SCIRES it
SCIentific RESearch and Information Technology
Ricerca Scientifica e Tecnologie dell'Informazione
Vol 15, Issue 2 (2025), 137-144
e-ISSN 2239-4303, DOI 10.2423/i22394303v15n2p137

CASPUR-CIBER Publishing, http://www.sciresit.it

CLOUD NATIVE TECHNOLOGIES FOR EFFICIENT AND ROBUST STORAGE, TRANSFER, AND CONSUMPTION OF CULTURAL HERITAGE DATA AND BEYOND

Franco Tommasi, Alessandro Longo*, Christian Catalano*, Andrea Chezzi**

*University of Salento – Lecce, Italy.

Abstract

For decades, setting up a server to host a multimedia database of cultural heritage data essentially consisted of installing the necessary software (primarily a web server and database) on a computer connected to the Internet and accessible from any desired node. Since the emergence of virtualization techniques, demands for efficiency, portability, cost-effectiveness, security, and practicality have progressively changed the tasks of service administrators and the knowledge required to fulfill them. This article aims to provide the reader with an update on the state of the art in setting up network services and software distributions, starting from the success of tools like Docker to the newest developments of OSTree, Bootc, and ComposeFS. It is also possible to extend this software management approach to the multimedia content itself to improve storage and transfer efficiency, collaboration among researchers, and the reproducibility of studies by leveraging technologies such as SPARQL, RDF and OCFL to structure content.

Keywords

Server setup, Virtualization, Containers, Docker, OSTree, Bootc, ComposeFS, Cultural Heritage

1. *Introduction*

Docker[1] containers have transformed GNU/Linux systems into an application platform and made possible the realization of a distributed operating system, *Kubernetes*[2]. Around containers, there is a vast ecosystem of sophisticated tools and services. However, the host operating system has not received the same level of attention and therefore does not share the same qualities as containers.

To address this, a particularly interesting approach is that of Bootable Containers, which makes it possible to leverage the container ecosystem to create and manage the operating systems of physical or virtual machines. Moreover, these systems benefit from *ComposeFS*[3] an innovative content-addressable storage that allows operating system images to be stored.

Thanks to ComposeFS, both the images used to boot the host system and those of containers can take advantage of file-based deduplication: each file present in the images is saved on disk only once.

In the case of the host system, this enables the implementation of atomic updates and an automatic rollback mechanism in the event that booting the new version fails. ComposeFS also leverages *FS-verity* to guarantee file integrity at runtime, thus allowing machines to be cryptographically sealed— a property previously seen only on image-based systems such as Android and ChromeOS.

As for the benefits of ComposeFS for application containers, it is interesting to note how this form of on-disk deduplication also extends to RAM, making it possible to instantiate a greater number of containers with the same amount of resources.

It will also be shown in the end how the described techniques may also be relevant in the fields of knowledge management and cultural heritage.

2. *From OSTree to Bootc and ComposeFS*

There are many immutable operating systems with atomic updates. In most of these, the system is booted from an image stored on a read-only disk. A first attempt to hybridize such immutable

[1] https://www.docker.com
[2] https://kubernetes.io
[3] https://github.com/composefs/composefs

systems with traditional GNU/Linux systems managed via packages is represented by *OSTree*[4]: it is a file-based content-addressable storage; like *Git*, OSTree iteratively stores the layers of a directory tree along with the files contained in it, inside a single file-based content-addressable object storage. Unlike Git, the checkout operation does not consist of swapping files between the object storage and a working directory to recreate the desired version; in OSTree, checkout takes place by creating directories and hard links to the files in the object storage to reproduce one of the stored directory trees. Therefore, in OSTree it is possible to have multiple checkouts in different paths, obviously in read-only mode; identical files in different checkouts are saved on disk only once thanks to the use of hard links.

RPM-OSTree (Red Hat Package Manager – OSTree)[5] allows booting the operating system from an OSTree checkout. It also integrates with the RPM package system of distributions such as Fedora and RHEL (Red Hat Enterprise Linux), allowing users to see which packages were used to generate the system image as well as to install additional packages through a layering system similar to that of containers.

OSTree uses its own transport protocol that enables efficient downloading of image updates by leveraging its file-based object storage. In the Docker container ecosystem, the image transport system is similar, but images are organized into layers; in fact, the layers are content-addressable and not the individual files.

