An Empirical Study of Package Dependencies and Lifetimes in Binder Python Containers

Tim Shaffer
University of Notre Dame

Kyle Chard
University of Chicago
Argonne National Laboratory

Douglas Thain
University of Notre Dame

***Abstract*—Containers are widely used in scientific applications as they provide greater precision and flexibility in controlling nearly every aspect of the software environment. They can also be easily shared, enabling researchers to run with the same environment on different host systems—an important requirement for scientific reproducibility. In this work, we studied logs of container launches from Binder, a publicly accessible online service for executing Git repositories. Binder dynamically builds and deploys containers following a recipe stored in the repository. These logs capture usage over several years, and include nearly 14 million container launches of around 70,000 unique repositories. To gain more insight about the types of containers and software environments in use for container-based scientific computing services, we downloaded the user-provided recipe repositories referenced in the logs and captured the software specifications and repository metadata. We discovered a number of interesting trends that may be of interest to site administrators provisioning container-based infrastructure for scientific computing in Python. Based on this analysis, we found that automatically generated containers present unique management challenges that are not well handled by straightforward caching. We used historical metadata on package releases from Pip and Conda to quantify difficulties in keeping previously built containers up to date with changing external dependencies. Finally, we proposed several management strategies for reducing infrastructure costs and improving user experience when managing a large container-based service, and back-tested these strategies against Binder launch activity and historical package metadata to demonstrate the value of dependency-oriented container management.**

I. Introduction

The use of containers for scientific applications has increased in recent years, driven by a number of advantages over previous methods of assembling and distributing applications. Rather than employing a single statically compiled executable or a simple script invoking site-provided software tools, modern scientific applications often consist of a wide variety of software packages written in high-level languages such as Python. This affords researchers greater flexibility in choosing the exact algorithms and procedures to use, but means that it is not feasible for administrators to prepare and maintain site-local installations of every software package that may be used. Instead, scientists and research software engineers must prepare full software environments by themselves, for example by compiling and installing packages on a shared filesystem. Containers serve as a more flexible alternative, as they allow researchers to control nearly every aspect of the software environment, from the very low layers (standard libraries, language runtimes, etc.), to precise versions of specialized libraries, to the applications themselves. Containers allow additional data, such as reference or training data, to be included as part of the environment. Containers can also be easily shared enabling researchers to run with the same environment in different host systems—an important requirement for scientific reproducibility.

There are a growing number of services that now make use of containers to provide dynamically deployed and customizable research environments. For example, many academic institutions, research labs, and supercomputer centers provide local installations of the JupyterHub service [1], and services such as Binder [2], Whole Tale [3], and Code Ocean [4], provide on-demand research environments for reproducible research computing. These services rely on clusters or cloud platforms for executing the containers as well as centralized storage for the container images that researchers generate. In order to make containers available via a service or at a computing site, however, administrators must provision storage and infrastructure for user-provided containers and/or disk images. Even minimal container technologies like Singularity [5] that generally operate on simple disk images can carry complicated storage requirements due to caches and layers used during container building.

When providing service to a large number of users at a site (or to the general public), it is important to be able to reason about the infrastructure cost involved. While there have been usage studies published for local [6] and shared file systems at scale [7], information on the usage of container-based applications is harder to come by. In addition, containerized applications are not composed solely of the file and directory data provided by users, but often involve contributions from various package managers and system components, whose contributions to resource utilization are not obvious from, for example, `stat()`ing files on disk and instead require more sophisticated handling of different configuration and setup files to evaluate. Modern container-based applications commonly employ several layers of package management in one container to realize complex software stacks on behalf of the user, for example APT (Advanced Package Tool) provides the base operating system, Conda provides the Python interpreter and a set of libraries, and Pip provides additional Python packages. Given the popularity of Python in scientific computing, it is important to consider these language-specific components when exploring container usage patterns.

In this work, we study logs of container launches from Binder [2], a publicly accessible online service for one-click interactive execution of Git repositories containing user-provided notebooks and applications. Binder builds a container for each repository based on the recipe specified in the repository. These logs capture usage over several years, and include nearly 14 million container launches from around 70,000 unique GitHub repositories. These launch logs, however, only record the sources of container recipes used and do not contain information about the actual containers or the software environments they provide. We therefore downloaded the user-provided container recipes referenced in the logs and measured various properties, including general filesystem properties and use of Python and other libraries provided by package managers. We observed requirements for nearly 8,000 unique Python packages via Pip and Conda, with a strong focus on scientific computing and machine learning applications. We discovered a number of interesting trends in this container usage data that may be of value to site administrators provisioning infrastructure, as well as provide insights into how modern Python applications are organized and distributed. Most Binder repositories referenced in the logs were fairly small (50% were less than 2 MB), with Pip and Conda the most popular package managers (75% of dependency specifications). Looking at the packages used, scientific and machine learning packages like Numpy, Pandas, and Scikit-learn were some of the most popular, but there was a very wide variety of other packages used. We also noted that most repositories (56% for Pip and 57% for Conda) had missing or incomplete specifications of dependencies, with the time a container was built determining what software versions would be included. From historical metadata on package releases for Pip and Conda, we found that these underspecified dependencies became out of date soon after the container was built (50% of these containers had out of date dependencies within 8 days).

Based on this analysis, we found that automatically generated containers present unique management challenges that are not well handled by straightforward caching of data. We used the historical release metadata for Pip and Conda to quantify difficulties in keeping previously built containers in line with changing external dependencies. Finally, we proposed several management strategies for reducing infrastructure costs and improving user experience when managing a large container-based service, and back-tested these strategies against Binder launch activity and historical package metadata to demonstrate the value of dependency-oriented container management.

