Preserving Containers

Klaus Rechert[1], Thomas Liebetraut[2], Stefan Kombrink[3], Dennis Wehrle[4], Susanne Mocken[5], Maximilian Rohland[6]

1,2,4,5,6 University of Freiburg
3 Ulm University

Abstract. Container technology has been quickly adopted as a tool to encapsulate and share complex software setups, e.g. in the domain of computational science. With growing significance of this class of complex digital objects their longevity is also of growing importance. This paper provides a detailed analysis of a container's long-term preservation risks. Based on this analysis, we propose an emulation-based preservation strategy to maintain access to software-based research methods by converting them into a generic archival representation for containers and providing a generic runtime environment.

Keywords. containers, long-term preservation, emulation

Introduction

Modern science is almost inconceivable without computer technology and sophisticated software, which is leading to novel research methodology. In particular, software is able to encapsulate a significant part of research, and thus, can be a crucial element of a research project's outcome by accompanying published articles and published data sets.

Computational science communities have already recognized the need for reproducible compute-based research results to improve scientific practice and to create sustainable results.[1] While publication and citation of research data has made progress recently (Callaghan 2014, Altman et al. 2015), management of software-based research methods remains an open challenge. "Software is a critical part of modern research and yet there is little support across the scholarly ecosystem for its acknowledgement and citation." (Katz, Niemeyer and Smith 2016). Concepts and practice of software citation are currently discussed [2] [3], but software also needs to be available, i.e. retrievable from a dedicated software-archive. [4] [5] Currently, several projects are working on software preservation concepts and services.

However, if software reproducibility is defined as "the ability for someone to replicate a computational experiment that was done by someone else, using the same software and data, and

1 Supercomputing 2017 Reproducibility Initiative, http://sc17.supercomputing.org/2017/02/07/submitting-a-technical-paper-to-sc17-participate-in-the-sc17-reproducibility-initiative/(online, version of Feb 22, 2017)
2 National Institutes of Health (NIH), https://grants.nih.gov/grants/guide/notice-files/NOT-OD-17-015.html
3 https://danielskatzblog.wordpress.com/2017/02/22/creation-publication-and-citation-issues-in-software-citation-versus-paper-and-data-citation/
4 Software Preservation Network http://www.softwarepreservationnetwork.org
5 UNESCO PERSIST Initiative, https://www.unesco.nl/digital-sustainability

then to be able to change part of it (the software and/or the data) to better understand the experiment and its bounds[.]"[6] then just availability of software is usually not sufficient. A software-based research process may contain multiple individual software components. And even with detailed documentation - if available at all - manually rebuilding a complex software setup is a laborsome and error-prone process, in particular because (implicit) operational knowledge is lost over time. Additionally, all of the preserved software's dependencies also need to remain available, e.g. operating system, libraries, build environment and a suitable hardware platform are necessary to run the whole software setup. Hence, reproducible computational science requires a portable, self-contained software setup and additionally, a defined runtime environment.

Driven by the demand for reproducible computational science, virtualization and container technologies have been adopted quickly by researchers (Meng et al. 2015, Boettinger 2015). Containers and virtual machines (VMs) are able to encapsulate a software environment, for instance, a complex software tool-chain including application-specific settings, into a single portable entity. They allow researchers to develop, prepare and test a complex software setup locally and to deploy this setup without additional effort in Cloud or HPC facilities. Hence, the configured environment can be shared and (re-)used independently of its original creation environment. Compared to VMs, low computational overhead is one of the main reasons for the rising popularity of containers.

Hence, containers already offer a set of features convenient for the preservation of (complex) software setups, e.g. portability. However, popular container implementations have been criticized for their poor backward compatibility.[7] To make scientific methods accessible, usable and citable in the long-term, the longevity of containers themselves needs to be ensured. This paper provides a detailed analysis on the archivability constraints of containers. Based on this analysis, we propose a preservation strategy to maintain access to software-based research methods by converting them into a generic archival representation for containers and providing a generic runtime environment.

Containers - A New Class of Digital Objects with Preservation Risks?

