HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

HMC PAPER | 3 | Versioning

Guidance on Versioning of Digital Assets

December 2022

HELMHOLTZ METADATA COLLABORATION

Keywords: Versioning, Digital Asset, Reproducibility, Version Control

DOI: 10.3289/HMC_publ_04

Citation: Helmholtz Metadata Collaboration; Guidance on Versioning of Digital Assets, 2022.

Acknowledgement

This publication was supported by the Helmholtz Metadata Collaboration (HMC), an incubator-platform of the Helmholtz Association within the framework of the Information and Data Science strategic initiative.

Call for Review

You are all invited to comment on this version. Please send your feedback by email to info@helmholtz-metadaten.de.

Version: 1.0

This document was generated in a FAIR manner. All previous versions are available on request.

Authors (ORCID)**:** Pier Luigi Buttigieg (0000-0002-4366-3088), Luigia Cristiano (0000-0002-1418-0984), Constanze Curdt (0000-0002-9606-9883), Ahmad Z. Ihsan (0000-0002-1008-4530), Thomas Jejkal (0000-0003-2804-688X), Christian Koch (0000-0002-4344-0837), Oonagh Mannix (0000-0003-0575-2853), Daniel P. Mohr (0000-0002-9382-6586), Anton Pirogov (0000-0002-5077-7497), Karl-Uwe Stucky (0000-0002-0065-0762)

HMC group: Cross-cutting Topic Working Group “FAIR Concepts and Implementation”

Editors: Sören Lorenz, Ants Finke, Christian Langenbach, Klaus Maier-Hein, Stefan Sandfeld, Rainer Stotzka

Contact: HMC Office
GEOMAR Helmholtz Centre for Ocean Research Kiel
Wischhofstr. 1-3
24148 Kiel, GERMANY
E-mail: info@helmholtz-metadaten.de

www.helmholtz-metadaten.de

Content

Abstract

Versioning of data and metadata is a crucial - but often overlooked - topic in scientific work. Using the wrong version of a (meta)data set can lead to drastically difference outcomes in interpretation, and lead to substantial, propagating downstream errors. At the same time, past versions of (meta)data sets are valuable records of the research process which should be preserved for transparency and complete reproducibility. Further, the final version of (meta)data sets may actually include errors that previous versions did not. Thus, careful version control is the foundation for trust in and broad reusability of research and operational (meta)data. This document provides an introduction to the principles of versioning, technical recommendations on how to manage version histories, and discusses some pitfalls and possible solutions. In the first part of this document, we present examples of change processes that require proper management and introduce popular versioning schemes. Finally, the document presents recommended practices for researchers as well as for infrastructure developers.

Intended scope and audience of this document

This document is intended to provide high-level guidance for versioning of data and metadata within the Helmholtz Metadata Collaboration (HMC) and the broader Helmholtz Association of German Research Centres (henceforth, the Helmholtz Association). It addresses stakeholders such as decisionmakers, researchers, as well as providers and developers of tools and other technical infrastructure.

This document aims to 1) raise awareness of the importance of good (meta)data versioning processes and 2) highlight key aspects that should be considered during the choice of (meta)data formats, tools and services.

In relation to metadata (data about data) themselves, this document focuses on data describing versioning processes and the assets they generate (i.e. versioning metadata). The data being versioned - and hence described by the versioning metadata - can include both subject data and its metadata (e.g. contextual, intrinsic to the files being versioned). Comprehensive and quality-controlled versioning metadata is essential to aligning to the FAIR Principles, and HMC's implementation thereof. For general guidance on HMC's interpretation of the FAIR principles, please read the **HMC FAIR interpretations document** [1].

Please note, due to the high variability of requirements and technical constraints across use cases and operational environments, this document cannot provide tailored advice for domain-specific and large-scale data. Therefore, we refer parties which have more bespoke (meta)data versioning needs to their HMC Hub representatives for individualized consultation. In the context of FAIR DOs, this PID Record is used to hold machine-readable information describing what is referenced by a certain PID allowing fast decision-making based on the PID Record before trying to resolve the referenced content. The contents of a PID Record, i.e., which keys are supported, their cardinality and value ranges for each key, are defined by a PID Kernel Information Profile (KIP). These PID KIPs are in the focus of this guidance document. In the following we describe PID KIPs in detail, their characteristics, and the motivation and process towards defining a Helmholtz KIP. At the end, we give a few examples and an outlook how to proceed based on what was presented.

