Software Heritage



The Source Code of Science

Archiving, Referencing and Reproducing Research Software

a doctoral course — beyond FAIR data

Roberto Di Cosmo

Software Heritage · Inria · Université Paris Cité

www.dicosmo.org · ORCID 0000-0002-7493-5349

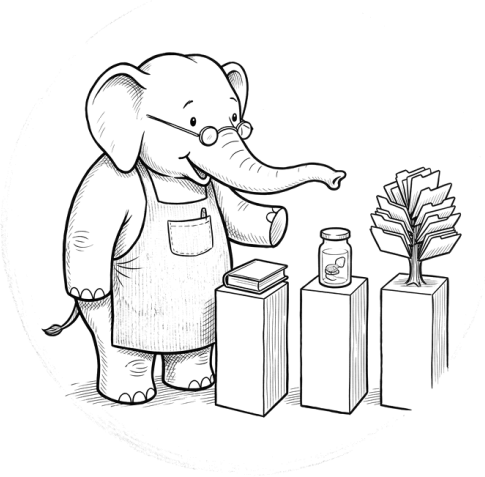
Version 1.6

This work is licensed under a Creative Commons Attribution 4.0 International License (CC-BY-4.0).

September 24, 2026

These notes are revised. The latest version is always at
hal.science/hal-05734828
How to cite them: Appendix I.

About these notes



These notes are a doctoral-level course on managing the *source code* of research: how to archive it for the long term, reference it at the right granularity, describe and cite it, and use it to make computational results reproducible. They are built on Software Heritage[1] and the SWHID — an identifier computed from the code itself, defined in Chapter 4 and now the international standard ISO/IEC 18670:2025. They address the FAIR principles and then go *beyond* them, following the CODE-beyond-FAIR roadmap for research software [39]. The acronyms in this paragraph are all explained, in order, in the chapters that follow; nothing here needs to be known in advance.

These notes are meant to be read *and* done: each practical chapter ends with a **Hands-on** box. Margin notes flag the key takeaways.

In one page (for the busy reader)

The claim. Source code is a first-class research output, but the habit of depositing a `.zip` with a DOI fails it: a DOI certifies a landing page, not the bytes; the deposit can change, has no granularity, and discards history. The fix is to archive the living repository in **Software Heritage** and reference it with a **SWHID** — an identifier computed from the content itself, recomputable and verifiable offline, and the international standard ISO/IEC 18670.
The evidence. The archive holds the development history of more than 29 billion source files from over 440 million projects. The SWHID is already accepted by ACM for the *Artifacts Available* badge; *eLife* has auto-archived every article's code since 2020; HAL, IPOL, swMATH, and Episciences are wired in; UNESCO, EOSC, France's National Plan, and the ANR all point research software at Software Heritage.
What to do. *Researchers*: archive your code, cite it with a SWHID (Chapter 5). *Editors / artifact-evaluation committee (AEC) chairs*: require a SWHID for the *Available* badge — a reviewer can verify it offline (Chapter 6, with a ready call-for-artifacts box). *Funders / policy makers*: name Software Heritage and the SWHID in conditions and policy (Chapter 7, with a model clause and a governance/sustainability note).

[1] In the interest of transparency: the author helped found Software Heritage, co-developed the `biblatex-software` style used here, and contributed to the SWHID standard; the recurring worked example (the Parmap library) is also his. The argument is meant to stand on its evidence, and the sources cited throughout are independent.

Contents

About these notes **2**

Contents **3**

1 Why these notes **5**
1.1 Software as the third pillar of open science 6
1.2 What is at stake 7
1.3 How to read these notes 8

2 Software is not data — and why "beyond FAIR" **9**
2.1 The living nature of software 10
2.2 FAIR for software: fit and friction 10
2.3 The CODE-beyond-FAIR roadmap 11

3 Foundations: forges, the archive, and the data model **13**
3.1 Forges are not archives 14
3.2 The universal archive 14
3.3 The Merkle DAG data model 16

4 Identifiers for digital objects **23**
4.1 Five concerns hiding behind one word 24
4.2 When is one DOI "better" than another? 24
4.3 Intrinsic versus extrinsic identifiers 25
4.4 Who must be trusted, and for what 26
4.5 The SWHID 28

5 ARDC in practice **31**
5.1 Archive 32
5.2 Reference 34
5.3 Describe 35
5.4 Cite and credit 37

6 Reproducibility and artifact evaluation **40**
6.1 The reproducibility crisis 41
6.2 Artifact evaluation and badging 41
6.3 Artifact evaluation with Software Heritage 43
6.4 Double-blind review, and dense pinpointing 44
6.5 Built into the record of science 45

7 Adoption, policy and outlook **46**
7.1 It is already happening 46
7.2 The policy landscape 48
7.3 The engine that carries it out: the OSPO 51
7.4 The wider mission: resilience, sovereignty, transparent AI 51
7.5 A fair question: what about Software Heritage itself? 52
7.6 Call to action 52

A Command cookbook **54**

B SWHID syntax cheat-sheet **56**
B.1 Core identifiers (spec §5) . . . 56
B.2 Qualified identifiers (spec §6) . . . 56
B.3 A worked example . . . 57

C Glossary **58**

D SWHID tool implementations **61**

E Policy reference: the instrument matrix **63**

F Typesetting software citations in venue classes **66**
F.1 ACM: `acmart` . . . 66
F.2 IEEE: `IEEEtran` . . . 67
F.3 Other classes . . . 67

G A model artifact-evaluation section **68**
G.1 The slot that is already there . . . 68
G.2 The common case: archive, then name the version . . . 68
G.3 When review is blind . . . 69
G.4 If you compute it yourself . . . 70
G.5 Model text for a call for artifacts . . . 71
G.6 The author's checklist . . . 72
G.7 For reviewers: there is nothing to check . . . 72
G.8 How it maps onto existing badges . . . 73
G.9 The questions a chair will ask . . . 73

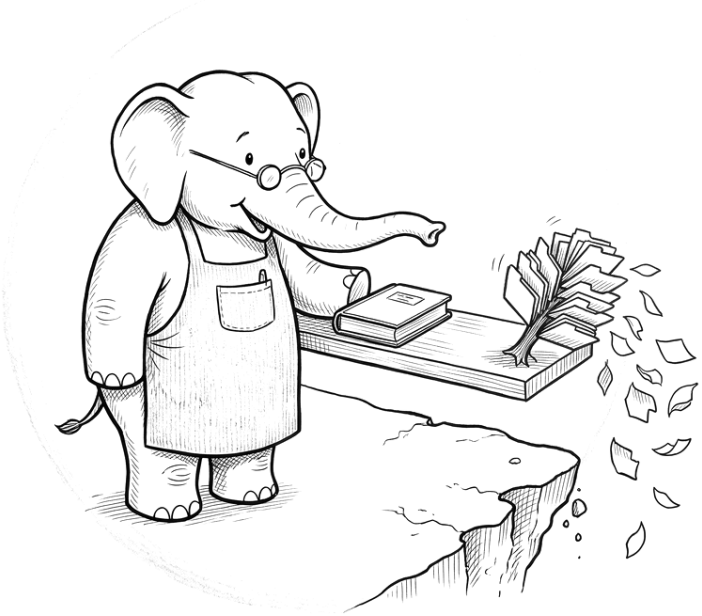
H Archie, and how these illustrations were made **74**
H.1 Why a mascot . . . 74
H.2 Archie, the elephant archivist . . . 74
H.3 How the images were made . . . 75
H.4 What each plate says . . . 76
H.5 What went wrong, and what fixed it . . . 77

I How to cite these notes **78**
I.1 Citing the work . . . 78
I.2 Citing one exact version . . . 78
I.3 Which document is which . . . 79

J How these notes name their own sources **80**
J.1 A key observation . . . 80
J.2 The mechanism . . . 80
J.3 This build . . . 81
J.4 Checking it yourself . . . 81
J.5 What the stamp leaves out . . . 81

References **83**

1 Why these notes



Source code is a research output in its own right, and the one we most often lose: the article is archived, the data deposited, and the code left on a forge that promises nothing. Reproducibility and preservation are only the most visible stakes; from the lone result to the institutional dashboard, every need rests on the same floor — archive, reference, describe, cite.

Learning objectives

- place software as a first-class research output, beside publications and data;
- name the two things at stake — reproducibility and preservation;
- see why a software forge is not an archive.

Software source code is a precious, unique form of knowledge. It can be turned, almost instantly, into something a machine will execute; and yet it is written to be read by people. As Harold Abelson put it in 1985, "programs must be written for people to read, and only incidentally for machines to execute". A program is a human creation in the same sense as an article or a proof — a record of how its authors understood a problem and chose to solve it — and in research it is, more and more, the place where the actual science happens: the model that was simulated, the data that were cleaned, the figure that was produced.

Publications, data, and software: source code is a first-class research output.

And yet, when a published result rests on code, that code is too often the first thing we lose. The article is archived, typeset, given a DOI; the data, increasingly, are deposited in a repository; the software — the one artifact that actually produced the numbers — is left on a personal web page, in a forge that may close next year, or nowhere at all. These notes are about taking that software as seriously as we take the paper, and about the concrete, already-available practice that lets you do so — starting with your next paper.

1.1 Software as the third pillar of open science

Open science rests on three pillars: publications, data, and software. The first two have had their revolutions. Open access freed the article; the FAIR principles and a dense network of repositories gave research data a home, identifiers, and metadata. Software, meanwhile, was treated as a mere means to an end — a tool you used and then forgot — rather than as a research output in its own right.

This is no longer tenable, and the numbers say so. Across *every* discipline, from mathematics to biology, from physics to the social sciences, a growing share of published work now depends on software the authors wrote themselves [30] (Figure 1.1). When the code *is* the experiment, pretending it is a detail does not make it one.

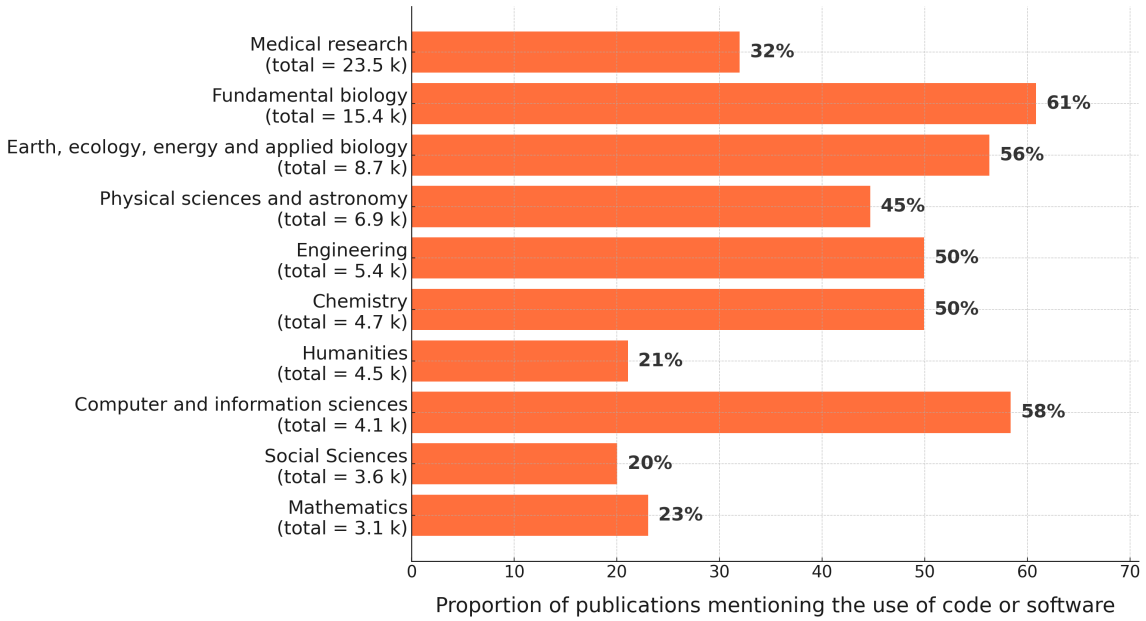


Figure 1.1: Across every discipline, a substantial and growing share of research articles produces or uses software. *Source:* French Open Science Monitor, 2025 edition.

What makes software special is also what makes it easy to mishandle: it is at once executable and human-readable; it *evolves*, often over decades, so that its development history is part of its meaning; and it never stands alone, resting instead on a deep web of other software it depends on (Figure 1.2). These are not the properties of a dataset — and, as we will see in the next chapter, they are exactly the properties that the familiar, data-shaped recipes fail to respect. Software is not data; it is a first-class citizen of the scientific record, and it deserves to be treated as one.

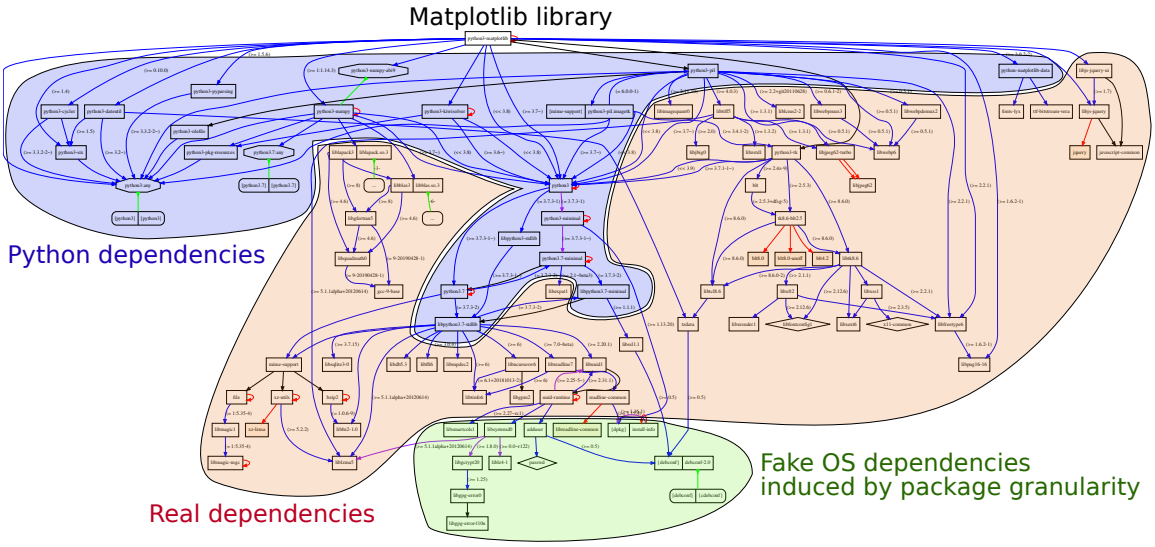


Figure 1.2: No software stands alone. The dependency graph of the `matplotlib` plotting library: blue marks its Python-level dependencies, orange the real system libraries beneath, green the artifacts of packaging granularity [37]. The individual labels are not meant to be read; the point is the *density* — a single research tool already rests on a deep web of other software, none of which the data-deposit model knows how to keep.

1.2 What is at stake

The two most visible stakes are worth naming plainly, first.

The first is *reproducibility*. A computational result is only as solid as our ability to get hold of the exact code that produced it and run it again — not code *like* it, not a later version with the bug quietly fixed, but *that* code, at *that* revision. We will return to the dispiriting state of the art in Chapter 6; for now it is enough to say that, far too often, the code behind a published figure cannot be found, cannot be built, or cannot be told apart from a dozen neighbouring versions.

The second is *preservation*, and here a single slogan does most of the work: *forges are not archives*. GitHub, GitLab, and their kin are wonderful places to *develop* software; they make no promise whatsoever to *keep* it. That promise has been tested, and broken, more than once: Google Code and Gitorious shut down in 2015, putting roughly a million repositories at risk [50, 48]; Bitbucket deleted more than 250,000 Mercurial repositories in 2020 [8]. The half-life of a URL cited in the scholarly literature is about four years [82]. Every one of those losses is a hole in the web of knowledge — a paper that points at code which is no longer there.

Forges are for developing code; they make no promise to keep it.

But reproducibility and preservation are only the most visible of a *plurality* of needs — and those needs stack, scale by scale, each resting on the one beneath it.

A *researcher* needs to archive and reference the exact code behind each result, so it can be retrieved and re-run; to find and reuse the software of others; and — not least — to get *credit* for the software they build themselves, as for any other research output.

A *laboratory or team director* needs to know what the group actually produces: to track its software contributions, fold them into activity reports and a public software page, and not lose them when the student who wrote them moves on.

A *research organization* — a university president, a funding agency — needs to know its *software assets* at all: for technology transfer, for honest impact metrics, for funding strategy, and for the fair career evaluation of the people who write software.

And here is the point that organises this whole note. Every one of these needs, from the lone result to the institutional dashboard, rests on the same minimal foundation — four things you must be able to do to a piece of software: **archive** it (so it can be retrieved), **reference** it exactly (so it can be identified), **describe** it (so it can be discovered and reused), and **cite** it (so its authors are credited). Those four verbs — **A**rchive, **R**eference, **D**escribe, **C**ite & credit, or ARDC — are the floor beneath all the layers. Get them right, and reproducibility, visibility, and assessment all become *possible*; skip them, and none of the higher needs can be met, however much metadata or policy is piled on top.

And yet, collectively, we have not been taking good care of this precious form of knowledge [1]. Closing that gap is the whole purpose of these notes.

From the lone result to the institutional dashboard, every need rests on one floor: ARDC.

1.3 How to read these notes

The good news is that ARDC is not aspirational: the practice exists, the tools are built, and you can adopt them today. It rests on two foundations — the Software Heritage archive [77], which preserves the world's source code with its full development history, and the SWHID, an identifier computed *from the code itself* (and now the international standard ISO/IEC 18670) that lets you point at exactly the code you mean. Chapter 5 makes each of the four verbs hands-on. We will meet the FAIR principles along the way and then, deliberately, go *beyond* them, for the reasons the next chapter makes precise [39].

These notes are meant to be read *and* done. The practical chapters end with **Hands-on** boxes you run on a repository of your choice — your own code, a library you depend on, a result you want to cite. Margin notes flag the one idea worth carrying away from each section; each chapter opens with a short summary and its **learning objectives**; and any unfamiliar term — SWHID, snapshot, forge, qualified identifier — is defined in the Glossary (Appendix C).

You do not need to be a computer scientist to use any of this. Different readers will want different routes through it:

Reading paths

Pick a row and follow it; the rest can wait. The chapters do build on one another — the SWHID (4) rests on the data model (3), and the practical core (5) uses both — so a skipped chapter is there when you need it.

Reader	1	2	3	4	5	6	7	App.
Researcher	●	·	·	◐	●	◐	·	A
Developer / intern	◐	·	·	◐	●	·	·	A
Reviewer / editor / AEC	●	·	·	●	◐	●	·	—
Funder / policy maker	●	·	·	·	·	·	●	E
Full course / student	●	●	●	●	●	●	●	all

● core ◐ skim / optional · skip. **1** Why **2** Beyond FAIR **3** Foundations **4** The SWHID **5** ARDC **6** Reproducibility **7** Policy. App.: **A** Cookbook **E** Policy reference.

If you write or run code for your research — whether a few lines of analysis or a library maintained for decades — these notes are for you. The tools are ready; the habit is the only thing missing. Let us build it, starting with your next paper.

2 Software is not data — and why "beyond FAIR"



Software is not data: it evolves, it comes in a thousand shapes, and it never stands alone. FAIR was written for data, and only its *findable* and *accessible* cross over — *interoperable* and *reusable* already mean something harder to anyone who writes code. What software needs is a progressive roadmap borrowed from free software: CODE, of which only *Open* — publish and archive — is mandatory.

