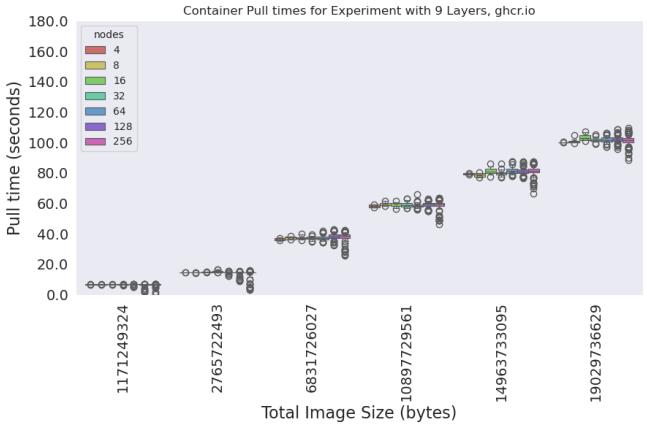

Figure 12. Pull times from a local registry (Google Artifact Registry) with an added local SSD improved pull times between 20-40 seconds and improved consistency of pull times across nodes in the cluster.

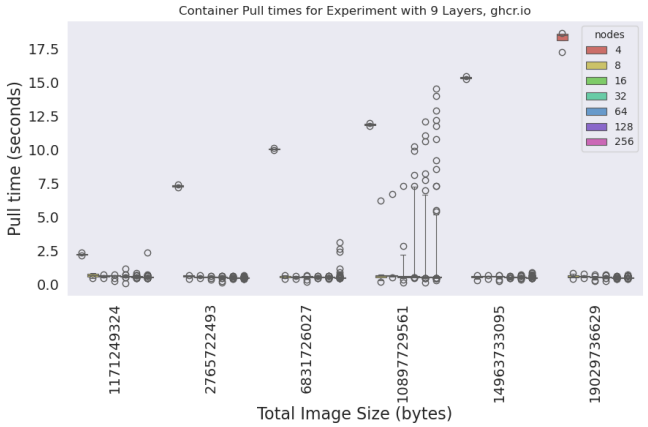

Figure 13. Image streaming pulling times across image sizes. The ability of the container to transition to running was consistently close to 1 second due to caching. Size 4 demonstrates that the first pull of a specific container in Google Cloud benefits from image streaming, but is not instantly available as it is not cached. Larger sizes benefit from a caching strategy [36]

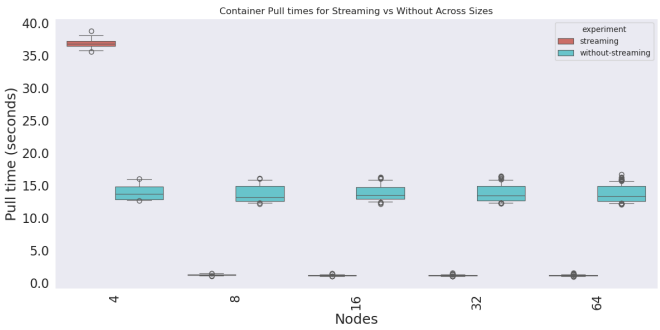

Figure 14. Image streaming pulling times across application images. The reported time for the container to start running was approximately 15x faster for applications LAMMPS, OSU All Reduce, AMG, and Minife. The smallest experiment size did not benefit from Google Cloud caching.

H. Node Coordination

Despite pull times not increasing as cluster size increases, the total time to run an experiment increased with number of nodes, resulting in 1383.65, 1392.84, 1408.48, 1426.84, 1535.19, 1940.99, 2884.18 seconds for our initial experiments for sizes 4, 8, 16, 32, 64, 128, and 256, respectively. This suggests that additional time is accumulated elsewhere, and the results of the node coordination tests give a hint to the source of this extra time. Figure 15 shows time differences between events across nodes. This is calculated as, for each container, the earliest timestamp recorded for the event subtracted from the latest across nodes. Doing this calculation across cluster sizes shows us the extent to which an event for a specific container is coordinated. A time of zero indicates that the nodes across a cluster had the event occur at the same time, while a value above that represents a stagger from that. These plots demonstrate that as the size of both containers and clusters increase, so does the variability of events for it between nodes – the largest container (19GB) on the largest

cluster size (N=256 nodes) has minimally two nodes that are pulled, created, and started approximately 50 seconds apart. This finding that extra time is accumulated as clusters get larger, a likely result of needing to wait for the slowest node across a large set, is interesting and warrants further exploration for behavior and solutions.