1 Introduction

Research data, publications, software, their associated metadata, and other kinds of digital assets are highly complex products of human labour which are usually developed in an iterative manner. Updates, extensions, revisions, and corrections are frequent and tracking these changes is essential to maintain trust and value of the Helmholtz Association's digital assets.

Thus, a digital "chain of evidence" that records who changed what and why over the life cycle of a digital asset is a key component of good data management. Much like their legal and financial counterparts, such chains must be rigorously preserved and secured to allow detailed inspection of how a digital asset was created and revised. Without this chain, and the transparency it brings, it is far more difficult to assess the trustworthiness of (research) data, especially as projects close and the individuals who generated the data are no longer available.

It is important to treat change (encompassing both the process of change and the documented history of changes) as a first-class citizen when building a FAIR digital research infrastructure to keep data machine-actionable and changes human-understandable. Below, we list some illustrative examples of change processes to clarify their central role in managing data, information, and digitised knowledge. These are:

- releasing a new version of a (meta)dataset or software
- fixing errors in a publication and/or its underlying research (meta)data
- releasing a new version of a published digital artefact in response to feedback
- retracting a published digital artefact when unfixable errors in its content or creation are detected and warning its users of such

Furthermore, change processes also drive the evolution of digital artefact in day-to-day work. For example:

- adding or removing content (such as text or scientific data)
- revising and updating existing content
- combining and integrating partial results from multiple different contributors

The processes of change noted above generally result in the following digital artefacts:

- The old and new version of a digital asset in its entirety
- A difference file (a "diff"), which contains only the lines/elements that have changed between versions of the digital asset
- A "patch", which contains the changes as well as a means to apply them to an older version of a digital asset in order to generate a newer one.
- A "changelog", which describes the changes to a digital asset in human-readable prose.
- An announcement - sometimes official - drawing attention to major or consequential changes (e.g. security vulnerability, major errors). This may be accompanied by a patch which has been marked as urgent or suitable for automatic application by an operating system or package management system.

2 Version tags: what they are and why you need them

A `version tag` is a `unique identifier` within a name space (e.g. that associated to a project, repository, artefact, file) which discriminates between different versions of an entity under version control. Good tagging will prevent confusion (sometimes catastrophic and irreversible) and error propagation. Thus, please consider the following:

- For two publicly released instances of the same artefact, equality of the version tag **MUST** imply equality of the artefact.
    - e.g. software source code is released by two sources which mirror one another. These sources ensure that the tags associated with versions of this source code are equivalent if (and only if) the released source code itself is *identical*.
- Given two artefact instances with different version tags, it **MUST** be possible to decide which version of the artefact is newer without requiring external information (e.g. lookup tables) to perform the comparison. That is, the version tag provides a total order of the instances: no two elements - even when viewed alone - are incomparable. This order **MUST** match the chronological evolution of the artefact.
    - e.g. a user examining only a list of version tags should be able to identify which versions are newer and which are older: following the major.minor.patch syntax, 1.1.0 is newer than 1.0.2
- The version tag **SHOULD** encode high-level semantic information, e.g. indicating the significance of the changes according to agreed-upon domain- and community-specific standards and significance thresholds.
    - e.g. the major.minor.patch syntax (1.2.1 vs 1.2.2 is a smaller unit of change than vs 1.3.0) is used to communicate the degree and consequentiality of change between versions. This relies upon consistent use of semantic conventions: if a major change is communicated as a patch or a minor change, considerable problems can arise downstream.
- The tags following the above principles **MUST** be machine-actionable, meaning that the version tag must consistently follow an agreed-upon standardized scheme.
    - e.g. if using the major.minor.patch syntax with numeric codes, there will be no instance of "1.3.bugFix", which violate the expected syntax and thus compromises machine readability. Note, ensuring and testing machine-actionability of version histories is a safeguard against errors in or misuse of semantic versioning conventions.