Therefore, the container image transport system is less efficient than that of OSTree; however, the recent introduction of the *zstd:chunked*[6] compression algorithm makes it possible to transfer individual files instead of entire layers, making the transfer of OCI (*Open Container Initiative*)[7] images comparable to OSTree in terms of efficiency.

A first step toward the container ecosystem was the addition of support for OCI images in OSTree as a transport system, encapsulating OSTree commits into OCI (*Open Container Initiative*) layers. This allowed reusing the OCI registry infrastructure to store OSTree images.

Bootc[8] is the evolution of RPM-OSTree. It uses ComposeFS and embraces OCI and the container ecosystem as much as possible. Despite the name "Bootable Containers," systems managed with Bootc are not containers, since there is no involvement of a container runtime like *Runc*, nor of *Namespaces*[9] and *CGroups*[10] in general, while ComposeFS uses *OverlayFS*[11].

An image consumable by Bootc is an OCI image but with the peculiarity of also containing the Linux kernel and other adjustments. Since it is a standard OCI image, it is possible to instantiate a container from it. Bootc can download an image from any OCI registry; after downloading, the content of the image is added to the ComposeFS object storage, meaning that the files inside the image are saved to disk. Despite the name, ComposeFS is not a filesystem: it uses regular files in a common filesystem such as *Ext4* or *Btrfs* as its object storage (Scrivano & Walsh, 2021).

The peculiarity of ComposeFS is that the object storage is responsible for storing the content of the original files, but not the metadata needed to recreate the complete directory tree with permissions, SELinux labels, and other file attributes. The format used to store the metadata reflects the structure of the original directory tree but without the content of the files.

For metadata, *EROFS*[12] is used, a read-only filesystem optimized for this purpose. In *EROFS,* files contain an attribute that is used by OverlayFS to link back to their respective files in the object storage. OverlayFS is then used to combine data and metadata, and from this the operating system is booted.

3. *Immutability and integrity*

In a system managed by Bootc, only /var and /etc are writable. The /var directory is dedicated to user additions, including personal files in/var/home. The /etc directory contains system configuration files that override the default behavior; managing /etc requires a three-way merge algorithm to reconcile any changes made by the user with those introduced by a system update.

[4] https://ostreedev.github.io/ostree/introduction/
[5] https://coreos.github.io/rpm-ostree/
[6] https://www.redhat.com/en/blog/faster-container-image-pulls
[7] https://opencontainers.org
[8] https://github.com/bootc-dev/bootc
[9] https://man7.org/linux/man-pages/man7/namespaces.7.html
[10] https://man7.org/linux/man-pages/man7/cgroups.7.html
[11] https://docs.kernel.org/filesystems/overlayfs.html
[12] https://docs.kernel.org/filesystems/erofs.html

The rest of the system directories, and particularly the executables in /usr, are mounted read-only. This also happens in systems managed by RPM-OSTree, but it does not constitute a particularly effective security measure. ComposeFS, on the other hand, significantly improves security since it supports *FS-verity*[13], a variant of *DM-verity*[14].

Image-based operating systems such as Android and ChromeOS use a Linux kernel feature, DM-verity, to ensure that the image is not altered at runtime. With DM-verity, the Linux kernel verifies the hash code of a block of data on disk before allowing a process to access it. FS-verity provides the same functionality but at the level of individual files. Thanks to the special file management provided by ComposeFS, it is possible to use FS-verity to verify the integrity of the entire operating system.

By completing the integrity verification chain starting from the machine's boot process via *UKI*[15] and *Secure Boot*[16], it is possible to create cryptographically sealed devices that are resistant to all attacks requiring disk alteration - including those in which the attacker has physical access to the machine (Longo, 2024).

4. *Automatic atomic updates*

In a package-managed system, an update is downloaded and applied while the system is still running. This can cause instability, and a reboot is recommended. In Bootc systems, updates are downloaded but not applied while the system is running. An update consists of new files in the object storage, which are referenced by a new image. At the next reboot, the system is started from the new image. If the boot fails, Bootc can automatically start from the previous image.

The failure can be detected by the image provider, and a new update can be attempted through a new image. This is particularly relevant in the *IoT* (Internet of Things) or embedded context, as it could be costly to physically restore a large number of devices that may be in hard-to-reach places and rendered unusable by a failed automatic update.

In general, automatic updates can ensure greater security through the rapid propagation of critical bug fixes (Longo, 2024).