Container technology (also called operating system virtualization) is promoted as a lightweight alternative to virtual machines, requiring less resources while maintaining portability. While virtual machines provide a virtual "partitioning" of physical resources, containers leverage the abstraction of the underlying operating system to provide exclusive environments for individual software setups. Thus, container technology virtualizes operating system features like starting and running applications, filesystem access or network connectivity, i.e. virtualizing their corresponding programming interfaces (operating system APIs).

For Linux-based container implementations, the virtualization of programming interfaces is built on a specific feature set of the Linux kernel that provide isolated environments - *namespaces* (Biederman 2006) and (process) control groups (*cgroups*) (Menage 2007). *Namespaces* allow users to create multiple isolated views on the operating system's interfaces. Anything running inside a namespace is separated from other *namespaces*, e.g. other containers or the underlying

6 https://danielskatzblog.wordpress.com/2017/02/07/is-software-reproducibility-possible-and-practical/
7 https://thehftguy.com/2016/11/01/docker-in-production-an-history-of-failure/

host system. *cgroups* are able to control and limit access to system resources shared with the host and among other containers.

Dependency Analysis

Due to the strict isolation, containers need to be self-contained, i.e. all software dependencies (libraries, applications etc.) have to be included within the container. Therefore, the main component of a container is a self-contained filesystem with installed and configured software components. The only remaining external or unresolved software dependency is the underlying operating system, i.e. the operating system's application binary interface (ABI).

The second container component is its runtime-configuration defining, for instance, filesystem mappings, e.g. shared folders between container and its host system, and the definition of an entry-point within the container (i.e. a script or program).

A container's *technical runtime* is composed of two components: a hardware component (the computer) and a software component. In general, the software component represents typically a basic Linux installation with an installed and configured (vendor specific) container runtime.

Furthermore, the isolation of a container simplifies the determination of (hardware) dependencies. For instance, applications running within a container usually have only a very abstract view on available hardware of the system, i.e. having only access to individual files instead of a raw hard disk. Direct hardware access has to be explicitly allowed for by *mapping* entries of the */dev*-subsystem into a container's namespace. Fig. 1 illustrates the full container software stack.

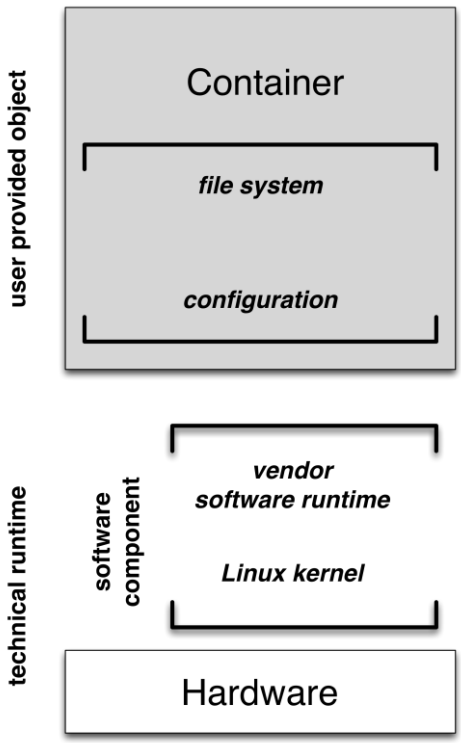

Figure 1. Container components and its technical runtime.

Unfortunately, in addition to these explicit dependencies, there are two further classes of (implicit) dependencies to be investigated. The first class is defined by specific dependencies of the com-

piled software binary. This includes dependencies on specific kernel features, because the software calls certain functions which may only be present from a certain kernel release. Similarly, software binaries depend on a CPU instruction set and highly optimized scientific software may require a very specific CPU version. In a long-term perspective, however, these risks seem manageable, as long as newer kernel versions and newer CPU generations remain available as an emulation environment. A second class of (implicit) dependencies are the containers' *expectations* on an external (technical) environment, e.g. the availability of network services such as licensing servers or data to be available at a certain remote server. This kind of (implicit) external dependencies pose not only the highest preservation risk for an individual container, but require specific effort and infrastructure to be identified and are expensive to monitor. However, every explicit or implicit dependency not only reduces the chances of successful long-term preservation but also limit the portability of a container. Hence, it should be also in the research communities' best interest to maintain external dependencies.

