LANDLORD: Coordinating Dynamic Software Environments to Reduce Container Sprawl

Tim Shaffer*, Thanh Son Phung*, Kyle Chard†, Douglas Thain*
University of Notre Dame* and University of Chicago†

Abstract—Containers provide customizable software environments that are independent from the system on which they are deployed. Online services for task execution must often generate containers on the fly to meet user-generated requests. However, as the number of users grows and container environments are changed and updated over time, there is an explosion in the number of containers that must be managed, despite the fact that there is significant overlap among many of the containers in use. We analyze a trace of container launches on the public Binder service and demonstrate the performance and resource usage issues associated with container sprawl. We present LANDLORD, an algorithm that coalesces related container environments, and show that it can improve container reuse and reduce the number of container builds required in the Binder trace by 40%. We perform a sensitivity analysis of LANDLORD using randomized synthetic workloads on a high-energy physics (HEP) software repository and demonstrate that LANDLORD shows benefits for container management across a wide range of usage patterns. Finally, we compare LANDLORD to offline clustering, and observe that the continuous churn in software necessitates an online approach.

1 INTRODUCTION

Containers are becoming the solution of choice for describing and distributing customized software environments. Technologies such as Docker [1] and Singularity [2] are increasingly used to deploy complex applications at different computing sites, without requiring that each software package be manually installed at each site. Container images being (by design) completely self-contained greatly simplifies management and deployment, but also limits opportunities for sharing common components. Much in the same way that a statically linked executable contains a full copy of each library used, container images necessarily include a complete set of dependencies.

For complex and multi-tenant applications, on-demand generation of containers is no longer a constant, static overhead when setting up applications; it is rather a dynamically varying and resource-intensive part of the application infrastructure, requiring management as a first-class activity. Adding new users, updating applications, and executing in different environments all require the creation, distribution, and storage of containers with the necessary dependencies for a set of tasks. Over time, these containers multiply: as a user's workload evolves, different tasks need different software (with versions changing regularly as packages are updated), and new containers are generated. Creation of a specialized container environment for a given task (which often takes minutes before task execution can begin) is one of the biggest hurdles in providing responsive service to users. In addition, related containers share many elements so a significant amount of storage may be wasted due to logical duplication resulting from container sprawl.

Most container-based services are implemented with an architecture like Figure 1. Here a *container service* is responsible for handling requests and constructing containers to satisfy those requests. After creation, the container is stored in a container cache and then transferred to local storage on a compute node, for example on a cloud instance or a High Performance Computing (HPC) node, for execution. Containers are typically cached to service repeated requests.

We developed the LANDLORD algorithm to take advantage of high-level information about the functionality of a container rather than the particular build steps or container contents. The key insight for LANDLORD is that a container may include a superset of a task's dependency requirements, which (with appropriate choice of packages) would allow a single container to satisfy the requirements of multiple tasks. LANDLORD merges compatible requirements from multiple tasks rather than building and maintaining many task-specific containers. LANDLORD takes an incremental approach, considering only the current service request against the currently materialized containers, and allocating the container with the closest similarity metric. This makes LANDLORD effective in practice as an online resource management tool.

Based on logs of container launches from the online Binder service, we found that employing LANDLORD to manage the set of cached containers resulted in a 20% decrease in I/O activity along with increased re-use of previously built containers, with the total number of container builds reduced by up to 40%. In addition, we prepared a synthetic workload based on high energy physics (HEP) applications to examine LANDLORD's sensitivity to workload and application requirements. We demonstrate that LANDLORD can provide benefits over a wide range of operating conditions. Finally, we examine the quality of containers maintained by LANDLORD over time by comparing the container hit rate of LANDLORD to an offline clustering algorithm, which shows that LANDLORD is a more stable and robust solution to the problem of container sprawl. This work is an extension of a previous conference paper [3] that analyzes an additional application workload and explores LANDLORD's use in large scale serverless applications.

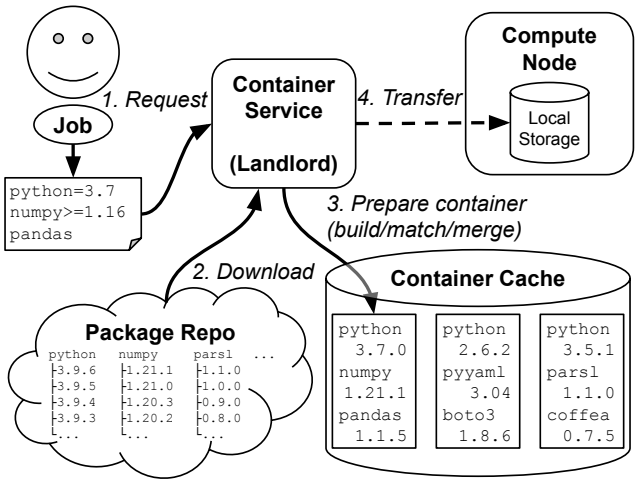


Fig. 1: Container Service Architecture.
In an on-demand container-based application, user jobs include a set of software requirements, given to a Container Service. The service may download packages from a global software repository to build a new container or identify an existing container that satisfies the dependency requirements. Either the container service or the compute system will then transfer the image to a compute node in a cloud/cluster. LANDLORD *is an algorithm that can be used by a service to manage container images.*

2 Background

The **container** has become a widely deployed tool for creating isolated, portable execution environments for complex applications. A container **image** is a filesystem image constructed from a declarative **specification** that indicates a sequence of actions necessary to construct the image. Docker [1] is a widely used example of a container management system, consisting of a local server that manages the container lifecycle, and a cloud service that permits publishing and sharing of container images. However, Docker is not widely deployed in the HPC context because its local service requires the use of node-local storage and elevated privileges. Instead, several alternative technologies have emerged, including Singularity [2], Shifter [4], and CharlieCloud [5], which make use of shared distributed filesystems for storage.

In these various forms, container technologies have become an integral building block for various services, applications, and computing paradigms: •**High Performance Computing (HPC) and High Throughput Computing (HTC)** applications often use containers in place of esoteric module and filesystem-based methods for configuring environments. Unlike persistent services, HPC/HTC workloads are expressed as a stream of discrete *jobs*, where each job may be associated with a pre-built container for execution. Some systems provide optimized container deployment mechanisms to avoid overloading the file system. HPC/HTC systems are used by many users and thus present opportunities for sharing containers to optimize performance.

•**Multi-tenant web services** like Binder, JupyterHub, and WholeTale [6] use containers to create customized execution environments for their users. These services dynamically create containers based on stated dependencies and in some cases by capturing environment changes by users.

•**Container orchestration systems** such as Kubernetes [7] allow users to declaratively define the high-level services/components of applications, while the orchestration layer manages the concrete resources (persistent storage, container instances, etc.). To aid in managing software environments in containers, Kubernetes package managers such as Helm [8] can instantiate specific versions of each software component and clean up outdated containers. These systems are often shared by many users with overlapping container definitions.

•**Workflow systems**, such as Parsl [9], and workflow languages, such as CWL [10], use containers to provide a common execution environment for tasks. While workflows typically operate on behalf of a single user, or small group of users, they may be composed of calls to different tasks with different environment requirements. This requirement presents an opportunity for container sharing.

•**Function as a Service (FaaS)** systems depend on containers to establish the environment for function execution. FaaS is an ideal use case for container sharing as environments are intentionally opaque, services are shared by many users with overlapping requirements, and FaaS providers aim to serve requests rapidly. Systems like funcX [11] are designed to use a container service to create containers on behalf of users, and deploy containers to compute nodes on-demand for function execution.