IV. Discussion

In this work, we amass a database of 77K Dockerfile and do a complete assessment of the trends and container ecosystem since 2014, observing that the number of layers has largely not changed, but image sizes are slowly getting larger – a trend we expect to continue with the growing number of machine learning images that are entering the ecosystem. We derive build practices from the data, noting that debian is the most popular base image, redundancy of layers is uncommon, and good practices to pin digests and perform multi-stage builds are uncommon. We finish our study with a set of experiments that first visually show the change in image similarity based on digests for three building strategies, and then performing a comprehensive pulling study that demonstrates using local SSD and a streaming approach can greatly reduce the time between onset of pull and having a running container. It was a surprising result that pull time does not increase with the number of nodes in the cluster, and that other scaling issues must be responsible for longer experimental runs on larger clusters. This finding is interesting and warrants further work.

While Google Cloud offers image streaming easily as an add-on to GKE, no similar easy install method exists for Amazon Web Services Elastic Kubernetes Service (EKS) and so as a supplement to this work we developed a daemonset [16] to automatically install the SOCI snapshotter to a cluster. We anticipate doing further work in the space of snapshotter plugins to further optimize how application assets are loaded with cache pre-fetching and on demand. From these observations, we recommend to the reader to use a streaming image approach when a registry is available that can provide the indexed images, and if not, to fall back to using local SSD for improved pulling times.

A. Docker Layers

The finding that Docker has references that set limits to each of 125 and 128 layers for the overlay fs driver was interesting and worth further exploring. As containerd does not set any maximum, we were able to test building and pushing the image “docker.io/tianon/test:many-layers-256” and it was successful. The limit was originally enforced because there were early issues with mounting layers (length of an argument to a syscall) that led to technical maximums. However, this early issue may not be relevant depending on the host operating system, kernel version, and container runtime being used. Different tools take different approaches to validating this -- – containerd and buildkit use a practical approach that does not enforce any checks, but then would propagate the error on mount failure, meaning that the limits are controlled by the kernel. Other tools like Docker hard code manual checks in the