Preservation Risks

The aforementioned properties are not only useful to share software-based research methods but also provide a solid foundation for long-term preservation. Portability, defined/limited external dependencies and a common runtime environment provide the basis for developing a preservation strategy. Furthermore, preservation planning of containers and their technical runtime can be decoupled: while preservation planning of individual containers focuses on determining and monitoring external dependencies (license servers, data dependencies, etc.), preservation planning of the technical runtime focuses on keeping the Linux container runtime usable by replacing outdated hardware with virtual hardware.

Even though containers build on the same technical foundations, currently popular container implementations such as *Docker*[8], *Singularity*[9] or *Shifter*[10] use different container representations and configuration formats. These representations require a specific technical runtime to be used. But ideally, only a small number of software runtimes are necessary to run a large number of containers, hence a common container representation and especially configurations reduces the preservation risk to the availability of a common software runtime. If we assume that the operating system interface rarely changes in incompatible ways, only a few variations of such runtimes are necessary.

With regards to long-term archival, we can assume that a software runtime instance is fixed and achievable. Hence, the long-term preservation risk of containers can be reduced to the availability of hardware required to host the software runtime, i.e. providing necessary hardware supported by the Linux kernel version used as software runtime. When compatible physical hardware becomes unavailable, virtual hardware can be used by means of emulation. In this case, specific properties of containers come in handy again. Typically, software running within containers has no access at all or only a very limited one to the underlying hardware (see discussion above). Furthermore, Linux operating systems support a wide range of hardware. Both properties ease the

8 https://www.docker.com/
9 http://singularity.lbl.gov/
10 http://www.nersc.gov/research-and-development/user-defined-images/

process of replacing physical hardware by emulators. A generic emulation-based preservation strategy is described by Rechert, Falcao and Emson 2016.

Implementation and Preliminary Results

In order to be able to reduce the containers' preservation risks to a generic emulation strategy, container and their runtime require a common archive format, such that for an archive's perspective all container "objects" share the same technical properties, i.e. technical dependencies. For this it is necessary to

1. define a common archive representation of container, as well as provide tool support to ingest and convert container "flavours" into this common format;
2. develop a common runtime for the common container archive format and keeping this runtime long-term available and usable.

Our development builds on the existing Emulation as a Service (EaaS) framework (Liebetraut et al. 2014, Rechert et al. 2012), which provides abstract emulation-based services, the foundation for an emulation-based preservation strategy, and is able to allocate and utilize compute resources in public or private Cloud services. Within the EaaS framework we distinguish between a rendering environment and a digital object to be rendered. A rendering environment is typically represented by a hardware emulator and a software environment (operating system and installed applications) and is able to render a certain class of digital objects. Digital objects are stored and maintained independently and are assigned either manually or in an automated way to a suitable emulation environment (Rechert et al. 2015). To make use of this model, in a first step, a suitable EaaS rendering environment for container has to be defined and implemented.

Integrating a Container Runtime

The Open Container Initiative (OCI)[11] is a governance body formed to create an industry standard for container technology. As part of their work a runtime specification (*runtime-spec*[12]), an image specification (*image-spec*[13]) and a reference runtime implementation (*runC*[14]) have been released.

For preservation purposes, we focus on the runtime-spec and the reference implementation *runC* as a generic container runtime. We have implemented an EaaS rendering environment for containers based on *runC*.

In a second step we have implemented an object *ingest* process for *Docker* and *Singularity* containers. The result of this process is a representation of the the container's file system and a runtime configuration. In OCI terms this is called *filesystem bundle*[15]. Of course, through converting a specific container format into a generic one, information is typically lost, for instance the container creation history represented by *Docker's* image layers. However, the resulting flat con-

11 Open Container Initiative (OCI), http://www.opencontainers.org
12 https://github.com/opencontainers/runtime-spec
13 https://github.com/opencontainers/image-spec
14 https://github.com/opencontainers/runc
15 https://github.com/opencontainers/runtime-spec/blob/master/bundle.md

tainer state is identical with the state of a specific container at the moment of execution, i.e. the container's filesystem is fully assembled, unpacked and usable. Extracting and archiving additional meta-data, e.g. the original `Dockerfile`, which documents the creation process, has to be addressed separately.

