CEOS

Working Group on Information Systems and Services
Data Stewardship Interest Group

Software Preservation White Paper

CEOS
Data Stewardship Interest Group

Doc. Ref.: CEOS.WGISS.DSIG.LSP
Date: September 2025
Issue: Version 1.0

Contents

SCOPE

This white paper is intended to assist data/software managers in the field of Earth observation (EO) with the task of ensuring the long-term preservation of EO-related software, thus improving accessibility and usability for potential future users. The intended audience comprises data and software providers, decision makers and scientists, and data managers/stewards for data centres and repositories. This document introduces the concept of software preservation, outlining its importance and justifying the importance of expending to effort to ensure effective preservation of EO software. Details of the main principles of software preservation are provided, as well as brief descriptions of the primary strategies that may be implemented by data managers. Consideration is also given to the typical challenges that may present throughout the preservation process.

INTRODUCTION

Long term digital preservation is the act of managing, maintaining, and preserving digital objects (information, documentation, data, etc.) so that they may be accessed and used by potential future users. However, without the necessary associated software, interaction with preserved data might not be possible, thus rendering it meaningless. Hence, software preservation can be a logical extension of the data curation process. Despite this potentially critical dependency, and the fact that long-term data preservation is widely regarded to be of high importance, only very limited consideration has been given to the preservation of software as a digital object.

When compared to many other types of digital object, software has many characteristics which make preservation significantly more challenging, particularly for people who was not involved in the development of the software. Specific hardware requirements as well as the typically large number of software libraries/dependencies, combined with particular operating system requirements and licence agreements, present a major technical barrier which generally consigns software preservation to being viewed as a secondary activity of limited utility.

Despite the associated difficulties with preservation, software is often a vital resource for the exploitation of data, and it is imperative that software is preserved to avoid obsolescence-related issues in the future. Preserving all associated dependencies to ensure that they can be met in the future is also of importance, in addition to ensuring that adequate documentation is maintained for running the software and interpreting results. The rapidly changing information technology landscape means that software preservation activities should be initiated as soon as possible to avoid additional difficulties.

WHY TO PRESERVE SOFTWARE

There are a few reasons that justify expending the effort required to preserve software, foremost of which is the simple fact that since data is being preserved, any associated software should be preserved alongside it to maintain the data's maximum value. Moreover, having access to the original software is crucial for any potential reanalysis/reproduction of earlier work, ensuring that researchers can understand how data was processed in the past, thus enabling meaningful comparisons with modern data

analysis. This ability to reanalyse data is of particular relevance in the field of environmental monitoring and analysis where long term data time series are necessary for understanding how the Earth's climate is changing. These time series typical consist of data from a range of different instruments that need to be reanalysed to permit accurate intercomparisons, hence full knowledge of how past data has been analysed is crucial.
Preserved software may also find use among the research community as a foundation for future advances as the software, or a portion of it, may present a viable starting point for the development of new methods and technologies, akin to modular software development efforts.
In addition to everything above, consideration must be given to the intrinsic heritage value that software may possess. Data processing software provides valuable historical context for scientific data as it reflects the analysis methods and technologies used over time. This insight into past analysis techniques may inform and inspire future research. Furthermore, software may possess some degree of cultural significance if it contributed in some way to a notable event/research/discovery and should therefore be preserved for posterity.

CHALLENGES AND COMPLICATIONS

A wide array of challenges face software preservation, from technical complications to legal issues and resource availability. A well planned and proactive approach to software preservation may help to avoid some of the most encountered challenges, particularly for larger, well-resourced organisations. The following provides a brief introduction to some of the most common difficulties that occur when preserving software.
As previously stated, a shortage of documentation and expert knowledge can significantly hamper the preservation process. A lack of expert knowledge may have a particular impact on the preservation of software that has been inherited from the original developers and where current users might only be experienced with using certain aspects of the application. In such cases, preserving the full functionality of the software may be difficult as data managers/preservationists may not be familiar enough with the entire software to accurately assess whether all aspects are functioning as designed.
The risk of partial or full obsolescence prior to beginning the preservation process poses a critical treat that is particularly pronounced for software that depends on any specialised hardware. The need to avert this risk further reinforces the importance of planning the preservation activities as soon as possible so that software and hardware may be preserved while they are still available. Similarly, early adoption of a preservation strategy will help to ensure that adequate documentation and expert knowledge are collected.
A major hurdle to be overcome during the software preservation process is the issue of intellectual property (IP) rights which may prevent the distribution of the software without first receiving explicit permission from the rights holder. Likewise, since software preservation often requires the manipulation of the software in a manner which may not be permitted by the licence agreements, simply preserving the software for personal use might not be permitted either. In such cases, effective preservation would depend on the owner of the IP granting specific permission – something which they might not be willing to do. Further complications arise from the fact that, in many cases, it may not even be possible to identify or contact the holder of the IP rights. In general, successfully navigating the