5. *Derivation and CI*

Since Bootc uses OCI images, it is possible to derive a new image through *Podman*[17] or Docker. The only difference is that the FROM directive inside the Containerfile (i.e., Dockerfile) refers to an image that also contains the kernel, which is ignored during the build phase of a derived image or when instantiating a container from that image.

Inside the Container file, it is possible to use a package manager to customize the image just as is done with containers. In addition, one can leverage the vast *Ansible*[18] ecosystem by executing playbooks inside containers (since Podman supports running Systemd inside containers) and use the resulting image to update machines managed by Bootc.

The advantage compared to the classic use of Ansible or scripts is that you don't have to explicitly manage the transition from a previous state to the desired system state: Bootc applies an update at system startup, and this update can be based on an image completely different from the previous one without affecting the state of the updated system in any way.

For example, a playbook that upgrades a Kubernetes node through Kubeadm would be forced to move from one minor release to the next, as imposed by Kubeadm; instead, the same playbook can be used to go through each minor release until producing a single final image on the registry. Bootc would download it and apply it to each node, with significant savings in terms of transferred data and computational resources.

Although a workstation user managed with Bootc could derive a new image locally and instruct Bootc to boot from it next time, the methodology that best takes advantage of Bootc is using the same CI infrastructure as for containers. Through CI, it is possible to perform automated tests that prevent regressions and also automate the updating of the base image or other software involved during the build phase. The use of layers in OCI images makes it possible to create a large number of variants of the same image without

[13] https://docs.kernel.org/filesystems/fsverity.html
[14] https://docs.kernel.org/admin-guide/device-mapper/verity.html
[15] https://wiki.archlinux.org/title/Unified_kernel_image
[16] https://www.dell.com/support/kbdoc/it-it/000145423/secure-boot-overview
[17] https://podman.io
[18] https://ansible.readthedocs.io

duplicating disk space and without rebuilding an already available layer, just like for containers.

The ease of derivation is a very important aspect for organizations, as it allows delegating the management of base images to third parties. Consider, for example, the public sector, where there could be base workstation images released at the national level, from which specific images for a region or a local authority could be derived—prepared for secure access to specific IT resources through VPN or similar.

Furthermore, it is extremely simple not only to change the image from which the system was booted but also the OCI reference from which updated images are being pulled: in fact, it is possible to try a completely different operating system and roll back without any risk of instability.

6. *ComposeFS for containers*

Podman, the container manager alternative to Docker but compatible with it, can use ComposeFS to store images on disk. Not only does this save disk space compared to storing Docker layers, but it also optimizes RAM usage: on Linux systems the page cache is based on inodes, so the same file is only loaded once into RAM. This is also true for files with identical content located in different images in ComposeFS, since they are effectively the same files accessed differently via OverlayFS; deduplication is preserved even in the presence of different metadata such as permissions. In addition to deduplication on disk and in RAM, there is also deduplication during image transfer: both Podman and Docker support zstd:chunked compression, which makes it possible to download only the missing files instead of entire missing layers (Scrivano & Walsh, 2021).

7. *The system as a platform*

In a traditional GNU/Linux system, a package manager allows modification of both the system and the applications integrated with it. With containers, the approach that treats the system as a platform has become established, and this is particularly true for an immutable system: although it is possible to modify the Bootc system through the derivation of new images or other mechanisms such as temporary overlays or Systemd-sysext, it is advisable to keep the system minimal and to make the most of containers and similar distribution systems, such as *Flatpak*[19] in the case of graphical applications. It is possible to have containers that are particularly well integrated with the system, such as those created with *Toolbx*[20] or *Distrobox*[21], which replicate the package manager experience, even by instantiating containers from the same image from which the current operating system is booted; this is especially desirable for creating software development environments.

Currently, Bootc's object storage and Podman's object storage are separate, and furthermore, Podman does not yet support ComposeFS in rootless mode. In the future, the goal is to allow Bootc and Podman to draw from the same object storage, thus deduplicating the files of the host system and those of the containers. Moreover, Flatpak is one of the main users of OSTree and could become the third user of a unified ComposeFS storage.

8. *Composability and optimization*

It has been shown through *Nix*[22] (a build and package management system) that it is possible to optimize the creation of OCI images so that they share a large number of layers, taking advantage of a fundamental fact: since Docker's original implementation, each layer has never indicated which layer should be applied previously; instead, each layer is simply identified by a unique code. During the instantiation of a container, a list of identifiers is looked through and the layers are assembled using OverlayFS.