3 The Problem of Container Sprawl

We define **container sprawl** as follows: given a large (and probably growing) number of tasks that require many overlapping software dependencies, creating a container to fulfill each task's requirements will lead to a combinatorial explosion in the number of distinct containers in use. As mentioned previously, container images do not allow for sharing components as is possible with local installations, site-wide modules, or copy-on-write filesystems. Instead, each container carries complete copies of all components. In the naïve case, each variation in task requirements results in the creation of a whole new container. In this scenario many identical copies of common base components and dependencies are stored across a set of similar but non-identical container images. In our container workload, for example, users wrote 505 distinct version specifications for the popular Numpy package. These specifications could be potentially complicated range requirements (`numpy<1.20.0,>=1.18.0`), exact builds (`numpy==1.18.4=py37h8960a57_0`), or they may lack any version information at all (simply `numpy`). A single version of Numpy could satisfy all three of the above examples, though a naïve approach to container management would prepare a different container for each of the three requirements. Since each task-specific variation exists as a completely separate container, coarse-grained caching does little to alleviate this duplication. Only tasks with identical sets of requirements can reuse existing containers. This proliferation of container images to the point of management difficulty is well known along with the related phenomenon for VMs called "image sprawl" [12].

4 Container Management Challenges

For any choice of container management scheme, there is *some* non-trivial management cost. This could be in the form of time and manual effort on the part of individual users, or a portion of the system's compute and storage resources used to create, manage, or cache containers. We briefly review several naïve solutions and outline why they fail to address requirements.

Imperfect Solution: Caching. The simplest approach is to cache containers such that they can be redeployed quickly without creation overheads. This approach has low per-task overhead, and tasks and requirements may be updated as frequently as desired. At large scale, however, the overall system efficiency suffers. Due to duplication of packages among images, the cache must store many identical copies of common base packages. To support a given workload, it becomes necessary to provision a cache much larger than the size of any repository. With the software repositories examined in this work consuming several terabytes of storage, the amount of cache space required grows quickly. In the case of an extremely well-provisioned system, it might be possible to retain *every* image. For large-scale and high-throughput computing the total size of applications and data can often grow to consume *any* available resources, so most effectively utilizing available resources is key. Thus it is necessary to balance management costs against system compute and storage constraints. When supporting multiple users with a potentially large number of container images, simply adding storage capacity to accommodate each user or application is not a sustainable solution over the life of a system [13]. Rather, it would be preferable to make better use of what site storage is available by reducing unnecessary duplication among container images.

Imperfect Solution: Full-repo Images. Rather than considering the precise requirements for each task, another way to reduce the number of containers in use is to place an *entire* software repository into a single image, which can then support a large number of tasks. A complete copy of the Python Package Index (PyPI) would be over 300,000 packages (with nearly 2.8 million released versions of those packages) and consume approximately 8.8 TB (at the time of writing). Unfortunately, this approach is likely to exceed a number of practical limits on container size. Individual worker nodes may have limited local disk space and be unable to store large container images. Even if the large container fits, it is likely that a given task does not need *all* of the repository simultaneously, so it is wasteful to transfer unneeded data. This concept is a driving influence on projects like Slacker [14]. It also becomes prohibitively expensive to update and transfer such large container images. The US collaboration of the ATLAS, ALICE, and CMS projects have experimented with CVMFS applications on computing resources at various supercomputers in the United States including Cori at NERSC [15]. When full-repo images were built and scaled out onto a large number of nodes inside the NERSC infrastructure, the entire process of generating the image and distributing it to compute nodes took around 24 hours, making it difficult to deploy up-to-date versions of the software on a regular basis. In addition, the process requires the administrators' manual involvement in image creation, deployment, and cleanup. As additional projects want to take advantage of the resources at NERSC, the administrative burden of managing multiple CVMFS images on multiple software versions increases accordingly. Taking this approach negates the flexibility and hands-off administration that containers were intended to provide.

Imperfect Solution: Layering. Docker allows container environments to be composed from reusable image layers. Docker can take advantage of modern filesystems like BTRFS [16] that provide efficient snapshots and transparent sharing of files and directories between different revisions. As a practical matter, Docker is generally not available in HPC environments for administrative and security reasons. Likewise, guest users at large sites do not generally have the ability to directly manipulate file system snapshots or export/load local filesystem volumes. More conceptually, layering images addresses a different problem than the issue at hand. With Docker, base images can be extended and refined over time by appending layers. When preparing to run external computing tasks, however, we must compose a set of largely independent pieces to fit specific task requirements, without any particular ordering relationship to previous images. It is therefore difficult to map this set of semi-independent pieces into a linear sequence of refinements that will fit future tasks. Furthermore, since layer-based deduplication can only operate on *identical* layers, any modification requires storing the complete contents of the layer (and all subsequent layers, since the identity of a layer depends on its parents). This leads to significant duplication in practice, with 97% of files stored within layers on the publicly accessible DockerHub being duplicates [17]. Content addressable storage has been proposed as a solution to this issue [18], but requires substantial changes to the container infrastructure and is not compatible with static disk images required in HPC environments.

Imperfect Solution: Block Deduplication. Another potential avenue to address container sprawl is data deduplication for disk images. The virtualization community has developed a number of solutions for efficiently deduplicating disk images [19] and running virtual machines with many incremental changes [20]. There has also been extensive research on deduplication [21], [22] of filesystem data [23], [24] and disk blocks [25], [26]. These techniques can be quite effective for container deduplication at the block level [27], as it is not difficult to identify duplicated files or blocks within container images. However, we lack the means to combine the extraneous copies; each container image by design contains complete copies of all data, and sharing of data across images is not possible for users of the system. Container images are concretely stored as files that may need to be frequently transferred between different sites or uploaded to cloud computing environments. Block deduplication only works with deep integration with the low-level storage infrastructure at a single site, and therefore places significant limitations on storage infrastructure.

Given: Cached container image collection I
Input : Container specification s, maximum container size m, similarity cutoff
Result: Suitable container image satisfying specification s

```
// Conflicts have infinite distance
C ← {i ∈ I | s ⊆ i ∧ d(s; i) ∈ ℝ} ;
if C ≠ ∅ then
    // An existing image satisfies s
    return arg min_{i∈C} d(s; i);
end
D ← {i ∈ I | d(s; i) <    ∧ size(Merge(s; i)) < m} ;
if D ≠ ∅ then
    // Merge with closest image
    m ← arg min_{i∈D} d(s; i) ;
    return Merge(s; m);
end
// Couldn't re-use or merge
return Insert(s);
```

Algorithm 1: LANDLORD Algorithm

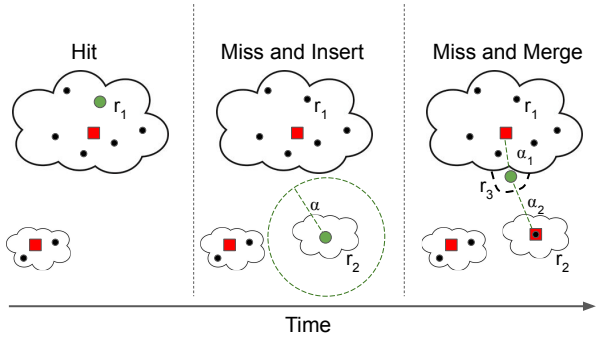


Fig. 2: LANDLORD Fundamental Operations

Hit: *A new request r_1 (green circle) is satisfied by a cached container (red square), so no further action is required.* **Miss and Insert:** *A new request r_2 is too far from all other containers (distance >) so a new container is created.* **Miss and Merge:** *A new request r_3 is close to an existing container so it is merged into the closest container, taking the union of the requirements from both.*

5 THE LANDLORD ALGORITHM

LANDLORD is an algorithm for managing a container store which makes online decisions to efficiently satisfy the dependency requirements for submitted requests. Rather than viewing a container as a sequence of shell commands to build layers or as a collection of arbitrary files, LANDLORD treats a container simply as an artifact that satisfies a set of requirements. It is therefore possible to check if an existing container satisfies a different set of requirements, or to combine sets of requirements to produce multi-functional containers. LANDLORD's main pseudocode and operations are shown in and Algorithm 1 and Figure 2, respectively. As each request to execute a task arrives, we consider whether the request is compatible with any existing container. If so, that is counted as a cache hit and the container is used to execute the task. On a cache miss, the distances between the request and all existing containers are measured using distance metric $d(a; b)$, where incompatible requirements are represented as an infinite distance. The choice of metric is discussed in section Section 5.3. If no container is within a critical distance , then a new container is inserted to satisfy the current request. Otherwise, the request is merged with the closest compatible container by adding the minimal packages needed to satisfy the request. If inserting or merging a container would overflow the available container cache space, then the least recently used container is removed. The result is that each request is satisfied by a sufficient container, and multiple requests may share a common container.