Learning objectives

- explain how software differs from data: executable, evolving, dependency-laden;
- say where the FAIR principles fit software and where their *I* and *R* vocabulary misleads engineers;
- place the CODE roadmap and its single mandatory step.

FAIR is a starting point, not the destination, for living software.

The previous chapter asked us to treat research software as a first-class output. The natural next question is: which recipe do we follow? For data, the community converged on a powerful answer — the FAIR principles [95] — and it is tempting to reach for the same recipe and simply declare software FAIR as well. This chapter argues, plainly, that the temptation should be resisted: not because FAIR is wrong,

but because software is not data, and the parts of FAIR that matter most for software either come for free from archiving or mean something quite different once code is involved.

2.1 The living nature of software

Recall what source code is: knowledge that a machine can execute and a human can read. As Harold Abelson put it, "programs must be written for people to read, and only incidentally for machines to execute"; and as Len Shustek of the Computer History Museum observed, source code offers "a view into the mind of the designer". That double nature already sets software apart from every other research output [1].

Three further properties make software a moving target rather than a frozen artifact.

It *evolves*. A dataset, once collected, is what it is; a piece of software that anyone finds useful is a *living entity*, continuously changing, with no state that can honestly be called final [39]. Its development history is not bookkeeping — it is part of the object's meaning, the record of how the design was reached.

It is *diverse*. Behind the single word "software" hides a startling variety of objects, differing in structure (a one-file script, or a million-line assembly of modules), lifetime (a fire-and-forget analysis, or a library maintained for decades), community (one researcher, one team, a worldwide crowd), size, and language — written in any of the more than nine thousand programming languages humanity has invented [39]. A rule that fits a tidy dataset will not fit all of this at once.

And it never stands *alone*. Every non-trivial program rests on a biotope: a deep stack of other software — libraries, build tools, an operating system beneath it all. We have already seen this (Figure 1.2): one plotting library drags in a whole forest of dependencies. In that biotope, the smallest change in a component buried far below can break the reusability of the code on top — or, worse, silently alter its correctness [39]. Data has no such problem; software has little else.

2.2 FAIR for software: fit and friction

The FAIR principles — *Findable, Accessible, Interoperable, Reusable* — were proposed in 2016 for research *data* [95], and they have been enormously successful there. In a nutshell, FAIR-for-data is about putting good metadata everywhere, which is exactly the right obsession for data. Serious, well-intentioned efforts have since tried to carry the same banner over to software: *Towards FAIR principles for research software* [56], the FAIR4RS principles [11], and FAIR computational workflows [49]. We share their higher-level goal completely; the disagreement is about *how* — and it is not a small one. Even when FAIR is carefully adapted to software, it falls short of the problems one actually faces [39].

The friction has two sources. The first is that "FAIR" has drifted into a buzzword with two faces: a vague, feel-good goal that policymakers invoke without looking inside, and a precise set of recommendations that, for software, mostly reduce to *putting metadata all over the place*. Metadata is necessary, but nowhere near sufficient for code, and it is a trap to let FAIR become the alpha and omega of how we treat software [39].

The second source of friction is sharper, and it is the one experienced developers feel at once. The *I* and the *R* of FAIR — *interoperable* and *reusable* — are not free-floating slogans in our field. They are precise, long-studied technical notions in software engineering: interoperability is about APIs, ABIs, data formats, and protocols; reusability is about modularity, portability, and architecture. These are properties that remain *very hard to achieve even for commercial software built by billion-dollar corporations*. When a researcher who writes code is told to make her software "interoperable and reusable", she hears a decades-old

engineering challenge — not a request to fill in a metadata field. The FAIR vocabulary, borrowed wholesale from data, thus confuses precisely the people best equipped to act.

So what, in FAIR, do we keep? The first two letters. *Findable* and *accessible* are real, achievable, and decisive for software — and, happily, they are exactly what a universal archive and an intrinsic identifier deliver, as the next two chapters show. The rest we should not paper over with a checklist: software interoperability and reusability are a distinct, harder, genuinely engineering problem, and pretending otherwise helps no one.

For software, interoperable *and* reusable *already mean something precise — and it is not what FAIR means by them.*

Objection, and response

"Aren't you dismissing a serious community effort? FAIR4RS deliberately re-interprets each principle for *software."* — Fairly put, and the FAIR4RS work [11, 56] is careful and well-meant; we share its goal of taking research software seriously. The disagreement is narrow and specific: re-interpretation keeps a *vocabulary* (interoperable, reusable) whose established engineering meanings then mislead practitioners, and it leaves the load-bearing problems (preservation, exact identity, provenance) to be solved by something *else* — which is exactly what archiving and intrinsic identifiers are. We are not against FAIR's aims; we are for naming the work by what it actually is. Read the FAIR4RS papers and judge for yourself.

2.3 The CODE-beyond-FAIR roadmap

If imposing data principles on software is the wrong move, what is the right one? Stop legislating from outside, and start learning from the community that has been building reusable software for half a century: free and open-source software. Rather than minting new principles and handing them to volunteers, we should adopt and adapt the philosophy that the FOSS world already lives by [39].

That philosophy has to respect the diversity we just described. A single strict gold standard, proclaimed from on high, would discourage exactly the people we most want to help — the scientist-developers who very often have no formal training in software engineering [18], and who contribute, frequently, on goodwill alone. So the recommendation is built, deliberately, as a *progressive roadmap*: a short list of *mandatory* first steps, then a longer ladder of *recommended* and *optional* ones, climbing toward a gold standard that nobody is required to reach all at once. The whole roadmap fits a four-word mnemonic whose verbs — *Open, Document, Execute, Collaborate* — reshuffle into a word you will not forget: **CODE** [39].

Open — publish the source on a public forge *and* archive it in Software Heritage — the two things the one *mandatory* rung asks of you; then license it openly and claim authorship.

Document — meaningful names and comments first, then API documentation, examples, and notebooks.

Execute — declare your software and hardware dependencies, then ship a container or a Guix recipe, a test suite, and a worked example with its expected results.

Collaborate — handle issues, explain how to contribute, set honest limits, and grow a community.

Above the mandatory rung, most of the work is simply choosing among well-worn tools — none of them required, all of them ordinary. Table 2.1 names the usual suspects for the **Document** and **Execute** rungs; the point is not to use all of it, but to know the menu exists.

***CODE** — Open, Document, Execute, Collaborate — and only* Open *is mandatory.*

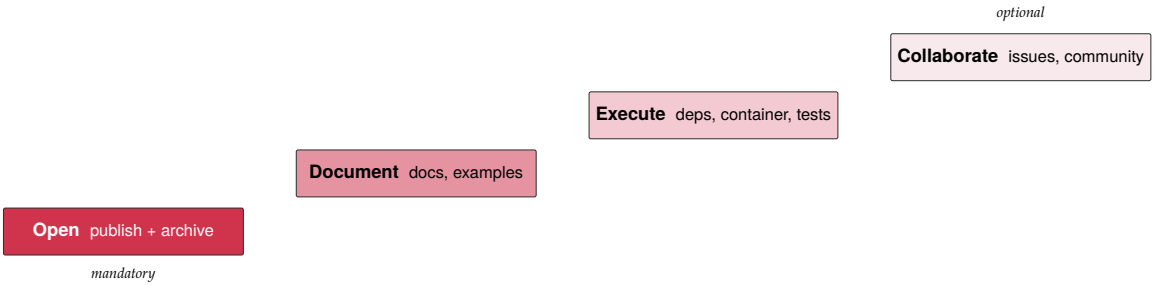


Figure 2.1: The CODE roadmap as a ladder: only the first rung — **Open** (publish the source and archive it in Software Heritage) — is mandatory; **Document**, **Execute**, and **Collaborate** are a progression you climb as far as your project warrants (the fill fades from mandatory to optional).

Concern	**Common tools (pick what fits; none mandatory)**
Version control	`git` — used by more than nine in ten professional developers [83] — with Conventional Commits, semantic versioning, and tagged releases that auto-archive to Zenodo and Software Heritage
Build	Make, CMake, `cargo`, `poetry`, `npm`
Scientific workflows	Snakemake, Nextflow, CWL
Pinned environments	conda, pixi, uv; Nix, Guix; Docker, Apptainer
Continuous integration	GitHub Actions, GitLab CI
Executable papers	Jupyter notebooks served by BinderHub
Reproducible builds	bit-for-bit determinism (`https://reproducible-builds.org/`)

Table 2.1: A sampling of the tooling behind the **Document** and **Execute** rungs of CODE. None of it is mandatory — the only required step remains **Open** (publish and archive) — but knowing which tools exist is half the battle.

Notice where the single hard requirement lands: *publish and archive*. Everything above it is a ladder you climb as far as your project warrants. This is liberating, not lax — it says, in effect, that the worst outcome is not imperfect software but *lost* software, and that “ugly code is always better than no code” [58]. It also tells us where these notes go next. The mandatory rung of **O**pen — archive your code, and be able to point at it — is precisely the **A**rchive and **R**eference of the ARDC practice at the heart of these notes. To make good on it we need two things first: somewhere to keep code for the long term, and a way to name exactly the code we mean. Those are the next two chapters.

3

Foundations: forges, the archive, and the data model



A forge is where code is developed; an archive is where it is kept — and the record of forges shutting down makes the distinction urgent. Software Heritage keeps the world's public source code, with its full history, in one Merkle DAG. That structure does more than store: it gives every file, directory, commit, release, and snapshot an identifier computed from its own content — gratis, verifiable, and needing no middle-man.

Learning objectives

- distinguish forges, distribution platforms, and archives;
- describe what a universal archive of source code collects, and at what scale;
- explain the Merkle-DAG idea and the three properties it buys (gratis, integrity, no middle-man).

A forge is where code is developed; an archive is where it is kept.

Before we can identify a piece of code, we need somewhere that reliably keeps it. This chapter builds that ground in three steps: why the places where we *develop* software are not the places that *preserve* it; what a universal archive of source code is, and how it works; and the simple, powerful data structure — the Merkle DAG — that turns "keeping the code" into "being able to identify it exactly," which is the subject of the next chapter.

3.1 Forges are not archives

Where does the source code behind a paper actually live? In practice, in one of three kinds of place, and it pays to keep them apart. There are *forges* — collaborative development platforms such as GitHub, GitLab, and Bitbucket, built for writing software together, with version control, issues, and pull requests. There are *distribution platforms* — CTAN, CRAN, PyPI, Debian — built to ship ready-to-use packages to users. And there are *archives*, built for one purpose: to keep things, for the long term. Software Heritage is an archive. GitHub is not, and was never meant to be.

This is not a quibble. For thirty years researchers have reached for one of two habits, and both have a hole in them. The older habit, inherited from the 1990s, is to drop a `.zip` on a web page or in a document repository, perhaps with a DOI attached: a poor user experience and — as the next chapter shows — a poor identifier. The newer habit, since the 2000s, is to leave the code in a forge and cite its URL. That is pleasant to use, and offers *no preservation guarantee whatsoever* [1].

How weak is that guarantee? The record is brutal, and it is recent. In 2015, Google Code and Gitorious shut their doors, putting on the order of a *million* repositories at risk and breaking links across the web of scholarship. In 2019, Bitbucket announced the end of Mercurial hosting; in July 2020 it deleted more than *250,000* repositories, research software among them. In 2022, GitLab.com floated a plan to erase any project left inactive for a year. And lest this seem a disease of commercial platforms only, in 2021 Inria unplugged its own ageing forge and broke the build chain of the entire OCaml package ecosystem. Earlier still, the hosting service Code Spaces (not today's GitHub Codespaces) ran happily from 2007 until 2014 — and then, one bad day, was gone, its data unrecoverable. Seven years of "everything is fine" is not preservation.

The deeper reason is that a URL is a promise nobody signed. As the very specification that defined them concedes, "there is no general guarantee that a URL…continues to [point to the same object]" [12]; Tim Berners-Lee's plea that "cool URIs don't change" carries its own sad diagnosis — URIs don't change, *people change them*. Measured rather than asserted, the rot is fast: studying the URLs cited in *Computer* and the *Communications of the ACM*, Spinellis found the half-life of a referenced URL to be about four years [82], and Lawrence and colleagues had found the same fragility throughout the scientific literature [57]. Digital information decays if left untended — the "bit rot" and "digital dark age" that preservationists have warned of for a generation [20, 6] — and source code, with its web of dependencies and its decades-long lives, decays faster than most. Keeping it is a job in its own right [61], and it is not the job a forge ever signed up for.

3.2 The universal archive

If forges will not keep our code, something must. Software Heritage is that something: an international, non-profit, long-term initiative whose mission is to *collect, preserve, and share* the source code of all publicly available software — a modern Library of Alexandria for code, built once and shared by industry, research, culture, and society as a whole [1, 37].

Source files
29,523,977,803

Commits
6,112,189,339

Projects
441,196,029

Directories
23,007,987,747

Authors
107,736,452

Releases
184,798,617

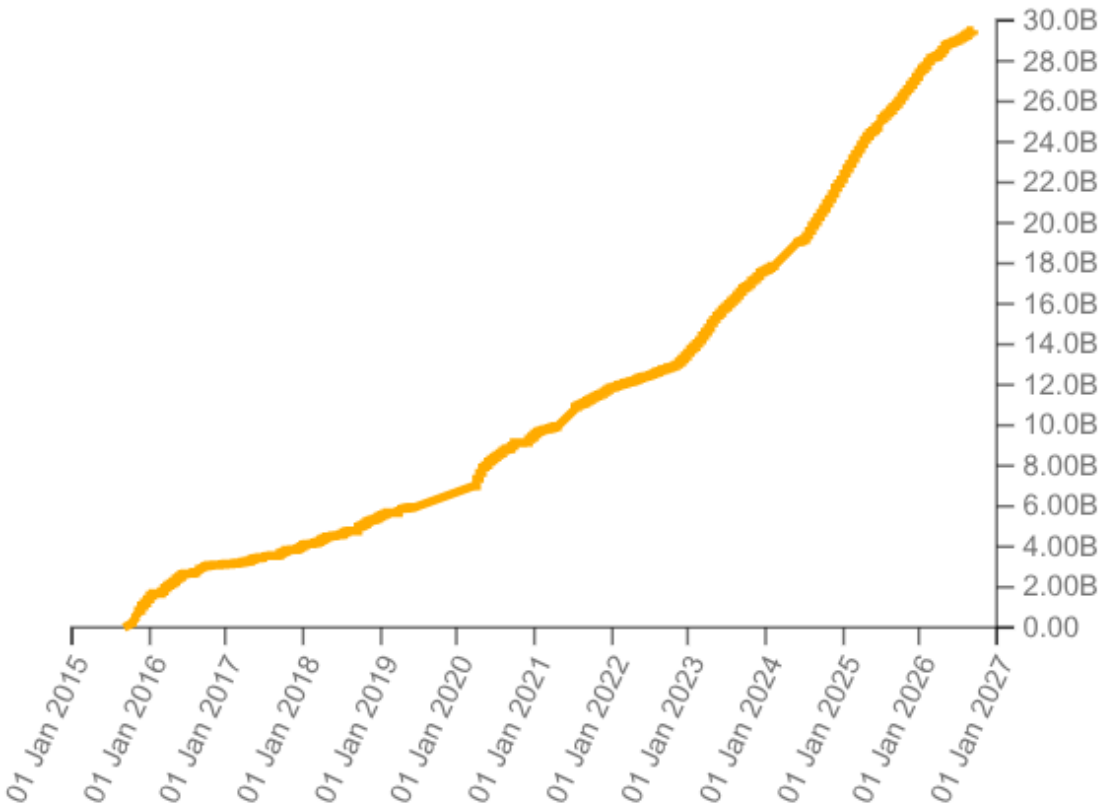


Figure 3.1: The growth of the Software Heritage archive over time, at the 2026-09-08 baseline. The curve traces source files; the other five counters grow alongside it. It holds the development history of hundreds of millions of projects — the largest corpus of source code ever assembled.

It fills itself automatically, by a division of labour worth understanding (Figure 3.2). *Listers* walk a hosting platform and enumerate the repositories (the *origins*) it holds; *loaders* then ingest each origin — its files, its directories, and, crucially, its full commit history — whatever the version-control system (Git, Mercurial, Subversion) or package format (a Debian source package, a PyPI tarball, …), *canonicalizing* them all into the one uniform data model described below. The harvesting runs across more than 5,000 distinct code-hosting and distribution platforms — the big public forges, per-institution GitLab and Gitea servers, Linux-distribution archives, and language package registries. New listers and loaders are contributed by a broad community, with support from the Alfred P. Sloan and NLnet foundations. The result, at the time of writing, is hard to overstate: the development history of more than *29 billion* unique source files from over *440 million* projects, some four petabytes of deduplicated content, organised as roughly sixty billion nodes joined by more than a

trillion edges[1] (Figure 3.1). It is, by a wide margin, the largest body of source code ever gathered in one place.

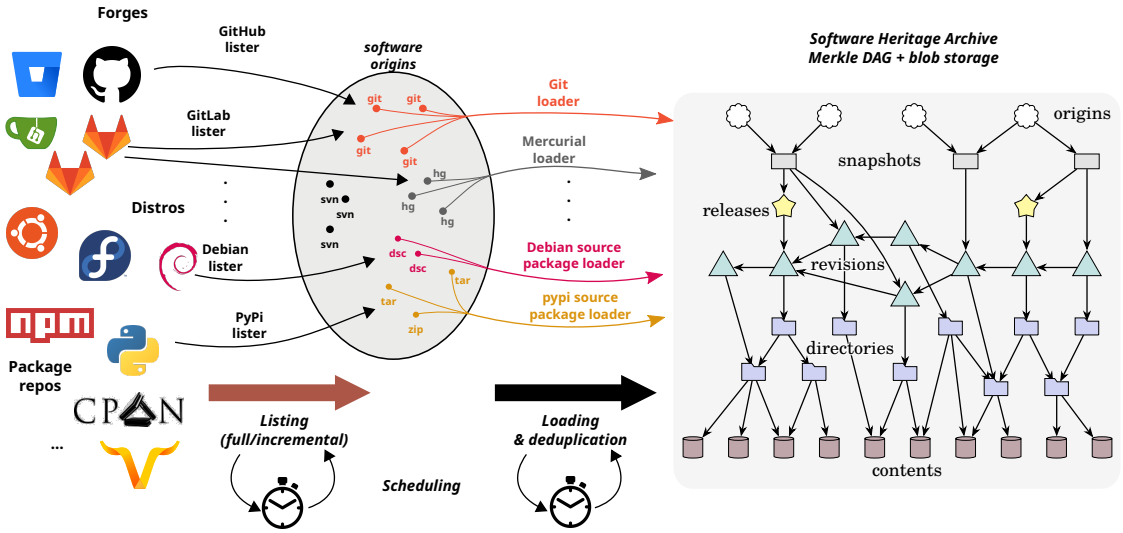


Figure 3.2: How the archive fills itself, and *canonicalizes* as it goes. *Listers* enumerate the repositories (*origins*) held by a hosting platform — forges (GitHub, GitLab, Bitbucket, Gitea), Linux distributions (Debian, Ubuntu, Fedora), and language package registries (PyPI, npm, CPAN, ...), across more than 5,000 such platforms. *Loaders* then ingest each origin, whatever its version-control system (Git, Mercurial, Subversion) or package format (Debian `.dsc`, PyPI tarballs, ...), and convert them all into *one* uniform representation: the Merkle DAG at right, with identical files and directories deduplicated across every origin. Heterogeneous sources in; a single, canonical model out.

The interesting question is not how big the archive is, but how it is *organised* — because the organization is what makes it more than a very large backup.