3 General recommendations

For users publishing data

1. **Inform yourself about the appropriate or standard community conventions concerning the versioning of the digital artefact that you're working with**. Communities publishing some artefacts (e.g. some ontologies) choose not to use semantic versioning schemes as described above, but opt for a date-based model. Become familiar with the scheme used for the digital artefact in question, consciously decided between alternatives if they exist. Please contact your HMC Hub representative or the **HMC Helpdesk** [2] if this choice is unclear.

2. **Favour archiving your digital artefacts in systems with in-built version control**. This feature should be one of many that would make this system a trustworthy archive (e.g. routines that check for data integrity, guarantees of long-term sustainability, professionally managed servers and services). Avoid ad hoc or project-level archives (i.e. those funded or supported only for the duration of a single project) for anything other than temporary storage. If no suitable system exists or is accessible (e.g. during long field expeditions), make some effort to record that files have changed, preserve important versions (i.e. checkpoints in your work which you may wish to return to) with a version tag and compute a checksum for each of these files. Note that the version control system used by an archive may conflict with that used for the specific type of digital artefact you work with (see point 1.). In this case, attempt to include the artefact-specific tags as additional metadata using qualified keys (in the sense of "key-value pair") such as `hasVersion` [3].

3. **Choose the simplest file format suitable for storing your data**. Version control - especially at higher resolution (e.g. tracking line-by-line changes) - greatly benefits from files being stored in a simple format. The format chosen for archiving and version control should also be as independent from specific software products as possible and follow a widely adopted domain-agnostic standard, if possible. For example, a CSV file is far more suited to fine-grained and informative version control than an XLS(X) file. The latter's complexity and dependency on commercial software make it obscure to most version control systems, and can easily introduce serious errors (see **this article** [4]). Furthermore, this ensures the long-term accessibility and reusability (both generically and in the sense put forth by the FAIR Principles) of the data by decoupling it from the availability of software. Consequently, this often overlaps with recommended formats for archiving, such as e.g. **recommended by RADAR** [5]. Naturally, the simplicity of the format should not compromise the utility of the data, and should support its intended use e.g. appropriate precision for numeric data or image rendering.

4. Good version control practices are important, and sometimes (but rarely) custom schemes are needed; however, they should never be obscure or esoteric: **other people using your data should not be required to understand your version control**

system. To make sense of the differences between multiple versions, you should explicitly mark official "releases" for your data using well-adopted and widely understood conventions, even if versions between the releases are tagged using a custom scheme.

5. Following a version scheme - with its tags and diffs - is powerful, but not sufficient for humans to understand the intent of the changes applied between two versions. Therefore, you should **provide human-readable descriptions about what has changed between successive versions** and associate these with the version history. This is typically done by including a changelog file (e.g. see **keepachangelog.com** [6]) and/or release notes in your project. This may be based on e.g. the history of your version control, which ideally is annotated in a way helping to understand the incremental changes systematically, e.g. by following the **Conventional Commits specification** [7], but should be additionally summarized in prose and in such a way that reduces the amount of information that a person must read to understand relevant changes.

For repository maintainers and software developers

Version control can be built on top of existing archiving solutions by adding or extracting suitable metadata from the archived files. For implementing version control software in a repository system, consider the following guidelines in order to ensure the trustworthiness of the system for prospective users:

1. **An object with a published PID should always remain available because others rely on its existence.** Deletion of such an object may only take place in justified, exceptional cases (e.g. the exposure of classified or secret data, passwords, data protected by copyright) and follow a well-defined and documented process aligned to your institutional policy and national legal frameworks (where extant). As is common for many repositories, a deletion should be obvious to every user and users should be actively and immediately notified. If possible, record deletion should be discouraged entirely; obsolete or erroneous records with published PIDs should rather be marked as deprecated, obsolete, etc. using a semantically qualified field or key (e.g. see this **obsolete ontology term** [8] and note that the term it is replaced by is also included as metadata). The obsolescence of the record should be made obvious to users, e.g. via the labelling of the record or other UI/UX features. Such objects may then be hidden from search under default settings (i.e. make them less findable), but shall remain accessible.