II. Background

A. Interactive Computing

Modern applications are often built around interactive computing environments, such as Jupyter Notebooks [8] and RStudio [9], which allow users to run computation, store previous results, and visualize outputs via a single interface. This approach allows researchers to explore data, conduct explorative analysis, and to document the procedures used alongside the code and data. The notebook or runtime can also be configured to execute the user's code on a remote compute node, with the local node (e.g. head node or laptop) only displaying results.

B. Scientific Reproducibility

Binder[1] is an online service that allows users to run interactive notebook-based applications [2] in the cloud. Users specify the required environment (usually in the form of a Git repository), which can include datasets, application code, software requirement specifications, documentation, etc. A containerized version of this environment is then built using the repo2docker [10] tool and deployed on one of several public cloud computing providers. The user can then interact with this containerized notebook via a web browser, allowing for simple sharing and collaboration. Binder relies on compute from several public cloud providers on a donation or grant basis. Users may also deploy private JupyterHub or BinderHub instances using their local resources.

Whole Tale [3] is an open platform for conducting reproducible research. The platform allows researchers to discover and ingest data from external sources, conduct analyses in various frontend environments (including Jupyter and RStudio), and export "Tales"—reproducible artifacts that capture the data, analysis, and compute environment. Whole Tale relies on containers to capture these artifacts in the Tale. Users select a base "environment" or define their own recipe specifying various dependencies. Whole Tale uses this information to build a container and deploy it on available HPC resources (located at the Texas Advanced Computing Center).

C. Serverless Computing

Serverless computing refers to a design paradigm in which application designers decompose computational tasks into fine-grained and self-contained components (often expressed as functions in a high-level language such as Python), with the execution provider allocating computational resources on demand. This approach can free users from capacity planning and provisioning decisions; the only information provided by the user is the code to run the task and any necessary input data. Projects such as FuncX [11] are designed for running scientific computing workloads in a serverless fashion. In practice, however, the application tasks are rarely pure functions: even if all input data is specified, any libraries and local data (such as reference or training datasets) needed by the task serve as implicit context that must also be available on the execution node. Containers offer a way for users to prepare a customized environment for each task that the execution provider can deploy automatically. Function-as-a-Service (FaaS) systems typically use containers internally to manage software environments, both among commercial cloud providers (e.g. Firecracker [12] is a lightweight container technology developed for AWS) and in scientific computing (e.g. FuncX). In providing this flexibility for the user, the

[1] https://mybinder.org/

execution provider takes on responsibility for building, storing, and transferring containers as needed. Thus in a similar way to execution sites provisioning resources for local Jupyter installations, administrators managing serverless execution providers must plan and provision resources for the container environments researchers build.

D. Common Issues

While diverse, these examples all use containers in similar ways: as a basis for creating a reusable and portable execution environment. However, all face similar challenges regarding how to efficiently manage a large number of containers for a large number of users both in terms of service levels (e.g., deployment time) and in terms of resource usage (e.g., storage costs). This raises a number of questions: how much storage should be allocated to serve a given number of users? should containers be built centrally (e.g., on the submit node) or on execution nodes? how long should previously built containers be retained? how long should user-provided container specifications be retained? It would be helpful when making these provisioning and policy decisions to have some measurements on containerized environments used in practice for a variety of applications.

III. Binder Container Dataset

The Binder project logs the launch of each interactive container along with some additional metadata such as the backend compute provider used and the source of the repository. These logs are available as a public dataset [13], and include records from 2018 to the present. Each launch record contains the following information:

1) `timestamp`
2) `version`
3) `provider` Repository source, e.g. GitHub or Zenodo.
4) `spec` Git repo and branch/tag/commit to use, e.g. `<github-id>/<repo>/<branch>`. This is provided by the user when launching the container.
5) `ref` Exact commit used at the time of resolving `spec`.
6) `origin` Compute provider the container instance ran on. This is selected automatically and not normally under the user's control.

The identity of the user invoking the container launch is not provided, so any conclusions must be based solely on the repository being launched. Note that the `ref` field is only present on records after June 2020. It is therefore not possible to determine the exact commit tag used for these older records. In our analyses, the general patterns observed in Binder launches did not show sensitivity to time (even considering only the more recent records for which we have exact commit data), so this limitation is not of great concern.

While the Binder launch records provide a detailed view of container activity over time, they do not include sufficient detail to explore what kind of containers were launched or how users designed containers. A more in-depth analysis requires fetching contents from each of the referenced repositories. We chose to limit this effort to repositories delivered via GitHub,

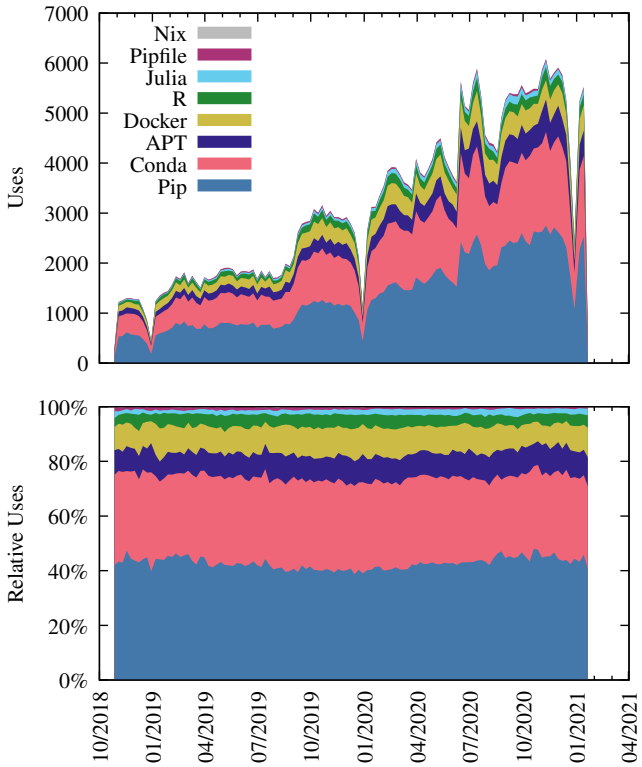

Fig. 1. Specification Types Used in Binder Container Launches

The distribution of dependency repository types used in Binder over time. Top: Absolute number of uses by type. Bottom: Relative proportion of uses for each type.