licencing aspects of software preservation will require a somewhat nuanced understanding of intellectual property rights, for which assistance from outside experts may be necessitated.

Beyond technical and legal hurdles, the adoption of modern software development methodologies (e.g. Agile, DevOps, etc.) introduced a major shift in terms of how software "completeness" is defined. In traditional software development, the development process yielded a discrete, finished product which could then be preserved; however, modern practices generally result in a continuous stream of incremental updates to software. As a result, there may not necessarily be a single definitive version of the software to preserve, greatly complicating the preservation process. In such cases preservation should be treated as a continuous process, preserving the entire version history of the software. Similarly, the general migration from traditional monolithic software applications (which can be preserved as single self-contained units) to microservices and cloud-native applications, which may consist of many decoupled components and external APIs, presents significant challenges for software curators.

The fundamental complication facing all preservation activities – software or otherwise – is the issue of resource availability, both in terms of personnel and finances. Beyond the expected costs associated with storing and hosting software, software preservation activities can involve sizable expenses, if employing a strategy of technical preservation for software that relies on specialist hardware that must be purchased and preserved. Direct personnel expenses may also be incurred if data managers lack the necessary skills and technical knowledge to carry out the preservation activities “in house”, thus necessitating the contracting of outside experts to assist with the process. Even if all the required expertise is available, preservation is still a time-consuming process that requires significant effort and commitment from the personnel involved. While larger entities may possess the necessary resources (expertise, staffing, and funding) for software preservation, for others the expense may be difficult to justify. In such cases, the approach to preservation would have to be tailored to the current capabilities of the organisation, and while it should be avoided, if possible, on occasion, a policy of *hibernation* or *procrastination* may have to be adopted until such time as resources are available to proceed with a more robust and reliable approach to preservation.

PRINCIPLES AND APPROACHES OF SOFTWARE PRESERVATION

Throughout the EO software preservation process, data managers/curators can ensure that the maximum benefit is obtained from their efforts by adhering to three simple principles:

1. Collect
2. Preserve
3. Share

These three principles, which correspond to the different phases of the software preservation process, help to guarantee the robust preservation of software while maximising benefit to, and usage by, potential future users of the software. Brief descriptions of each are provided below.

Collect

As the first step in the software preservation process, all the constituent components of a software environment are to be collected/retrieved. This includes source code, libraries, the operating system, dedicated hardware, and any other dependencies which are identified. It is impossible to predict the evolution of the technology landscape, hence, during the collection phase, no assumptions are made on what aspects of a software environment may or may not be available in the future. As change history represents an intrinsic aspect of a software application, the version history, if available, should also be retrieved for inclusion in the preservation activity. Efforts to collect all the relevant components should commence early in the lifecycle of the piece of software, ideally during the development phase, as this helps to avoid difficulties in "tracking down" these components when preserving the software. Notably, if the current userbase is entirely separate from the developers, and the original developers are not involved in the preservation process, difficulties may arise if the current users are not fully aware of all the software's dependencies.

In addition to the software itself, it is imperative that all relevant documentation is also collected for preservation. While collection of installation guides and software manuals should be relatively trivial, it is important to consider the expert knowledge and understanding held by the developers and current users of a piece of software. If this knowledge is lost, the utility of a piece of software may potentially be diminished for a future user, especially if existing documentation is found to be inadequate. This risk may be averted through the collection of detailed explanations from expert users describing how they used the software accompanied by relevant screenshots and video screen captures which may be packaged with the preserved software for the aid of future users. As people retire or move on to different work their expertise could be lost at any time, especially in the case of highly specialised software with a small userbase, therefore, efforts should be made to ensure that this expert knowledge is harvested early in the lifetime of a software application. To further assist potential non-expert future users, test datasets (where applicable) can be prepared along with "tutorials" which may guide the user through the data processing procedure, while also serving as a test to confirm to the user that the software is working as designed.