2. Whenever practically feasible (e.g. factoring in memory constraints), **published artefacts with an assigned PID shall be immutable and the whole version history of an artefact should be transparent.** For each artefact, its hosting repository shall provide a landing page from which the latest as well as all previous versions can be accessed.

3. Because frequent or small-scale changes can easily inflate a version history (e.g. with many typo corrections), maintainers and developers should consider installing **services which provide a "preview" mode to inspect drafts prior to commits**. Alternatively, a series of small-scale or incremental commits/changes on a development branch can be "squashed" into one commit to the master or main branch in version control systems such as git. Developers and maintainers should consider such approaches/capabilities in order to support detailed work while simultaneously maintaining uncluttered version histories.

4. Furthermore, **installed systems should make users very aware when they are about to perform any irreversible, final publications/commits** and request confirmation. If possible, mandatory automatic validation (e.g. through **continuous integration using automated unit tests** [9]) ensuring that changes comply to a series of tests or domain-specific community standards shall be performed to reject faulty artefacts and give informative feedback when errors are encountered.

5. **The PID of any digital artefact (i.e. subsuming all of its versions) shall be distinct from, but related to, the PIDs of its individual versions (if these exist).** *If* both the artefact (e.g. a dataset) and some or all of its versions are issued PIDs then: the version-level PIDs and the artefact-level PID shall be connected in both directions by suitable metadata annotations. That is, given an artefact *A* with a PID *P* and versions *A.1*, *A.2*, *A.3* with PIDs *P.1*, *P.2*, and *P.3* respectively, there will be some metadata allowing systems to link P with P.1...P.3.

6. If the implementation/system you are working with issues PIDs intelligible to human users (please ensure this choice was made carefully) then: **PIDs exposed for direct view by humans should also have a clear human-readable syntactic relationship**. The PIDs of versions shall contain the general PID and also provide information about version. For example, a dataset with a PID A25K22 shall have version PIDs as A25K22-vX for version X of the dataset. This should allow a human who knows a versioned PID (A25K22-vX) to easily access the general artefact (A25K22) even without technical mediation (e.g. in the case that the PID trivially maps to an URL).

7. **Each file stored in a repository system must have a checksum to verify its integrity.** Unless there are clear reasons for choosing a different hash sum algorithm, opt for the state-of-the art algorithm (at the time of writing, we recommend the use of SHA-512). Avoid SHA-1 or MD5 as these are known to be cryptographically insecure (i.e. attackers can craft collisions) and thus are not suitable to ensure file integrity. The used hash sum algorithm MUST be stored along with the hash sum in the corresponding metadata record to enable verification. Note, that many off-the-shelf systems (e.g. git) still use **SHA-1** [10] for reasons such as backwards compatibility. While this may be sufficient for git functionality, please do not assume this accomplishes the same level of integrity assurance as a dedicated system based on SHA-512. If you are deploying a brand-new system, consider implementing a version of git with more secure hash sums.

8. **Any version history of a digital artefact should be stored independently from the artefact itself and remain accessible even if the artefact is deleted**. In line with **sub-principle A2** [11] of the FAIR Principles, version histories - as key metadata about a digital artefact - should be preserved independently of that artefact. That is, the version history should be stored in a different file and ideally backed up in a separate location (even if this is not feasible for the primary artefact due to its size, security concerns, or other constraints). If the digital artefact is deleted or not available anymore, the version history (consisting of changelogs, hash sums, etc.) should remain accessible.
9. **Destructive updates (i.e. which modify the existing history, such as forced pushes in git) should be disabled by default**, because they break the interoperability between repositories with modified and unmodified histories. If a destructive update is necessary (e.g. for technical or data security reasons), a maintainer should be contacted, a formal record of that action and the person doing it should inform all users of the repository in an appropriate way for them to take action and update their repositories as well.