TABLE I. Specification Types in Binder Repositories

Format	Occurrences	Percentage
Pip	54,177	36.9%
Conda	34,466	23.5%
APT	10,627	7.24%
Docker	10,187	6.94%
R	4,846	3.30%
Julia	2,543	1.73%
Pipfile	913	0.622%
Nix	35	0.0239%

We successfully cloned 146,745 repositories from GitHub. Percentages give the percent of repositories containing each specification type; a single repository can contain multiple specification files. Note that many (44,049 repositories) did not contain any environment specifications at all, and were used solely to bundle files and data directly in the repository.

since they account for the vast majority (97%) of repositories referenced. The remaining ∼3% of launches used GitLab, Gist, Git URLs, Zenodo DOIs, Figshare DOIs, Dataverse DOIs, and Hydroshare resources. Since our goal in this work is to explore packaging and container use patterns, we were interested primarily in the various package specifications used by repo2docker to construct the container environment. Due to variability in how containers are specified in the Binder launch records, it is not possible to simply identify repositories by Git commit. Instead, we considered the exact commit

only if available (the most recent 32% of records, after June 2020), and fell back to the branch/tag specified in the records. This resulted in around 166,000 distinct source repositories referenced in the dataset. We attempted to download each source repository, recording all the the package specification files in use. These were recorded verbatim to allow for further processing. Since we are also interested in general container use patterns, we captured the filesystem metadata for each repository, that is, the layout and sizes of the user-provided files/directories. This information is useful for exploring the transfer/storage cost of working with these containers. Due to the large number of GitHub repositories referenced in the Binder launch records, the process of collecting this packaging information required a significant amount of time. To avoid throttling, we fetched repositories in serial with small delays inserted. Repositories that were since deleted or made private after being launched via Binder could not be cloned, so these repositories were noted but skipped (12% of GitHub repositories, around 20,000). The repositories were built for a variety of data encodings and filesystems, so additional care was necessary to work around unexpected data formats and special/reserved files. This procedure took roughly two weeks to process the full set of GitHub repositories used in the Binder launch records.

The resulting dataset provides a view into the packaging and container design practices in use over time for widely varying users and projects. Figure 1 shows the number of Binder launches over time, broken down based on the types of package specification formats in use. In order to examine the contents of these specification files, some additional processing was required. We chose to focus our analysis on the most commonly used specification types, Pip and Conda. As shown in Figure 1, this selection covers the majority of repositories. The later analyses involving the packages in repositories looked only at Pip and Conda specification files. We validated these specification files, excluding any repositories where we could not successfully parse and operate on the specifications. Since some of our analyses in Section IV take into account the set of transitive dependencies or resulting dependency sizes, it was also necessary to limit the package sources under consideration. In situations where users fetched packages from arbitrary URLs or used custom build scripts (such as Dockerfiles, which are free to make arbitrary changes to the container), it is not possible to determine the actual dependencies used in the container without building it. We therefore limited our analysis of Python dependencies to repositories that used only static and declarative dependencies available in public repositories, such as the Python Package Index (PyPI) for Pip and publicly accessible Conda channels. Even with these restrictions on "pure" software environments, there was still a fairly large sample (∼85,000 repositories) under consideration.

As Table I shows, Pip and Conda together were used in nearly 75% of specifications in the repositories. APT also makes up a significant portion of the specifications used, but provides system packages largely orthogonal to the Python dependencies provided via Pip and Conda. Dockerfiles are also commonly used, but present additional difficulties for analysis. An interesting point to note is that a large number of repositories did not use *any* specifications at all. These repositories simply used the default environment provided by repo2docker, which includes a recent Python version and Jupyter notebook. These repositories thus used Binder solely as a way to package notebooks, code, and data into a more convenient container form, giving little consideration the the software environment in use.

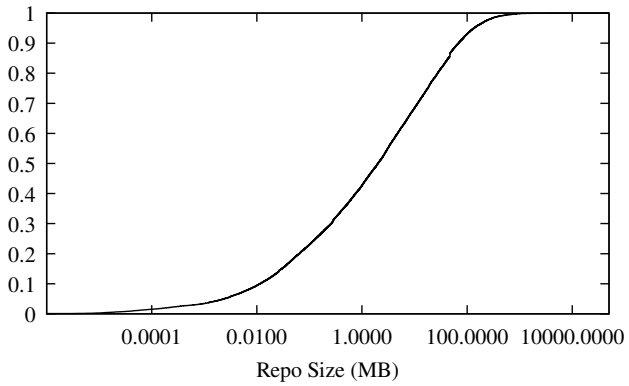

Fig. 2. Total sizes of Binder repositories.

IV. Observations

We explored a wide variety of properties of the Binder repositories, with some of the interesting results presented here in cumulative distribution function (CDF) form to highlight the distribution of data. We focused on three primary areas: general properties of the repositories (e.g. size and filesystem-related properties), Python usage, and package manager usage. Since the focus of this work is Python applications, we limited our package manager analysis to Pip and Conda, which account for most of the Python packages used (see Figure 1). Due to the wide variation in properties across repositories, most of the results here are presented as CDFs with a logarithmic scale on the x axis.