5.1 Container Management as Clustering

The problem of container management can be viewed as a variation on the general problem of clustering, shown in Figure 3. Briefly, the system considers a set of requests, each consisting of one or more constraints upon software packages. Multiple requests that are "close" may be gathered together into a cluster that can be described by the union of the package constraints. These constraints are solved (if possible) into a concrete list of packages that is then used to materialize one container image. The goal of the container management system is to find a suitable clustering, subject to two opposing constraints: each individual container must be small enough to deploy to an individual cluster node, and the sum of the sizes of all containers must fit in the shared cache space. However, there are several complicating factors that prevent the straightforward application of a conventional clustering algorithm:

1) The system must respond to requests in a timely way as they arrive. This requires an online algorithm that addresses both request similarity as well as cache resource constraints. To provide an acceptable service to interactive users, the total work for a single request must be bounded and limited to resources relevant to that specific request. A given user making a request should not "pay" the cost of operations that provide no benefit to that user.

2) The available operations that can be performed on containers are limited and relatively expensive. The manager may create a new container from a specification, merge new packages into an existing container, or delete an unused container. However, there is no fundamental capability to "transfer" packages from one container to another, short of deleting and recreating containers to effect the transfer. These operations may move GBs of data and take minutes.

3) Both the stream of requests and the state of the package repository evolve over time, as new packages become available and of interest to users. As a result, it is not generally possible to determine compatibility of specifications a priori, because they may have incompatible implied dependencies. Instead, it is necessary to evaluate the requests at a given point in time to determine compatibility, consulting the package repository to determine if packages actually exist that satisfy logically compatible requirements. In practice, change in specifications and software repositories over time is significant; our prior study of this dataset found that due to changes in the software repository, containers became out of date on average ten days after they were built [28].

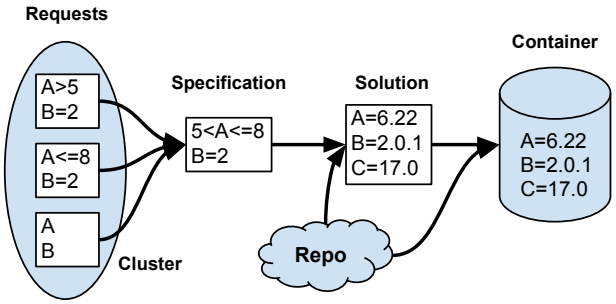


Fig. 3: Clustering Container Requests

Similar requests for packages may be clustered together, resulting in a common specification. The available packages in the software repository are used to generate a solution listing a set of concrete packages. The packages from the software repository are combined to build a usable container. The final container contents will constrain future merges with new requests.

5.2 Package-Level Coordination

Rather than treating each container as a black box of arbitrary files, we can consider it as a *set of packages* drawn from a package repository. While a build script gives a sequence of steps to produce a final container image, it does not give information about the desired properties of the resulting image. If we were building images by layering, there would be very limited options for optimizing storage or safely determining whether an existing image can be reused. Rather than trying to recover information from build scripts or previously built images, the specifications used to construct them offer higher-level information about their functional characteristics and more opportunities for management and optimization. Specifications provide minimal requirements that an image must fulfill without specifying anything about the exact image contents.

Specifications afford another opportunity to a container management system: unlike build scripts or recipes, it is possible to automatically manipulate or combine specifications. Since LANDLORD operates by composing sets of requirements, it is possible to add to or adjust a specification while guaranteeing that the requirements of a request are satisfied. A composite specification can be formed by first taking the union of all requirements from two or more specifications, then taking the intersection of all sets of versions for each repeated requirement name. For instance, Figure 3 shows the outcome of composing three different specifications into one. This kind of composite image could be used in place of any of its constituent specifications, since it meets the minimum requirements given in each. Note that in some cases, incompatibilities among requirements (e.g., packages or versions) make combination impossible.

While caching and merging specifications give a mechanism to reduce unnecessary duplication among stored container images, we still do not know which specifications to merge. Choosing randomly or by order of task submission, for example, is liable to join specifications with little in common. This would increase the sizes of images to be transferred among worker nodes, while doing little for deduplication. Instead, we want to merge specifications with many common components. We now introduce a simple metric for similarity between specifications that LANDLORD uses to automatically manage an image store, with a tunable parameter controlling how aggressively to reduce duplication and increase storage utilization.

5.3 Similarity Metric

A key requirement in improving storage behavior for a collection of container images is the ability to quickly identify containers that are "similar" as candidates for optimization. Rather than examining the containers themselves, we will compare the specifications used to generate them. We chose the weighted Jaccard distance under appropriate choice of set elements as it has several desirable properties for grouping sets of packages and is simple to compute. When working with package repositories, each package is usually assigned a name/version string that is defined to be unique within the repo. Public package repositories generally support explicit version constraints, so two specifications may include constraints that cannot be simultaneously satisfied. For LHC applications this is a non-issue, since CVMFS is append-only and all previous versions remain available. For other sources of software (e.g., Python package repositories), we represent conflicting requirements as an infinite distance between specifications. As discussed in Section 5.1, we are not concerned with the particular version strings (as long as they are compatible, i.e. there exists a package version satisfying all constraints). We therefore consider only the set of package names and their storage sizes (weights) when computing distances between compatible specifications.

For sets A and B, the weighted Jaccard distance d_j is:

$$d_j(A;B) = 1 - \frac{\sum_{i \in (A \setminus B)} size_i}{\sum_{i \in (A \cup B)} size_i}$$

When considering container sprawl as a clustering problem as discussed in Section 5.1, the weighted Jaccard distance serves as a metric on the collection of all finite sets of packages. This metric captures several desirable properties when dealing with specifications. First, the weighted Jaccard distance considers specifications with significant storage overlap to be close. This results in similar specifications being grouped together. Second, the inclusion of unrelated components increases the weighted Jaccard distance between two sets. In the case of a full-repo image for example, there would be overlap with any given specification. The large number of other packages included in the full-repo image, however, would cause the weighted Jaccard distance to become large for specifications that require only a few small-sized packages. This naturally penalizes bloated containers which are expensive to create, update, and transfer. In addition, a constant-time approximation of the Jaccard metric (MinHash [29]) is available for making an efficient first pass at selecting similar images when the number of packages or components is large.

The fundamental operation for LANDLORD's storage optimization strategy is merging container specifications that are "close enough". Using the weighted Jaccard distance metric, we can quickly identify cached specifications that are similar to a new request. To decide if two specifications are "close enough" to optimize, we define the parameter

as the maximal weighted Jaccard distance between closely related specifications. Since weighted Jaccard distance is by definition between 0 and 1, must be in the same range. This parameter is something like the "globbiness" of the system. Using the parameter, we can define a simple algorithm for managing and optimizing a central image store.

Choosing near zero requires that specifications are extremely similar before considering them for merging. In the extreme case with = 0, only *identical* images will be considered close, so no images will be merged. This corresponds to a simple cache without LANDLORD's optimization. Choosing to be larger makes it more likely for images to be considered similar and merged. This results in more augmented images that serve multiple tasks. In the extreme case of = 1, *every* pair of images is considered close and merged if possible. This results in large container images that accumulate many specifications. Using the parameter, it is possible to continuously vary between these two extreme behaviors.

It is important to note that while specifications consist of package names and version constraints, generating an image from a specification requires selecting a concrete version for each package requirement. Thus building the same specification at a later time might result in a different selection of concrete package versions. This does not present a problem for LANDLORD, since we retain the original specifications. Thus it is always possible to check whether a given concrete image satisfies the requirements of a specification, and to select concrete packages that satisfy the union of requirements from merged specifications (or find that no such package exists and the specifications are thus incompatible). Our previous research [28] found that version specifications are quite lax in practice, making it easy to find overlapping and compatible selections of package versions.