Preserve

Several different strategies exist for preservation of software (see *Software Preservation Strategies* below) while the choice of which strategy to adopt depends on a few factors including total expense, technical know-how, the existence of a current active userbase, and the expected demand for the software in the future. For software that has an active user

community or is expected to see somewhat regular use in the future, then adopting a "live" preservation approach would be appropriate, maintaining the software in an operational, or near-operational, state. Conversely, a more passive approach to preservation may be taken for software that is expected to only see minimal or sporadic use in the future and for which only limited funding for preservation can be justified.

Regardless of the preservation strategy adopted, all the previously collected components (software, dependencies, documentation, etc.) are to be preserved in their entirety, again without making any assumptions on future importance. For an end user with no prior experience with a piece of preserved software that is looking to utilise it, access to comprehensive documentation is essential. Software should always be preserved such that it is packaged together with all related documentation. While associated documentation such as user guides may already be stored in a repository/archive, simply pointing to this is inadequate as it intrinsically ties the longevity of the preservation activity to the status of the documentation archive.

If the preserved software is not packaged in a manner that allows it to be immediately executable, then specific installation instructions should be provided to aid a future user in recreating the original functionality of the application. As technology evolves over time these installation guides should be periodically updated to reflect the common computer systems of the day. Likewise, software that is preserved in an operational state (e.g. containerised) should also be periodically assessed and updated considering technological developments to maintain this status – software which is immediately executable today might not be so in the future.

Just like with the preservation of any other digital object, guaranteeing redundancy is central to the software preservation process, and many of the principles of preservation of e.g. data can also be applied to software. For reliable preservation of software, redundancy needs to be ensured both in terms of file format and storage. For storage, preservation best practices are to be followed with regards to multiple identical copies ideally stored in different locations and media formats, while also following media refreshment procedures. Ensuring redundancy of format is of particular importance in the case of preservation strategies which rely on an underlying technology (e.g. containerisation). In these cases, it is unwise to depend entirely on one specific technology as any unexpected obsolescence would have significant knock-on effects for the preservation activities. For example, if using containerisation, effort should be made to not rely solely on Docker for containerising the software, but to also utilise another form of containerisation.

Share

As important as the actual preservation activities, the sharing of software is a fundamental part of the software preservation process. To achieve the full benefit of preservation activities, preserved software should be indexed, organised, and most importantly, discoverable. Discoverability refers to the ability to locate and access a digital object in a meaningful and efficient manner and is crucial to maintaining its relevance and value to users over time. As with any other digital object, the persistent identification of preserved software is an ongoing challenge. The ability to unambiguously locate and access the software is crucial to its continued use – a property that is not guaranteed by simply relying on a URL to refer to the object, and simply put, preserved software that is not readily

accessible will inevitably die. Associating a persistent identifier (PID) with the software ensures that it will continue to be discoverable and accessible even if it is renamed or moved to another server or organisation.
As for the form which software should take for distribution to users, the most appropriate choice depends largely on the nature of the software and on the preservation strategy adopted, and should be tailored to the specific needs, whether it be, for example, releasing the software as a containerised application, in the form of a pre-built binary, or simply as source code to be built by the user. Factors to consider include the expected future demand for the software and the level of expertise of future users (source code may be inappropriate for non-technical users, for example).
Preserved software should have clear licencing details and, wherever possible, should be released as free/open source. Restrictive licencing is generally detrimental to the future value of any preserved software. As time goes on, the identity of the rights holder becomes less clear and they may become uncontactable, thus interfering with future reuse of the software, not to mention that restrictive licencing totally inhibits any potential formation of a community around the continued development of the software as a form of "live preservation" (see *Cultivation* below).

In the Space context three specific cases have been considered:

Future Missions:

- Recommendations about software engineering best practice such as clear licensing; clear documentation; use of commonly adopted and modern programming languages; modular design; revision management and change control; established software testing regime and validated results; separation between data and code; and clear understanding of dependencies all make the work of software preservation easier. If the already existing Software Engineering Best Practices cover all preservation requirements, only a statement, with the standards references, could be highlighted.