A. General Repository Properties

The original Binder launch dataset is stored in chronological order, so we initially attempted to plot the properties explored here in the same way. Figure 1 shows the types of specification files in use over time in this way. An increasing trend in volume of container launches as well as seasonal variation are clearly visible. When we present the same data as proportions of the total launches, however, this time variance disappears.

Figure 2 shows the distribution of total sizes of the Binder repositories. This was computed by summing the sizes of all the files present in a Binder repository. We do not consider the size of packages to be downloaded as specified in the Pip, Conda, or APT files. We note that most of the repositories are small (around half take up 1 MB or less), but it is quite common for repositories to contain significantly more data. Some of the largest repositories took up tens of gigabytes of

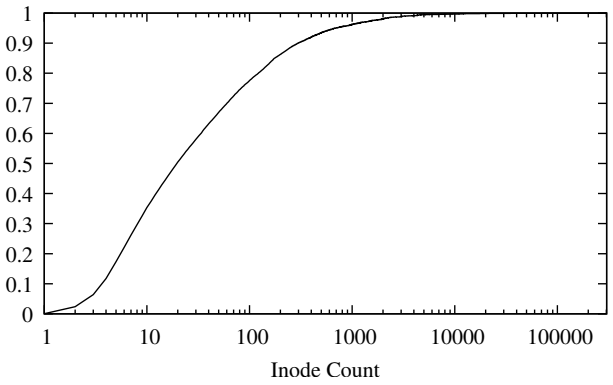

Fig. 3. Number of Inodes in each Binder Repository

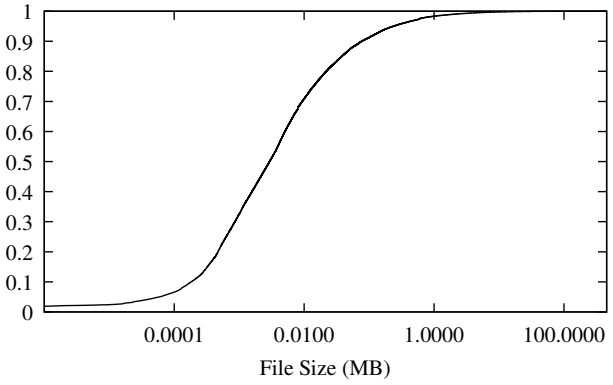

Fig. 4. Sizes of Files Across All Binder Repositories

space. On examination, the largest repositories were bundling reference or training data used for machine learning applications. This was a recurring trend when examining outliers and heavy users: containers for machine learning were generally responsible for the most extreme resource usage among the Binder repositories we observed. Figure 3 shows the number of inodes included in each Binder repository. Again the majority of repositories were small, but some included a very large number of filesystem entries. Table II gives the frequencies of file extensions across Binder repositories. Figure 4 shows the distribution of file sizes across all Binder repositories. This result is similar to previous studies of shared filesystem usage patterns [6], with repositories generally consisting of a large number of small files. Approximately 2% of the files were empty. As with the other measurements, some outliers included much larger files (up to approximately 450 MB). The largest files observed were reference or training data used for machine learning.

These general measurements of the Binder repositories may be useful for administrators provisioning storage or defining resource limits on containers. Assuming a container system that allows for sharing of base operating system layers, fairly modest limits would be sufficient for most users (e.g. 100 MB would be sufficient for over 90% of repositories). The distributions of file size and number of inodes are important considerations for a storage backend. Depending on the filesystem in use, the number of inodes available may become a significant limitation. On shared filesystems in particular, these can often become a precious resource that require per-user quotas. Many shared filesystems are optimized for bulk access to large files, which is the opposite of the usage patterns we observe here. It would thus be better to treat container storage in the same way as user home directories than as bulk scratch storage. For a container service that operates on disk images, however, it is not necessary to store the unpacked container contents, instead storing each image in a single large, self-contained file. Even simpler storage without full POSIX filesystem semantics such as an object store would be well suited for hosting disk images. The primary concern in this situation is the total storage space available. These considerations may affect the choice of container system, filesystem, and storage/inode limits used for serving containers to users at large sites.

TABLE II. Frequency of File Extensions

File Extension	Occurrences	Percentage
.py	3,881,323	12.6%
.png	3,157,166	10.2%
.txt	2,897,633	9.44%
.jpg	2,128,205	6.94%
.js	1,605,104	5.23%
.ipynb	1,511,060	4.92%
.md	1,194,985	3.89%
.json	840,116	2.74%
.csv	833,132	2.72%
.html	771,808	2.52%
Other	11,862,608	38.7%

B. Python Usage

We next consider several container properties particular to Python. In addition to libraries, users have the option of requiring a particular version of Python itself. Pip does not support this functionality, but Conda can distribute the full Python environment, along with its required base libraries. repo2docker also supports another configuration file to set the Python version used, but internally this configuration is simply translated to a Conda requirement. Table III shows the most commonly used Python versions across the Binder repositories. Note that a significant number of repositories did not require a specific version, which will result in Conda providing the most recent version at the time the container is created. Section V discusses this issue further.

C. Library Usage

Finally, we examined the usage patterns of Python libraries among the Binder repositories. As mentioned in Section III, we only considered repositories with static dependencies for publicly accessible Conda and/or Pip packages. Since Pip and Conda are by far the most commonly used specification formats, this still left a large sample of Binder repositories to examine.

Table IV lists the most commonly used Pip and Conda packages across the Binder repositories. As expected, widely-

TABLE III. Python Versions Requested

Version	Total Uses	Percentage	Version	Total Uses	Percentage
3.0	2	0.0202%	2.7	722	7.28%
3.4	13	0.131%	2.*	1	0.0101%
3.5	135	1.36%			
3.6	2,393	24.1%			
3.7	4,214	42.5%			
3.8	1,986	20.0%			
3.9	42	0.424%			
3.*	406	4.10%			

If no version of Python was requested, repo2docker includes a default version (3.7 at the time of writing).