3.3 The Merkle DAG data model

Archiving the world's code so that anyone can *identify* a precise piece of it calls for a particular kind of identifier. We want identifiers that are *intrinsic* — derived from the content itself, not handed out by a registry — and that work not only for single files but for the *tree structures*, directories-within-directories, that are the natural shape of software. A classical construction gives us exactly this. We meet it first in the abstract, then name the five kinds of object Software Heritage builds from it, and finally watch it at work on a real repository.

What a Merkle DAG is

Ralph Merkle, looking for an efficient way to authenticate many items at once, proposed marrying a tree to a cryptographic hash function: label every leaf with the hash of its data, then label every internal node with the hash of *its children's labels*, repeating all the way to the root [63]. The resulting *Merkle tree* (Figure 3.3) has a remarkable property: the label of the root is a fingerprint of the *entire* structure beneath it. Change a single byte in a single leaf and every label on the path from that leaf to the root changes with it — the root's label simply cannot stay the same unless every byte below it is unchanged.

[1] Software Heritage Graph Dataset, version 2026-06-04: 58,675,740,295 nodes and 1,101,018,460,510 edges, read from `graph.nodes.count.txt` and `graph.edges.count.txt` in `s3://softwareheritage/graph/2026-06-04/compressed`.

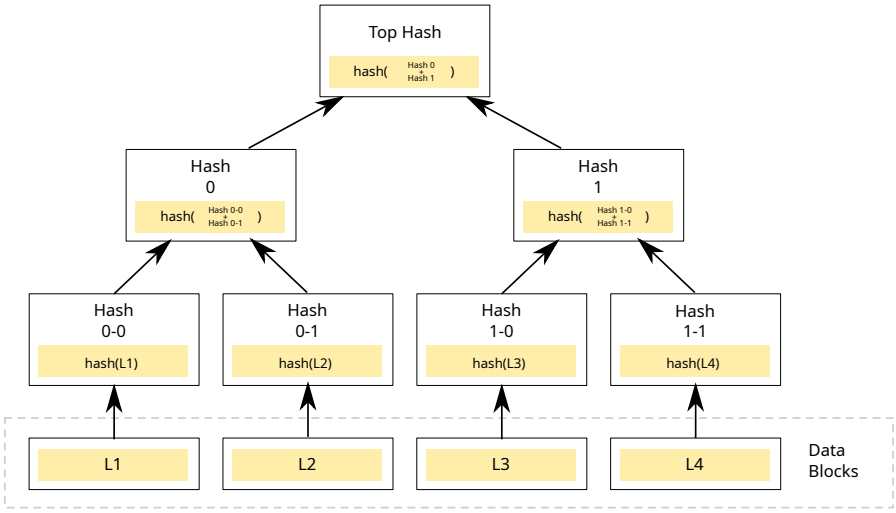


Figure 3.3: A Merkle tree, in Merkle's original construction. Each data block (L_1–L_4) is hashed; each internal node holds the hash of its children's hashes; and the single *Top Hash* fingerprints the entire structure. Change any block and every hash on the path up to the root changes with it. Software Heritage applies the same idea to files and directories — and, by storing identical subtrees once, turns the tree into a DAG.

Software adds one twist a plain tree does not capture: the same file, and the same directory, recur *everywhere* — the identical license text sitting in a thousand projects, the identical sub-library *vendored* — copied wholesale — into hundreds. Because a node's identifier is computed purely from its content, two identical subtrees receive the *same* identifier, so they can be stored once and pointed at from every place they occur. A tree with this sharing is no longer a tree but a *directed acyclic graph*: a Merkle DAG. (It stays acyclic for a simple reason — a node's identifier depends on its children's identifiers, so no node can ever contain itself.)

Gratis, integrity, no middle-man — a Merkle DAG delivers all three at once.

This single idea buys three properties we could not otherwise have together. Such identifiers are *gratis* — you can mint billions of them at no cost; they guarantee *integrity* — change one byte and the identifier changes, so an object cannot be altered without its identifier changing; and they need *no middle-man* — no central authority has to exist for the identifier to mean something. Integrity *and* no middle-man, at once, is exactly the combination a registry cannot give you — which is why the same construction underpins Git, Nix, IPFS, and the blockchains.

The five kinds of node

Software Heritage stores *all* the code it has ever collected, with its full history, in *one* such Merkle DAG [28, 71]. Its nodes come in five kinds. Each is identified by the hash of its own content, and each is, in the main, built out of the kind below it — so the five form a layered structure, contents at the bottom and whole-repository snapshots at the top (Figure 3.4):

content (`cnt`) the raw bytes of a single file — a *blob*, in Git's vocabulary. It is a leaf of the DAG: its identifier is computed directly from its bytes, and it knows nothing of its own filename.

directory (`dir`) a list of named entries, each pointing at a content (a file) or another directory (a subdirectory), together with its permissions. Its identifier is the

hash of that list, so it fixes the *exact* contents and layout of the tree beneath it.

revision (`rev`) a commit: a directory (the snapshot of the source tree at that point) plus metadata — author, date, message, and a link to the parent revision(s). The chain of parents is the project's history.

release (`rel`) a named, often signed, milestone — a tagged revision such as `v2.3.0` — carrying its own author, date, and annotation.

snapshot (`snp`) the complete state of a repository *as archived at one visit*: the full list of its branches and tags and what each pointed to at that moment. It is the entry point from which everything else hangs.

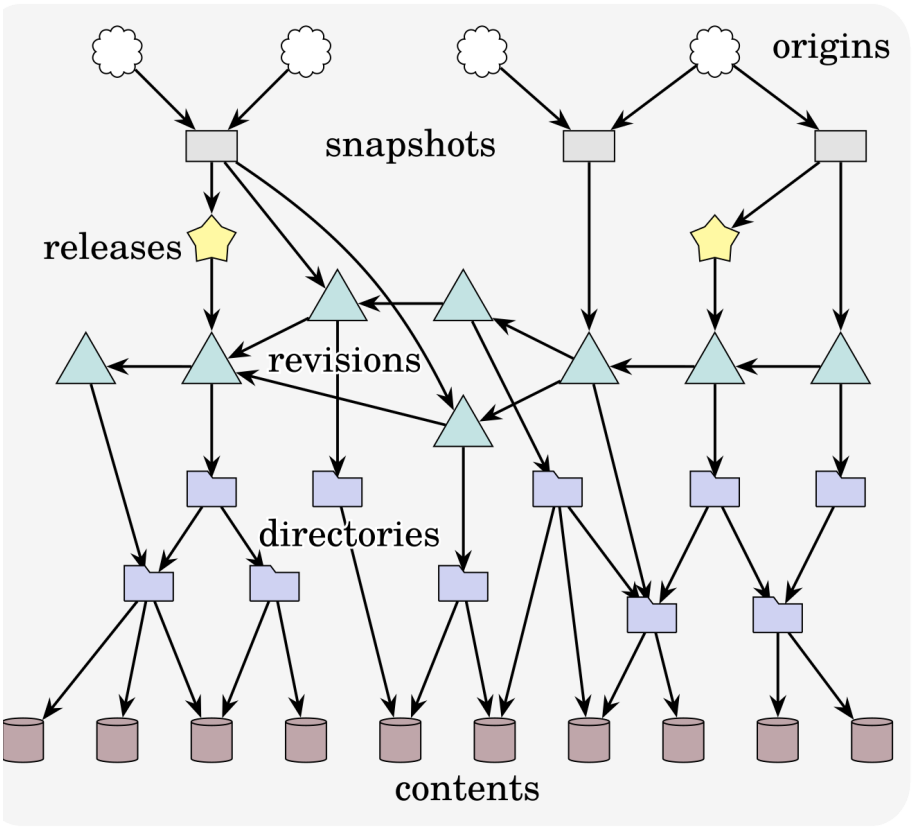


Figure 3.4: The five kinds of node, layered into one Merkle DAG. From the top: *origins* (the repositories crawled) point at *snapshots*, which point at *releases* and *revisions*, which point at *directories*, which point at the *contents* (file blobs) at the bottom. Because every node is identified by the hash of everything beneath it, a node reached from several parents — the same directory or file shared across revisions and across repositories — is stored exactly once.

A worked example: hashing your way up

The five layers are easiest to believe when you watch them being built, bottom to top, on a real repository. The figures that follow do exactly that, and they share one running thread:

the GPL v3 license file, whose Git-style hash `94a9ed02…` you can follow as it is carried upward from layer to layer.

Start with a **content** (Figure 3.5). Feed a file — here the text of the GPL v3 — to a hash function and you get several digests; the one Software Heritage uses as the node's identifier is the Git-style `sha1_git` (in red, `94a9ed02…`). That hash *is* the identifier of this file's content, anywhere it ever appears.

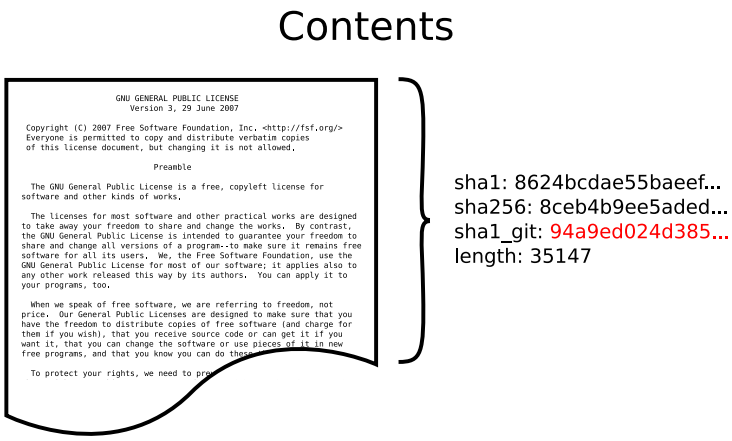


Figure 3.5: A `content` node: the raw bytes of a file (the GPL v3 license) hashed to an identifier. Software Heritage uses the Git-style `sha1_git` (`94a9ed02…`, red) — so this content's SWHID will be `swh:1:cnt:94a9ed02…`, exactly the example met in the next chapter.

Now gather files into a **directory** (Figure 3.6). A directory is just a list: one line per entry, giving its permissions, its type (`blob` for a file, `tree` for a subdirectory), the *identifier of the object it points to*, and the entry's own filename. Notice the `LICENSE` line: it carries precisely the `94a9ed02…` we just computed. Hash this whole listing and you get the directory's identifier, `515f00d4…` — an identifier that pins down every file in the tree and exactly which bytes each contains.

A **revision** (Figure 3.7) wraps a directory in history. Its content is a small record: a `tree` line — here `515f00d4…`, the very directory we just built — a link to the parent revision, the author and committer, and the commit message. Hashing that record gives the commit's identifier, `963634dc…`: change the tree, the message, or the parent, and the commit is a different commit, with a different identifier.

Two further layers sit on top. A **release** (Figure 3.8) is a named, usually annotated and signed, marker attached to one revision — the object behind a tag such as `v0.0.51`. And a **snapshot** (Figure 3.9) records the *whole* state of a repository at the moment it was archived: every branch and tag, and the object each pointed to — exactly what `git show-ref` would print. The snapshot is the root from which a complete archived repository can be recovered.

Stack these five layers and let identical subtrees be shared, and you have the whole picture: *one* Merkle DAG in which contents feed directories, directories feed revisions, and those feed releases and snapshots (Figure 3.10). Because every node is identified by the hash of everything below it, identical files and directories — of which the world's software has a staggering number — are stored exactly once, and the archive as a whole becomes, in effect, a global ledger of public code whose integrity anyone can verify [37].

That is the foundation. We now have somewhere to keep code, and a data model in which every artifact already *has* an identifier, computed from its content. The next chapter

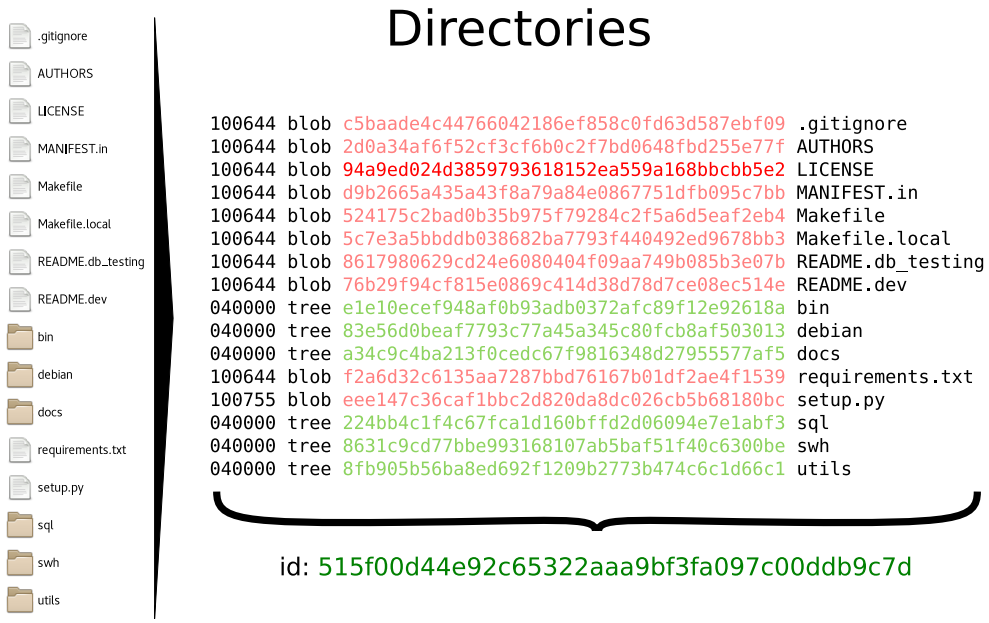


Figure 3.6: A `directory` node: a list of entries, each naming a `content` (file) or a `tree` (subdirectory). The `LICENSE` entry points at the very content of Figure 3.5 (`94a9ed02…`); hashing the whole list yields the directory identifier `515f00d4…`.

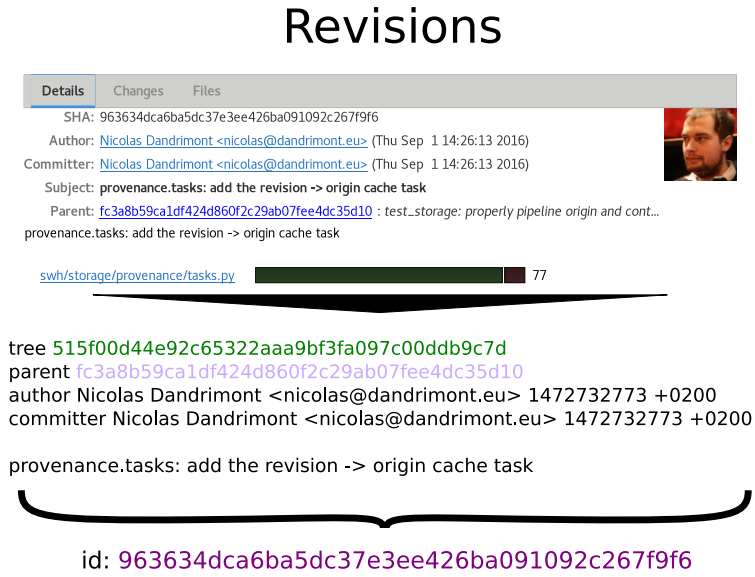


Figure 3.7: A `revision` (commit) node: a `tree` (the directory of Figure 3.6, `515f00d4…`), a parent link, author/committer, and a message — hashed to the revision identifier `963634dc…`.

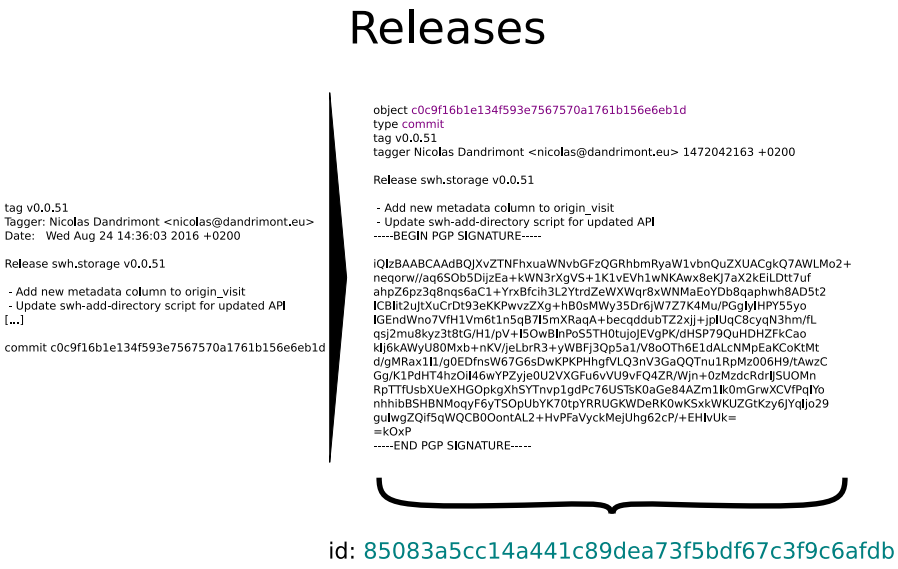


Figure 3.8: A `release` node: an annotated, signed tag (`v0.0.51`) pointing at one revision, with its own tagger, date, and message — hashed to the release identifier `85083a5c`….



Figure 3.9: A `snapshot` node: the full set of a repository's branches and tags and what each pointed to at archival time (the output of `git show-ref`) — hashed to the snapshot identifier `b464cad1`….

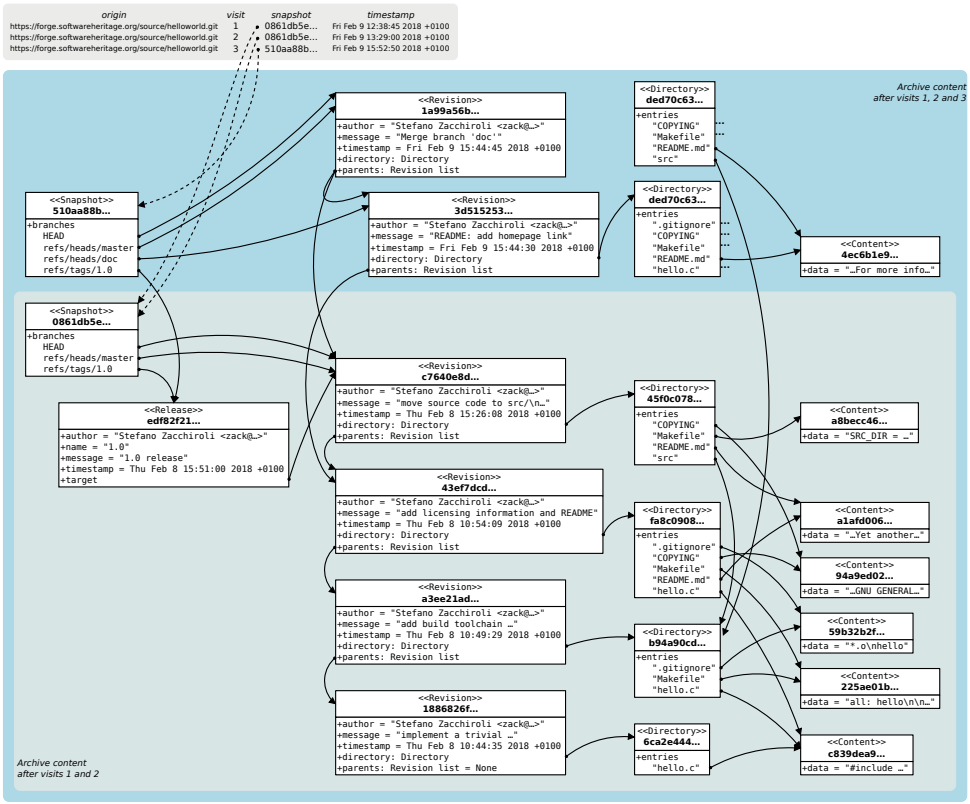


Figure 3.10: The Software Heritage data model: one Merkle directed acyclic graph in which file *contents* (`cnt`) are gathered into *directories* (`dir`), directories into *revisions* (`rev`, i.e. commits), and those into *releases* (`rel`) and *snapshots* (`snp`) — the five layers of the worked example above. Each node's identifier is the hash of everything beneath it; identical subtrees are stored once.

takes that identifier seriously — and shows how it untangles a confusion that has dogged digital identifiers for decades.