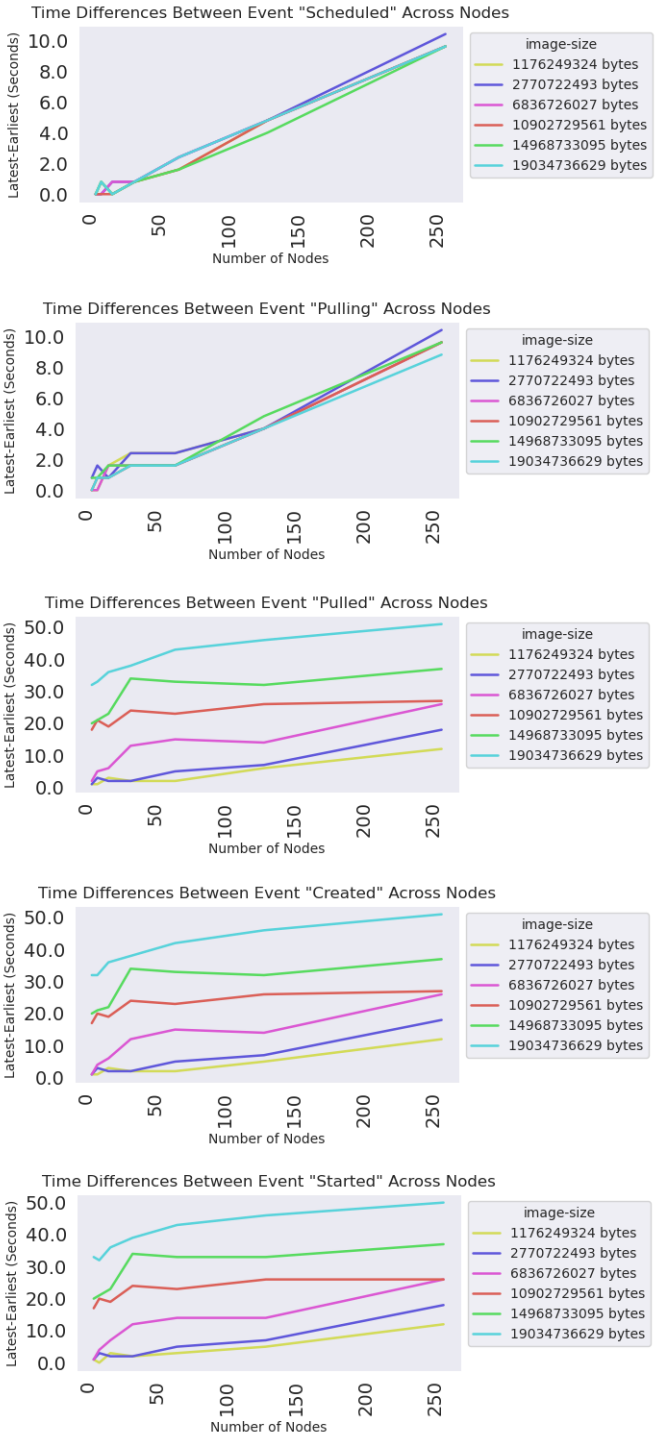

Figure 15. Time differences between events across nodes. This is calculated as the maximum - minimum timestamp across nodes for a cluster sizes, reflecting the extent to which an event for a specific container is coordinated. A time of zero indicates that the nodes across a cluster had the event occur at the same time, while a value above that represents a stagger from that.

code, which might fail earlier, but do not always reflect the true limit enforced by the user's particular kernel. With respect to Docker, the failure often comes after pulling the layers, which arguably is a check that should be done earlier. This was an interesting finding because it represents a cultural practice and established knowledge that is more of a gray area. In practice, many empty or metadata layers are relatively harmless since they are ignored or not relevant to image extraction.

B. Limitations

We recognize that our choice of a software database that would provide scientific images is only a slice of the entire container ecosystem, and this choice was intentional to not include many service-oriented images that might run websites, databases, or other applications not directly related to science.

While using a different compression algorithm can also reduce extraction, we aimed to test solutions that were readily available in the common software being use to build containers, which typically is not containerd [29]. Another viable solution that was not tested here is to preload base images using a daemonset [32], a great idea given containers with a large shared base image. That approach would not have fit our study as our base image was chosen to be minimal and insignificant to the pull time. While we used Google Cloud for this work, the use of the open source project Kubernetes that is available across clouds, and general pattern to use a more performant filesystem can be applied to other cloud environments. A logical next stage of work is to understand how patterns of application data retrieval work with various pulling strategies. For example, requiring download of large data after container startup could have a detrimental effect to application performance. In these cases, optimizing an initial pull to better run in parallel could be an optimal choice.

V. Conclusion

Best practices are often prescribed with little attention to how reasonable they are, or how well they fit into a user workflow or incentive structure. Our work demonstrates that the number of layers is not a salient variable to worry about, but rather total image size. Our takeaways are that 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. This improvement becomes more salient when using expensive resources such as GPU, or an auto-scaling strategy that provisions new nodes that do not have images cached. We suggest container streaming as an ideal strategy for quickly starting containers that are large, however caution should be used if large amounts of new data are needed for application execution later in the run. Local SSDs can consistently improve performance without these detrimental effects.

While we cannot say that these benefits extend to other clouds, those using Google Cloud should consider pulling images to a smaller cluster first to take advantage of caching along with image streaming. Layers should (and cannot) go over the registry limit of 10GB, and given this limitation, developers will need to consider strategies for provisioning large models that are intended to be used with containers. As ML images get larger it will be more important to address these issues.

Finally, we suggest to the reader that although the specific strategy chosen for building and pulling might vary based on the experimental resources and application characteristics, it is responsible to have awareness about costs, and strategies for improvement. Given contention for resources such as GPUs, there is an opportunity cost of the extra time used on the resources. Nodes that have excess pulling time are not available for anyone else to use. We encourage the community to think about the costs of their experiments, and to further explore this interesting space of work.

Acknowledgments

Thank you to the AWS EKS team for interesting discussion on the SOCI Snapshotter, and Daniel Milroy for his feedback on the manuscript. Thank you to the OCI Slack for discussion on layers, and to the larger HPC and cloud communities for continuing to make this space of development interesting and gratifying to work in.

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 LLNL-CONF-871325

References

[1] D. Moreau, K. Wiebels, and C. Boettiger, "Containers for computational reproducibility," *Nature Reviews Methods Primers*, vol. 3, no. 1, p. 50, 2023.

[2] Honeypot, *Kubernetes: The documentary [PART 1]*, Jan. 2022.

[3] L. W. et al., "Bare-metal vs. hypervisors and containers: Performance evaluation of virtualization technologies for software-defined vehicles," in *2023 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, 2023, pp. 1–8.

[4] J. Baumgartner and L. et al., "Performance losses with virtualization: Comparing bare metal to vms and containers," in *International Conference on High Performance Computing*, Springer, 2023, pp. 107–120.

[5] G. Hu, Y. Zhang, and W. Chen, "Exploring the performance of singularity for high performance computing scenarios," en, in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, IEEE, Aug. 2019, pp. 2587–2593.