Historical Missions:

- Recommendations take into consideration different scenarios based on data uniqueness, available funding, etc. *Example of Recommendation: if reduced funding is available and similar instrument dataset already preserved then hibernation and procrastination recommended (i.e.* keep in original format without any further effort).

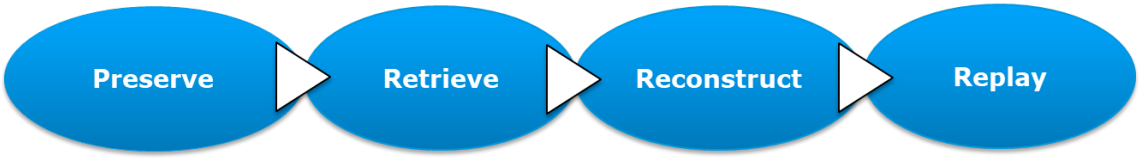
Current Missions:

- Mixed approach. For any new development, the Future Missions recommendations should be implemented, otherwise the approach to be followed is the Historical missions one.

SOFTWARE PRESERVATION STEPS AND ATTRIBUTES

The following steps in Software Preservation have been identified:

- ***Preserve***: a copy of a software "product" needs to be stored for long term preservation. There should be a strategy to ensure that the storage is secure and maintains its authenticity over time, with appropriate strategies for storage replication, media refresh, format migration etc.
- ***Retrieve***: to retrieve it at a date in the future, it needs to be clearly labelled and identified, with a suitable catalogue. This should provide a search on its function and origin (provenance information).
- ***Reconstruct***: The preserved product can be reinstalled or rebuilt within a sufficiently close environment to the original that it will execute satisfactorily. For software, this is a particularly complex operation, as there are many contextual *dependencies* to the software execution environment which are required to be satisfied before the software is executed at all.
- ***Replay***: To be useful later, software needs to be executed and perform in a manner which is sufficient close in its behaviour to the original. As with reconstruction, there may be environmental factors which may influence whether the software delivers a satisfactory level of performance.



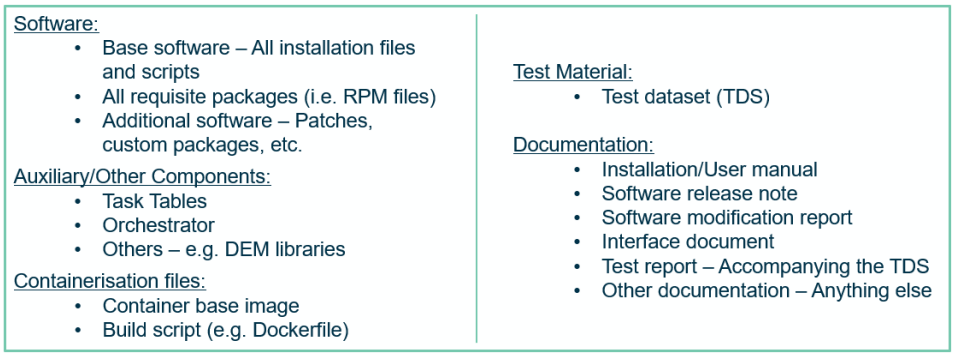
The identified software preservation attributes to be considered are the following:

Attribute	Definition	Why it is Important for Preservation
Functionality	The set of operations the software performs and the specific data schemas/types it requires to produce valid outputs.	Preserving correct functionality is at the core of software preservation. Rebuilt software may "run", but might not "function". Functionality may be confirmed through the use of test datasets preserved with the software.
Environment	The specific "stack" required for execution, including the CPU architecture, OS kernel, and runtime versions.	Version shifts in a compiler or OS can render the source code unbuildable or the binary non-executable.
Dependencies	External assets not included in the primary software, such as	This is the most common point of failure. If a dependency is no longer publically available, it might not be

	standard libraries, drivers, or other dependent software.	possible to rebuild the preserved software.
Composite Nature	The recognition that software is a bundle of heterogeneous files: source code, build scripts, documentation, etc.	Preserving only the primary software is often insufficient for long-term viability. Losing the build scripts or documentation removes the context of how the software was meant to exist.
Architecture	The structural design of the system, including memory management, storage, and input/output protocols.	Software often relies on specific input/output behaviours to handle datasets.
User Interaction	The interface through which a human or system interacts with the software (CLI, GUI, etc.).	A user's ability to interact with the software needs to be preserved. If a GUI's graphical library is no longer supported, the software's utility is lost even if the backend logic survives.