4 Identifiers for digital objects



The word "identifier" bundles five different concerns, and almost every argument about DOIs is really an argument about one of the other four. Pull them apart and the point becomes plain: no classical persistent-identifier system lets you check that the bytes you were handed are the bytes that were named. An identifier computed from the content does, and for source code it is the SWHID.

Learning objectives

- separate the five concerns bundled into the word "identifier";
- contrast intrinsic and extrinsic identifiers, and say why it matters;
- read and build a SWHID, in both its core and qualified forms.

The previous chapter left us with a quietly remarkable fact: in the Software Heritage data model, every artifact already *has* a name, computed from its own content. Before we put that name to work, we should be clear about what a name is *for* — because almost everything people believe about digital identifiers is tangled in a confusion that a few minutes of care can dissolve.

4.1 Five concerns hiding behind one word

Identification is one thing; location, metadata, organization, and curation are four others. Keep them apart.

When we say that a digital object "has a DOI", we are quietly running five different concerns together. Pulling them apart is the single most clarifying move in this whole subject:

Identification giving the object a unique name — the identifier proper.

Location a way to *access* or resolve the object. The ISBN names a book; the library catalogue tells you which shelf it sits on. Two different jobs.

Metadata what is *known about* the object — authors, title, date — usually held in a registry.

Organization who keeps the system coherent: the governance, the registry operator, the institutions that keep the lights on.

Curation the *quality* of the metadata — how it is checked and maintained, and who decides which objects deserve metadata, or even a name, at all.

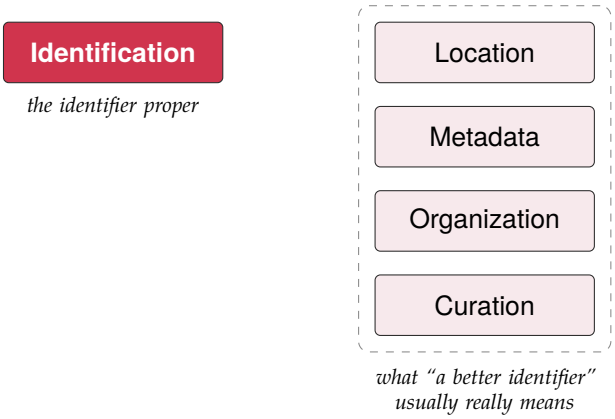


Figure 4.1: The five concerns bundled into "has a DOI". Only the first is identification; the other four are where almost every argument about identifiers actually lives.

Only the first is *identification* (Figure 4.1). The other four are real and important — but they are not the same thing, and most arguments about identifiers turn out to be arguments about one of the latter four, wearing identification's clothes. One example makes this unforgettable.

4.2 When is one DOI "better" than another?

Here is a question that sounds simple and turns out to be a perfect test of the distinction: are the DOIs minted by one registration agency "better" than another's — Crossref's, say, than DataCite's? The honest answer is *yes and no* at the same time, and which half you land on depends entirely on which of our five concerns you have in mind.

At the level of *identification* there is no difference to find. A DOI is a DOI. Crossref and DataCite are both *registration agencies* accredited by the same International DOI Foundation; both mint identifiers resolved by the same underlying Handle System [7, 85], through the same `doi.org`. They are so much one system that when `doi.org` suffered an outage in 2015, it took down *every* agency's DOIs at once. Hand someone a bare DOI and, as a rule, they

cannot even tell you which agency registered it. As *identifiers*, then, they are the same kind of object, minted and resolved in exactly the same way [52]; "better" has nothing to bite on.

And yet the question is not idle, because there *is* a real difference — it just lives in *organization*, *metadata*, and *curation*, not in the identifier. Crossref grew up around the scholarly literature, DataCite around research data and software; the two run different metadata schemas, serve different communities, and accumulate metadata of differing richness. But that richness, as the two organizations themselves put it, "hinges on what depositing members actively provide, rather than being automatically generated by the DOI assignment process itself" [25]. The quality lives in the organization and its curation, not in the identifier. So the answer is yes *and* no — and it sounds paradoxical only because all five concerns were crammed into one word. Pull them apart and the paradox evaporates: as *identification*, no DOI is better than another; as *organization, metadata, and curation*, they can differ a great deal.

4.3 Intrinsic versus extrinsic identifiers

There is a deeper lesson hiding in that example, and Norman Paskin — long the voice of the DOI itself — put his finger on it: "Digital Object Identifier" is best read as *digital identifier of an object*, not *identifier of a digital object* [70]. That distinction is the whole game.

An *extrinsic* identifier — a DOI, a Handle, an ARK — is a digital identifier *of* something, assigned by an organization and recorded in a register. It says nothing about the bytes it names; it is a pointer whose meaning is maintained, on goodwill, by several parties at once. The resolution can be repointed; the content at the far end can change; and there is *no intrinsic way to notice* [35] (Figure 4.2). Anyone who has clicked a DOI and met "not found" has felt the consequence. Such an identifier delegates to an organization not only *location* but *identification* itself: without the register, it means nothing.

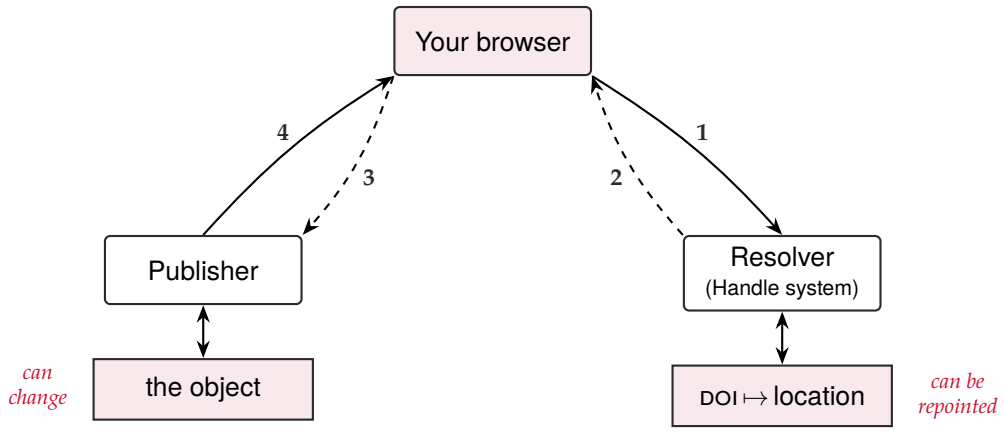


Figure 4.2: How a DOI actually resolves, in four steps. (**1**) the browser sends the identifier to a resolver — the Handle System, the *same* for every DOI, whichever agency minted it; (**2**) the resolver looks the DOI up in its DOI ↦ location register and hands a location back to the browser; (**3**) the browser forwards that to the publisher; (**4**) the publisher returns the object from its own store. The two double-headed links are the lookups that make it work — and they are exactly the two mutable joints, invisible from the identifier itself: the register can be *repointed*, and the object at the far end can *change*. What holds them steady is not the technology but the goodwill of whoever runs each box — the trust surface the next section takes apart.

There is a subtler cost still, and it lands on the very word that sells these identifiers: *persistent*. Persistence is not a property the technology confers. The Handle System is the machinery beneath the DOI, and its own specification concedes as much: resolving a handle "does not guarantee persistence, which is a function of administrative care", and "the persistence of handles depends more on administrative policies than the technology itself" [85]. The "P" in *persistent identifier*, then, is a promise kept by an *organization*, not a guarantee written into the name: a DOI resolves for exactly as long as someone keeps minding the register and paying for the service, and the day that care lapses it is as mortal as any URL. Persistence is administered, never computed.

An *intrinsic* identifier — a Git commit hash, a SWHID — is an identifier *of* a digital object, computed from the object's own bytes. It needs no organization to *mean* something; it needs one only, and separately, to help you *locate* what it names. As we saw in Chapter 3, this is exactly what the Merkle construction buys: identifiers that are gratis, that guarantee *integrity*, and that need *no middle-man* [36]. Map this back onto our five concerns and the elegance is plain: an extrinsic identifier entangles identification with location, metadata, organization, and curation all at once; an intrinsic identifier *separates* identification from all four. That separation is no technicality. It is the difference between a name you must trust someone to keep, and a name anyone can check.

4.4 Who must be trusted, and for what

An intrinsic identifier is computed from the content; an extrinsic one is assigned by a registry.

That last sentence deserves more than rhetoric, because "trust" does a great deal of quiet work in every argument about identifiers and is almost never unpacked. Let us unpack it. Naming a digital object and later getting it back involves four separable acts of faith:

Minting someone must issue the name — and never issue the same one twice.

Resolution someone must keep running the machinery that turns that name into a location.

Integrity someone must ensure the bytes you are handed are the bytes that were named.

Availability someone must still be holding those bytes when you come looking.

Every identifier system is, in the end, a claim about which of these four you may stop worrying about. The useful question to ask of one is therefore not "is it persistent?" but *whom am I trusting, and for which of the four?*

The identifier literature asks this in terms of the *mechanisms* a system offers. Three are common to any system worth the name: *generation* of a label, *assignment* of that label to an object, and *retrieval* of the object from the label. Three more are optional: *verification* — checking that a label and an object actually correspond — *reverse lookup*, recovering the label from an object in hand, and *description*, fetching the object's metadata [35]. Survey the familiar systems against that list, and one row is conspicuously empty (Table 4.1).

Read that table slowly, because it is this chapter in miniature. *No* classical persistent-identifier system provides verification. Not one of them lets you ask, from the identifier alone, the only question a scientific claim really needs answered: *are these the bytes that were named?* The three mechanisms they do provide sit entirely on the minting-and-resolution side of our list. Integrity was never in scope — and, to be fair to them, was never promised. So of the four acts of faith, an extrinsic identifier asks you to make all four, and hands you the means to check none.

Mechanism	Handle	DOI	ARK	PURL
Generation	yes	yes	yes	yes
Assignment	yes	yes	yes	yes
Retrieval	yes	yes	yes	yes
Verification	—	—	—	—
Reverse lookup	—	—	—	—
Description	yes	yes	yes	—

Table 4.1: What the familiar persistent-identifier systems actually offer. Generation, assignment, and retrieval are universal; *verification* — checking that a label and an object correspond — is offered by none of them. A dash marks a mechanism the system does not provide. *Source:* the survey in Di Cosmo, Gruenpeter, and Zacchiroli [35].

Classical PID systems offer generation, assignment, and retrieval — but no verification. *Integrity was never in scope.*

This is not an oversight; it follows from what those systems were built for. A DOI must be able to name a journal article, a museum specimen, a person — things whose "bytes" may not exist at all, and for which an organization really is the only possible guarantor of uniqueness and persistence. That generality is bought at a price, and the price is precisely the property research software needs most. Set the requirements out plainly — identifiers must be *gratis*, because there are billions of objects to name; they must guarantee *integrity*, because reproducibility rests on it; and they must need *no middle man*, because the practice has to outlive any one institution — and the survey behind Table 4.1 could find no existing system offering the last two together [35]. The Merkle construction of Chapter 3 is what finally does, and it does so by striking two of the four acts of faith off the list outright: minting needs no issuer, because the name is computed rather than granted; and integrity needs no guarantor, because the name *is* the check. What remains — resolution and availability — is an archive's job, and, being a job about *keeping* rather than *meaning*, it is one you can hedge: mirror it, replicate it, audit it from the outside.

Abstractions are easy to nod along to, so make that empty row concrete. A dataset is deposited in a repository, given a DOI, and cited in a dozen papers. Some time later — for good reasons or bad; a privacy takedown, a corrected outlier, an inconvenient record quietly dropped — rows are removed and the deposit is updated in place. The DOI still resolves. The landing page still looks right. The title, the authors, the citation count: all unchanged. Every one of those dozen papers still points at "the data", and not one of them now points at the data its results were computed from.

Notice that nothing has been violated. No rule was broken and no system failed; the registry is behaving exactly as designed, because the identifier layer never promised otherwise. That is the whole point: *with only a DOI, there is no way to detect the change at all.* You cannot tell, from the identifier, whether the object behind it is still the one the paper meant — and neither can the reviewer, the reader, or the author herself five years later. The redaction is not so much hidden as *unknowable*.

Now give the same dataset a content-derived identifier, and the edit stops being a matter of trust altogether: the redacted dataset simply has a different hash, and anyone who recomputes it sees the mismatch at once — offline, with no registry to consult and no one's permission to ask. The change may of course be entirely legitimate. What changes is that it becomes *visible*, and so can be discussed. An intrinsic identifier does not prevent redaction; it makes redaction detectable, which is all a scientific record can reasonably ask of one.

A DOI cannot tell you the data were altered. A content hash cannot help but tell you.

Run the experiment once more, in the other direction, on something of your own. Pick the single dataset or program your group would least like to lose — the one a decade of results quietly rests on. Suppose that five years from now

it is gone; or, worse, still sitting exactly where it always sat, but subtly not what it was. What would you need, *today*, to be able to tell the difference *then*?

Not a URL, which will have rotted. Not a folder of files, which cannot testify about itself. What you need is a single name that pins every file in the thing at once — and that is exactly what a *directory* identifier gives you. As Chapter 3 showed, the hash of a directory node is computed from the hashes of its entries, which are computed from theirs, all the way down to the bytes of every file; one short string therefore stands for an entire tree, and nothing can be added, removed, renamed, or altered anywhere beneath it without changing that string. Archive the thing, write down that one identifier, and "is this still what it was?" becomes a computation instead of an argument.

You can go one further step for almost nothing. Keep a *manifest*: a short plain-text file listing, for the handful of artifacts your group genuinely depends on, what each one is and the identifier that pins it. Commit it beside your code, publish it with your data-management plan, attach it to the thesis. It is a few lines of housekeeping, and it converts "we are fairly sure this is the right version" into something a reader, a reviewer, or a successor can check for themselves years from now — without having to reach you, and without having to trust you.

4.5 The SWHID

One directory identifier pins a whole tree; a short manifest of them pins everything your group cannot afford to lose.

The intrinsic identifier for source code is the SWHID — the *Software Hash Identifier* — and it is simply the name of a node in the Merkle DAG of the previous chapter. In its core form it reads `swh:1:`*`type`*`:`*`hash`*, where the type is one of `cnt` (a file's content), `dir` (a directory), `rev` (a revision, i.e. a commit), `rel` (a release), or `snp` (a snapshot of a whole repository). Thus `swh:1:cnt:94a9ed02…` *is* the text of the GPL v3 license, wherever it appears, and `swh:1:rel:22ece559…` is release 2.3.0 of the Darktable project — not a record pointing *at* them, but a name computed *from* them, that you can recompute yourself.

Identification, though, is not location, and the SWHID keeps the two cleanly apart. When you do want to point a reader into the archive — to say not merely *which* code but *where it came from, and in what context* — you add qualifiers: `origin` (the URL the code was harvested from), `visit` (which archived crawl), `anchor` (the node the `path` is rooted at), `path`, and `lines`. The core name still identifies; the qualifiers merely locate. Figure 4.3 lays out a fully qualified SWHID with every part labelled; Appendix B gives the full grammar.

A fully qualified SWHID is precise but long — it can run to several lines, and, dropped raw into a sentence, it overflows the page. The remedy is the oldest trick in hypertext: show the reader the short, readable *core* identifier, and carry the full qualified one — resolver prefix and all — inside the *link*. Hover, and the browser reveals the full target; click, and the archive opens the exact lines. You keep readability in the narrative and full functionality on the click. So this very sentence can name the core mapping routine of the Parmap library by a core SWHID you can actually read, `swh:1:cnt:43a6b232768017b03da934ba22d9cc3f2726a6c5`, while the link behind it carries the full qualified identifier and opens lines 192–228 of `src/parmap.ml` in the archive. In LaTeX this is one `\href` macro; Chapter 5 shows how `biblatex-software` does the same automatically in a bibliography, through a single configuration switch.

None of this is a private convention. The SWHID is an international standard, ISO/IEC 18670:2025, recognized by SPDX, registered with IANA as the `swh:` URI scheme, and recorded as Wikidata property P6138 [54].

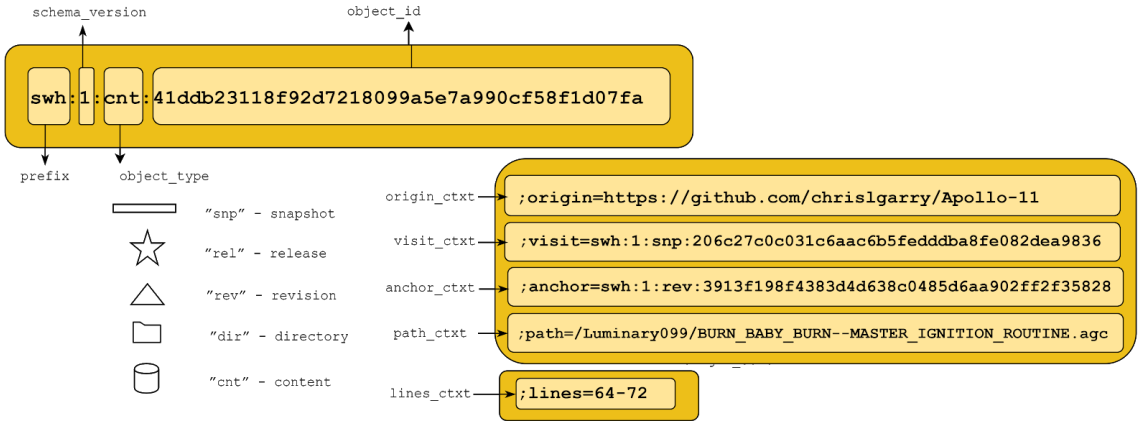


Figure 4.3: Anatomy of a fully qualified SWHID (schema version 1). The *core* identifier (top) is four colon-separated parts: the `swh` prefix, the schema version (`1`), the object type (`cnt` here — one of the five at left: `snp`, `rel`, `rev`, `dir`, `cnt`), and the object's hash (`object_id`). The optional *qualifiers* (right) add context without changing what is identified: `origin` (where the code was harvested), `visit` (which archived snapshot), `anchor` (the node the path is rooted at), `path` (the file within that tree), and `lines` (the span within that file). The example points at lines 64–72 of the Apollo-11 guidance routine `BURN_BABY_BURN`.

Getting a SWHID: retrieve one, or compute one

Two different needs, and it is worth not confusing them. If the code is *already in the archive* and you only want its SWHID, nothing need be installed: the archive's red *Permalinks* tab hands you a fully qualified SWHID for whatever file, directory, or line range you are looking at, and the *updateswh* browser extension will archive a GitHub/GitLab/Bitbucket repository and then do the same. But that is the archive *reporting* an identifier it computed — not a check of your own bytes. To *compute* a SWHID from local files, or to *verify* that a claimed one still matches them — offline, with no archive and no registry in the loop — you need a command-line tool: **Python** (`pip install 'swh.model[cli]'`, the `swh identify` command) or the **Rust reference tool** used in the labs below (`cargo install swhid`). Appendix D sorts the ecosystem by exactly this question — what computes a SWHID, and what is built for something else. *Installing one is your Lab 0*; for a class, a shared container avoids per-machine setup.