TABLE IV. Most Frequently Used Packages.

Pip Package	Total Uses	Conda Package	Total Uses
matplotlib	20,517	numpy	11,006
numpy	20,361	matplotlib	10,068
pandas	16,269	python	9,691
scipy	9,574	pandas	8,719
ipywidgets	8,299	scipy	5,369
seaborn	5,669	pip	4,945
voila	4,456	seaborn	3,303
pillow	4,293	ipywidgets	3,261
scikit-learn	3,818	scikit-learn	3,175
ipython	3,505	jupyter	2,194
Other	205,749	Other	166,433

used scientific and machine learning packages were the most commonly used. The other packages, however, make up a substantial portion of those seen. This "long tail" of dependencies makes it difficult to prepare standard/multipurpose containers to support many users; there is a common core of regularly used packages, but there is no straightforward way to choose a selection of other packages that maximizes future utility of the container. This in part explains the necessity of using container-based solutions when dealing with ever growing numbers of computational researchers: the cost of storing a large number of specialized containers is less than the operational and administrative cost of coordinating the precise and ever-changing requirements of a large number of users into site-provided environments.

We also examined the sizes of the software environments requested for the containers, both in terms of number of packages and size on disk. This is complicated by the fact that the set of packages explicitly requested in the container specification is generally not the same as those that will actually be installed during container build. Package managers recursively include necessary dependencies of the packages, significantly increasing the size of software environments. We therefore considered both the direct dependencies (those listed explicitly by the user) and the total dependencies (the closure of all indirect and transitive dependencies collected by the package manager). Figure 5 gives the distribution of the number of packages for Pip and Conda, and Figure 6 shows the same plot for on-disk storage.

In terms of both number of packages and storage, we observe that set of user-specified packages is much smaller than the total set required for most containers. This would indicate that comparatively few users include a complete

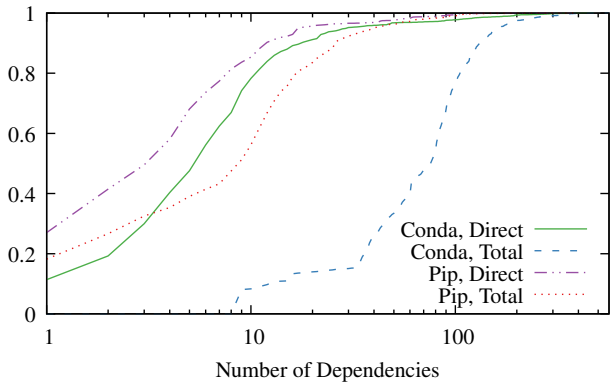

Fig. 5. Number of Python Dependencies per Binder Repository
Direct indicates dependencies directly listed in the user-provided specification file; Total indicates the dependencies after recursively collecting all indirect dependencies in addition to the direct ones.

environment specification for their containers. This can also be a source of surprises when estimating storage costs, as the bulk of the packages for a container are not explicitly listed in the recipes and are only realized when the container is actually built. We note here that the storage required for the Binder repository itself (Figure 2) and the storage for the software environments assembled when building containers (Figure 6) show markedly different distributions. Especially for Conda installations, the storage required for the realized container normally far outweighs the storage used by the Binder repository used to create it, even considering the arbitrary user-provided data that can be included in Binder repositories. Standard techniques to reduce container storage like Docker's reuse of layers offer little benefit here, since only *identical* layers can be reused; only Binder repositories with specification files that are bit-for-bit identical can share layers containing the Conda and Pip environments. This effectively limits any layer-based deduplication to the base operating system image; each realized container will have to include a complete copy of all software dependencies as well as the contents of the Binder repository. Site administrators may benefit from treating these two types of storage separately, e.g. using durable storage with tighter limits for Binder repositories while keeping realized containers on bulk scratch storage. Realized containers can be rebuilt as needed, allowing for substantial space savings.

The dependency specifications themselves also give some important insights. Both Pip and Conda allow for flexible dependency specifications, given by a package name and optional version constraints. Multiple version constraints can be given for a single dependency, and the constraints can include complex logical predicates for precise control over dependency versions. For the greatest reproducibility and consistency over time, all dependencies for a repository should specify an exact version, ensuring that software environment can be rebuilt. Looking at the Binder repositories in Table V, however, we observe that most specification files included only partial or

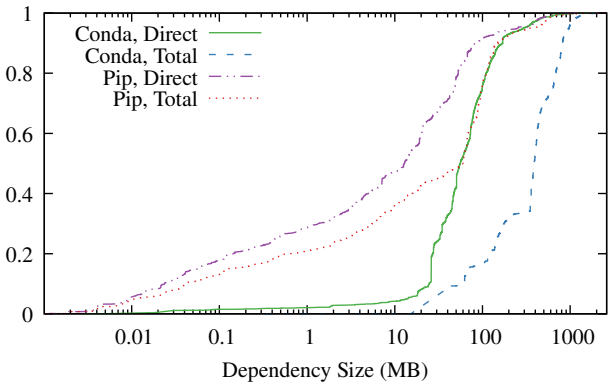

Fig. 6. Size of dependencies per Binder repository

TABLE V. Completeness of Specifications in Repositories

Specification Completeness	Pip	Conda	Total	Percentage
Complete Versions	15,305	7,681	22,986	44.1%
Partial Versions	6,952	6,653	13,605	26.1%
No Versions	12,163	3,354	15,517	29.8%

Repositories that included a version constrain on every requirement are considered to have Complete Version information. Likewise repositories that included no versions constraints at all (i.e. listed only package names) are considered to have No Version information. Partial Versions indicates that some of the dependencies in a specification file did not include version information.

no version information. These underspecified requirements are satisfied by the most recent package versions at container build time. Thus the software environments in a large proportion of containers based on underspecified repositories are sensitive to the time at which they were built.