EO Software Preservation Module

The EO Software Preservation Module is a collection of everything required to adequately preserve the software for future use. Not everything is mandatory, but if it exists it should be preserved. In the table below lists the elements to be considered during the preservation process, highlighting the need to also preserve containerisation files if relying on this approach for preservation.



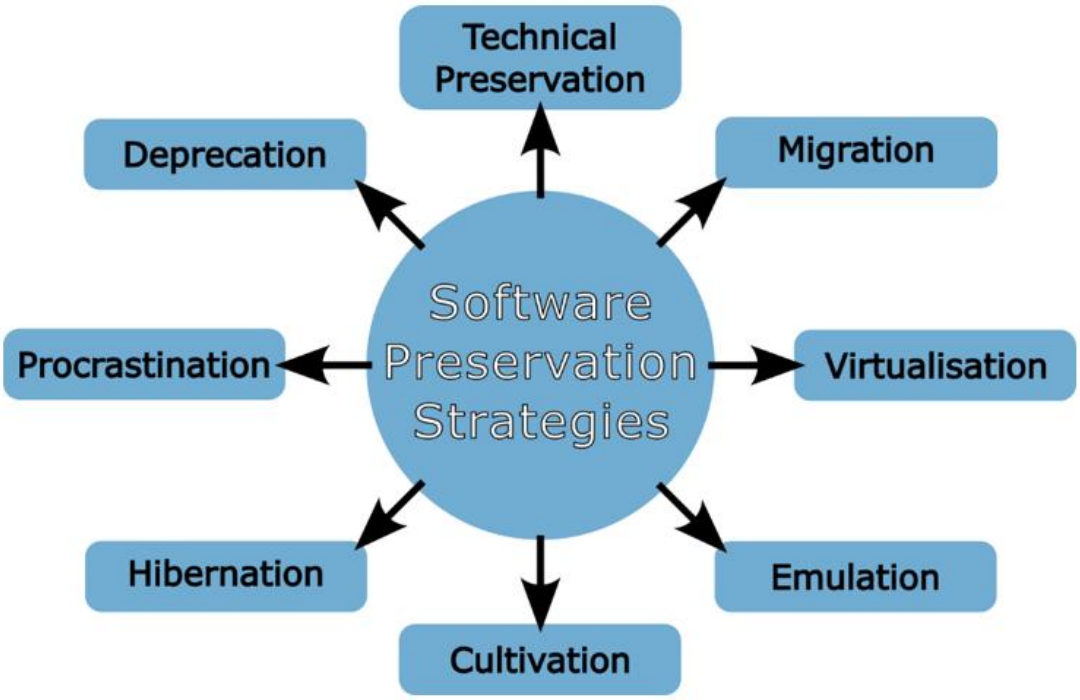
SOFTWARE PRESERVATION STRATEGIES

When it comes time to initiate the software preservation process there exist a few strategies available to the data curator responsible. The choice of strategy depends on the exact nature of the software to be preserved, and in some cases more than one strategy, or a hybrid approach, might be adopted. The decision on which strategy to be put in place for the mission related software preservation is demanded to the Appraisal decision process

applying the CEOS EO Data Collection Appraisal Procedure. The decision ultimately depends on software relevance, complexity, storage media and archiving format, technical aspects and cost.

While software preservation activities should begin as early in the software lifecycle as possible, over time the chosen preservation strategy may evolve with the software itself. Early in the software lifecycle, strategies that ensure preservation of all of the technical details of the piece of software may be suitable, eventually giving way to strategies such as *virtualisation* as the software reaches a mature/stable phase such that the current state of the software is accurately preserved. As the software approaches the legacy phase, the choice of strategy to follow will largely depend on the current and expected user-base as well as the expected utility of the software in the future, with strategies of varying complexity and robustness being available to the software curator.

The following provides brief descriptions of the primary software preservation strategies available.