This means that the same layer can be shared across very different images and applied at different "heights." Fully exploiting this implementation detail means ensuring that images share as many layers as possible; while this is obvious for images derived from one another, for independently built images it is not: even the smallest difference in building a layer results in two distinct final layers.

ComposeFS largely solves this problem, since its file-based deduplication is independent of the layer organization. Therefore, layers take on a more logical than physical meaning, providing an

[19] https://flatpak.org
[20] https://containertoolbx.org
[21] https://distrobox.it
[22] https://nixos.org

additional degree of freedom compared to packages, without extra costs.

Since no established term yet exists to define this approach, let us try to introduce one based on the analysis so far. The package system allows software to be assembled based on a graph of relationships; we define this modularity as "intercomposability", since it is closely tied to the "internal" functioning of the various elements that make it up. Iterating intercomposability is a trivial approach, but to distinguish it from the others we call it "multicomposability", emphasizing that each level of composability is independent of the internal implementation of the others (Fig. 1).

Container images, on the other hand, are organized into freely composable layers, with the peculiarity that the final result is always obtained from a specific ordered series of layers, from the deepest to the most superficial. We therefore define this as "overcomposability."

The ability to adopt two different types of composability for two distinct phases or aspects - in our case, one closer to the development phase and the other to the execution phase - allows us to introduce another type of composability; we can define it as "transcomposability."

It differs from multicomposability by the integration between its levels: each is implemented with its own mechanism that considers that of the other levels, with the goal of providing the same functionalities as a multicomposable system, but more efficiently.

The term "composability" highlights more the variety of possible configurations when assembling a system, compared to "modularity," which concerns dividing a system into parts designed to interface with one another.

In the specific case we have examined, we thus have transcomposable operating systems complete with applications and development environments; furthermore, this transcomposability extends to third-party software by reusing the container platform, the standard in the software industry, and Flatpak, the standard in Linux workstation systems.

Beyond combining operating systems, applications, and development environments, there are other fields that can benefit from this approach. The requirement is wanting to access the same files in read-only mode, but organized into different folder structures, with different metadata, and possibly managed with a package system (Longo, 2024).

An example is knowledge management, understood as large amounts of integrated information, shared publicly or among members of an organization, perhaps through Hypermedia systems such as the Web.

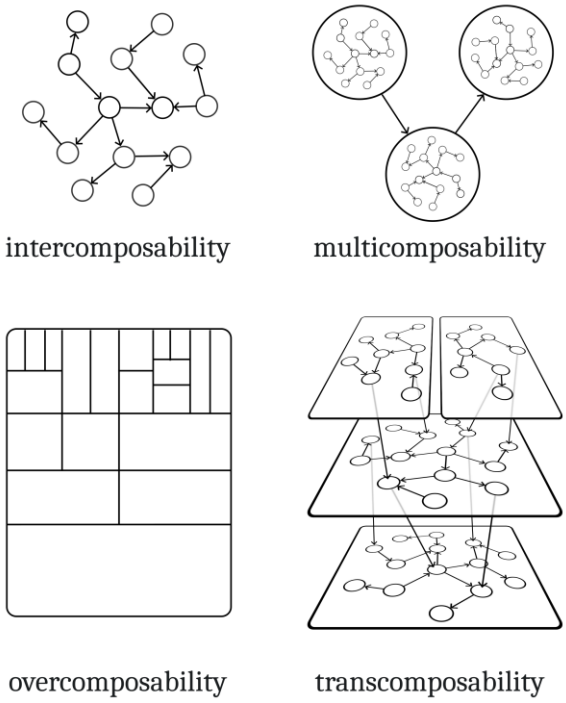

Fig. 1: Intercomposability, multicomposability, overcomposability and transcomposability.

9. *Extending the approach to multimedia content*

In preserving cultural heritage, it is important to balance several aspects:

- enabling access for a highly heterogeneous user base, for example through multimedia entertainment applications for the general public and advanced research tools for domain scholars;
- efficiently transferring and archiving even very large content such as high-resolution images and videos;
- structuring data according to common standards to enable collaboration among different parties.

The field of software development offers many ideas and tools that can be applied broadly to the structured and efficient organization of data. The analogies unfold on multiple levels:

1. Software and knowledge can be stored on disk and transferred over the Internet in the form of files.
2. Software is often organized into packages among which there are dependency relationships, forming the so-called dependency graph. When a user requests to install a package, a package manager traverses that graph, deriving one or more trees from the request and satisfying it by installing all the packages encountered along the path. Knowledge can be organized in graphs and queried with tools such as SPARQL[23] (Arenas & Pérez, 2011; Harris & Seaborne, 2013).
3. A common form of collaboration is when parties base their work on that of others, which may also be updated over time. In software, it is essential to provide a continuous flow of updates, and thus technologies such as OCI containers are adopted, whose images enable a derivation mechanism through which one can benefit from future updates to the base image.