A potential issue with this automatic merging strategy is "bloated" images that accumulate infrequently used dependencies and increase overhead indefinitely for future tasks. The weighted Jaccard distance gives a natural way to capture and address this effect. As an image becomes bloated due to repeated merges, its distance from any individual request increases. After sufficient growth, the image will become too far from any request to be considered. Without regular use, the bloated image will eventually be evicted from the cache. Choice of therefore places an upper limit on the amount of undesirable bloat in images. Later, we examine the effect of the parameter by simulating image management over a large number of application requests.

Figure 4 shows as an example a single simulation of LANDLORD with = 0:75 and cache size of 1.4 TB processing 250 HEP tasks. First, we note that most of the operations are merges. This is to be expected, due to the high value. The total bytes written also closely tracks merges, indicating that merging is the dominant source of I/O. We still see inserts over the course of the simulation. At more extreme values, we expect to see one of these operations dominate. As the data in cache continues to rise, the set of containers eventually reaches the cache limit, after which the delete count increases. Over the course of the simulation, inserts and deletes are filling and emptying the cache such that it remains close to its storage limit. We also observe the number of cache hits continue to rise despite deletions. As we will see, merging allows for a greater proportion of hits even if the amount of data remains constant, due to deduplication. The cache limit then ensures that infrequently used parts are eventually removed.

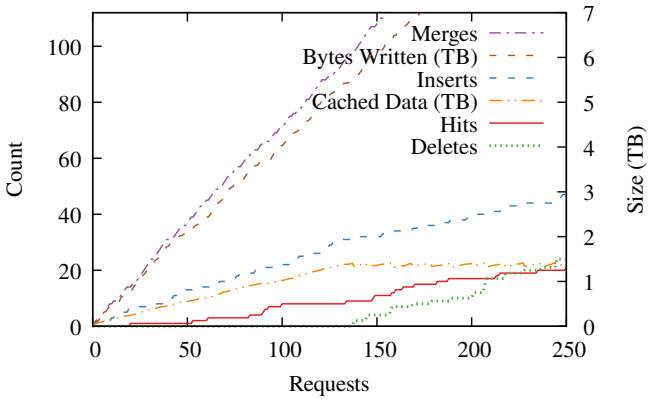


Fig. 4: Behavior of a single simulation.
The x axis shows the number of requests handled so far (the actual time for container creation and application run is not available). Here the cache is filling during the first ∼125 container launches, then containers are deleted to meet the storage limit.

5.4 Deployment of LANDLORD

We designed LANDLORD to allow for flexible deployment, either as an end user or an infrastructure provider. Since individual users may need to run tasks across many sites and will need to work without special privileges, the most straightforward way to employ LANDLORD is as an automated step during task submission. The first step is to prepare a specification for each task. In the simplest case, the user explicitly provides this information by annotating each task or providing a specification file. Alternatively, a workflow system might automatically generate container environments in order to portably execute tasks across remote resources by inspecting the enclosing software environment. Users then set up their particular task submissions to wrap invoked tasks with LANDLORD. On task submission, LANDLORD first scans its configured cache for existing images that are "close" to the task's specification, creates/updates images in the cache as necessary, and finally launches the task inside the prepared container. LANDLORD first observes or infers the package dependencies of submitted applications, then generates the execution environment needed by each application. As required, it creates, merges, or deletes container images in order to balance the total storage consumed by containers against the size of individual containers. LANDLORD allows for a limit on the total storage used, and removes the least recently used images (an LRU eviction policy) to free up space when necessary as a result of an insert or merge. As a future extension, more sophisticated caching policies [30] may be able to achieve better cache performance, but LANDLORD's design does not depend on the particular policy.

While a user-level approach is a good fit for a single unprivileged user, administrators may wish to employ

LANDLORD for site-wide container management such as a Binder-like service or even a batch system. The same core functionality of LANDLORD can easily be adapted into a plugin for a site's batch system, where the system carries out the same per-task steps as above for each task submission. In addition to batch systems, there are other situations where LANDLORD's approach is applicable. With a pilot job system, for example, users are effectively operating a "user-level scheduler". Users could use this same approach to connect LANDLORD to transparently optimize container storage without requiring application changes. The Binder service could also employ LANDLORD as a sub-component, where the main service hands off each notebook launch to LANDLORD to optimize via merging or reuse, if possible, before performing the actual build. Other container-based systems like WholeTale would likewise integrate LANDLORD in a straightforward manner as part of the container build process without requiring architectural changes.

6 CONTAINER-BASED WORKLOADS

To evaluate LANDLORD, we simulated the operation of a container service employing LANDLORD under two different workloads, 1) a large-scale, predominantly Python Binder execution trace; and 2) a synthetic HEP workload using software dependencies from a large mixed repository.

6.1 Binder Workload

Binder [31] is an online service that allows users to launch interactive browser-based notebook applications. Users specify a Git repository, DOI, or other supported format which contains software specifications and/or static data to be included in the notebook environment. On receiving a user request, the public Binder service prepares to launch a container using JupyterHub [32] on one of several cloud compute backends. The repo2docker [33] tool examines the specifications given in the source repository, then carries out any necessary build steps to produce a Docker container for the repository. Each computing backend caches previously built Docker containers for a short period of time, so that if another notebook is requested using the same source repository the cached container can be used. After the build, a container with Jupyter notebook is launched and connected to the user's browser. Individual sites also use the BinderHub [34] software to provide interactive notebooks using local compute resources, like a cluster.

Logs of notebook launches on the public Binder service are periodically published [35], which include the time of each launch and the specific repositories requested. In previous work [28], we downloaded the repositories referenced in the logs to in order to collect the actual software specifications requested for each notebook launch. Using these software specifications, we can replay the sequence of notebook launches to examine usage patterns and caching on a real, large-scale workload with 18 million container launches. A cursory analysis of these Binder containers provides a number of interesting insights. First, software environments specified by users are generally not completely specified. A large proportion of container environments (55%) include one or more packages without any version constraints. The contents of such environments thus depend on when the container is prepared. This does, however, afford some flexibility in selecting package versions.

Second, the usage patterns of containers observed on Binder are far from uniform. The top 10 containers make up 65% of observed launches. Most of these popular containers are demo or example notebooks featured on the websites of projects or software tools, including IPython, JupyterLab, and spaCy. The most popular container, a demo notebook linked on the Jupyter website, accounted for 35% of launches by itself. We also note a very long tail: while some containers were only launched once, many of the containers were launched occasionally over a long period of time, indicating a large working set to be kept in cache.

Finally, we observed a large amount of duplication among the Binder container environments. Popular packages like Numpy or Python itself appeared in the majority of containers, with each container necessarily storing a distinct copy of these packages. Further, some of the popular packages (especially machine learning frameworks like Tensorflow) actually consist of a large number of subcomponents and bring with them a large set of dependencies. We thus observed a high degree of duplication of the data in cache during our simulations.

Our full analysis in [28] includes a more in-depth examination of the Binder workload and the software environments in use. In addition, the dataset we compiled based on this workload is available at [36].

6.2 High Energy Physics Workload

The Worldwide LHC Computing Grid (WLCG) consists of more than 170 computing centres in 42 countries, which provide 1.5 exabytes of storage and around 1.4 million CPU cores. During normal operation the WLCG runs over 2 million tasks per day, with global data transfer rates over 260 GB/s. [37]. The CernVM File System (CVMFS) [38] filesystem is used to publish the software used by all of the major LHC experiments at computing sites around the world. Researchers at CERN use CVMFS as the primary means of distributing the analysis and simulation software they develop to the WLCG. Each experiment maintains a repository of current and previous software versions, allowing stable and uniform access to large software collections that vary over time. For reproducibility and reliability of results, it is important that the *same* applications run at all sites across the globe, and that all previous versions of application code be available and usable when needed.