Hands-on: watch the name follow the bytes

~10 min · needs any SWHID tool (Appendix D) · in-class, offline

Install the Rust reference implementation of the SWHID standard — `swhid-rs`, released under the MIT license — with `cargo install swhid`.[a]

1. Compute a content SWHID for any file: `swhid content --file file.c`.
2. Change a single byte and recompute — the identifier changes completely.
3. Restore the byte; the original identifier returns, to the character. No registry was ever consulted, and no authority was asked — the name simply follows the bytes.

You can confirm the round trip in one step with `swhid verify file.c swh:1:cnt:…`: it exits cleanly only when the bytes still match the name.

[a] `https://github.com/swhid/swhid-rs`. A pre-built binary for your platform is also available from the project's releases page. The reference implementation tracks ISO/IEC 18670 directly; Appendix D surveys the other tools.

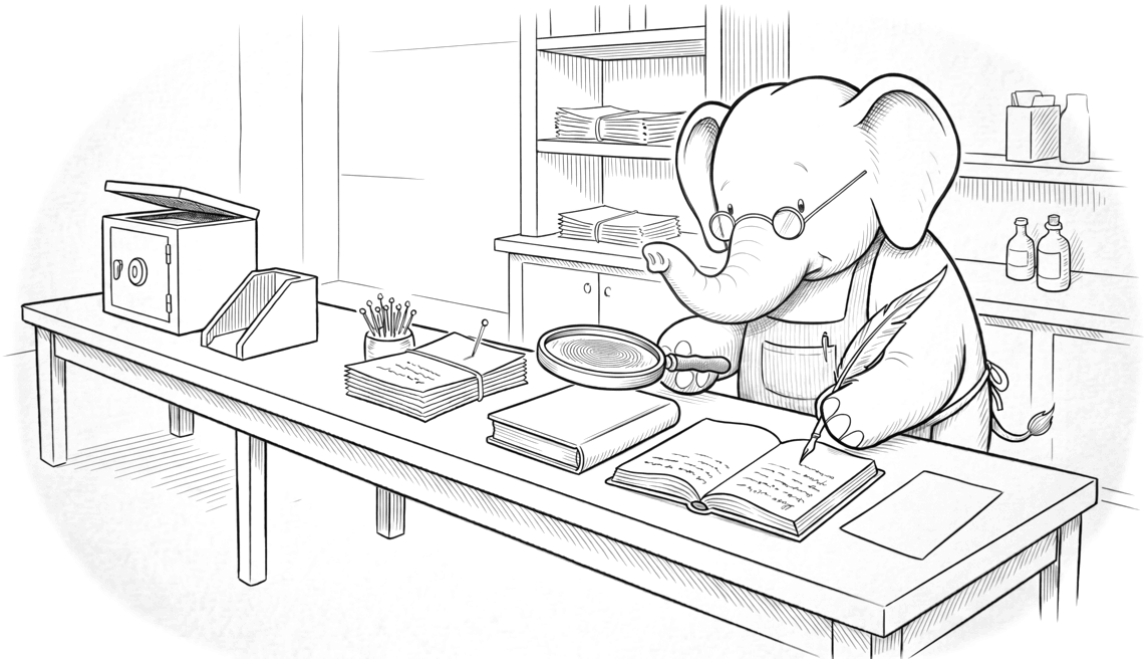
Hands-on: a readable link

~10 min · needs LaTeX and a browser · in-class

Reproduce the split above on a routine of your own: put the full qualified SWHID (resolver prefix, `origin`, `path`, `lines`) in the *target* of a `\href`, and show only the core SWHID as the visible text; click to confirm it opens exactly those lines. Rather than hand-build the `\href` each time, these notes use a one-line `\swhref` helper — visible core, full qualified target — which you will meet again in Chapter 5.

We now have what the rest of these notes need: an intrinsic identifier we can trust — one that identifies code exactly, that anyone can verify without asking permission, and that, when we choose, can also tell us where the code lived. The remaining chapters are about putting that identifier to work: archiving the code, referencing it at the right granularity, describing it, and citing it. That practice is ARDC, and it is next.

5 ARDC in practice



This is the operational heart of the notes. Archiving is one paste of a clone URL, or a webhook that does it for you thereafter; referencing is the craft of matching the identifier to the claim — project, release, revision, or a span of lines. Describing and citing then cost a `codemeta.json`, a license, and a bibliography style that carries the SWHID. Four verbs, adoptable one at a time.

Learning objectives

- archive your code by the channel that fits your situation (forge or tarball);
- reference it at the granularity a claim needs, with a qualified SWHID;
- describe it (`codemeta.json`, `CITATION.cff`, an SPDX license) and cite it with `biblatex-software`.

Archive once; then reference at the granularity your claim needs.

With the archive and the SWHID both in hand, we can finally do the work: keep the source code for the long term, and name exactly the code we mean. This chapter is the operational heart of these notes. It turns the principle "treat software as a first-class output" into four concrete verbs you can apply to a repository of your choice — your own code, a library you depend on, a result you want to cite. The verbs are **A**rchive, **R**eference, **D**escribe, and **C**ite & credit; together they spell ARDC, and each ends with a hands-on lab.

5.1 Archive

If you only do one thing

Go to `https://save.softwareheritage.org`, paste your repository's clone URL, and submit. Your code, with its history, is now preserved and has a permanent SWHID. Everything else in this chapter is worthwhile polish on top of that one irreversible win.

Recall the one hard requirement from the CODE roadmap: *publish and archive*. Publishing on a forge you already do. Archiving means depositing the code, with its history, into Software Heritage, and there are four ways in, in rising order of automation [1].

The simplest is *Save Code Now*: go to `https://save.softwareheritage.org`, paste the clone URL of a public repository, and the archive will visit it — Git, Mercurial, Subversion, no matter. One step better is the *updateswh* browser extension (Figure 5.1), which puts a single "archive this" button on the repository pages of GitHub, GitLab, and Bitbucket.[1] Better still, for code you maintain, is a *webhook*: wire your forge so that every release triggers archival automatically, and you never have to remember again. And there is a fourth channel, the SWORD-based *deposit* (`https://deposit.softwareheritage.org`) — but it is *not* a general path: it is reserved for trusted partners and registered organizations — a journal, an institution, or HAL (France's national open-access repository) — who hold credentials to push a tarball with its metadata straight into the archive. An ordinary researcher does not have those credentials, and does not need them: the easy paths below cover exactly the case the deposit was built to handle.

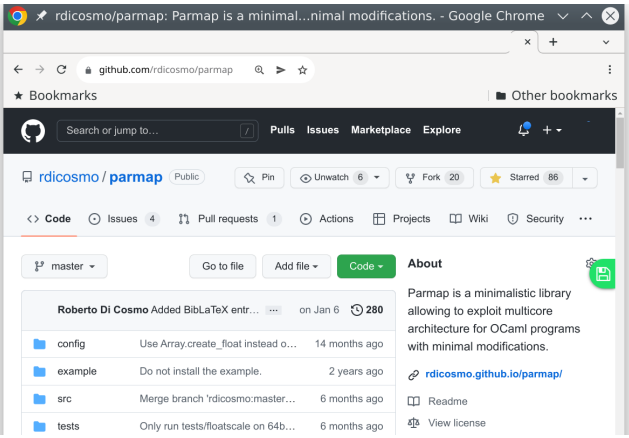


Figure 5.1: The *updateswh* browser extension on a repository page: its button shows the archival status at a glance — already in Software Heritage, or one click away via *Save Code Now*.

When all you have is a tarball

The three public channels above all assume one thing: that your code lives in a forge, behind a clone URL. *And yet* a great deal of research software does not — it is a `.zip` or `.tar.gz` attached to a paper, a snapshot with no public Git remote. For that case, two paths are open to you, and neither needs partner credentials.

The easiest, by far, is to go through *Zenodo*. Since the Zenodo–Software Heritage integration launched in late 2024, software you deposit on Zenodo is *automatically* sent on to Software Heritage, provided the files are publicly accessible; Software Heritage computes

[1] Install links and documentation: `https://www.softwareheritage.org/browser-extensions/`; source at `https://github.com/rdicosmo/updateswh`.

its SWHID, and Zenodo links that SWHID to the record's DOI and shows both on the landing page [79]. So you upload your archive once, and walk away with *both* identifiers — the DOI for discoverability in the scholarly literature, and the SWHID for the verifiable identity of the exact bytes. It is the same division of labour we have insisted on throughout: *a DOI for findability, a SWHID for integrity* — and here you get both, for the price of one upload, with no forge required.

The second path stays inside Software Heritage: *Save Code Now* can also ingest a `.zip` or tarball directly, rather than a clone URL. This is open to any registered account (the account is free; an anonymous visitor can still submit only a version-control URL), but a tarball deposit enters a *review queue* and is archived only once a moderator has approved it — and that queue is not always cleared quickly enough for time-sensitive needs, such as a journal that wants a paper's artifacts archived as soon as it is accepted. Holders of the *ambassador* role bypass the queue, their tarball deposits being ingested directly; but the ambassador role is far more than a sign-up. It is a vetted community programme — ambassadors are volunteers who actively promote and support Software Heritage in their field, and one joins by applying to the programme.[2] So for a one-off the Zenodo route above is simpler and immediate; for archiving tarballs regularly — shepherding a journal's artifacts, say — the ambassador route is worth pursuing.

However the code got in, the result is the same: it is now kept, and it now has a SWHID. The official Software Heritage HOWTO walks through the whole recipe, and its three steps — prepare, save, reference — are exactly the ground the next two sections cover, before we go on to describe and cite:

The official HOWTO

1. **Prepare** your public repository: add a `README`, an `AUTHORS` file, and license information (a `LICENSE` file at the root, or a `LICENSES/` directory).
2. **Save** your code in Software Heritage: manually with the *updateswh* extension; manually via *Save Code Now* (pick the version-control system, enter the repository URL, click submit); or automatically through the API.
3. **Reference** your code: get the SWHID of a full directory, or of a precise code fragment.

Hands-on: archive any repository, and automate it

∼15 min + archival wait · needs a forge account · take-home (archival is asynchronous)

1. Pick a repository worth keeping — your own, or one you depend on. Archive it with the **updateswh** extension or at `https://save.softwareheritage.org`; wait for the visit to reach status `full`.
2. Then make it automatic: add a release *webhook* (GitHub, GitLab, Gitea, Bitbucket, SourceForge) so each new tag is archived without your lifting a finger.

Over time, and when it does not work

For a *maintained* project, make archival a habit, not an event: the release webhook above means every tagged version is preserved as you go, and keeping `codemeta.json` and `CITATION.cff` updated alongside each release keeps your citations current. A few common snags: a visit that stays `pending` or `partial` usually just needs time (large histories take a while) — re-check rather than re-submit; *private* repositories cannot be archived until they are public; a *monorepo* is archived whole, and you reference the sub-directory with a `path` qualifier; and a very large artifact is better archived as its git repository than as a tarball.

[2] Software Heritage Ambassador programme: `https://www.softwareheritage.org/ambassadors/`.

5.2 Reference

Archiving gives the code an identifier; *referencing* is the craft of choosing *which* one to quote, and at what granularity. Here the identifiers of Chapter 4 come into their own, because a paper rarely depends on "the repository" — it depends on something far more specific. Match the identifier to the claim [36]:

the project "Inria created OCaml and Scikit-learn" — name the project (a `snp`, or the origin).

a release "2D Voronoi diagrams were introduced in CGAL 3.1.0" — name the release (`rel`).

an exact state "this result was produced using the exact commit behind it" — name the revision (`rev`).

a code fragment "the core routine `simplemapper` sits at lines 192–228 of `src/parmap.ml` in release 1.1.1" — name the content, with its `path` and `lines`.

For a paper you will usually want a *qualified* SWHID: the core identifier itself, plus the `origin`, `visit`, and `anchor` qualifiers that let a reader walk straight to it in the archive and see where it came from. And when no precise version is known — when you simply want to point at "the project, as the archive saw it" — an origin-qualified SWHID serves as a Wayback-machine-style pointer into the crawl history.

Which SWHID to cite? When in doubt, a directory

To reference a version of the code *as a whole*, Software Heritage recommends a full, qualified *directory* (`dir`) SWHID [80]: print the short core `dir` identifier, and carry the full qualified one in the hyperlink behind it — exactly the split the `\swhref` helper makes. The core `dir` identifier is the most useful one for reproducibility, because it is computed *locally, from the directory's content alone*: it stays the same no matter which release, commit, or repository that directory belongs to, and anyone can recompute it from a copy of the files. A `rev` identifier, by contrast, is tied to one commit's history, and a bare `cnt` names only a single file. So prefer the directory, and add an `anchor` qualifier linking it to a release or revision, so a reader can still recover the version and origin from the archive.

Here is the last rung made concrete — the `simplemapper` routine of the Parmap library, named to the exact line:

```
let simplemapper init finalize ncores' compute opid al collect =
  flush_all ();
  let ln = Array.length al in
  set_ncores (min ln (max 1 ncores'));
  ...                                   (* src/parmap.ml, lines 192-228 *)
```

named by `swh:1:cnt:43a6b232768017b03da934ba22d9cc3f2726a6c5` — the visible text is the short core identifier; the link behind it carries the full qualified SWHID and opens exactly these lines in the archive.

Hands-on: reference code in a paper

~15 min · needs an archived repo · in-class once the artifact is archived

Take a result of your own that rests on code, and build its reference with the Rust reference tool `swhid-rs` (`cargo install swhid`; see Appendix D). Compute the `cnt` SWHID of the precise file — `swhid content --file analysis.py` — and the `rev` SWHID of the exact commit, which is simply `swh:1:rev:` prepended to `git rev-parse HEAD`. Add the `path` and `lines` qualifiers by hand, then paste each qualified SWHID into

`https://archive.softwareheritage.org/` and confirm it highlights exactly the code you meant.

5.3 Describe

An identifier keeps code findable; *description* makes it understandable and reusable, and it is where a little effort pays off enormously. Three files do most of the work, and you should add them in order of growing ambition [15].

Start with a `README`: the human entry point, answering *what, why, install, run, cite, license* in plain words — ten minutes today, and the single highest-value thing you can do. As a project grows, add a `codemeta.json`: a rich, machine-readable description — authors, dependencies, funding, archival identifiers — in a schema.org-based vocabulary that forges and Software Heritage can read to generate citations automatically. Do not write it by hand; use the CodeMeta generator (Figure 5.2).[3] From the CodeMeta file you can in turn generate a `CITATION.cff`, the lightweight format that makes GitHub show a "Cite this repository" button [40].

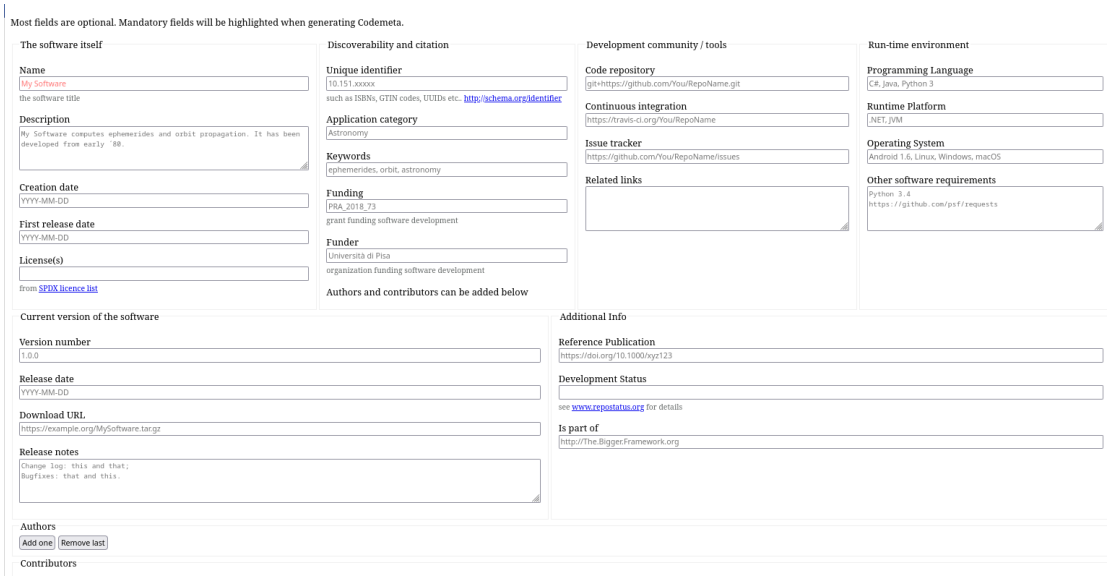


Figure 5.2: The CodeMeta generator: a browser form that writes a `codemeta.json` for you, from fields for the software itself, its discoverability and citation, its development tools, its run-time environment, and its authors — no hand-editing of JSON-LD.

And do not forget the license — the most consequential one-line file in your repository. By the Berne Convention, code is copyrighted the moment it is written, and the default is *all rights reserved*: without a license, nobody may legally reuse your work, however open you meant it to be. Yet roughly *four in five public repositories on GitHub carry no license at all* [9], and research code is no exception: in the corpus of French public-research software archived in Software Heritage, nearly three-quarters of the projects carry no license information whatsoever (Chapter 7). Choose one — permissive (MIT, BSD, Apache-2.0) or copyleft (the GPL family) — declare it machine-readably with an SPDX identifier, and, if you like, automate the bookkeeping with the REUSE specification. (Resist the temptation

[3] `https://codemeta.github.io/codemeta-generator/`; the CodeMeta vocabulary and crosswalk: `https://codemeta.github.io/`.

to reach for Creative Commons: it was not designed for software.) The community has distilled all of this into the FAIR Software Five recommendations.[4]

None of this is much work. Declaring a license is one line at the top of each source file, and a citable record is a few lines of YAML:

```
# SPDX-License-Identifier: MIT      (one line, at the top of each file)

# CITATION.cff
cff-version: 1.2.0
title: My Research Tool
authors:
  - family-names: Curie
    given-names: Marie
version: 1.0.0
license: MIT
```

About four in five public repositories carry no license — so by default nobody may legally reuse them. A one-line SPDX tag fixes it.

To put your software forward as a citable output — onto your CV, into your institution's portal and the literature — one further step helps: reference it in a curated catalogue. In France that means depositing in HAL, which is wired directly into Software Heritage (Figure 5.3); elsewhere, registries such as OSPO-RADAR play the same role.



Figure 5.3: The HAL / Software Heritage synergy: a researcher deposits a curated record of a piece of software in the HAL open-access portal, while the source code itself is archived in Software Heritage — the two linked by the code's SWHID.

Once your software is in HAL it becomes *queryable* as a first-class output. A single call to the HAL API lists every piece of software an entity — a lab, or a person — has produced over a span of years; for all software produced under Inria between 2020 and 2024:

```
https://api.archives-ouvertes.fr/search/?q=*&wt=json
   &fq=docType_s:SOFTWARE&fq=collCode_s:INRIA
   &fq=producedDateY_i:[2020 TO 2024]
```

returns well over a hundred records, each with its SWHID. Narrow to a single author — `authIdHal_s:roberto-di-cosmo` — and you get that researcher's archived software (here the Parmap and OcamlP3l libraries, each carrying the SWHID of its source). The HALtools front-end turns the same query into a ready-made HTML list for a CV or a lab page.[5]

[4] https://fair-software.eu/

[5] HAL API: https://api.archives-ouvertes.fr/; HALtools: https://haltools.archives-ouvertes.fr/.