4 Proper management of version histories

For version control of small files, especially text-based, we currently recommend using the version control software git. Git emerged from the field of software engineering and is most widely adopted for version control of code. Extensive documentation on using git can be found online, a first introduction can be found e.g. in **this guide** [12].

Using git for versioning of small and text-based files has many advantages. Systems built on top of git, such as the popular code hosting platforms GitHub and GitLab, extend its functionality and make it more accessible. It supports line-by-line comparison between versions of text-based files, as well as the merging and recovery of different versions. Furthermore, it supports collaborative work across distributed teams. Note, with additional extensions, git's functionality extends beyond plain-text files.

Using git or a similar software does not guarantee high standard version history.

The more complex the version management, the more training with such software is required. Below we illustrate this notion with a few useful, but potentially dangerous git functions.

Software like git can make rewriting the version history easy, and therefore it becomes quite easy to damage or corrupt the version history. There are functions for rewriting the version history of a local repository in order to prepare it for integration with a shared repository. These functions are used to streamline and clarify the changes that are to be submitted to a shared repository. However, these functions should not be used (or used with extreme caution) on the shared repository itself (see recommendation 9. in the subsection "For repository maintainers and software developers"): they do indeed rewrite history, which could negatively impact collaboration and change tracking. Only trusted and trained maintainers of shared repositories should consider applying rewrite functions and should follow the guidelines on history rewriting in the section below.

While software like git maintains very granular version histories, it is important to differentiate the internal change management from what will be exposed to external users of the resulting digital artefact. Internally, each individual commit is significant to the development and maintenance of the artefact. External users would not need such detail and should be provided with tagged releases that usually subsume multiple commits. The versioning provided by software like git is for internal use by people working on your artefact. Therefore, you should provide releases of your data with a reasonable version scheme (as discussed above) for people who want to work with your artefact.

Software or data publications based on git(-like) systems should point to a specific tagged and named release provided by a repository (instead of just the commit or the entire repository hosted at e.g. GitHub or GitLab), as this prevents breakage of links in case that a history rewrite in fact did occur.

Versioning of larger data files

Versioning of larger and, especially, non-text-based data is challenging; however, it is possible to keep track of different versions by storing the essential metadata of that data. These metadata include the hash sum (i.e. a fingerprint of the data) alongside the attached version tag.

For version control of moderately large files (currently, in the order of 10-20 GB), git extensions are available. While it is generally not feasible to perform diffing and merging of large data files, git does allow basic version tracking and integrity verification.

There are two major extensions to allow the management of large files by git: Git LFS and git-annex. Git LFS (which is not fully open-source) is supported by GitHub and GitLab, and is easy to set up on both the client and server side. It stores files on a central server alongside one or more git repositories. Alternatively, git-annex is a decentralized, open-source solution that allows large files from distributed sources (i.e. hosted on different servers) to be linked to git repositories. Both extensions vary in many technical details, so the most suitable extension should be selected after assessing the technical requirements of the task at hand.

We cannot give concrete recommendations on versioning of data with a total size beyond the hard drives of a regular computer, but suggest to try applying the guidance in this document as well as in the corresponding **RDA recommentations** [13] as much as it seems feasible and appropriate.

How long should version histories be stored and maintained?

General Federal guidelines for the length of time that (meta)data should be stored do exist. For example, the DFG requires that research data is stored for at least 10 years, to - for example - have a basis to audit research activities if needed. The version histories described above are an integral part of the provenance of these (meta)data and should also be archived, and not simply the final versions of (meta)data. **This also applies to (meta)data connected to "failed" experiments or other research activities**, which are just as valuable as those that are published. Exceptions exist for large files, however, the means to reproduce all artefacts should be available (e.g. source data and the code needed to generate them).

We generally recommend that the version histories and the (meta)data they pertain to are maintained for as long as feasibly possible. If deletion must occur, a documented process of selecting what to delete and why should be undertaken with the conclusions archived with the remaining (meta)data.

Appendix A1: High-level technical notes

Examples of version tags

1. An incremented natural number (e.g.: 1, 2, 3)

This is the minimal reasonable scheme satisfying the outlined properties.