The upcoming High Luminosity upgrade to the LHC is expected to increase the amount of data generated by a factor of thirty [39], so the WLCG is working to greatly expand its computational capacity through algorithmic improvements, use of accelerators and specialized hardware, and leveraging additional computing resources. HPC resources are an appealing source of computing power to supplement the WLCG, but HPC sites often impose restrictions on network activity and system configuration, preventing WLCG tasks from running directly on HPC resources. Containers offer a potential means for importing software environments to HPC sites without the CVMFS infrastructure available at WLCG sites. CVMFS retains all

	Minimal Image	Full-Repo Image
`alice-gen-sim`	6.0 GB	450 GB
`atlas-gen`	2.7 GB	4.8 TB
`atlas-sim`	7.6 GB	4.8 TB
`cms-digi`	8.4 GB	8.8 TB
`cms-gen-sim`	6.1 GB	8.8 TB
`cms-reco`	7.3 GB	8.8 TB
`lhcb-gen-sim`	3.7 GB	1.0 TB

Fig. 5: Benchmark applications for LHC experiments.

historical versions to ensure reproducibility and backwards compatibility, making simple garbage collection impossible. Since transferring the entire container repository for every task is prohibitively expensive, it is necessary to create tailored images based on a required subset of the full software repository. There are a number of potential approaches to work around the explosion in task-specific images but none are satisfactory.

We consulted with the developers of CVMFS as well as HEP researchers at our university collaborating with CERN to determine how current users interact with CVMFS. We expect significant variability in files accessed and total size among different users and experiments. We nonetheless observed that certain core components are used near-universally. While multiple versions and variations might exist, these components have a very high likelihood of appearing in every container image. These components correspond to the base frameworks, setup scripts, calibration data, etc. needed for most tasks. Based on anecdotal evidence from WLCG researchers, we expect to see these components in a large proportion of tasks across all simulated tasks from different users and experiments. There is also a large set of components that must be available and are used in some applications, but which are very rarely used overall. It is important to make these "long tail" components available to researchers, but it would be wasteful to include them universally when they are rarely used.

Figure 5 shows several container-based LHC benchmark applications [40]. Here `gen` (generation), `sim` (simulation), `digi` (digitization), and `reco` (reconstruction) are phases of the experiment pipelines, with each phase running as a separate workflow. Minimal Image indicates the size of the container image that includes only the subset of the repo needed to run that particular workflow, while Full-Repo Image gives the size of the experiment's entire software repository, which falls in the range of terabytes. Note that while these measurements give a rough idea of task requirements, there is substantial variation in WLCG jobs in production. The CMS experiment for example runs more than 1,200 unique workflows carrying out the general phases above, with many different software builds, versions, and customization [41].

7 Evaluation

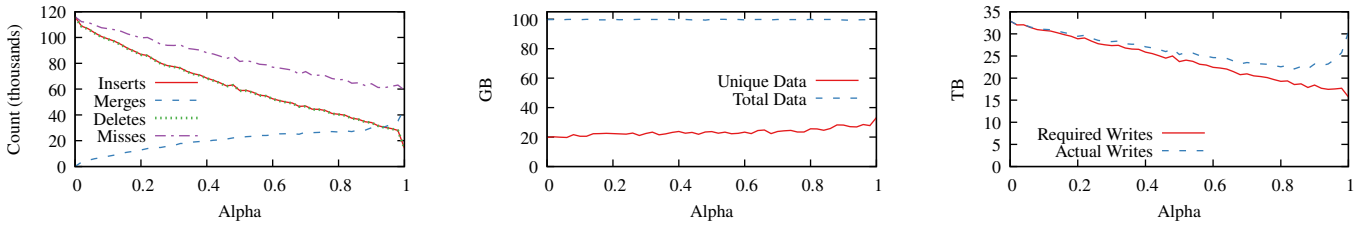
Both workloads take the form of a stream of tasks to be launched, with each task carrying some set of software requirements. For our evaluation, the container service is responsible for preparing a container environment satisfying each task's software requirements. Software packages directly required by tasks can themselves depend on additional packages. Thus to prepare a complete container environment, the container build process must assemble the set of the direct and indirect dependencies. To determine both the dependencies of software packages and metadata such as the package sizes, we collected package metadata for the repositories used in both workloads (PyPI and Conda repositories for the Binder workload, and build metadata from CVMFS packages for the HEP workload). To simulate a workload, we processed each task launch in turn, recursively collecting any software dependencies and passing the complete dependency list to LANDLORD, which built a new container, merged with an existing one, or identified an existing container satisfying the requirements.

7.1 Binder Workload

Sweeping Over . To evaluate the behavior of LANDLORD, we simulated the results of optimizing the container cache for a large, varied workload. Starting from an empty container cache, we replayed the Binder dataset at varying choices of . This dataset gives a sequence of container launches, along with the software environment required for each container. We are therefore able to compare the cache performance and I/O overhead as a result of LANDLORD's merging strategy. Our goal in this evaluation is therefore to choose so as to minimize the storage and compute costs associated with maintaining a collection of images.

Sweeping over the range of values (in steps of 0.05), we can immediately see differences in the frequency of simulated operations. Figure 6a shows the upper range of values where behavior differences appear. From the lower values on the left, the insert and delete counts are the primary (or only) operations, with number of hits relatively constant. This corresponds to a simple LRU-based cache. The insert count is slightly higher due to cache filling, but the two tend to move in lockstep (in the figures the two nearly overlap). As increases, image merges become more frequent. The merge count steadily increases throughout most of the upper range, while inserts and deletes decrease. This suggests that at high values, the cache space would be more efficiently used, with some of the duplication merged out. In the extreme case with `= 1`, every request is merged if possible, hence the reduced number of misses and predominance of merges at the far right of Figure 6a.

Overhead of LANDLORD. Under LANDLORD's approach, we use compute and I/O capacity during task submission in order to improve utilization of storage space. With excessive merging, however, this additional I/O cost can become prohibitively expensive. To quantify this computational and I/O overhead, we used package repository metadata to estimate the cumulative amount of data written over the course of simulated cache operation. We use cumulative write size as a metric for overhead/latency that is independent of specific hardware or disk performance. Figure 6c shows the amount of data written during simulations over a range of values. "Required I/O" is the total amount of data actually requested by each task over the course of the simulations. Note that a cache hit



(a) Total Cache Operations (b) Estimated Duplication in Cache (c) Estimated I/O Overhead

Fig. 6: Binder workload over a range of values

Note that in Figure 6a the number of deletes and inserts is too close to distinguish. Compared to the length of the Binder workload, the time spent filling the container cache is negligible. The delete count therefore very closely tracks the insert count when presenting only the total operations at the end of the workload.

would not require any I/O, so that even when replaying the same workload, differing cache performance changes the required I/O. "Actual I/O" is the total amount of data written to cache over the course of simulations. The actual I/O is greater than what was requested by the user because LANDLORD includes additional packages in containers as part of its merging behavior. This measurement is simply the sum of the data written for each insert and merge. If, for example, an image were evicted and then re-inserted later in a simulation, then the cost of generating and writing the image would be added again.

Without merging (low), the actual I/O in the system closely follows the required I/O. For a simple cache, these two metrics would be identical. As increases, the effects of updating and merging images become apparent. We first note that the required I/O falls as increases. This is because a greater proportion of task requests can be fulfilled directly from the cache. Since more of the requests can be satisfied by previously merged images in the cache the system needs to handle fewer misses, which leads to a corresponding decrease in the required I/O and latency as a result of container builds. From the perspective of the system, this is beneficial as a larger proportion of tasks can be handled with no extra time or compute cost. For latency sensitive applications like interactive notebooks, maximizing the hit rate may be desirable even at the cost of increased compute. Figure 7 highlights this tradeoff: increasing initially leads to decreasing miss rate and I/O overhead.