Hands-on: describe, and watch the citation appear

~20 min + re-archival wait · needs the archived repo · take-home (asynchronous)

Create a `codemeta.json` with the CodeMeta Generator, commit it, and re-archive the repository. Then open the project in the Software Heritage archive and watch the citation panel populate from your metadata — and use it to produce the citation you actually want.

5.4 Cite and credit

Before the mechanics, the principle. The FORCE11 *Software Citation Principles* settled what had long been contested: software that underpins a result is a legitimate, first-class research product, to be cited like any article or dataset, and cited with enough metadata — authors, version, and a *persistent, verifiable* identifier — to find and credit the exact code a claim relied on [76]. That last word, *credit*, is where it gets genuinely hard. A research code is rarely the work of one pair of hands, and the contribution that matters is rarely only the code: testing, documentation, design, and the unglamorous labour of long-term maintenance all earn a place on the author line. *And yet* an attribution mined automatically from the `git` history sees only commits, and quietly writes the non-coding contributors out of the record [4]. Crediting a collective — "The Givaro group" rather than a roster of names — keeps the record honest about a team effort, but at the cost of the individual recognition on which academic careers still turn. No bibliography style can resolve that tension; what a good one *can* do is carry the version, the license, and the verifiable identifier on every reference, and leave the human judgement of credit to people.

So, finally, the mechanics. Cite software as you would cite anything else in research: in the bibliography, as a first-class reference. For decades there was no proper way to do this, and software citations were forced into `@misc` entries that lost the version, the license, and any verifiable identifier [4]. The `biblatex-software` style fixes that [27, 29]. It adds four entry types — `@software` for a project, `@softwareversion` for a release, `@softwaremodule` for a component, and `@codefragment` for a precise span of code — each carrying a `swhid=` field. The power comes from `crossref`: shared metadata (authors, repository, license) is written *once* on the parent `@software` entry, and each version, fragment, or snapshot adds only what is specific to it. So citing several versions and several fragments of the *same* project costs almost nothing:

```
@software{parmap,                       % the project: shared metadata, written once
  title   = {The Parmap library},
  author  = {Di Cosmo, Roberto and Danelutto, Marco},
  url     = {https://github.com/rdicosmo/parmap},
  license = {LGPL-2.0}}

@softwareversion{parmap-1.1.1, crossref={parmap},          % a released VERSION
  version = {1.1.1}, date = {2020},
  swhid   = {swh:1:rel:373e2604d96de4ab1d505190b654c5c4045db773;
             origin=https://github.com/rdicosmo/parmap}}

@codefragment{simplemapper, crossref={parmap-1.1.1},       % a code FRAGMENT
  subtitle = {Core mapping routine},
  swhid    = {swh:1:cnt:43a6b232768017b03da934ba22d9cc3f2726a6c5;
              path=/src/parmap.ml;lines=192-228}}

@softwareversion{parmap-dev, crossref={parmap},            % a whole-repo SNAPSHOT
  version = {dev snapshot 2017-05},
  swhid   = {swh:1:snp:2a6c348c53eb77d458f24c9cbcecaf92e3c45615;
             origin=https://github.com/rdicosmo/parmap}}
```

In the text you then write `\cite{parmap-1.1.1}` for the release, `\cite{simplemapper}` for the routine, or `\cite{parmap-dev}` for the archived snapshot — and the bibliography prints the shared metadata (authors, repository, license) *once* on the project entry, then gives each version, fragment, and snapshot only its own specifics and a live back-reference to its parent, with the right SWHID type (`rel`, `cnt`, `snp`) rendered automatically. These very notes are typeset that way: in the References, the Parmap release and the `simplemapper` fragment each point back to the one shared Parmap record — *of* it, *from* it — rather than repeating its authors and license, and the CGAL module is likewise printed as *part of* its release.
The fourth entry type, `@softwaremodule`, names a *component* of a larger project — a distinct, often separately-authored part, perhaps added only at a certain version. Parmap is too small to want one; a deliberately *modular* library like CGAL [88] is the natural case. Its *2D Voronoi Diagram Adaptor*, written by Menelaos Karavelas and first shipped in CGAL 3.1 (here in release 5.0.2 [89]), is exactly such a module [55]:

```
@software{cgal,                                      % the modular PROJECT
  title  = {The Computational Geometry Algorithms Library},
  author = {{The CGAL Project}}, url = {https://cgal.org/}}

@softwareversion{cgal-5-0-2, crossref={cgal},      % a released VERSION
  version = {5.0.2}, date = {2020},
  swhid   = {swh:1:rel:636541bbf6c77863908eae744610a3d91fa58855;
             origin=https://github.com/CGAL/cgal}}

@softwaremodule{cgal-voronoi, crossref={cgal-5-0-2},  % a MODULE
  author       = {Karavelas, Menelaos},
  subtitle     = {2D Voronoi Diagram Adaptor},
  license      = {GPL}, introducedin = {CGAL 3.1}}
```

The module carries its own author and the version that *introduced* it, and inherits the project's title and the release's SWHID through `crossref` — so all four types (project, version, module, fragment) are now in play, each pinned by the identifier it deserves.
The package is on CTAN and in TeX Live, and has shipped in the ACM `acmart` class since 2022, so the SWHID renders natively in an ACM paper. It also solves, for the bibliography, the very readability problem of Chapter 4: a single configuration switch, `shortswhid`, chooses whether the *full* qualified SWHID is printed or only the short *core* one — and in both cases the complete identifier stays live in the hyperlink (Figure 5.4).

The payoff is that a software citation becomes *reproducible*: a SWHID in the bibliography resolves to the exact bytes you used, with no dependence on a URL that may not outlive the paper. These notes' own bibliography is built this way — the Parmap library, for instance, is cited as a versioned software entry carrying its SWHID, and a routine within it as a code fragment.[6]

the bibliography is
hat still resolves
URL is gone.

Hands-on: cite software with biblatex-software

~20 min · needs LaTeX with biber · in-class

1. Write a `@softwareversion` entry for the code you archived, with its `swhid=` field; add a `@codefragment` child via `crossref` for a specific routine.
2. Produce the variants — a project-level `@software`, a `@softwaremodule` — and compare how each renders.
3. Drop the entries into a real venue template — `acmart` or `IEEEtran` — and confirm the SWHID renders there too; Appendix F gives the exact preamble for each.
4. Flip the option both ways — `\ExecuteBibliographyOptions{shortswhid=true}` vs `false` — and rebuild: watch the printed identifier shrink to its core while the link stays whole (Figure 5.4).

[6] See the `biblatex-software` entries [33, 34] in the references.

shortswhid = false *the full qualified SWHID is printed*

[SW Rel.] Roberto Di Cosmo and Marco Danelutto, *The Parmap library* version 1.1.1, 2020. Inria, University of Paris, and University of Pisa. LIC: LGPL-2.0. URL: https://rdicosmo.github.io/parmap/, VCS: https://github.com/rdicosmo/parmap, SWHID: ⟨swh:1:rel:373e2604d96de4ab1d505190b654c5c4045db773;origin=https://github.com/rdicosmo/parmap;visit=swh:1:snp:2a6c348c53eb77d458f24c9cbcecaf92e3c45615⟩.

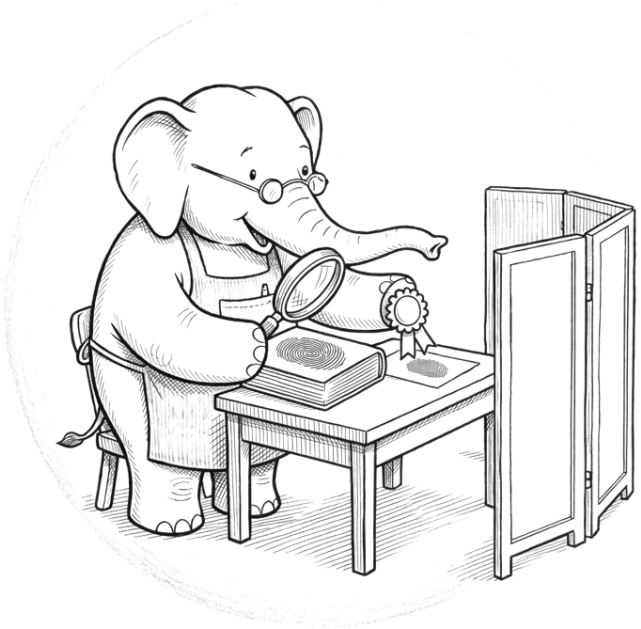
shortswhid = true *only the core is shown — the link is unchanged*

[SW Rel.] Roberto Di Cosmo and Marco Danelutto, *The Parmap library* version 1.1.1, 2020. Inria, University of Paris, and University of Pisa. LIC: LGPL-2.0. URL: https://rdicosmo.github.io/parmap/, VCS: https://github.com/rdicosmo/parmap, SWHID: ⟨swh:1:rel:373e2604d96de4ab1d505190b654c5c4045db773⟩.

Figure 5.4: Real `biblatex-software` output for the same Parmap release with `shortswhid=false` (top) and `shortswhid=true` (bottom). Both encode the *same* fully qualified identifier: the switch changes only what is *shown*, never what the SWHID resolves to — the top prints the qualifiers (`;origin=…;visit=…`) in full, the bottom keeps just the readable core. In the note's own References — and in any document built with this style — each SWHID is a live link to the archive.

That is ARDC: archive the code, reference it precisely, describe it, and cite it — a practice you can adopt incrementally, one verb at a time. The remaining question is what all this rigour *buys* us. The answer, and the sharpest test of the whole approach, is reproducibility and the evaluation of research artifacts: the subject of the next chapter.

6 Reproducibility and artifact evaluation



Artifact evaluation is the exam this practice was preparing for, and it now runs beyond computer science. The *Artifacts Available* badge asks for a durable archive and a unique identifier — exactly what archiving and a SWHID deliver — but where a DOI gives a reviewer a link whose persistence is a function of administrative care by third parties, a SWHID lets her recompute the name from the bytes and check it, offline. A directory SWHID settles double-blind review as well.

Learning objectives

- relate artifact evaluation and the ACM badges to Software Heritage;
- verify an artifact's SWHID offline, as a reviewer would;
- handle a double-blind submission with a directory SWHID.

Artifact evaluation is the question this whole note was answering.

How much of computational science can actually be re-run? It is an uncomfortable question, and the honest answers are sobering — which is why our community has built a process to force the issue: *artifact evaluation*, now standard at computer-science conferences, checks that the software behind a paper actually exists, runs, and supports its claims. It is the sharpest test of everything these

notes have argued for, and it turns out that Software Heritage answers its central need almost perfectly. ARDC, in other words, was the preparation; this chapter is the exam.

6.1 The reproducibility crisis

Begin with the finding nobody likes to quote. When Collberg and Proebsting went looking for the code behind several hundred papers from top computer-systems conferences, they found that, for a large majority, the code could not be obtained, or could not be built [23]. Not refuted — simply *unavailable*, or unbuildable, and so impossible to check at all. You cannot reproduce what you cannot run; and, as the free-software tradition has always insisted, you cannot truly inspect what you cannot read. A result resting on code is only as solid as our ability to get *that* code, at *that* version, and run it again. A SWHID secures the first half of that sentence — the *identity* of the source, exactly; getting the *same build* out of it, bit for bit, is a separate (and harder) problem, the province of reproducible-build systems such as Guix and Nix.[1] The two are complementary: an intrinsic identifier pins what you built from, a reproducible build pins what you get. The whole apparatus of artifact evaluation exists to push back on both fronts.

6.2 Artifact evaluation and badging

Artifact evaluation began, fittingly, as a contest: at ESEC/FSE 2011, where the first prize went to Vouillon and Di Cosmo. What started as an experiment is now a near-universal standard, codified by the ACM *Artifact Review and Badging* policy [2], which defines three independent families of badge (Figure 6.1):

Artifacts Available the artifacts are placed in a publicly accessible archival repository, with a unique identifier.

Artifacts Evaluated the artifacts are documented, consistent, complete, and exercisable (*Functional*) — and, a notch higher, structured carefully for reuse (*Reusable*).

Results Reproduced the paper's results were independently obtained by others, with the authors' artifacts (*Reproduced*) or without them (*Replicated*).

The practice has swept the field — programming languages, software engineering, systems, security, databases, high-performance computing — to the point of routine: at EuroSys 2025 some 93% of papers earned the *Functional* badge, and nearly half pursued *Results Reproduced*. Some communities go further still: in computer graphics, the *Graphics Replicability Stamp* awards a public seal only to papers whose code has been independently built and run to reproduce the results — hundreds of papers now carry it (Figure 6.2).[2]

Artifact evaluation beyond computer science

And the idea no longer stops at computer science. The same uncomfortable question — *can the result actually be re-run?* — is now being put, in each discipline's own idiom, far outside our field; and, tellingly, the answers are hardening into standing institutions rather than one-off favours to a reviewer.

[1] https://reproducible-builds.org/.
[2] http://www.replicabilitystamp.org/.

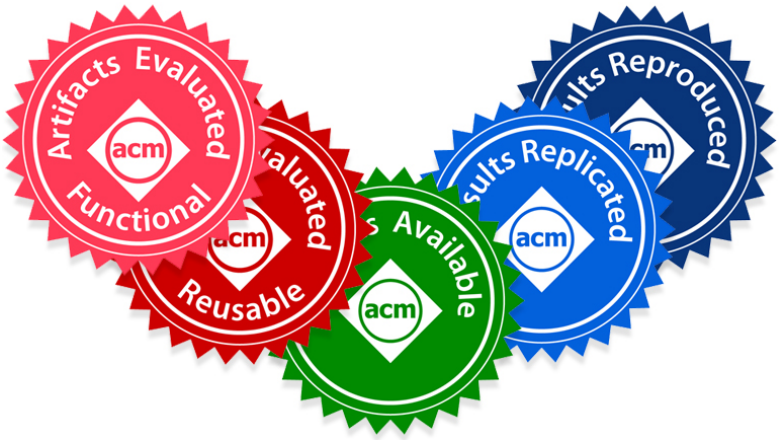


Figure 6.1: The ACM *Artifact Review and Badging* badges. The *Artifacts Available* badge — a public archival repository plus a unique identifier — is the one Software Heritage satisfies most literally.

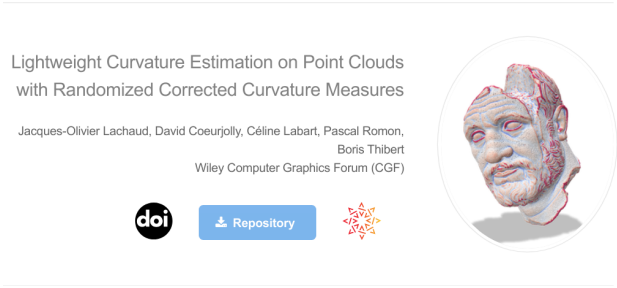


Figure 6.2: The Graphics Replicability Stamp: an independent reviewer rebuilds and runs a paper's code, and only on success is the paper awarded the public stamp — reproducibility certified by re-execution, not by author assertion.

Economics moved first, and moved hardest. Since 2019 the American Economic Association has subjected *every* empirical paper accepted at its journals to a mandatory code-and-data audit, run before publication by a dedicated Data Editor — not a badge an author may opt into, but a gate the paper must pass [5]. Beside that public mandate sits a non-profit service, *cascad*, which independently re-executes the code behind a study and issues a reproducibility certificate; it has signed off on more than 170 of them to date [19].

Other fields have built instruments of their own. In the life sciences and computational geography, *CodeCheck* pairs each submission with an independent *codechecker* who re-runs the analysis and issues a citable certificate, in partnership with venues such as *GigaScience* and the Reproducible AGILE conference [41]. Image processing sets perhaps the most exacting bar of all: *IPOL* publishes each algorithm together with a peer-reviewed reference implementation and a runnable in-browser demo, so that a reader reproduces the result in the very act of reading the paper [53]. And security has joined at the level of policy — USENIX Security made artifact availability *mandatory* in 2025 [94].

Two efforts aim past any single discipline. *ReScience C* publishes independent re-implementations that reproduce a paper's findings from scratch [75], and NISO's RP-31:2021 lifts the whole badging vocabulary out of computing into a discipline-neutral recommended practice [68] (Table 6.1). Across every one of these the instrument differs, but the need underneath is identical: a durable, publicly resolvable, *verifiable* handle on the exact code that was checked. Intrinsic identification does not care which discipline minted the artifact — a SWHID names bytes, not a field — which is precisely why the same ARDC

discipline serves an economist, a geographer and a systems researcher without changing a line.

Field	Venue / body	Instrument	Mandatory?	URL
Economics & social science	AEA journals	AEA Data Editor audit	Yes (2019)	aeadataeditor.github.io
Economics & social science	CNRS, HEC Paris, Université d'Orléans	*cascad* certification (170+)	No (opt-in)	cascad.tech
Life sciences & geography	GigaScience, Reproducible AGILE	CodeCheck	No (opt-in)	codecheck.org.uk
Image processing	IPOL journal	Reference impl. + online demo	Yes (built-in)	ipol.im
Computer graphics	SIGGRAPH, TVCG, ...	Graphics Replicability Stamp (478) [51]	No (opt-in)	replicabilitystamp.org
Security	USENIX Security	Artifact availability	Yes (2025)	usenix.org
Cross-domain	ReScience C	Independent re-implementation	—	rescience.github.io
Pan-disciplinary	NISO	RP-31:2021 badging standard	—	niso.org

Table 6.1: Reproducibility and artifact-evaluation instruments across research fields. The instruments differ by discipline; what they share is the need for a durable, verifiable handle on the exact code under review — the gap ARDC fills with a SWHID.

6.3 Artifact evaluation with Software Heritage

Reproducibility policy has outgrown computer science — but the handle it needs is the same everywhere: a verifiable SWHID.

Look again at the first badge. *Artifacts Available* asks for two things: a publicly accessible archival repository, and a unique identifier for the object. For decades this has been met by the legacy recipe — a zip deposited on Zenodo, figshare, or in the ACM Digital Library, with a DOI. We have already seen, at length, why that recipe is weak for *source code*: a DOI certifies a landing page rather than the bytes (Chapter 4); the deposit can change, carries no granularity, and discards the development history (Chapter 3). Everything the badge really wants — a durable archive and a unique, verifiable identifier — is exactly what ARDC delivers: archive the living repository in Software Heritage, and reference it with a SWHID.

This is not a departure from policy; it is the rigorous way to follow it. Software Heritage is already among the archives ACM accepts for the *Available* badge, the SWHID is now ISO/IEC 18670, and the software-engineering community has explicitly asked that it be used for artifact evaluation [32, 54]. The gain is sharpest for the *reviewer*. A DOI gives a reviewer a link to follow and a record to trust; a SWHID gives something a DOI never can — the ability to *recompute the identifier from the bytes under review and check that it matches*, offline, in under a minute. If it matches, the reviewer holds cryptographic proof that the artifact in hand is exactly the artifact the authors named. If it does not, that too is a finding.

Hands-on: verify an artifact as a reviewer

~10 min · needs a SWHID tool + the artifact files · in-class, offline

You are handed an artifact and the SWHID the authors published for it.

1. Verify a file against its claimed content SWHID: `swhid verify file.c swh:1:cnt:<hex>` — it exits 0 only if the bytes match (Appendix D).
2. Re-compute the directory SWHID of the whole artifact tree and compare it with the one in the paper. No archive, no network, and no trust required.