2. A date (+ time) in **ISO 8601 format** (e.g.: 2021-05-21, 2021-06-17T18:30:00+02:00, [14])

This versioning scheme is similar to the natural number, but contains additional machine-readable information about the actual point in time when the version has been published and therefore the 'age' of the version at hand. This may or may not be significant to the user (e.g. older versions of an ontology or standard are more widely supported and implemented), but is equivalent to the natural numbering, from the machine point of view.

The granularity of this timestamp (i.e. whether it should include time) should be reasonably chosen to be proportional to the release frequency, i.e. if multiple versions might be released on a single day, then clearly time must be included, but when the release cycle length is of the order of weeks, then time is not useful information in the version tag. For more information on date-based versioning schemes in practice, see e.g. **CalVer** [15].

3. A nested version number (e.g. 1.0.1, 1.1.3, 2.0.4)

This versioning scheme is universally adopted by software projects and used with various depths (typically using a sequence of 3 numbers). In general, increasing an earlier number in the sequence indicates a more significant change. Whenever a number is increased, all following numbers in the sequence are reset to zero.

> ***Remark:*** *Technically, it is possible to maintain multiple versioning lines, e.g. for software that updated from 1.7.1 to 2.0.0 there might still be updates provided that are versioned 1.7.2, 1.7.3 etc. In this case, the version tag describes not a linear chain anymore, but a tree, and technically breaks the "linear ordering" requirement. This practice is typically used for providing "legacy versions" with updates and there is still a clear linear evolution of the "main branch". It is quite unlikely that this makes sense in other contexts where versioning is important, so we will maintain the assumption of a "linear version history", ignoring this case in the scope of this document.*

If there is no agreement on the thresholds for incrementing certain numbers in such a version tag, then from the machine point of view, this schema is also not more informative than a plain increasing number. Therefore, a nested version number SHALL follow standardized semantics, like **SemVer** [16]. This improves machine-actionability and allows to determine automatically whether a certain version is usable in a specific context.

Removing data from a version-controlled repository

Removing version-controlled content in a non-disruptive way is a complicated and time-consuming task. Avoid adding any data to a version control system that is likely to be unnecessary or temporary.

Completely removing all versions of a file from a version-controlled repository is possible, but not recommended: such removals do not just affect that file, but the history of the entire repository.

Version-controlled repositories are intended to archive and share data with all of their history. However, if removal is (absolutely) necessary (e.g. due to legal, privacy or security issues), please consider the technical guidance provided in Appendix A2.

Appendix A2: Fully removing files from a version-controlled repository

Before removing a file entirely from the history of a repository (i.e. so it "does not leave a trace"), the following broad steps should be considered:

- Think very carefully about what you're about to do!
  - Test the process in a simulated environment / copy of the repository
  - Read the manual of your version control software carefully
  - Define the deletion / removal procedure carefully
  - Create a backup of the repository in case things go wrong
- For a repository storing code: How to fix the code in the end? The answer should be part of the designed procedure!
- Write an email to every person managing a clone of the affected repository noting:
  - Which content should be deleted?
  - Why should this content be deleted?
  - A description of the consequences of deleting and ignoring your request
  - A description of the procedure you have designed
- Speak to every person managing a clone of the affected repository to verify that they have read your email, have understand your email, and have understood the consequences of the removal.
- You should also check in this way if removing file content is the right way.

If you still want to remove the content and all other parties affected consent, perform your removal process. Below, the general process in a git repository is illustrated.

- First of all, request every person managing a clone of the concerned repository deleting clones by writing an email:

    - Repeat the information (which, why, consequences, procedure) from last email (e. g. by replying)
    - request deleting the clones (and backups and other copies)
- Speak again to every person managing a clone of the concerned repository to check if everyone has read your mail, has understand your mail and has done the requested action(s) of deleting.

Assume the file `bad_file` should be removed from the repository `foo`, where `foo` is assumed to be a directory (but could also be a server accessible by ssh, https, etc.):

```
git clone foo bar
cd bar
git filter-branch --index-filter \
    'git rm --cached --ignore-unmatch bad_file' HEAD
cd ..
git clone bar baz
rm -rf foo bar
mv baz foo
```

Repeat this for all necessary branches.