At higher , however, we see another trend. Each time a merge occurs, the resulting image must be written out *in its entirety*. Thus when merges are frequent at very high , some data will be written and re-written many times to satisfy new task requests. Thus while extremely high makes better use of available storage space, LANDLORD introduces a significant amount of overhead in the form of repeated I/O operations. At the far right of Figure 7, the actual I/O increases well above that of a naïve cache configuration. Despite the decreased number of container builds, the size of each container grows large enough to result in a net increase in I/O. Since a merge entails rebuilding a container image in its entirety, frequently merging large containers becomes prohibitively expensive.

Appropriate choice of thus gives administrators a way to improve utilization of available storage and reduce total I/O using LANDLORD. Figure 7 suggests a wide range of

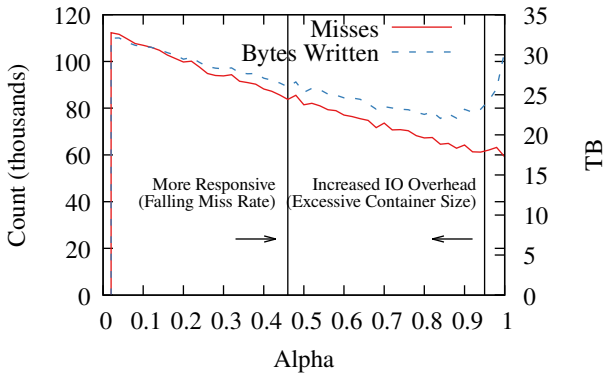


Fig. 7: User responsiveness under LANDLORD

values that result in decreased I/O cost and reduced miss rate over the course of the Binder workload. Even at high where the actual I/O increases, the frequency of cache misses in Figure 6a continues to fall to its minimum at = 1. For situations where minimizing latency is important (e.g., interactive computing), it is reasonable to pay this additional overhead in order to minimize cache misses.

While LANDLORD achieves definite improvements in responsiveness and storage utilization on the Binder dataset, certain properties of that workload are particularly favorable to LANDLORD's design. First, there is a very high degree of reuse of certain individual Binder containers. In that dataset, the median number of times a given container was launched was only 2 times, but the most popular containers were launched hundreds of thousands of times. There is also a high degree of overlap among the the packages that users requested in Binder containers. Common packages like Numpy and Python itself occur in most container specifications, and despite the dataset containing approximately 150,000 unique container specifications, there were only a total of around 10,000 different packages requested (not counting distinct version requirements). LANDLORD therefore had ample opportunities to perform optimizations. Finally, conflicting version constraints in the container specifications meant that it is not possible to merge containers indefinitely. If two container specifications request different versions of the same package, there is no way to satisfy both requirements and therefore the specifications cannot be merged. The user-provided specifications, therefore, naturally limit

(a) Total Cache Operations (b) Duplication of Data in Cache (c) Cumulative I/O Overhead

Fig. 8: HEP workload over a range of values

container bloat and prevent pathological behavior such as repeatedly merging the entire workload into one massive container (with enormous accompanying I/O overhead).

7.2 High Energy Physics Workload

We performed a sensitivity analysis using simulated high energy physics (HEP) tasks to evaluate LANDLORD 's worst-case behavior, and to demonstrate LANDLORD 's application to a non-Python software repository with markedly different organization than the Python repositories used in the Binder workload. This simulated workload differs from the Binder dataset in a number of key ways. First, tasks in the simulated dataset had uniform launch frequency. This forces LANDLORD to handle a very large “working set” of containers. Second, package selections for containers were selected at random from the available CVMFS repositories. This ensures that any overlap between containers is due entirely to common dependencies inherent in the software collection rather than bias in application or workload. Finally, CVMFS software packages are organized such that containers can include an arbitrary selection of packages without con ict. We therefore have the possibility of merging all task requirements into one large container. While real workloads are much closer to the Binder dataset discussed earlier, this analysis allows us to explore extremes in LANDLORD 's behavior and shows that employing LANDLORD does not worsen performance even under extremely unfavorable conditions. Using randomly generated speci cations also allows us to demonstrate that LANDLORD is not suitable for compacting arbitrary collections of data, but is speci cally suited to reducing duplication among software packages that share overlapping dependencies.

Simulating HEP Tasks.

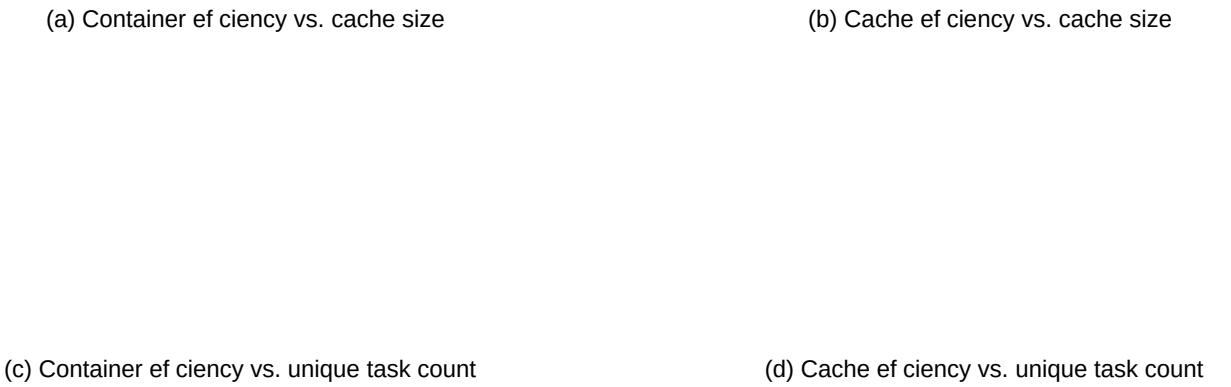
For each simulated request, we chose a random selection of packages and then added any dependencies, repeating until we collected the complete set of packages and dependencies (i.e. the closure of the package dependencies). This image simulation scheme captures the structure inherent in the software collection, in that packages in addition to those requested are automatically included so as to ensure a functional image. The initial selection of packages, however, is simply uniformly random. To evaluate the effects of container contents, we also generated completely randomized images consisting of packages chosen in a uniform random way without regard for dependency relationships. To ensure that total size (or at least total number of packages) is comparable to images generated by the previous method, for this approach we started with an image request generated via the previous scheme (uniform random core selection with dependencies added). We considered only the total number of software packages in the resulting image, and then chose the same number of packages uniformly randomly from the entire repository, ignoring package dependencies. While images generated in this way are very unrealistic, they allow us to isolate the effect of dependency relationships among packages. By comparing results with random images to those with the previous image generation scheme, we can compare the general case of containers as collections of arbitrary data to the speci c focus of this work, i.e. containers with selections of software packages with dependency relationships.

To generate an image for a simulated task request, we randomly made an initial selection of up to 100 packages. We then used one of the two schemes (dependency tree-based or random) to expand the initial selection into a full image. Repeating this procedure, we created streams of container speci cations for simulated tasks.

Since our simulation uses random simulated requests, there is variability between individual simulations. Thus for a given choice of cache size, task count, etc. we repeated the simulation 20 times and reported the median behavior over the runs. The bands in the plots of storage utilization show the standard deviation over the runs. At each choice of (in steps of 0.05) we performed a set of 20 simulated runs, allowing us to plot various measurements of the system versus . Figure 8 shows operational metrics for cache management in the HEP workload.

Metrics for Cache Utilization. When sweeping over the range of values, there are a number of metrics available to summarize each simulation run. Many, however, are highly coupled with the particular workload and system con guration, and dif cult to compare as we vary the parameters of the simulations. Simply comparing cache hit rate with results for the Binder dataset, for example, would not be meaningful due to the stark difference in degree of container contents and reuse. In addition, storage use and I/O overhead for the Binder workload are only estimates based on package metadata for the Python repositories. The repositories used in the HEP workload provide exact storage and I/O information, allowing us to focus more strongly on the cache and container contents. We therefore chose to de ne two metrics, cache ef ciency and container ef ciency, to indicate the effective utilization of the container storage independent of system con guration.