A basic runtime configuration (`default.json`) has been pre-configured and will be enriched by the conversion process. Furthermore, defaults can be altered to satisfy the needs of different host environments. For instance, specific namespaces can be set up or specific devices can be mounted.

Example Docker

The conversion process for *Docker* uses the `docker export` command to extract the container's file system. The conversion process further uses `docker inspect` to determine the container's entry point and environment variables, specified during the creation of the container. For instance, the official *MariaDB* image from the *Docker Hub* provides a `docker-entrypoint.sh` as entry point and adds `mysqld` as command that can be altered during the creation of a container, resulting in the final command line "`docker-entrypoint.sh mysqld`". If no entry point is defined, the default entry point is set to `/bin/sh`. The `docker export` command only exports the root file system of a container, mounted volumes are not included in the generated archive. Therefore, it is necessary to archive volumes separately. Changes that are made during the runtime of the *filesystem bundle* are stored in the root file system by default. If a volume does contain data needed for the execution of the bundle, e.g. configuration or research data, it has to be mounted manually in the `config.json`. Currently, the conversion process is not able to recreate the network environment of converted containers. The basic configuration does not utilize network namespaces, such that contained applications share the network configuration with the host machine. If it is desired to isolate the network, appropriate namespaces can be set up. This shortcoming might be an issue especially if the container to be archived is part of a multi-container environment where some containers are linked with others, e.g. an application with a database backend. These links are not part of the Open Container Specification and are set up by the *Docker* Daemon. To recreate the functionality of such containers it is necessary to rebuild the needed network environment and provide the *runC* bundle with a suitable /etc/hosts file. Alternatively, the application configuration has to be changed, which would imply a content-related analysis of the container.

Example Singularity

Similar to the *Docker* conversion process, the configuration is generated based on the same default.json, which is altered to satisfy the needs of different host environments. The command for the *runC* configuration is *Singularity's* `/.run.sh` script, which not only contains the command to be executed but also sets the environment variables defined by the author. If the script is not present, a shell `(/bin/sh)` is used as command.
Because of *Singularity's* focus on portability images created with it already contain all information needed for a successful execution. As described above, the run script sets up the environ-

ment of the contained application. Setups consisting of multiple, linked containers are not intended, since Singularity does not utilize network namespaces. Because of this, a correctly authored *Singularity* image should pose no problem. Successful migration from an bare-metal HPC setup into a cloud-computing environment has been shown using a singularity container of *GROMACS*. The container was originally designed for the JUSTUS HPC Compute Cluster[16].

Access

A container prepared in the aforementioned way can then be accessed using the EaaS framework. Containers map directly to rendering environments in the EaaS terminology and can be started just like any other environment using the provided EaaS REST API by posting a JSON request similar to the following:

```
HTTP POST "http://emulation.solutions/api/components"
{
   "containerId": "gromacs"
}
```

Because the EaaS framework's core component is an abstract rendering environment, containers fall into the same category. The selected environment in this case is the *GROMACS* container which automatically starts its computation when it is invoked and prints results to the standard console output. The framework will return a session ID, which enables allows the user to control the session and send further related requests.

In order to provide different types of access, each component in the EaaS framework has a set of so-called control URLs. Once a session is initiated and a session ID has been retrieved, a control URL can be requested for this session. Depending on the session type the user is either able to interact with the running instance (HTML5) and/or is able to retrieve the container's output.

Since containers usually provide no interactive user interface, we have implemented a *headless* access method, which invokes the container's entrypoint and retrieves output from standard- and error-out output (*stdout* and *stderr*). Both outputs are available as zipped text files. Consequently, in order to retrieve the container's result, the component's control URL. This URL can be downloaded e.g. using a regular web browser at any time. Data transfer does not start until the container's computation is finished. Once downloaded, the ZIP archive contains the mentioned output text files. This mechanism can later be extended to also put a certain directory into the archive where the contained application is known to store its results. A network-based live-interaction mode is currently work in progress. Furthermore, research data needs to be integrated, ideally directly via DOI references.