For the *organiser*, the move is equally small: name Software Heritage in the call for artifacts, and ask authors to put the qualified SWHID of the precise artifact in the paper. Each badge then maps cleanly onto a Software Heritage–native check:

ACM badge	Software Heritage–native check
Artifacts Available	archived in Software Heritage; a qualified SWHID in the paper
Evaluated — Functional	the reviewer fetches the *exact* SWHID-pinned tree; no "which zip?"
Evaluated — Reusable	`codemeta.json` + `CITATION.cff` + an SPDX license, under that SWHID
Results Reproduced	reproduction starts from a verifiable, immutable identifier

Table 6.2: Each ACM badge mapped onto its Software Heritage–native check.

can verify a SWHID minute — a DOI ing of the kind.

Ready to paste: a call for artifacts

For an organiser, here is language to drop into a call for artifacts, and a reviewer check:

> *Authors of accepted papers must archive the source code behind their results in Software Heritage (`https://save.softwareheritage.org`) and include in the paper the* qualified *SWHID of the precise artifact. A DOI (e.g. via Zenodo, which forwards to Software Heritage) may be supplied additionally for discoverability. For double-blind submission, give a* directory *SWHID, which reveals no author or origin.*

Reviewer check (offline, <1 min): resolve the SWHID at `archive.softwareheritage.org/<swhid>`; recompute it from the bytes (`swhid verify`); confirm it pins the *exact* code the claims rest on. A mismatch is itself a finding. *When it doesn't fit:* for artifacts that cannot be public (licensed data, large binaries, special hardware), require the public part archived and the rest escrowed — Software Heritage solves *identity and availability*, not buildability, which *Functional*/*Reusable* still test by running.

6.4 Double-blind review, and dense pinpointing

Artifact evaluation often happens under *double-blind* review — where reviewers do not know who the authors are, nor the authors who their reviewers are — which seems to pose a problem: how do you share your code without revealing who wrote it? Tools that rewrite a repository's history to anonymise it are fragile, and they leak. The intrinsic identifier sidesteps the whole difficulty. A *directory* SWHID is computed from content alone — no author, no origin, no commit metadata, nothing to scrub — and it works for any technology, not just Git. Submit the `dir` SWHID; reviewers confirm the bytes are exactly yours while learning nothing about you.

The same idea scales to a paper that pinpoints code throughout its body. Modern papers already do this densely: *Source Code Hotspots* ties its findings to exact passages, naming each with a qualified SWHID so a reader can open the precise lines in the archive [66]. And the part that carries *provenance* is precisely the qualifiers — `origin`, `visit`, `anchor` — while the core hash, the identifier proper, reveals nothing about authorship. So the *only* difference between the double-blind submission and the camera-ready version (the final, de-anonymised paper) is removing, then restoring, those qualifiers: the core hashes are identical, and every pinpoint a reviewer checked during blind review stays valid, to the character, in the final paper. One mechanical, fully reversible edit (Figure 6.3) — no history-rewriting, no risk of a leak.

nd vs camera-ready: restore, the – the content hash never changes.

Hands-on: a double-blind artifact

~10 min · needs a SWHID tool · in-class, offline

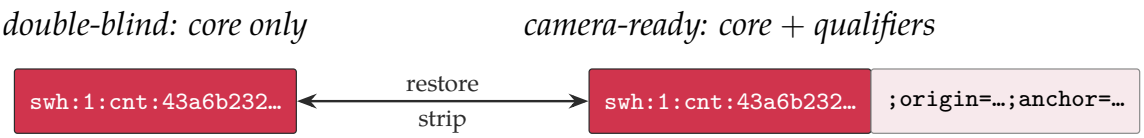


Figure 6.3: The core hash (red) is identical on both sides and reveals nothing about authorship; only the locating qualifiers are stripped for blind review and restored for the camera-ready version. Every pinpoint a reviewer checks survives the round trip unchanged.

Export your artifact tree cleanly into an empty directory with `git archive` — never hash the checkout itself, which carries `.git/` and your build output — compute the directory SWHID of the export (Appendix D), and check it against `git rev-parse HEAD^{tree}`: the two must print the same hash. Submit *that*. It names your exact bytes and nothing else. After acceptance, archive the repository and add the `origin/anchor` qualifiers — the same identifier, now pointing home.

If the artifact is already a public repository, the shorter path is to archive it first and read the identifier off the archive; Appendix G sets out the whole workflow as a programme chair would state it.

6.5 Built into the record of science

Best of all is when none of this is left to the author's goodwill. Since November 2020 the journal *eLife* has archived the code behind every article in Software Heritage automatically, embedding the SWHID in the article's machine-readable metadata and validating it before publication [42]. Availability stops being a promise a reviewer checks once, and becomes a durable property of the published record itself — which is, in the end, exactly where it belongs.

That is the case for building artifact evaluation on Software Heritage. What remains is to look outward: at how widely this is already being adopted, the policy that now backs it, and the larger mission it serves. That is the final chapter.

7 Adoption, policy and outlook



None of this is hypothetical: journals, funders, and national plans already point at Software Heritage and the SWHID, and France has gone past recommendation to statute. The archive serves more than open science — supply-chain traceability, transparent AI, resilience when a platform fails. The tools are built and the policy is written; what is missing is the habit, and, one level up, four choices only funders and policymakers can make.

Learning objectives

- survey where this practice is already adopted, and the policy that backs it;
- situate Software Heritage in the resilience, sovereignty, and transparent-AI agendas;
- identify what to do as a researcher, a reviewer, or an institution.

We have built the argument and shown the practice. It remains to look outward: to see that this is already happening, that policy now stands behind it, and that the same infrastructure serves a mission far larger than open science. And then to ask what you can do, starting — one last time — with your next paper.

7.1 It is already happening

Open science is one payoff of software preservation; resilience, transparency, and sovereignty are others.

None of this is hypothetical. Journals and platforms across disciplines have wired Software Heritage into how they work: the HAL national repository offers a curated software-deposit workflow; IPOL publishes image-processing papers with runnable code and a SWHID; *eLife* archives the code behind every article automatically; *swMATH* references mathematical software in the archive; and Episciences journals such as JTCAM ask authors to cite their software with `biblatex-software` (itself shipped in the ACM article class since 2022) [1]. The connections keep multiplying: Episciences now links each published article to its code through a SWHID [81]; *Zenodo* forwards its software deposits to the archive (Chapter 5); and code-publishing venues such as Dagstuhl's *DROPS* interconnect the same way.[1] Papers themselves now carry not one identifier but many, pinpointing their code passages with qualified SWHIDs throughout the text [66].

And the stakes are real, not niche. French law took the measure of this directly: article 163 of the 2022 "3DS" law required the government to report to Parliament, within eighteen months, on the software produced by public research — a mandate answered in 2023 by the *Rapport Boulet* [16, 74]. Built on 1 331 detailed responses, it found long-lived, widely-used, high-impact work: half of the software reported was more than nine years old, over a third (36%) had more than a hundred users, nearly two-thirds (62%) reported impact beyond academia, and almost a quarter (23%) already fed technology transfer — and it was overwhelmingly open, with only about a tenth kept proprietary. This is not a fringe of scientific practice; it is a load-bearing part of it, and it has been going uncared-for.

That survey, though, was deliberately narrow: a stock-take of *production and valorization*, answering a precise legal question, not an examination of the code itself. Point the same lens at the actual corpus — the software French public research has archived in Software Heritage, some 32 694 projects carrying contributions from the country's universities and research organizations — and a sharper picture emerges (Figure 7.1). Only about a fifth of those projects carry any license at all, so by default most of them cannot legally be reused; and roughly seven in ten live on a single commercial platform, `github.com`, a concentration that is itself a systemic risk. The law asked *how much*; the archive can show *in what state* — and the state is exactly what the rest of these notes are written to improve.

Recognition, not just adoption, has begun to follow. In 2022 the French ministry created a standing national award for free and open-source research software — the *Prix science ouverte du logiciel libre de la recherche* — and its inaugural edition drew 129 nominations, from which the jury singled out four winners and six honourable mentions (*accessit*) [14]. The roll of winners shows exactly the breadth these notes keep asserting, made concrete: the proof assistant *Coq* from computer science,[2] *scikit-learn* from machine learning,[3] *Faust* from computer music and audio,[4] and *Gammapy* from gamma-ray astronomy[5] — four fields, one practice. The prize is now annual; its second edition, in 2023, drew 66 projects [65]. The reasoning behind it — why a country should reward research software at all, and how a jury should judge it — has itself been written up as a reusable blueprint for other nations to copy [14]. Career credit for the people who write software, the very thing this chapter's call to action will ask of institutions, is here already national policy.

Coq, scikit-learn, Faust, Gammapy: a national award whose winners span four disciplines — software matters in every field, made concrete.

And we are no longer measuring this by hand. The French Open Science Monitor we met in Chapter 1 (Figure 1.1) does not count software paper by paper: it mines the open-access literature automatically, with an improved software-mention extractor in the *Softcite* lineage — the same dataset the Chan Zuckerberg Initiative used to train the model that surfaced the hidden footprint

[1] https://drops.dagstuhl.de/.
[2] https://coq.inria.fr/
[3] https://scikit-learn.org/
[4] https://faust.grame.fr/
[5] https://gammapy.org/

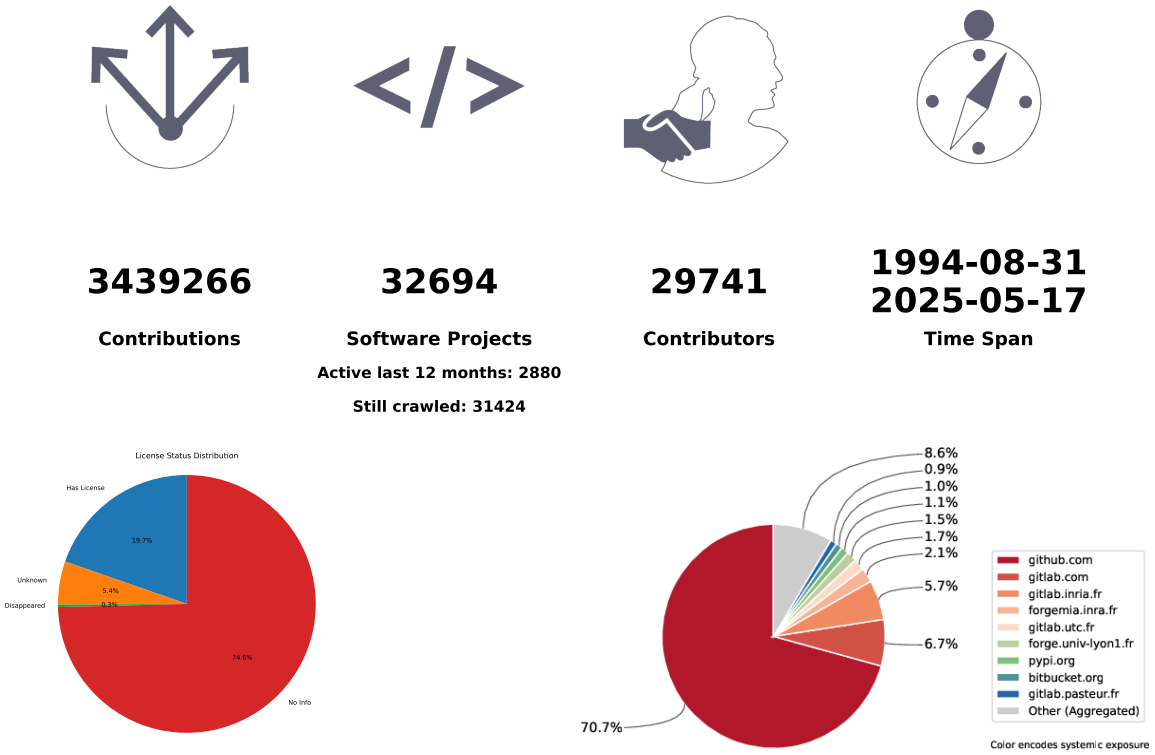


Figure 7.1: The French public-research software corpus archived in Software Heritage, as of 2025. *Top*: the bird's-eye view — some 32 694 projects (2 880 active in the last twelve months), 3.4 million contributions by 29 741 contributors, spanning 1994–2025, with 1 270 projects preserved although they have since vanished from their forge. *Bottom left*: only 19.7% of projects carry a license — four-fifths have none at all, so by default they cannot legally be reused. *Bottom right*: hosting is heavily concentrated, with 70.7% of projects on a single commercial platform, `github.com` — a systemic single point of failure. (Software Heritage, Acad-Hal-Fr analysis, v1.0, 2025.)

of open-source software in the biomedical literature [21]. Building it on open technologies was always a bet that others could run it too — and the bet is paying off. Under a joint UNESCO and French impulse, the method is being generalized into an **Open Science Monitoring Framework**: a shared, reproducible recipe for monitoring open science with open tools, rather than with the proprietary bibliometric databases the task has long depended on [24]. The software strand of that framework has its own dedicated infrastructure — **SoFAIR** — which identifies software mentions across the scholarly literature and links each one back to the archived code in Software Heritage, closing the loop between a citation in a paper and a verifiable SWHID [78]. So the very figure that opened our case is itself on the move: a national dashboard becoming a worldwide instrument, and what a community can finally measure, it can finally choose to value.

7.2 The policy landscape

The direction of travel, at every level, is now unambiguous. The **UNESCO Recommendation on Open Science** (2021) calls for source code to be released openly, and for open-science infrastructures to be "organized and financed upon an essentially not-for-profit and long-term vision" [92]. The European **EOSC SIRS report** (2020) is more specific still: *archive in Software Heritage, use the SWHID*, and treat open source as the default for research

software, with deviations to be justified [38]. The **Paris Call on Software Source Code** (2019) had already urged the world to "recognise software source code as a fundamental enabler in all aspects of human endeavour" and to build shared infrastructures to preserve it [93]. And at the national level the message is the same: France's second **National Plan for Open Science** names Software Heritage and the SWHID explicitly, the country's research-infrastructure roadmap now lists the archive, and a standing College for research software carries the agenda forward [64, 30]. Strip away the acronyms and a single instruction remains: archive in Software Heritage, and identify with a SWHID.

A fair objection is that a report, however specific, is still just a report — a recommendation can sit on a shelf. This one did not. In 2021 the EOSC Association stood up a *Task Force on Infrastructures for Quality Research Software*, chartered to foster the tools and services that let researchers archive, reference, describe, share, and reuse their software, to raise its quality technically and organizationally, and to win recognition for the people who build and maintain it [43]. And in 2022 the **FAIRCORE4EOSC** project began building it: one of its work packages implements the very architecture the SIRS report had sketched, wiring Software Heritage to the repositories, publishers, and aggregators around it through concrete connectors — InvenioRDM, Episciences, Dagstuhl, swMATH, Zenodo, OpenAIRE [47]. The line runs 2020 report, 2021 task force, 2022 implementation: not a recommendation gathering dust, but a blueprint being wired into the European research infrastructure.

There is, though, a difference in *kind* that France draws more sharply than anyone, and it is worth making explicit. Everything cited so far — recommendations, plans, roadmaps, funding conditions — is *soft* policy: it urges and it nudges, and a determined laggard can still decline. France has also *legislated*. The 2016 *Law for a Digital Republic* — the "loi Lemaire" — made the source code of software both *produced by* and *received by* the administration an administrative document, open by default and communicable on request [73]. A 2021 directive of the Prime Minister then turned the principle into machinery: it opens public-administration data and source code by default, stands up a dedicated "open-source software mission", and installs in each ministry a network of data, algorithm and source-code officers — the AMDAC — charged with carrying the obligation out [72]. The effect is not a pledge but a pipeline: code written for the French public administration now flows into Software Heritage through `code.gouv.fr`, on the authority of a statute rather than a recommendation.[6] The distinction is not pedantry: a recommendation tells you what good practice looks like; a law makes its absence a fault.

Soft policy urges; France's 2016 loi Lemaire *and its 2021 Prime-Ministerial directive* compel *open source by default.*

Funders have followed. France's national research agency, the ANR, asks in its own funding guidelines that project software be released under a libre license and its source archived in Software Heritage, with the grant referenced [3] (Figure 7.3).

Step back from France, though, and the same instruction reappears in jurisdiction after jurisdiction — which is what turns a national success story into a landscape. At the European level it predates all of them: already in 2018 the Commission formally called on every Member State to adopt a national action plan "reinforcing the preservation and re-use of scientific information" — publications, datasets, and other research outputs, source code among them [44]. France's plan is, in this light, one Member State doing what the Commission asked; others are doing the same. Nor does the agreement stop at Europe's edge: in the United States, NASA's open science-information policy makes the software behind agency-funded research an expected, openly available output in its own right, alongside the data and the papers [67].

[6] The public-sector codebase is published and archived through `https://code.gouv.fr/` (formerly `code.etalab.gouv.fr`); see `https://www.etalab.gouv.fr/code-etalab-gouv-fr-les-codes-sources-de-logiciels-publies-par-des-organismes-publics`.



Figure 7.2: Two of the documents that now point research software at Software Heritage: the EOSC *Scholarly Infrastructures for Research Software* report (left, 2020) [38] and France's second National Plan for Open Science (right, 2021–2024) [64].

> Enfin, conformément au 2ème Plan national pour la science ouverte, L'ANR recommande que les logiciels développés durant le projet soient mis à disposition sous une licence libre[30] et que les codes sources soient stockés dans l'archive Software Heritage[31] en indiquant la référence au financement ANR.

Figure 7.3: From the ANR's funding guidelines: project source code is to be stored in Software Heritage under a free license, with the ANR grant referenced.

Funders, too, have moved beyond France, and tellingly the lever several of them reach for is the one these notes keep coming back to — recognition. Germany's Deutsche Forschungsgemeinschaft, the country's central research funder, added *research software* to its template academic CV in 2022, so that the code a researcher writes can be listed, and weighed, next to their publications [26]. And the funders have begun to act not one agency at a time but together: Science Europe — the association of Europe's major public research-funding and research-performing organizations — has taken research software into its open-science work through a dedicated working group, with a software strand at the 2022 Open Science European Conference [30]. That is a different kind of policy actor from the ones above: not a ministry issuing an instrument, but the funders themselves, collectively *drafting* the policy upstream of any published text. The signatures differ, the languages differ, the legal instruments differ; the instruction does not.

The full set of instruments behind this convergence — international, European, and national, deduplicated into one matrix with issuing body, jurisdiction, date, mandate, and link — is collected in Appendix E.

The policy is unambiguous: archive in Software Heritage, identify with a SWHID.

A policy we helped shape: the 2019 "upload filters" episode

Policy is not only something that backs us from above — it is something a community can help write. When the European Union finalised its **Copyright Directive** (Directive (EU) 2019/790) in the spring of 2019, one provision — the so-called "upload filters" — would have obliged platforms to screen everything their users post against rights-holders' databases [45]. Drafted with music and video in mind, it swept up the entire machinery of modern software development: the forges where code is written, the distribution platforms that carry it (Maven, PyPI, CRAN, CTAN), and preservation archives such as Software Heritage. Source code that legitimately reuses or quotes other source code would have been caught in a filter built for pop songs. After sustained engagement from the open-source and

research-software community — in which Software Heritage helped raise policy-makers' awareness, briefed NGOs, and opened channels to governments — an explicit exclusion was obtained for open-source software developing *and* sharing platforms, and the "not-for-profit" qualifier that would have gutted it was struck. The lesson is worth keeping: the same community that asks for policy to protect its infrastructure can, when it shows up, keep bad policy from breaking it.

7.3 The engine that carries it out: the OSPO

Policy written at the top does not execute itself. A national plan can name Software Heritage and the SWHID, a funder can ask for an open license and a deposit — and still nothing changes in a lab unless someone, locally, turns the principle into the default path. In industry that someone has a name: the *Open Source Program Office*, or OSPO — a small competence centre that owns an organization's open-source strategy, its compliance, and its security, and that has by now become common enough to warrant an annual census of its own [59, 91].