We de ned cache ef ciency as the ratio of unique data to total data in the cache. In our case, this is equivalent to the



(a) Container ef ciency vs. cache size

(b) Cache ef ciency vs. cache size

(c) Container ef ciency vs. unique task count

(d) Cache ef ciency vs. unique task count

Fig. 9: Effects of Simulation Parameters on System Ef ciency

ratio of the size of the unique packages to the total cache size taken from Figure 6b and Figure 8b. If many images contain copies of the same packages, the cache ef ciency decreases. This metric captures duplication within the cache across all images. With no merging there is a high degree of duplication, so the cache ef ciency is low. On the other end of the spectrum, maintaining a single, large image containing all data results in cache ef ciency of 100%, because nothing is duplicated.

We de ned container ef ciency as the ratio of the size of the requested container (a set of requested packages plus all dependencies) to the size of the container the system actually used for the task. In the absence of merging, these two are equal so the container ef ciency is 100%; tasks are run with exactly what was requested. By merging to allow for image reuse, we include additional, unrequested data in container images. The container ef ciency measures this difference between requests and containers. In the extreme case of = 1 with a single large image, for example, the container ef ciency is poor because the entire repository is used for every request, regardless of size. These two extreme cases, no merging among many images and a single merged image, can both be useful in some situations. Rather than de ning where these limits fall, we discuss choosing limits and compare our two application workloads in Section 8.

Sensitivity Analysis. In Figure 9, we plot ef ciency curves for a range of simulation conditions. The left column shows container ef ciency, while the right column shows cache ef ciency. In the rst row, the number of tasks and the amount of repetition are constant while the cache size is varied. In the second row, the number of unique requests is varied with the other parameters constant.

The size of the cache has an inverse relationship with both the container and cache ef ciency. As seen in Figure 9a and Figure 9b, a larger cache can of course hold a larger number of images, but since each image contains signi cant duplicated portions, increasing cache size tends to decrease cache ef ciency. Conversely, small caches more quickly evict images so that ineffective merges tend not to remain in cache too long. A larger cache also allows for more opportunities to merge images, leading to decreased container ef ciency. When deciding how to handle a request, a large cache full of images is much more likely to contain an image suitable for merging. With a small cache, opportunities to merge are much more dependent on the order of requests.

The effect of varying the number of unique tasks is less pronounced than the effect of cache size. As seen in Figure 9c and Figure 9d, streams of 500 and 1000 unique tasks show nearly indistinguishable behavior, indicating that by 500 tasks the system has reached a steady state. Continuing with an arbitrarily long stream should not result in signi - cant performance changes. However, 100 unique tasks were not suf cient to ll the cache and reach a steady state. In this case the container ef ciency is slightly decreased over , suggesting that some ineffective merges had not made their way out of the cache. Cache ef ciency in this case is slightly increased. This would suggest that before reaching a steady state, the cache contents are more assorted and some unnecessary data remains cached.

Effects of Package Dependencies. Figure 10 shows a representative simulation with both dependency-based and random synthetic image types included. In the purely random case, there is no correlation between different images. Thus, it is much more dif cult to nd images similar enough to merge until the value is very lax. This would indicate that our merging strategy is not applicable to arbitrary collections of data. Random images show little to no effect for most values. Our merging strategy, which takes advantage

Fig. 10: Impact of dependencies on duplication.
The Cache and Container ef ciencies of the two image types are plotted together, showing how randomized image contents greatly reduce the ef cacy of LANDLORD .

Fig. 11: LANDLORD vs. Periodic Of ine Clustering
Because of the continuous variation in the workload, the quality of the of ine cluster is initially high but decays quickly. LANDLORD used = 0:7. Every 10% of the dataset the clusters were recomputed based on the most recent 5%.

of duplicated content included as a result of dependencies in software, would be ill advised for situations that are not known to follow similar patterns of duplication. Even with random task requests, the tree structure of package dependencies is produces pronounced duplication in the cache, leaving room for optimization.

7.3 Comparison with Of ine Clustering

It is natural to consider whether better results could be obtained by periodically performing a complete global clustering of requests using an of ine algorithm. As a comparison point, we choose AGNES [42], a classic approach to hierarchical clustering. AGNES rst treats all points as singleton clusters. Then at each subsequent iteration, it determines the centroids of all clusters, computes all pairs of distances between these centroids, and if possible groups two nearest clusters into one. The eventual outcome is a tree structure culminating in one cluster at the top. The algorithm can terminate early if a desired number of clusters or other global constraint is reached.

AGNES is a natural counterpart to LANDLORD , because it makes use of the same fundamental operations on containers, which may be created, merged, or deleted, but not arbitrarily changed. A few adjustments are needed to apply AGNES to the container management problem. First, LANDLORD immediately resolves a request into a concrete container. For example, it may resolve python=* to python=3.6 or python=3.7, depending on when each request arrives. In contrast, AGNES considers many requests at once, clusters them together, and then resolves the cluster into a concrete container. Thus, more information is taken into account for each container. Secondly, a full application of AGNES on n nodes requires considering all $O(n^2)$ distances at each step to nd the closest pairs to merge, which is prohibitively expensive. To reduce this cost we instead sample 10K distance pairs at each step, and select the closest. Third, naive AGNES will result in at least n iterations corresponding to n merges of clusters, as each merge removes two clusters and adds a new one. This combining with the sampled pairs of distances also introduces a high computational cost, thus we cap the maximum number of iterations to 1,000 to avoid this. (A parameter study shows increasing iterations to 2000 only increases the average hit rate slightly from 94.6% to 94.9%) Finally, since the container cache has a physical storage limit, we need a way to detect and pick useful or popular clusters (each of which eventually realizes to a container) once the clustering process nishes. To do this, we de ne the popularity of each cluster to be the sum of the frequencies of the unique requests in that cluster and continually add the containers materialized from those clusters having the highest popularity until the cache is full.

Figure 11 compares the performance of LANDLORD against periodic of ine clustering with AGNES, on the entire Binder dataset. Both algorithms warm their caches up with the rst 20% of the dataset. The LANDLORD algorithm is run as normal, with each request being processed as it arrives and incrementally updating the container cache. AGNES, on the other hand, operates on statically generated clusters and treats any request not satis ed by a cached container as a miss (requiring the requested container to be built and inserted). AGNES periodically clears its cache every 10% of the dataset and reclusters using the most recent 5% (most recent 200K events). The percent hit rate is shown over time. As can be seen, LANDLORD maintains a hit rate in excess of 98%, while periodic of ine clustering with AGNES results in a brie y higher hit rate that falls off quickly as the request mix evolves. The fall in AGNES' container hit rate is signi cantly steeper in the interval (2.5, 3) due to an update to the Jupyter Lab package used in a popular demo repository. This shows that a drastic change in package speci cations of popular requests could degrade the performance of an of ine algorithm, where as LANDLORD is more robust to these frequent changes. From this, we conclude that an of ine clustering algorithm – even if superior to AGNES– would not be suitable for this problem space because of the rapid evolution of the requests and software environment. Furthermore, each epoch of clustering would require a very large expense to reconstruct (and cache) the entire set of containers corresponding to the new clusters discovered.

8 Tuning Alpha

We close by discussing the considerations for a user or infrastructure provider employing Landlord in practice. Constraints at each site such as the amount of scratch storage available for caching container images and upper bounds on the computational cost to prepare each container ultimately dictate the viability of any particular approach. Landlord provides a good deal of exibility to match the properties of a given execution site and workload(s).

Across both the Binder trace and our simulations, we found that the choice of was not particularly important, as long as it falls within a wide "operational zone" (0.65 to 0.95). Despite noticeable variation in ef ciency for Binder due to sampling from only one workload, trends in Landlord's behavior are still visible. Figure 12 shows that choosing extreme values of results in a large number of overlapping container images or excessive overhead creating and updating massive images. These extremes correspond to the naïve approaches discussed previously, i.e. many single-use containers or a single all-purpose container, respectively. Choosing anywhere within the operational zone strikes a reasonable balance between storage utilization and overhead. A new application employing Landlord should choose a moderate (e.g. 0.8) to start, with ner tuning possible to meet speci c application or site requirements. A moderate choice of allows Landlord to avoid extremely poor behavior in either direction, without attempting to attain "optimal" performance. Landlord thus offers a lightweight mechanism to avoid cases of pathologically poor performance.