The presented REST API can be used completely automated, which can also serve as a verification tool to assert that a container ingested into the EaaS system produces the same results as the original container.

16 JUSTUS Compute Cluster, https://www.uni-ulm.de/einrichtungen/kiz/service-katalog/high-performance-computing/justus/

Discussion & Outlook

Containers are and will be interesting research topic in the domain of digital preservation and in particular, reproducible science, not only because of their widespread use but also because of their technical characteristics. This initial study suggests, that if the long-term preservation of containers focuses on their technical runtime, e.g. by structured archival of the software runtime, the preservation risks can be reduced to a generic emulation strategy, which seem to be manageable in a (cost) efficient way (compared to the number of objects to be preserved). However, preserving containers is only a complementary strategy to software archiving and not a substitute. Containers are able to capture a complex software-based research method with built-in quality control (completeness), but for individual re-use of software components an independent software archive strategy is necessary.

While this work provides the necessary conceptual and technical groundwork for preserving containers, a main success factor lies still within the scientific community. Research communities need to agree on usage and access pattern to support interoperability of their research methods. Common, documented access to external services, creates awareness of external preservation risks. This should also include maintaining external dependencies.

References

Altman, Micah, Christine Borgman, Mercè Crosas, and Maryann Matone. 2015 "An introduction to the joint principles for data citation." *Bulletin of the American Society for Information Science and Technology* 41 (3): 43-45.

Callaghan, Sarah. 2014. "Joint declaration of data citation principles." *Data Citation Synthesis Group: FORCE11*.

Smith, Arfon M., Daniel S. Katz, and Kyle E. Niemeyer. 2016. "Software citation principles." *PeerJ Computer Science* 2: e86.

Katz, Daniel S., Kyle Niemeyer, and Arfon M. Smith. 2016. "Strategies for biomedical software management, sharing, and citation." *PeerJ Preprints* 4: e2640v1.

Meng, Haiyan, Rupa Kommineni, Quan Pham, Robert Gardner, Tanu Malik, and Douglas Thain. 2015. "An invariant framework for conducting reproducible computational science." *Journal of Computational Science* 9: 137-142.

Boettiger, Carl. 2015. "An introduction to Docker for reproducible research." *ACM SIGOPS Operating Systems Review* 49 (1): 71-79.

Biederman, Eric W., and Linux Networx. 2006. "Multiple instances of the global linux namespaces." In *Proceedings of the Linux Symposium*, vol. 1: 101-112.

Menage, Paul B. 2007. "Adding generic process containers to the linux kernel." In *Proceedings of the Linux Symposium*, vol. 2: 45-57.

Goecks, Jeremy, Anton Nekrutenko, and James Taylor. 2010. "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences." *Genome biology* 11 (8): R86.

Rechert, Klaus, Patricia Falcao, and Tom Emson. 2016. “Towards a Risk Model for Emulation-based Preservation Strategies: A Case Study from the Software-based Art Domain.” In *Digital Preservation (iPRES) 2016 13th International Conference on*: 139 - 148.

Liebetraut, Thomas, Klaus Rechert, Isgandar Valizada, Konrad Meier, and Dirk Von Suchodoletz. 2014. "Emulation-as-a-Service-The Past in the Cloud." In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*: 906-913. IEEE.

Rechert, Klaus, Isgandar Valizada, Dirk von Suchodoletz, and Johann Latocha. 2012. "bwFLA – A functional approach to digital preservation." *PIK-Praxis der Informationsverarbeitung und Kommunikation* 35 (4): 259.

Rechert, Klaus, Thomas Liebetraut, Oleg Stobbe, Isgandar Valizada, and Tobias Steinke. 2015. "Characterization of CD-ROMs for Emulation-based Access." *Digital Preservation (iPRES) 2015 12th International Conference on* (2015): 144.