V. Invalidation of Containers Over Time

Based on these observations, an important distinction arises between generated container environments and the recipe repositories from which they are derived: since in practice users very often leave the version(s) of dependencies unspecified (see Table V), the contents of the container are sensitive to the time when the repository is built. Even while the contents of the base repository remain the same (no changes to the dependency specifications or additional data provided by the user), the available versions of these dependencies change over time as new versions are released. Thus we cannot treat generated container environments as immutable data for the purposes of comparison and caching. Short time discrepancies of this kind are often harmless (e.g. a dependency releases a new minor version that provides essentially the same functionality), but the greater the time difference between container build and use, the more likely that at least one of the dependent packages undergoes a significant change that makes the previously built container functionally different. We assume here that the correct behavior for a container management service is to match the result of a user building a container manually (i.e. downloading the source repository and running repo2docker themselves).

As a case in point, one of the most commonly launched containers on Binder is a demo[2] used in the documentation for Jupyter Notebooks to introduce basic functionality and provide an interesting environment for users to explore (i.e. it includes several scientific and machine learning packages such as Numpy, NetworkX, Pandas, and Scikit-learn). This repository was last modified approximately two years ago at the time of writing, and has been regularly used on Binder since then. Also note that this repository does not specify versions for many of its dependencies. Treating the generated container as plain data without respect to the details of the dependency specifications, it would be perfectly reasonable to keep using a cached build until the source repository is updated. Even with a strategy like least recently used (LRU) in place to limit container storage, the popularity of this repository would ensure that the previously built container always stays in cache. A user looking at the repository and launching it through the Binder service might be surprised, however, to find that the machine learning packages actually present in the container may be up to two years old, and that building the same repository locally gives the current versions of dependencies. This demo repository provides a prominent illustration, but is unlikely to cause real harm due to version mismatch. When generated containers are used behind the scenes for scientific reproducibility or serverless computing, however, we have a source of confusion and bugs. If a user registers a software environment without fully specifying the software versions needed (i.e. only includes dependencies on packages without specifying version constraints, as was very commonly seen among Binder repositories), it would be incorrect to cache this environment indefinitely. If the user wants a feature added to one of the dependencies in a subsequent release, the container system would need to rebuild the environment. If the user-provided specification has not changed, however, the container system would not have any way to detect that the previously generated container has become outdated.

The ideal solution here is that all users fully specify versions of all dependencies. As we have seen, however, this does not reflect actual use. When designing future container-based scientific platforms, or when training scientists and research software engineers, it might be prudent to encourage/require all environment specifications to be complete. Wherever incomplete specifications are allowed, it is important to consider the time factor in built containers diverging from the source repositories/specifications in addition to any storage constraints. To measure this effect, we collected all of the Python repositories in the Binder records which included one or more dependencies without version information (see Table V). For each of these underspecified repositories, we first determined the current dependency versions at the time the repository was launched using historical metadata for Pip and Conda releases. We then looked forward in the historical release data to determine the amount of time until the dependency versions no longer matched those at launch

[2]https://github.com/ipython/ipython-in-depth/tree/master

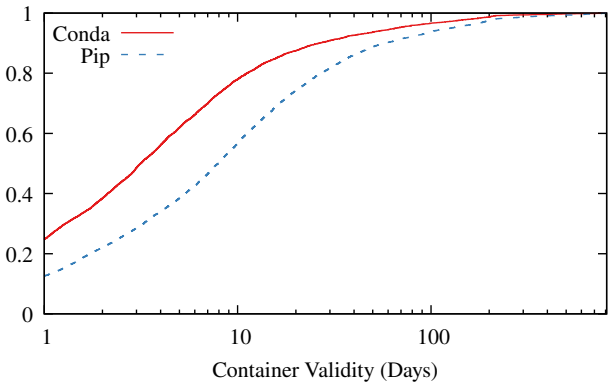

Fig. 7. Time Underspecified Containers Remain Valid

This figure shows the fraction of under-specified containers that remain up-to-date after a given amount of time. For example, approximately 20% of containers using pip dependencies were out of date after two days, because a dependent package had released a new version since the container was first launched.

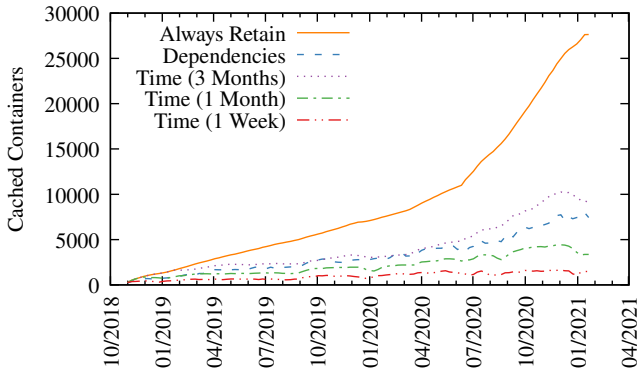

Fig. 8. Number of cached containers.

time for the underspecified dependencies. Figure 7 shows this distribution of the validity periods for built containers. We observe that most containers became out of date with respect to the packages available at the time within 10 days. Of course, many of these updates were likely to be minor and non-breaking changes; however, since it is not possible to automatically determine which updates are functionally significant (packages use a variety of versioning schemes with varying stability guarantees), the most conservative approach would be to rebuild containers on every update. This raises the question of how to balance container storage, compute time spent building and rebuilding containers, and mismatch between source repositories and cached containers.