The compute and transfer cost in the highly merged case and the cache ef ciency in the unmerged case, serve as limits on the viable range of values for a system and its users/applications. Figure 12 highlights the operation of Landlord at varying , serving as a guideline we have found to be applicable across a variety of applications. Moving from the left side of the graphs, the miss rate of the cache begins to fall signi cantly. Choosing too low results in higher job latency than necessary and makes poor use of available storage. On the right, the likelihood of merging increases. As shown in Figures 6c and 8c, the amount of I/O and compute to update images becomes much larger if is set too high. Applications/workloads that prioritize latency would be best served by setting as high as possible, though setting = 1 may result in excessive overhead, especially in the absence of other constraints (e.g. package version con icts) that limit merging. There is no general rule for the placement of these limits, which depends strongly on the performance characteristics of the execution environment, as well as the priorities of the administrators in optimizing the system.

9 Conclusions and Future Work

Large-scale and multi-tenant applications based on container technologies must increasingly treat on-demand generation of containers as a dynamically varying and resource intensive part of application infrastructure, requiring management as a rst-class activity. The mechanisms available to dynamically create and manage containers, however, lead to container sprawl without careful management and monitoring, and alternatives like static clustering of dependencies are not suited to varying application workloads. We analyzed two large-scale container-based application workloads and demonstrated how Landlord, despite being a simple algorithm with a single tunable parameter , can improve container reuse and application latency across a range of conditions and application usage patterns. Our analysis shows that Landlord's behavior is not highly sensitive to choice of , and as future work it may be possible to extend Landlord to automatically respond to poor system utilization by tuning appropriately.

(a) Binder Workload

(b) HEP Workload

Fig. 12: Ef ciency of Landlord. Note that since CVMFS packages for the HEP workload cannot have version con icts, Landlord is able to produce one giant container at very high , hence the sharp jumps in ef ciencies. Version con icts among packages prevent this effect in the Binder workload.

Acknowledgements

This work was supported by NSF grant OAC-1931348 and a DOE Graduate Computer Science Fellowship.

REFERENCES

[1] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[2] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017.

[3] T. Shaffer, N. Hazekamp, J. Blomer, and D. Thain, "Solving the Container Explosion Problem for Distributed High Throughput Computing," in *International Parallel and Distributed Processing Symposium*, 2020, doi: 10.1109/IPDPS47924.2020.00048.

[4] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," *Proceedings of the Cray User Group*, 2015.

[5] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 36:1–36:10. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126925

[6] A. Brinckman, K. Chard, N. Gaffney, M. Hategan, M. B. Jones, K. Kowalik, S. Kulasekaran, B. Ludäscher, B. D. Mecum, J. Nabrzyski, V. Stodden, I. J. Taylor, M. J. Turk, and K. Turner, "Computing environments for reproducibility: Capturing the "Whole Tale"," *Future Generation Computer Systems*, vol. 94, pp. 854–867, 2019.

[7] "Kubernetes: Production-grade container orchestration," https://kubernetes.io/, 2021.

[8] "Helm: The package manager for Kubernetes," https://helm.sh/, 2021.

[9] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive Parallel Programming in Python," in *28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019.

[10] M. R. Crusoe, S. Abeln, A. Iosup, P. Amstutz, J. Chilton, N. Tijanic, H. Ménager, S. Soiland-Reyes, and C. A. Goble, "Methods Included: Standardizing Computational Reuse and Portability with the Common Workflow Language," *CoRR*, vol. abs/2105.07028, 2021.

[11] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "funcX: A federated function serving fabric for science," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, Jun 2020, pp. 65—76.

[12] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala, "Opening black boxes: Using semantic information to combat virtual machine image sprawl," in *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2008, p. 111–120.

[13] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez, "Storage challenges at Los Alamos National Lab," in *28th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, April 2012, pp. 1–5.

[14] 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.

[15] L. Gerhardt, W. Bhimji, S. Canon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter, and V. Tsulaia, "Shifter: Containers for HPC," *J. Phys. Conf. Ser.*, vol. 898, no. 8, p. 082021, 2017.

[16] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-Tree Filesystem," *ACM Trans. Storage*, vol. 9, no. 3, Aug. 2013.

[17] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. K. Paul, K. Chen, and A. R. Butt, "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, 2021.

[18] H. Fan, S. Bian, S. Wu, S. Jiang, S. Ibrahim, and H. Jin, "Gear: Enable efficient container storage and deployment with a new image format," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 115–125.

[19] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proceedings of the Israeli Experimental Systems Conference*, ser. SYSTOR '09, 2009, pp. 7:1–7:12.

[20] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "Snowflock: Rapid virtual machine cloning for cloud computing," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09, 2009, pp. 1–12.

[21] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani, "Demystifying data deduplication," in *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*. ACM, 2008, pp. 12–17.

[22] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey, "Redundancy elimination within large collections of files." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 59–72.

[23] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system." in *Fast*, vol. 8, 2008, pp. 1–14.

[24] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 73–86.

[25] P. Nath, M. A. Kozuch, D. R. O'hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups, "Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines," *management*, vol. 7, no. 5, p. 20, 2006.

[26] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM, 2009, p. 7.

[27] Z. Cao, H. Wen, F. Wu, and D. H. Du, "ALACC: Accelerating restore performance of data deduplication systems using adaptive Look-Ahead window assisted chunk caching," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 309–324. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/cao

[28] T. Shaffer, K. Chard, and D. Thain, "An Empirical Study of Package Dependencies and Lifetimes in Binder Python Containers," in *IEEE International Conference on e-Science*, 2021.

[29] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, June 1997, pp. 21–29.

[30] S. Jin and A. Bestavros, "Popularity-aware greedy dual-size web proxy caching algorithms," in *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, 2000, pp. 254–261.

[31] Project Jupyter, Matthias Bussonnier, Jessica Forde, Jeremy Freeman, Brian Granger, Tim Head, Chris Holdgraf, Kyle Kelley, Gladys Nalvarte, Andrew Osheroff, M. Pacer, Yuvi Panda, Fernando Perez, Benjamin Ragan Kelley, and Carol Willing, "Binder 2.0: Reproducible, interactive, sharable environments for science at scale," in *Proceedings of the 17th Python in Science Conference*, Fatih Akici, David Lippa, Dillon Niederhut, and M. Pacer, Eds., 2018, pp. 113 – 120.

[32] Project Jupyter, "JupyterHub," https://jupyter.org/hub, 2021.

[33] ——, "repo2docker," https://github.com/jupyterhub/repo2docker, 2021.

[34] ——, "BinderHub," https://github.com/jupyterhub/binderhub, 2021.

[35] Mybinder.org events archive. https://archive.analytics.mybinder.org/. Accessed July 2021.

[36] T. Shaffer, K. Chard, and D. Thain, "Binder Software Environments," 2021, doi: 10.5281/zenodo.4891790.

[37] https://home.cern/science/computing/grid.

[38] J. Blomer, P. Buncic, R. Meusel, G. Ganis, I. Sfiligoi, and D. Thain, "The Evolution of Global Scale Filesystems for Scientific Software Distribution," *IEEE/AIP Computing in Science and Engineering*, vol. 17, no. 6, pp. 61–71, 2015, doi: 10.1109/MCSE.2015.111.

[39] The HEP Software Foundation, J. Albrecht, A. A. Alves, G. Amadio, and et al., "A Roadmap for HEP Software and Computing R&D for the 2020s," *Computing and Software for Big Science*, vol. 3, no. 1, p. 7, Mar 2019.

[40] https://gitlab.cern.ch/hep-benchmarks/hep-workloads.

[41] https://indico.cern.ch/event/759388/contributions/3311664/attachments/1814435/2964911/hpc_production.pdf.

[42] Z.-H. Zhou, *Machine Learning*. Springer Singapore, 2021.