OCI images are generally assembled by leveraging package managers of GNU/Linux systems (APT[24], DNF[25]) or of programming languages (PIP[26], Cargo[27]). As of today, there does not appear to be a package manager for multimedia content. To implement one, existing archival systems could be leveraged, perhaps those that already organize knowledge into graphs. Some online repositories provide SPARQL endpoints that allow querying the graph with queries such as "paintings by authors of Italian nationality born between 1700 and 1800." Projects such as Schema.org[28] seek to standardize node types and relationship labels, and RDF[29] is a common format to encode graphs as triples and enrich the Web with semantic metadata (Cyganiak, Wood, & Lanthaler, 2014; Özsu, 2016). Once the resource URIs are obtained, the package manager can download them and organize them as files in directories on disk. The result can be packaged as an OCI image and saved to an OCI registry.

Regarding directory organization, there are specifications such as the Oxford Common File Layout (OCFL) specification. OCFL, in particular, leverages directories, JSON metadata, and plain text files to ensure properties such as immutability, versioning, and robustness (Hankinson et al., 2019; Jefferies et al., 2022). Both OCI (Open Container Initiative) and OCFL use JSON files to store checksums — for layers in the case of OCI, and for files in the case of OCFL.

It is possible to use OCI as a wrapper around OCFL by simply including the entire OCFL object (its files and UNIX directories) within an OCI image. This approach introduces some redundancy in JSON checksums: OCI checksums would be used to efficiently download and de-duplicate layers while ensuring their integrity, whereas a subsequent program could verify the integrity of individual files according to the OCFL specification. However, this latter verification step would be redundant during the download phase, since corruption of a single file would necessarily imply corruption of the entire layer containing it.

The introduction of zstd:chunked compression and ComposeFS would bring substantial benefits when embedding OCFL objects within OCI images:

1. Since OCFL employs directories for versioning, it is likely that multiple versions within the same OCI image contain duplicate files in different directories. These files may include large multimedia assets, and their duplication would lead to very large OCI images. With zstd:chunked and ComposeFS, each file would be transferred, stored on disk, and loaded into RAM only once, regardless of how many times it appears in the OCFL object layout.
2. When accessing a file stored on disk, it is desirable to verify its integrity via checksums. Users may access files through various programs, and in principle each program would need to implement its own verification mechanism. With ComposeFS, however, integrity verification occurs at the kernel level through FS-verity, and is automatically executed whenever any program accesses a file via a standard

[23] https://en.wikipedia.org/wiki/SPARQL
[24] https://en.wikipedia.org/wiki/APT_(software)
[25] https://en.wikipedia.org/wiki/DNF_(software)
[26] https://en.wikipedia.org/wiki/Pip_(package_manager)
[27] https://github.com/rust-lang/cargo
[28] https://schema.org/
[29] https://en.wikipedia.org/wiki/Resource_Description_Framework

system call. Consequently, no modifications to existing Linux software would be required. This property also holds for Linux containers (notably Podman, due to its ComposeFS support) running on Microsoft Windows or Apple macOS environments.

The advantages described above apply to any layout similar to OCFL and can already be exploited with current technologies. Nevertheless, it would be beneficial to address the redundancy introduced by combining OCFL and OCI, possibly by replacing OCFL with a lighter and more human-readable layout, while delegating to OCI the responsibility for ensuring robustness, completeness, parsability, and versioning. In fact, OCI is one of the most successful standards in the software industry, it has multiple robust and efficient implementations, both client- and server-side, and is supported by many tools and online services. The layout specification could then focus on the semantic structure, for example by mapping the directory tree to a systematic traversal of a graph derived from SPARQL or RDF data — with, for instance, the first directory level representing nationality, the second authors, and the third their works. This approach preserves the freedom to organize content hierarchically while maintaining maximal efficiency, ensuring that each file is transferred and stored only once per machine.

OCI images can contain not only multimedia content but also the application that enables its consultation. For example, they can contain a Web server such as Apache that serves content organized into web pages. Organizing both multimedia files and the software applications to consume them in the same format—OCI—makes it possible to exploit ComposeFS de-duplication.