VI. Reducing Outdated Container Use

To evaluate the impacts of different container management strategies, we simulated the effects of replaying the Binder repository launches using historical Python package metadata to determine the past versions of dependencies that would have been included in containers, as well as the degree to which build containers became out of date. Note that since our evaluation depends on historical Conda and Pip package metadata, we only considered those repositories which contained static Conda and/or Pip dependencies. When determining if a previously built container was out of date, we considered only the underspecified dependencies, since these are subject to change based on conditions external to the repository.

We chose several management strategies to evaluate:

1) **Always Retain:** containers are cached indefinitely
2) **Dependencies:** containers are deleted as soon as one of their underspecified dependencies received an update
3) **Time:** containers are deleted after a fixed period of time (1 week, 1 month, and 3 months after their build).

For these simulations, we consider the general case where there is no hard limit on storage for containers. In practice there would be some sort of site-local storage limit or budget for commercial cloud services. These constraints can be applied alongside the strategies presented here.

For administrators of a local notebook provider, container management service, or serverless computing platform, there are several key operational characteristics. First, it is important to cache and reuse previously built containers. Since the time to build a container is typically on the order of tens of minutes depending on the particular contents, it is highly desirable to reuse containers when re-launching the same repository to reduce the time users spend waiting. For serverless computing, where the execution time of a task is often measured in seconds, the overhead of rebuilding containers for every task is unacceptable. Administrators must also consider the storage space used to store containers. The particular container technology available at a site significantly impacts storage. Docker's layer model can store common layers (e.g. base operating system components) only once and share them between multiple containers. Container runtimes that operate on disk images (e.g. Singularity) require a complete copy of all container contents in each image, significantly increasing the storage cost. This does come with advantages, such as simpler management and the ability to store images using any available shared filesystem, object store, etc. We also measured the extent to which repository launches used outdated containers under each of the strategies by counting the total number of outdated launches, as well as computing the median amount of time by which each container was out of date (the difference in time between the container launch and the dependency update that rendered the container outdated).

Figure 8 shows the number of containers stored under each strategy over the course of the Binder dataset. As expected, the number of containers monotonically increases under the Always Retain strategy. The time-based strategies significantly reduce the rate of increase of cached containers, with the more aggressive options holding the total number of containers mostly constant. The Dependencies strategy shows the exact number of valid containers at each point, so the difference between Dependencies and Always Retain gives the number of containers that have become outdated due to dependency up-

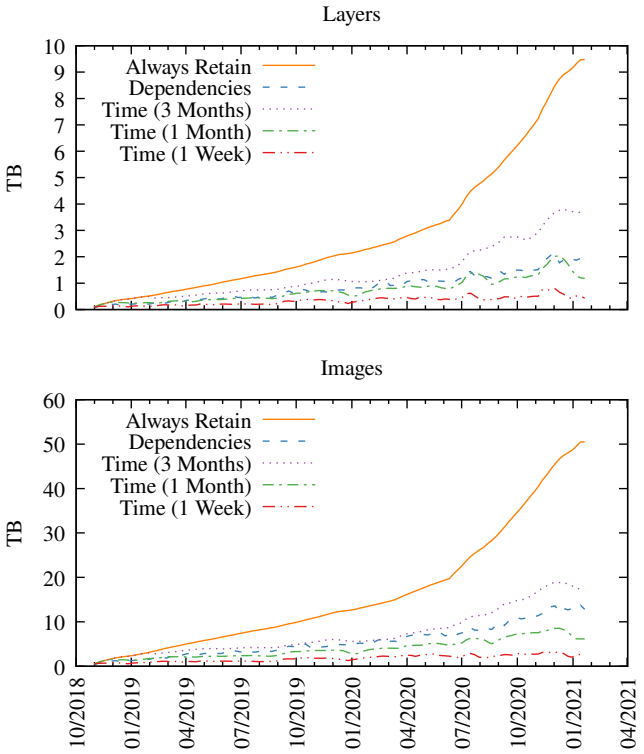

Fig. 9. Storage Used By Container Cache

Top: storage only for the repository-specific layers.
Bottom: total cost of repository-specific and base components as for disk images.

dates. This strategy only removed containers for dependency-related reasons, so like Always Retain it does not reflect the working set of repositories in use at a given time. Figure 9 shows the estimated storage required for the container cache under each strategy. We compute the storage measurements both for repository-specific layers (software dependencies plus user-provided data from the repository) as in the Docker model, as well as the size of complete disk images as used with other container systems. These measurements follow the same trends as Figure 8. They also give a sense of the scale of resources involved when operating a large public notebook service. While storing these container caches using commercial cloud providers would incur significant costs (at the time of writing, storing the full set of containers under the Always Retain strategy using AWS ECR for layered images and AWS S3 for disk images would cost approximately $950 and $1150, respectively, per month), provisioning enough local storage to hold all the containers built over these simulations would be feasible for individual sites.

In addition to the storage required, we also examine the number of container builds required under each strategy. Since we are not actually carrying out the builds, and since the Binder launch records do not include details on the time spent building containers, we cannot determine the amount of time or CPU-hours used for building. Table VI gives the number of

TABLE VI. Summary of Container Cache Simulations

Strategy	Builds	Evictions	Outdated Launches	Median Age
Always Retain	27,641	0	5,913,039	64 days
Dependencies	92,831	85,400	0	0 days
Time (3 Months)	40,227	31,117	5,675,235	21 days
Time (1 Month)	58,052	54,691	5,055,746	8 days
Time (1 Week)	106,794	105,319	3,567,261	2 days

All numbers are totals over the entire simulation using the Binder launch records. The Median Age refers to the amount of median time between a launch and the container becoming out of date for each repository with outdated container launches.

builds over the course of each simulation. The Always Retain strategy gives the minimum number of builds required to carry out the repository launches from the Binder records. Under the Dependency strategy, containers are removed exactly when their underspecified dependencies become out of date. This conservative approach requires a large increase in the number of builds. As a practical matter, it would be difficult to track this for a large collection of container images without the benefit of historical data. The Time-based strategies also increase the number of builds compared to Always Retain, since some actively-used containers are discarded and rebuilt.