Technical Preservation

A conceptually straightforward approach to software preservation, technical preservation is the preservation of the entire technical environment, including all necessary hardware and software. In preserving the software environment, the operating system, drivers, libraries, and any other dependencies must be preserved, without making any assumptions on what components may or may not be available at some point in the future. Likewise, hardware preservation covers all the components of the technical environment including hard drives, processors, etc.

While seemingly a robust technique for software preservation, technical preservation is unlikely to be a sustainable long-term solution. As technology advances, hardware will eventually become obsolete and fall out of production, at which point, replacing any faulty hardware will become impossible. While stocking up on old hardware may prolong the life of the preserved system, it is unlikely to be able to address the issue in perpetuity. Technical preservation is also hindered by its lack of scalability, being best suited for small, simple systems which will require little to no distribution to any potential userbase. While source code preserved in an online repository (e.g. Software Heritage) may be readily distributed to future users, the utility of the software becomes increasingly difficult over time as hardware necessary to run the software becomes obsolete.

Summary:

Easiest way to ensure that there will always be hardware on which to run your software is to preserve the hardware (and its operating system and any other reliant software). Only practical for preservation of small-scale systems that will not require distribution to any future userbase. Prone to failure due to the reliance on the physical preservation of hardware.

Advantages:

- It is easy and clearly defined
- Can change to Hardware Emulation at later date

Disadvantages:

- Costly, especially when hardware fails
- Does not guarantee future access if dependent on other hardware/software (e.g. networking)
- Maintenance (over time hardware components will wear out and must be replaced; if the hardware is no longer manufactured components may become scarce and expensive)
- Isolation (if the software only works with very specific hardware, the userbase is limited to those people with the right hardware)

Migration

Migration is when the software which is to be preserved is moved to a new platform before the current one becomes obsolete. This process may involve re-writing, or translating, the software from one programming language to another, as well as recompiling and reconfiguring the software source code to generate new binaries adapted for the new operating environment.

For translation of software, access to the original source code is a necessity, as well as an intrinsic understanding of how the software functions to accurately replicate its behaviour. Availability of extensive documentation is also of great benefit to the migration process. However, even with full source code and documentation availability, the technically demanding and time-consuming process means that migration as a method of software preservation is typically only practical for specific applications that do not consist of an overly large amount of source code.

There are generally two approaches which may be adopted for software migration:

- Complete re-writing of the code allows the software to be used on a completely different system
- Continual migration, keeping code up to date with the latest changes to the hardware and software that code relies on

Migration is a robust means of preservation, ensuring that the software remains fully functional on modern systems, however successful implementation requires significant effort to be expended. Accurate migration of software requires a deep understanding of all aspects of the software, hence migration activities are best performed by the actual developer of the software, where possible.

Summary:
Updating software is required to maintain the same functionality, porting/transferring before platform obsolescence. Periodic transfer from one HW and/or SW configuration to another, or from one generation of computer technology to a subsequent one. As the risk of failure scales with the complexity of the software, particularly for non-expert users, end users planning to apply a policy of software migration should accurately assess their understanding of the software before proceeding and should ideally limit migration to rather simple pieces of software.

Advantages:
- Enables access on other platforms
- Retain ability to retrieve/access/use data exploiting technology progress

Disadvantages:
- Requires continued effort for development; significant effort if complex SW; time-consuming, costly

Virtualisation

As a simple definition, virtualisation enables legacy software to be run on modern computers by imitating the original software environment. In doing this, virtualisation is used to create independent, self-contained, virtual instances of an environment which are then run on the host machine while being able to directly access the underlying hardware of the host system. As an extension to virtualisation, containerisation technology packages applications with their dependencies into a single “container” and allows them to be run as isolated units. Containers provide a level of consistency and standardisation and can be easily moved between different environments without any changes, thus simplifying the migration and distribution of preserved software. The wide range of well-documented, relatively simple, containerisation solutions currently available (Docker, Podman, rkt, etc.) mean that it is an ideal approach for software preservation activities based on virtualisation. However, for preservation activities relying on a virtualisation approach, it should be noted that the fact that virtualisation requires the virtual systems to be capable of running on the host hardware may pose a limitation as it raises the possibility of hardware obsolescence becoming an issue at some point in the future.
Different types of hardware virtualisation include:

- **Full virtualisation** – complete simulation of the actual hardware to allow software to run unmodified.
- **Partial virtualisation** – some but not all the target environment attributes are simulated.