This enables efficient management of embedded devices for consuming multimedia content, with the following advantages for an organization involved in the preservation of cultural heritage:

1. reuse of the same technologies employed in creating and updating a Web portal also for embedded devices such as those used in museums;
2. efficient storage and transfer of files among servers (hosting the web portal), PCs (for operators who manage the contents and for researchers), and embedded devices (for end users);
3. use of tools widely adopted in the software industry, including automation via Continuous Integration platforms.

The same SPARQL endpoints could be deployed via OCI images.

By automatically retrieving and organizing heterogeneous resources, the package manager could provide researchers with a reproducible and verifiable environment in which the provenance of every file is explicitly documented. This would not only strengthen the transparency of scientific results, but also reduce duplication of effort, since the same resources could be shared and reused across projects without unnecessary redundancies.

In the field of cultural heritage, the availability of curated OCI images would enable institutions to distribute thematic corpora—such as collections of manuscripts, audiovisual archives, or annotated datasets—in a way that preserves both their internal organization and their connections to external references. Researchers could then compose new collections starting from these corpora, layering additional metadata, annotations, or interpretive frameworks, while continuing to rely on the same underlying sources.

The reproducibility of studies could be ensured by adopting the OCI format not only for multimedia content but also for the source code used for statistical processing and for producing scientific studies from source code, such as LaTeX. This approach would help ensure that what is published is consistent with sources and the processing reproducible.

More generally, the introduction of a generic, dependency-aware package manager would bridge the gap between software engineering practices and knowledge management. Just as a software build can be reconstructed from a dependency graph, academic and cultural collections could be reconstructed from explicit references to the materials that constitute them. This would contribute to reproducibility, verifiability, and long-term interoperability—three pillars increasingly recognized as essential both in scientific research and in the preservation of cultural assets.

REFERENCES

Arenas, M., & Pérez, J. (2011). *Querying semantic web data with SPARQL*. Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.

Cyganiak, R., Wood, D., & Lanthaler, M. (2014, February 25). *RDF 1.1 concepts and abstract syntax*. World Wide Web Consortium. Retrieved from https://www.w3.org/TR/rdf11-concepts

Hankinson, A., Brower, D., Jefferies, N., Metz, R., Morley, J., Warner, S., & Woods, A. (2019). The oxford common file layout: A common approach to digital preservation. *Publications, 7*(2), 39.

Harris, S., & Seaborne, A. (2013, March 21). *SPARQL 1.1 query language*. World Wide Web Consortium. Retrieved from https://www.w3.org/TR/sparql11-query

Jefferies, N., Metz, R., Morley, J., Warner, S., & Woods, A. (2022, November 15). *Oxford Common File Layout specification v1.1*. Oxford Common File Layout. Retrieved form https://ocfl.io/1.1/spec

Longo, A. (2024) *Metodi Cloud Native nel settore workstation*. [Tesi di Laurea triennale in Sistemi Operativi, Università del Salento, Dipartimento di Ingegneria dell'Innovazione].

Özsu, M. T. (2016). A survey of RDF data management systems. *Frontiers of Computer Science 10*(3), 418-432.

Scrivano, G., & Walsh, D. (2021). *Pull container images faster with partial pulls.* Red Hat Blog. Retrieved from https://www.redhat.com/en/blog/faster-container-image-pulls

WEB SOURCES

Bootc. Retrieved from https://containers.github.io/bootable/

ComposeFS. Retrieved from https://github.com/composefs/composefs

DM-verity. Retrieved from https://docs.kernel.org/admin-guide/device-mapper/verity.html

*Docke*r. Docker Inc. Retrieved from https://www.docker.com/

EROFS. Retrieved from https://docs.kernel.org/filesystems/erofs.html

Flatpak. Retrieved from https https://flatpak.org

FS-verity. Retrieved from https https://docs.kernel.org/filesystems/fsverity.html

Kubernetes. Cloud Native Computing Foundation – CNCF. Retrieved from https://kubernetes.io

OCI -Open Container Initiative. Retrieved from https://opencontainers.org

OSTree. Retrieved from https://ostreedev.github.io/ostree/introduction

OverlayFS. Retrieved from https https://docs.kernel.org/filesystems/overlayfs.html

Podman. Red Hat. Retrieved from https://podman.io

RPM-OSTree (Red Hat Package Manager – OSTree). Retrieved from https://coreos.github.io/rpm-ostree