We finally consider the use of outdated containers under the various strategies. We are interested both in the frequency of launch for these containers, as well as the degree to which their dependencies are out of date. Looking at Table VI, all of the strategies (except Dependencies of course) launch a comparable number of outdated containers. Aggressive Time-based removal (1 week) reduces this somewhat. The median age of outdated containers, however, shows greater variation. As before, the Dependencies strategy never launches outdated containers so its median age is zero. The Time-based strategies impose an upper limit on how out of date a container may be, equal to their time limit. In the simulation, however, all three time-based strategies kept the median age much lower than their respective limits. Thus in practice a generous limit on the worst-case behavior is still helpful for improving the average case. As expected, under Always Retain the containers tended to be most out of date. Comparing the storage costs, number of builds required, and age of cached containers, setting a Time limit of 1 month on retaining built containers offers a good balance. Despite not being optimal along any axis, this policy is simple and avoids extremes of storage and compute usage while ensuring reasonably up-to-date software in containers. Sites with specific requirements (e.g. absolutely minimize storage) may benefit from the other strategies, but removing previously built containers after 1 month would serve as a good general recommendation based on the activity on Binder and Python package activity over several years.

VII. Related Work

To aid in managing software environments in containers, Kubernetes package managers such as Helm [14] can instantiate specific versions of each software component and clean up outdated containers. PyWren [15], a Python library for running

Python functions on Amazon Lambda, supports determining dependencies semi-automatically, though the user may need to employ Conda themselves to package more complicated environments. Landlord [16] manages container caches for distributed computing by merging container specifications, but is focused on reducing storage and does not consider out of date software. There has been extensive research on deduplication of filesystem data [17] and disk blocks [18], as well as container caching [19] to improve storage use and latency. Filesystem usage studies such as [6] provide insights into the usage patterns and properties of files/directories across local machines. Large-scale studies at supercomputing sites [7] provide characterizations of data and IO behavior for high-performance applications.

VIII. Summary

Myriad systems—from campus deployments of JupyterHub through to online scientific reproducibility services—make use of containers as a way of encapsulating complex computing environments and abstracting underlying system heterogeneity. In this paper we analyzed a large dataset of 14 million container launches spanning over 166,000 GitHub repositories used by the Binder service. After downloading these repositories we found that 75% used either Pip or Conda package managers and that most repositories are small (around 50% are less than 2 MB). We further explored Python usage in these repositories, identifying diverse use of Python packages and quantifying both the number and size of indirect and direct packages imported. Finally, we investigated the time that cached containers may remain valid by looking at the rate at which underspecified dependencies were updated. Our results showed that approximately 20% of repositories would be out of date after only two days. This sensitivity to time means that straightforward caching techniques are not sufficient for container-based applications. Our analysis using Binder logs and historical package release metadata shows that dependency-aware management can effectively mitigate usability and reproducibility issues where dependency version information is incomplete. We also give a simpler time-based strategy that container systems should use to manage cached containers.

Acknowledgments

This work was supported in part by NSF grants OAC-1931348, OAC-1550588, and OAC-2004894, and by the U.S. Department of Energy under Contract DE-AC02-06CH11357.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists, Office of Science Graduate Student Research (SCGSR) program. The SCGSR program is administered by the Oak Ridge Institute for Science and Education (ORISE) for the DOE. ORISE is managed by ORAU under contract number DE-SC0014664. All opinions expressed in this paper are the author's and do not necessarily reflect the policies and views of DOE, ORAU, or ORISE.

References

[1] Project Jupyter, "A gallery of JupyterHub deployments," https://jupyterhub.readthedocs.io/en/stable/gallery-jhub-deployments.html, 2021.

[2] 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.

[3] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17310695

[4] "Code Ocean," accessed April 16, 2021. [Online]. Available: https://codeocean.com/

[5] 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. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[6] J. R. Douceur and W. J. Bolosky, "A large-scale study of file-system contents," *SIGMETRICS Perform. Eval. Rev.*, vol. 27, no. 1, p. 59–70, May 1999. [Online]. Available: https://doi.org/10.1145/301464.301480

[7] B. Xie, S. Oral, C. Zimmer, J. Y. Choi, D. Dillow, S. Klasky, J. Lofstead, N. Podhorszki, and J. S. Chase, "Characterizing output bottlenecks of a production supercomputer: Analysis and implications," *ACM Trans. Storage*, vol. 15, no. 4, Jan. 2020. [Online]. Available: https://doi.org/10.1145/3335205

[8] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. development team, "Jupyter Notebooks: a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Scmidt, Eds. IOS Press, 2016, pp. 87–90. [Online]. Available: https://eprints.soton.ac.uk/403913/

[9] RStudio Team, *RStudio: Integrated Development Environment for R*, RStudio, PBC., Boston, MA, 2020. [Online]. Available: http://www.rstudio.com/

[10] Project Jupyter, "repo2docker," https://github.com/jupyterhub/repo2docker, 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." ACM, Jun 2020. [Online]. Available: http://dx.doi.org/10.1145/3369583.3392683

[12] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434.

[13] The BinderHub Federation, "binder-launches," https://binderlytics.herokuapp.com/binder-launches, 2021.

[14] https://helm.sh/.

[15] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *ACM Symposium on Cloud Computing*, 2017, p. 445–451.

[16] 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.

[17] 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.

[18] 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.

[19] 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