It should be noted that hardware virtualisation is not the same as hardware emulation. In hardware emulation, a piece of hardware imitates another, while in hardware virtualisation, a hypervisor (a piece of software) imitates a particular piece of computer hardware or the entire computer.

Summary:
Hardware virtualisation or platform virtualisation refers to the creation of a virtual machine that acts like a real computer with an operating system. Software executed on these virtual machines is separated from the underlying hardware resources. As virtualisation requires architecture parity (e.g. x86) between the preserved software and the host environment, it is best suited to software that has been developed relatively recently. In the case of architecture mismatch, a strategy of emulation (see *Emulation* below) may be necessary.

Advantages:
- No need to update/migrate software running on a Virtual Machine when underlying hardware changes

Disadvantages:
- Need to virtualise SW to run on virtual machines

Emulation

Related to virtualisation, emulation also allows legacy software which was originally designed to run on now obsolete hardware to run on modern computers. This functionality is achieved by using software to imitate the original system, including hardware, and create an emulated environment in which the preserved software can now be run, thus eliminating the need to maintain old hardware far beyond its effective end of life.
While emulators offer an effective means of preserving software, there are some drawbacks. In particular, the technology behind emulators is rather complex, and their development requires a certain degree of expertise, time, and expense. Hence, if an adequate emulator is not already available, the adoption of emulation as a software preservation strategy might not be a viable option for data curators that do not possess the technical know-how. Even in situations where an appropriate emulator is already available, data curators should be mindful of the possibility that the emulator may, at some point in time, cease to be supported, thus necessitating the in-house development of a replacement. Additionally, when compared to virtualisation, emulation is significantly more resource-intensive and therefore may not present a good strategy where scalability and ease of distribution of preserved software environments are desirable.

- **Emulate hardware platforms** with another piece of hardware, typically a special purpose emulation system

- **Emulate applications & Operating System:** ability of the software to emulate (imitate) another software (e.g. application or operating system)

Summary:
Emulation addresses the original HW & SW environment of the digital object and recreates it on a current machine. The emulator allows the user to have access to the software on a current platform as if it was running in its original environment.
In the context of software preservation, emulation is generally analogous to virtualisation with the choice of implementing full emulation likely being driven by a mismatch between the original software architecture and the needs of the host environment (e.g. preserving x64 software in an ARM environment). Consequently, the older the preserved software is, the more suitable emulation is as a strategy.

Advantages:
- Emulating hardware is easier to manage
- If the emulation layer continues to be developed, software can continue to be run indefinitely

Disadvantages:
- The appropriate emulator needs to be found; in the case of old hardware that is rare, no emulator might exist
- Need all aspects of hardware to be emulated correctly
- The emulation software itself could become obsolete

Cultivation

Cultivation is the process of keeping software "alive" by moving to an open development model, involving additional contributors to distributing the responsibility and expertise associated with the software. As more contributors join the project knowledge is spread around and the future of a project is less likely to depend entirely on one or two people, thus ensuring the continued life of the software. Adopting a cultivation approach is a long-term process which requires firm commitment from the original developers of the software, releasing source code publicly with appropriate licensing, and expending the effort to stimulate the creation of a self-sustaining community around the software. Throughout the cultivation approach, governance for the software must be established, and data curators should be prepared for the possibility of the community collapsing, thus requiring a pivot to an alternative strategy to ensure continued preservation of the software.

Summary:
Cultivation is the process of opening development of an own software. It aims at keeping software "alive" by moving to a shared development model through:
- Distributing and spreading knowledge about the software to minimise single point of failures (e.g. departure of a developer)
- Building a self-sustaining community of developers working together and sharing efforts to keep software up to date

Naturally cultivation is best suited to software for which there is already an active user community willing to take on the continuous development of the software.

Advantages:

- Increases chances of further development of software and possible migration to other platforms

Disadvantages:

- Long process, requiring more coordination
- Possibility for loss of direction control
- Time investments into building the community, which requires work to understand what the community wants and how to appeal to them

Hibernation

For software that has been assessed to be unlikely to see any regular usage in the future (e.g. to be used only for sporadic reanalysis of data/checking of results), hibernation may be an appropriate strategy for preservation. Similarly, hibernation may be appropriate if the software currently lacks a user community, but one may come into existence at some point in the future.