[6] S. Buchanan, J. Rangama, and B. et al., "Container registries," *Introducing Azure Kubernetes Service: A Practical Guide to Container Orchestration*, pp. 17–34, 2020.

[7] O. Containers, *Opencontainers/distribution-spec*, en, 2024.

[8] O. Containers, *Opencontainers/image-spec*, en, 2024.

[9] *Dockerfile reference*, en, https://docs.docker.com/reference/dockerfile/, Accessed: 2024-10-2, Sep. 2024.

[10] *Moby/moby github issue*, en, 2024.

[11] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 181–195.

[12] Honeypot, *Kubernetes: The documentary [PART 1]*, Jan. 2022.

[13] I. Docker, *Multi-stage*, en, https://docs.docker.com/build/building/multi-stage/, Accessed: 2024-10-2, Sep. 2024.

[14] N. Zhao, V. Tarasov, and A. et al., "Large-scale analysis of docker images and performance implications for container storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 918–930, 2020.

[15] J. L. Chen, D. Liaqat, M. Gabel, and E. de Lara, "Starlight: Fast container provisioning on the edge and over the WAN," *NSDI*, pp. 35–50, 2022.

[16] V. Sochat, *converged-computing/soci-installer: soci installer release 0.0.0*, version 0.0.0, Oct. 2024. DOI: 10.5281/zenodo.13895822.

[17] V. Sochat, *converged-computing/container-crafter: Container Crafter v0.0.0*, version 0.0.0, Oct. 2024. DOI: 10.5281/zenodo.13871919.

[18] I. Docker, *"best practices"*, en, https://docs.docker.com/build/building/best-practices/, Accessed: 2024-10-2, Sep. 2024.

[19] V. S. et al., "The research software encyclopedia: A community framework to define research software," *Journal of Open Research Software*, vol. 10, no. 1, p. 2, Mar. 2022.

[20] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[21] I. Docker, *Cache*, en, https://docs.docker.com/build/cache/, Accessed: 2024-10-2, Sep. 2024.

[22] Mikal, *Interpreting whiteout files in docker image layers*, en, https://www.madebymikal.com/interpreting-whiteout-files-in-docker-image-layers/, Accessed: 2024-10-2.

[23] V. Sochat and D. M. et al., *Converged Computing Performance Study Release v0.0.0*, version 0.0.0, Sep. 2024. DOI: 10.5281/zenodo.13738496.

[24] S. Shudler, N. Ferrier, and I. et al., "Spack meets singularity: Creating movable in-situ analysis stacks with ease," in *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2019, pp. 34–38.

[25] V. Sochat, *converged-computing/ensemble-containers: Ensemble Containers*, version 0.0.0, Oct. 2024. DOI: 10.5281/zenodo.13887339.

[26] V. Sochat, *Singularityhub/guts: Release v0.0.1*, version 0.0.1, Mar. 2023. DOI: 10.5281/zenodo.7703378.

[27] V. Sochat, *singularityhub/shpc-guts: Singularity Registry HPC Guts v0.0.1*, version 0.0.1, Mar. 2023. DOI: 10.5281/zenodo.7703380.

[28] D. Community, *Docker documentation: Number of layers is not documented*, en.

[29] V. Sochat and C. K. et al., *Hpc containers community survey 2024*, May 2024. DOI: 10.5281/zenodo.11206333.

[30] S. Section, *Jobs*, https://kubernetes.io/docs/concepts/workloads/controllers/job/, Accessed: 2023-9-1.

[31] R. Authors, *Kubernetes-event-exporter: Export kubernetes events to multiple destinations with routing and filtering*, en.

[32] T. He and W. Chiang, *Tips and tricks to reduce cold start latency on GKE*, en, https://cloud.google.com/blog/products/containers-kubernetes/tips-and-tricks-to-reduce-cold-start-latency-on-gke, Accessed: 2024-10-2, Jan. 2024.

[33] AWS, *Soci-snapshotter: A containerd snapshotter plugin which enables standard OCI images to be lazily loaded without requiring a build-time conversion step*, en.

[34] K. Tokunaga, *Startup containers in lightning speed with lazy image distribution on containerd*, en, https://medium.com/nttlabs/startup-containers-in-lightning-speed-with-lazy-image-distribution-on-containerd-243d94522361, Accessed: 2024-11-10, Apr. 2020.

[35] D. Community, *Contrib/mkimage/debootstrap at 03e2923e42446dbb830c654d0eec323a0b4ef02a · moby/moby*, en.

[36] G. Cloud, *Use image streaming to pull container images*, en, https://cloud.google.com/kubernetes-engine/docs/how-to/image-streaming, Accessed: 2024-11-10.