If the repository stores code, it is most likely that the code is not working anymore. As part of the removal process, **the code should be fixed and operational**.

After this process, share the new repository with all parties managing or working with a clone of the affected repository. Ensure that it is back on the server and all parties are informed.

If - and only if - all parties acknowledge that the new repository is functional and they are satisfied with the changes, delete your backup (ideally after a delay of several months).

Note, that if content is removed as shown above:

- There are likely some traces left, which cannot be removed by operations on the repository alone (e.g. snapshots, backups, ...).
- You will not be able to (and SHOULD NOT attempt to) reproduce signed commits. Do not sign commits of others. If your process rewrites signed commits in the version history, add a new commit at the end with a trusted signature. This would verify the data and version history with at least your signature.

Keep in mind that removing content is a complicated and time-consuming task, which is not lossless (e.g. invalidating signatures and commit hashes). Avoid adding any data to a version control system that is likely to be unnecessary or temporary.

Glossary

audit: A methodical examination and review [17]

backup: A copy of data, which may be used to restore the original data to its state at the time the copy was made. The backup of an entire system is a form of snapshot. To ensure the safety of the data in a backup, the backup files should be maintained in systems independent of those where the original data resides, ideally in a separate physical location, and should not change once copied. To preserve data integrity, programmes that validate accurate bit-level copying has taken place should be used. [18]

chain of evidence: A series of events which, when viewed in sequence, account for 1) the actions of an agent or agents during a particular period of time or 2) the location of a piece of evidence during a specified time period. (Adapted from [19])

changelog: A human-readable record of changes made throughout the lifetime of the project. Changelogs are usually very brief, highlighting only the major differences between versions of a digital asset.

checksum: A bit sequence which is computed from the whole content of a digital asset. It should be very sensitive to changes of the content (e.g. single bit change).

checksum representation: A alphanumeric string, which maps bidirectionally to a checksum.

commit: A process during which a set of changes made to a state of a version control repository is transferred, as a unit, into that repository, defining its successive state. The repository can be local or remote, depending on the version control system used.

diff (file): A machine-readable representation of the difference between two files. Typically, a diff represents changes between two versions of the same file in a machine-readable format. Comment: this is distinct from the "diff" program used in Unix systems to detect difference between files.

hash: A bit sequence which is computed from the content of a digital asset. For the same content it always produces the same hash. Comment: A hash can be computed from a part of the digital asset, such as the magic byte header. This is not recommended. Some systems perform file normalisations (e.g. normalising line endings) before computing hashes.

hash function: Not all are secure, or change when the content of a digital asset changes. Cryptographic hash function: A hash function which makes it difficult to create identical hashes for different content of a digital asset. Deriving the content of the digital asset from a cryptographic hash is not possible without a cipher.

hash representation: An alphanumeric string, which maps bidirectionally to a hash.

intermediate files: Files generated as artefacts of a digital process and required for the generation of the intended output of that process. Those files can speed up the recomputation on changes and therefore it is desirable to keep them where the recomputation takes place,

but not necessary (or even harmful, due to their size or compatibility across systems) to include them in the version control.

patch (file): A set of changes - which may be distributed in an executable file or as stand-alone diffs (see "diff") intended to modify a digital asset. Generally, patches are designed to fix, update, or otherwise improve a digital asset such as a program [20]. Comment: this is distinct from the "patch" program used in Unix systems to apply patch files.

release: A collection which packages digital assets for distribution as a whole. Comment: Typically, a release is made to capture the state of digital assets at a certain point in a version history.

release notes: Documentation associated with a release.

snapshot: A record of state of a system at a particular point in time, usually allowing the restoration of that system to that point. Snapshots can be recorded in multiple ways with varying demands on compute resources, and are implementation dependent. (Adapted from [21]) Comment: This term is mostly used in the context of system virtualisation and file systems.

stateful system: A system which is designed to record events which change its components (or their properties) or user interactions (Adapted from [22]).