In preparation for hibernation of software, effort needs to be made to ensure that everything necessary to rebuild the software in the future is archived. In particular, source code or pseudo-code, rigorous technical documentation, usage guides/examples, and test data sets are critical for any future user attempting to rebuild the software to its full functionality.

The status of hibernated software should be assessed periodically to determine if a potential need to resurrect the software is forthcoming, at which point a decision should be made on whether to resurrect it, maintain the software in a state of hibernation, or to begin activities to migrate to a more robust preservation strategy which maintains the software in an operational state.

Summary:

Hibernation aims to preserve the knowledge of how to resuscitate/recreate the exact functionality of the software later. Applicable for example when:

- the software has come to the end of its useful life, but there is the possibility that it might need to be resurrected to double-check analysis or prove a result in future
- Currently there is no user community, but one may arise in the future

Advantages:

- After the initial gathering of all necessary components, hibernation requires very little effort to maintain

Disadvantages:

- Without rebuilding the software, it is difficult to determine if all necessary knowledge and components have been collected. Any components missed during the hibernation process may not become apparent until it is too late.
- Preparing software for hibernation can be resource heavy, and if the software is never resurrected this might be considered a waste of resources

Procrastination

Procrastination is the act of doing nothing at the current time to allow for preservation of software. For well-maintained, active software, procrastination may be an acceptable short-

term strategy, however one runs the risk of having to expend significant effort in the future to keep software alive if certain aspects become obsolete and cease to function fully. Best practice is to begin the process of preparing for preservation very early in the lifecycle of a piece of software, therefore maintaining a policy of procrastination runs contrary to these best practices and should be avoided.

Deprecation

Not technically a means of preservation, deprecation is when active use of software comes to an end but, unlike hibernation, no effort is expended to prepare the software in such a way as to facilitate its potential future recreation, and it is simply archived as is. Should a need to resurrect the software arise in the future, any prospective user will hence find it exceedingly difficult to achieve full functionality of the software due to the lack of documentation, test data sets, etc.

Considering the highly disruptive, and likely permanent, nature of software deprecation, the strategy should only be adopted as an absolute last resort. Prior to making the decision to deprecate software, every effort should be made to identify any current userbase. If it is not feasible to maintain software despite the existence of an active userbase, e.g. due to lack of funding, then current users should be given the opportunity to "adopt" the software and take over responsibility for development and governance (see *Cultivation*). In any case, sufficient time should be allowed prior to shutting down the software to allow any users/contributors to be notified of the intent to deprecate.

FUTURE READING

N. C. Hong, "Digital Preservation and Curation: The Danger of Overlooking Software", in *The Preservation of Complex Objects, Vol. 1, Visualisations and simulations*, pp. 25–35 (2012). ISBN 978-1-86137-6305

R. A. Lorie, "Long Term Preservation of Digital Information", *JCDL '01: Proceedings of the 1st ACM/IEEE-CS joint conference on Digital libraries*, USA, pp. 346–352 (2001). https://doi.org/10.1145/379437.379726

S. M. Morrissey, "Preserving Software: Motivations, Challenges and Approaches", *Digital Preservation Coalition*, (2020). https://doi.org/10.7207/twgn20-02

E. M. Corrado, "Software Preservation: An Introduction to Issues and Challenges", *Tech. Serv. Q.*, **36**(2), pp. 177–189 (2019). https://doi.org/10.1080/07317131.2019.1584983

B. Matthews, A. Shaon, J. Bicarregui, C. Jones, J. Woodcock, and E. Conway, "Towards a Methodology for Software Preservation", *iPRES 2009: Proceedings of the 6th international conference on the preservation of digital objects*, USA, pp. 132–140 (2009). https://escholarship.org/uc/item/8089m1v1

CEOS.WGISS.DSIG.LSP, Issue 1.1,February 2026, CEOS EO Data Collection Appraisal Procedure, https://ceos.org/document_management/Working_Groups/WGISS/Interest_Groups/Data_Stewardship/Best_Practices/CEOS%20EO%20Data%20Collection%20Appraisal%20Procedure.pdf