system state: The contents and properties of a system at a defined timepoint (Adapted from [22])

tag: A string associated to a digital asset, intended for convenience. Comment: Tags are relatively informal labels, used for convenience. They need not be unique or persistent.

version: A state of a digital asset at a certain time. Comment: Versions can be lost or never stored unless they are under version control.

version control: A system for archiving and managing versions of one or more digital assets.

version identifier: A string which follows a scheme to classify versions of a digital asset in a human-readable way.

version history: A record of sequential changes undergone by digital assets as their versions change.

version tag: A tag which is unique associated with a version of a digital asset. Comment: Version tags should be human readable, and clearly communicate the order of versions. If these tags follow a strict model (e.g. semantic versioning conventions), they can also be used by machines.

References

[1] Helmholtz Metadata Collaboration; An interpretation of the FAIR principles to guide implementations in the HMC digital ecosystem, HMC Paper (2022). doi:10.3289/HMC_publ_01.

[2] Helpdesk, *Helmholtz Metadata Collaboration*. https://helmholtz-metadaten.de/en/hmc-helpdesk (accessed December 12, 2022).

[3] "hasVersion", DCMI Metadata Terms, Dublin Core. http://purl.org/dc/terms/hasVersion.

[4] Lewis, D; Autocorrect errors in Excel still creating genomics headache, *Nature* (2021). doi:10.1038/d41586-021-02211-4.

[5] File Formats, RADAR, FIZ Karlsruhe. https://radar.products.fiz-karlsruhe.de/en/radarabout/dateiformate (accessed December 12, 2022).

[6] Keep a Changelog. https://keepachangelog.com/en/1.0.0/ (accessed December 12, 2022).

[7] Conventional Commits 1.0.0. https://www.conventionalcommits.org/en/v1.0.0/ (accessed December 12, 2022).

[8] "obsolete orange juice", The Environment Ontology (ENVO). iri:http://purl.obolibrary.org/obo/ENVO_00000337.

[9] Wikipedia contributors; "Continuous integration", *Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Continuous_integration&oldid=1101336748 (accessed December 12, 2022).

[10] Anderson, T; Git takes baby steps towards swapping out vulnerable SHA-1 hashing algo for SHA-256, *The Register*. https://www.theregister.com/2020/02/05/git_sha_256_work/ (accessed December 12, 2022).

[11] A2: Metadata should be accessible even when the data is no longer available, GoFAIR. https://www.go-fair.org/fair-principles/a2-metadata-accessible-even-data-no-longer-available/ (accessed December 12, 2022).

[12] Dudler, R; git - the simple guide. https://rogerdudler.github.io/git-guide/ (accessed December 12, 2022).

[13] Klump, J, Wyborn, L, Downs, R, Asmi, A, Wu, M, Ryder, G, & Martin, J; Principles and best practices in data versioning for all data sets big and small. Version 1.1. (2020) *Research Data Alliance*. DOI: 10.15497/RDA00042.

[14] Wikipedia contributors; "ISO 8601", *Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=ISO_8601&oldid=1126256903 (accessed December 12, 2022).

[15] Calender Versioning. https://calver.org/ (accessed December 12, 2022).

[16] Semantic Versioning 2.0.0. https://semver.org/ (accessed December 12, 2022).

[17] "Audit", Merriam-Webster.com Dictionary, Merriam-Webster. https://www.merriam-webster.com/dictionary/audit (accessed December 12, 2022).

[18] Wikipedia contributors; "Backup", *Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Backup&oldid=1095660717 (accessed December 12, 2022).

[19] “Chain of Evidence Law and Legal Definition”, US Legal. https://definitions.uslegal.com/c/chain-of-evidence/ (accessed December 12, 2022).

[20] Wikipedia contributors; "Patch (computing)", *Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Patch_(computing)&oldid=1090884314 (accessed December 12, 2022).

[21] Wikipedia contributors; "Snapshot (computer storage)", *Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Snapshot_(computer_storage)&oldid=1060645463 (accessed December 12, 2022).

[22] Wikipedia contributors; "State (computer science)", *Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=State_(computer_science)&oldid=1080643841 (accessed December 12, 2022).