Universities are standing up the same thing, adapted to research. A university OSPO is the local execution engine that connects national open-science policy to the daily life of a group: it makes open-by-default the path of least resistance, and absorbs the license and compliance questions so that researchers need not; it runs the archiving and citation plumbing — the `CITATION.cff` file, the SWHID, the curated deposit — so that ARDC (Chapter 5) becomes routine rather than heroic; it trains people in reproducibility and continuous integration; and it sees to it that software contributions are recognized and counted in careers. Registries such as OSPO-RADAR (Chapter 5) are exactly the kind of catalogue such an office keeps current. In short, the OSPO is where the policy of the previous section meets the practice of the whole of these notes.

7.4 The wider mission: resilience, sovereignty, transparent AI

Policy needs a local engine: the university OSPO is where open-science mandates become ordinary research practice.

Open science is only one of the things this archive makes possible, and stepping back reveals why it matters beyond the library. The same uniform record of the world's source code is the foundation for several urgent agendas at once. It underpins *software supply-chain traceability*: an intrinsic identifier is exactly what a software bill of materials (an SBOM) needs, and what regulators reaching for the EU Cyber Resilience Act will require [46, 10]. And the scale of that dependency problem is not guesswork: industry audits find open-source components in 96–97% of the commercial codebases they examine, making up around three-quarters (some 77%) of the code, with an average of over five hundred distinct open-source components in a single application [87, 13]. Modern software is, by an overwhelming margin, assembled rather than written — and every one of those components can move, change, or quietly disappear, so preserving the *dependency graph*, not merely the top-level repository, is the whole of the supply-chain argument. It underpins *transparent AI*: the very large code corpora that train today's models — The Stack, on which StarCoder is built — are drawn from Software Heritage, and provenance for that data is becoming a legal as well as an ethical demand [60]. And it underpins *resilience*: because the archive holds the code with its history, a build system can fall back to it when the upstream platform fails. Guix has done exactly this since 2019 — when a package vanishes from its registry, the build continues from Software Heritage [31, 37]. The point is not academic. If GitHub or PyPI disappeared tomorrow, much of the digital world would not merely slow down — it would stop.

This reframes the whole enterprise. The goal is not to *own* the infrastructure but to ensure that nothing essential can vanish — a shared, open, non-rival archive that benefits

all while leaving each free to build on it. That is digital sovereignty rightly understood: not autarky, but *resilience*.

7.5 A fair question: what about Software Heritage itself?

These notes have argued that platforms fail and that we should plan for it — so they must turn the same lens on the archive they recommend. Software Heritage is a non-profit initiative launched in 2016, hosted by Inria and run as an international, multi-stakeholder effort under an agreement with UNESCO; it is funded by a range of institutional members and sponsors rather than by any single company or state, with foundational support from bodies such as the Alfred P. Sloan and NLnet foundations. Its answer to its own single-point-of-failure risk is a *network of independent mirrors* — full copies of the archive held by partner institutions, so the corpus survives the loss of any one operator. It is the same resilience-by-redundancy the archive offers your build chain, turned on itself. For an editor or a funder weighing a ten-year commitment, that — not a metaphor — is the due-diligence answer.

It is equally worth being clear about the *limits*. A SWHID secures the *identity* and *availability* of source code; it does not make a build deterministic (Chapter 6), and it cannot open code that was never public. Adoption, though real and growing, is still offered as one option among several at most venues rather than a default. The claim of these notes is not that the work is finished — it is that the infrastructure, the standard, and the practice are now ready to be made the default.

7.6 Call to action

Which brings us, finally, back to you. The tools are built, the policy is written, the adoption is under way; the one thing still missing is the habit. So make it, at three scales.

As a researcher: archive the code behind your next result in Software Heritage, reference it with a SWHID at the granularity your claim needs, describe it, and cite it. That is the whole of ARDC, and you can do it this week.

As a reviewer, an editor, or a member of an artifact-evaluation committee: ask authors for archival proof — a SWHID — rather than a fragile URL. It is one small habit, but applied every time, it changes the norms of a whole community.

As an institution or a funder: write Software Heritage into your open-science policy, become a member or run a mirror, and — not least — count high-quality software contributions in the careers of the people who make them.

Ready to paste: a grant condition and a policy line

Grant condition (it costs the grantee nothing — archival is free):

> *Source code produced with this grant must be released under an open license and archived in Software Heritage, and the resulting SWHID reported in the project's outputs. (Further documentation, testing, and packaging are encouraged but not required.)*

Open-science / sovereignty policy line:

> *We adopt intrinsic, standard identifiers (SWHID, ISO/IEC 18670) and a shared, non-profit archive (Software Heritage) for research software, ensuring that publicly funded code remains verifiable, citable, and recoverable independently of any commercial platform.*

A funder can audit compliance across a whole portfolio with one HAL-API call (Chapter 5's `collCode_s` query), no per-project chasing required.

A working agenda for funders and policymakers

The three scales above are things one person can act on tomorrow. But the deepest leverage sits one level up — with those who write the calls, set the rules, and sign the cheques — and there the choices are systemic, not personal. Four of them, in particular, decide whether the practice this note describes becomes the default or stays a curiosity. They are worth stating plainly, as an agenda.

First, default to open. Software produced with public money should be open source as a matter of course, and any departure should have to be justified — not the reverse. This is the European line (§7.2) taken to its conclusion: where the EOSC SIRS report asks that deviations be "properly motivated" [38], a funder can go one step further and make open the default, putting the burden of proof squarely on the exception. Closed-by-default is the costly choice; it is the one that should have to argue for itself.

Second, avoid balkanization. The instinct, once a need is recognized, is for each institution to stand up its own archive, its own identifiers, its own registry. That way lies a patchwork of silos that do not interoperate and that each carry the full cost of survival alone. The alternative is to build on shared, open, non-profit infrastructure — Software Heritage for the source code itself — so that effort accumulates instead of fragmenting.

Third, fund the commons as a commons. A mutualized infrastructure is not free because it is shared; it needs recurrent, predictable funding rather than one-off project grants, and it must be evaluated as the long-lived service it is. Above all it must be funded for what it predominantly *is*: not a server bill, but the people who curate, maintain, and run it. The human component is the larger part of the cost, and the first to be cut when infrastructure is mistaken for hardware.

Fourth, incentivize intelligently. If we want researchers to write, share, and maintain good software, the careers of the people who do it must reward exactly that — high-quality contributions, counted as first-class scholarly output. The temptation is to reach instead for a number (papers, citations, downloads) and let it stand in for worth. Goodhart's law warns where that ends: "when a measure becomes a target, it ceases to be a good measure" [84]. Keep a human in the loop, and judge contributions on their merits rather than on a metric that will, predictably, be gamed.

A four-point agenda for research-software policy

For those who design the rules, not just follow them:

1. **Default to open.** Publicly funded research software is open source by default; exceptions *must* be justified.
2. **Avoid balkanization.** Build on common, shared, open, non-profit infrastructure (Software Heritage), not per-institution silos.
3. **Fund the commons as a commons.** Recurrent funding and proper service evaluation, recognizing that the human component is the predominant cost.
4. **Incentivize intelligently.** Count quality contributions in careers; avoid purely numerical indicators; keep a human in the loop.

> "Software source code is a precious, unique form of knowledge, and for too long we have treated it as an afterthought. We no longer have to. The argument is settled and the machinery is ready; what is needed now is simply that we use it. Record your code, share it, and value it [39]. Start with your next paper."

A Command cookbook



There are two interchangeable local toolchains for computing and checking SWHIDs, and they produce the *same* identifiers for the same bytes — pick whichever suits your environment. **Path A** is `swhid-rs`, the Rust reference implementation of the standard (ISO/IEC 18670, MIT license): fast, standalone, and the one that tracks the specification most closely. **Path B** is the original Python implementation — `swh identify` (from `swh.model`) and `swh scanner` — which adds two things the Rust CLI does not yet cover: snapshot identifiers, and scanning a working tree against the archive. Appendix D compares all the implementations in full. The two paths below mirror each other operation for operation; the shared section at the end is the same whichever you chose.

Path A — the Rust reference tool (`swhid-rs`).

```
# --- Install -------------------------------------------------------------
cargo install swhid                  # swhid-rs (Rust reference impl., MIT)
cargo install swhid --features git   # ...with VCS (revision) support
# or download a pre-built binary from the project's releases page:
#   https://github.com/swhid/swhid-rs

# --- Compute core SWHIDs locally -----------------------------------------
swhid content --file file.c          # -> swh:1:cnt:...
swhid dir src/                 # -> swh:1:dir:...  (Merkle hash of tree)
swhid parse "swh:1:cnt:<hex>"        # validate / pretty-print a SWHID

# A revision SWHID is just the git commit hash:
echo "swh:1:rev:$(git rev-parse HEAD)"     # -> swh:1:rev:...

# --- Verify bytes against a claimed SWHID (exit 0 iff they match) --------
swhid verify file.c swh:1:cnt:<hex>        # PATH first, then SWHID
```

```
swhid verify src/   swh:1:dir:<hex>

# Snapshot SWHIDs and tree scanning: use Path B below.
```

Path B — the Python tools (`swh.model` + `swh.scanner`).

```
# --- Install -------------------------------------------------------------
pip install 'swh.model[cli]'         # swh identify
pip install swh.scanner              # swh scanner

# --- Compute SWHIDs locally (auto-detects type, or force with --type) ---
swh identify file.c                  # -> swh:1:cnt:...
swh identify --type directory src/   # -> swh:1:dir:...
swh identify --type directory ./export/    # a CLEAN export -> swh:1:dir:...
#   (never a live checkout: see the caveat below)
swh identify --type snapshot repo.git/     # -> swh:1:snp:...  (snapshot)

# --- Verify bytes against a claimed SWHID (exit 0 iff they match) --------
swh identify --verify swh:1:cnt:<hex> file.c    # SWHID first, then PATH

# --- Scan a working tree against the archive -----------------------------
swh scanner scan ./repo                    # what is already archived?
```

Shared — archive and resolve (no install needed).

```
# Archive a public repo:  https://save.softwareheritage.org
# Resolve any SWHID:      https://archive.softwareheritage.org/<swhid>
```

One caveat for `dir` hashes. A directory SWHID is computed over everything below the directory you point the tool at — including a `.git/` directory, build output and editor droppings — and it depends on file *permissions* as well as content. To match what the archive stored from a git repository, never hash a live checkout: compute over a clean export (`git archive` unpacked into an empty directory), whose hash equals `git rev-parse <commit>^{tree}` by construction. If a directory check mismatches where a content check would pass, suspect a polluted working copy first, and permission bits second.

B SWHID syntax cheat-sheet

This appendix summarises the SWHID grammar. It is *not* normative: the authority is the *SWHID Specification, version 1.2* [86], published openly on swhid.org and standardised, word for word, as ISO/IEC 18670:2025 [54]. The section pointers below refer to that specification.

B.1 Core identifiers (spec §5)

A *core* SWHID is four colon-separated fields, `swh:1:`*`type`*`:`*`hash`*:[1] the scheme `swh`; the schema version (`1`); a three-letter object *type*; and a 40-digit, lowercase-hex object *identifier*. That identifier is a SHA-1 computed over a git-style serialisation of the object — so a revision SWHID is exactly the Git commit hash with `swh:1:rev:` prepended. The five types are:

Code	Object	Identifies
`cnt`	content	the bytes of a single file
`dir`	directory	a directory tree (names, permissions, and child hashes)
`rev`	revision	a commit, with its parents, author, and date
`rel`	release	a named release (annotated tag)
`snp`	snapshot	a repository's full state at one visit (all branches and tags)

B.2 Qualified identifiers (spec §6)

A *qualified* SWHID appends a sequence of qualifiers to a core identifier, each introduced by `;` and written `name=value`.[2] The core *identifies*; the qualifiers *locate* and *contextualise*.

Qualifier	Kind	Meaning
`origin`	context	the URI the code was harvested from
`visit`	context	the snapshot SWHID of the visit
`anchor`	context	a node (`dir`/`rev`/`rel`/`snp`) the `path` is rooted at
`path`	context	an absolute path from the anchor's root directory
`lines`	fragment	a line range within a `cnt` (1-based, inclusive; `A` or `A-B`)
`bytes`	fragment	a byte range within a `cnt` (0-based, inclusive)

The rules (spec §6): each qualifier appears at most once; `visit` requires `origin`; `anchor` requires `path`; and the fragment qualifiers `lines`/`bytes` apply only to a `cnt`. The recommended order is `origin`, `visit`, `anchor`, `path`, then `lines` or `bytes`.

[1] https://www.swhid.org/specification/v1.2/5.Core_identifiers/
[2] https://www.swhid.org/specification/v1.2/6.Qualified_identifiers/

B.3 A worked example

The specification's own example pins lines 9–15 of one file, in one revision, of one archived repository:

```
swh:1:cnt:4d99d2d18326621ccdd70f5ea66c2e2ac236ad8b;
  origin=https://gitorious.org/ocamlp3l/ocamlp3l_cvs.git;
  visit=swh:1:snp:d7f1b9eb7ccb596c2622c4780febaa02549830f9;
  anchor=swh:1:rev:2db189928c94d62a3b4757b3eec68f0a4d4113f0;
  path=/Examples/SimpleFarm/simplefarm.ml;
  lines=9-15
```

(written on a single line in practice). To compute or verify a SWHID, see Appendix D.

C

Glossary

Anchor
: a context qualifier of a SWHID: the node (`dir/rev/rel/snp`) against which a `path` is resolved.

Archive
: a system whose purpose is to *keep* content for the long term — as opposed to a forge, which is built to *develop* it.

ARDC
: Archive, Reference, Describe, Cite & credit — the four operations that make up the practice at the heart of these notes.

Artifact evaluation (AEC)
: the process, now standard at computer-science conferences, of checking that the software behind a paper exists, runs, and supports its claims; carried out by an *artifact-evaluation committee* (AEC).

Badge (ACM)
: one of the ACM *Artifact Review and Badging* marks: Artifacts Available; Artifacts Evaluated — Functional / Reusable; Results Reproduced / Replicated.

`biblatex-software`
: the LaTeX bibliography style for citing software, adding four entry types and a `swhid=` field.

CITATION.cff
: the Citation File Format, a lightweight YAML file describing how to cite a repository.

CODE
: the beyond-FAIR roadmap for research software — Open, Document, Execute, Collaborate; a mnemonic word, not an initialism read off in order.

CodeMeta
: a JSON-LD vocabulary for software metadata, written to a `codemeta.json`.

Content (`cnt`)
: the bytes of a single file; the leaf of the Merkle DAG.

Core SWHID
: `swh:1:`*`type`*`:`*`hash`*; the part that *identifies*, a fingerprint of the content alone.

Deposit
: the SWORD (Simple Web-service Offering Repository Deposit) archival channel, reserved for trusted partners and registered organizations (HAL, journals, institutions) that hold push credentials — not an ordinary registered account, which uses Save Code Now — to push code that is not in a public forge.

Directory (`dir`)
a directory-tree node: names, permissions, and the hashes of its children.

DOI
Digital Object Identifier; an *extrinsic* identifier resolved through the Handle System.

Extrinsic identifier
a name assigned by a registry; it means nothing without that registry.

FAIR
the Findable, Accessible, Interoperable, Reusable principles — designed for research *data*.

FAIR4RS
the effort to adapt the FAIR principles to research software.

Forge
a collaborative development platform (GitHub, GitLab, Bitbucket, ...) — *not* an archive.

Intrinsic identifier
a name computed from the content itself; verifiable offline, with no registry and no middle-man.

Lister / loader
the components that, respectively, enumerate the repositories on a hosting platform and ingest each one into the archive.

Merkle DAG
the content-addressed directed acyclic graph in which Software Heritage stores all code and history; each node is named by the hash of everything beneath it.

Origin
a context qualifier: the URI a piece of code was harvested from.

Path
a context qualifier: an absolute path from the anchor's root directory.

Qualified SWHID
a core SWHID plus context (`origin`, `visit`, `anchor`, `path`) and fragment (`lines`, `bytes`) qualifiers.

Release (`rel`)
a named release node (an annotated tag).

REUSE
a specification for declaring per-file licensing in a machine-readable way.

Revision (`rev`)
a commit node — identical to the underlying Git commit hash.

Save Code Now
the web entry point (`https://save.softwareheritage.org`) that triggers archival of a public repository.

Snapshot (`snp`)
the full state of a repository at one visit — all of its branches and tags.

SPDX
: the standard catalogue of machine-readable software-license identifiers.

SWHID
: *Software Hash Identifier* — the intrinsic identifier for source code (ISO/IEC 18670:2025). Originally read "Software Heritage Identifier"; since the specification it is "Software *Hash* Identifier".

Visit
: a crawl of an origin by the archive; also the context qualifier giving that visit's snapshot SWHID.

SWHID tool implementations

D

A SWHID is, by design, something anyone can compute: the whole point of an intrinsic identifier is that it owes nothing to a registry, so one you could only obtain from a central service would betray the idea. Now that the SWHID is an international standard (ISO/IEC 18670:2025), a small ecosystem of independent implementations has grown up around it — in several languages, at several levels of ambition — and they all compute, for the same bytes, the same identifier. That is the standard doing its job.

It helps, though, to keep one distinction sharp, because the ecosystem is easy to misread. To *obtain or check* the SWHID of a software artifact, you want one of the dedicated calculators: two command-line tools that *compute and verify* a SWHID from local files — the Rust reference implementation `swhid-rs`, released under the MIT license and the one these notes reach for first, and the Python `swh.model` / `swh identify` CLI, the most complete in coverage (content, directory, revision, and snapshot) — or, to do the same arithmetic *inside your own programs*, one of a few language libraries. The rest are *companion* tools, built for other jobs: `swh scanner` reports what the archive already holds, the *updateswh* extension and the *Permalinks* tab *fetch* a ready-made SWHID from the archive, and the resolvers turn a SWHID back into the code it names. Several of these do compute SWHIDs internally — `swh scanner`, for one, hashes your files precisely so it can look them up — but none is the blessed way to *get* an artifact's SWHID. Table D.1 sorts the ecosystem along exactly that line.

A practical rule of thumb: reach for `swhid-rs` when you want the reference behaviour and a single fast binary; fall back to `swh identify` for snapshots and the deeper archive workflow. But keep the grouping in mind: the first two groups are the ones you reach for to *obtain or verify* an artifact's SWHID, while the companion tools are built for other jobs — the browser route fetches one the archive already computed, the resolvers turn an identifier back into code, and `swh scanner` hashes your tree only to check what is already archived. For a one-off *lookup*, then, you need install nothing at all — the resolver at `https://archive.softwareheritage.org/` will take any SWHID and hand you back the code it names.

Table D.1: Implementations and companion tools for the SWHID standard (ISO/IEC 18670), grouped by job: what *computes* a SWHID — at the command line, or as a library — and what is built for something else (scanning, archiving, resolving).

Name	Language / ecosystem	What it does & how to use it
Compute (and verify) a SWHID at the command line		
`swhid-rs`	Rust (MIT)	*Reference implementation* of ISO/IEC 18670. Computes and verifies core SWHIDs. `cargo install swhid` (add `--features git` for revision support), or grab a pre-built binary from the releases page. `swhid content --file F`, `swhid dir DIR`, `swhid verify PATH SWHID`, `swhid parse "swh:1:…"`. https://github.com/swhid/swhid-rs
`swh.model` (`swh identify`)	Python	The most complete CLI: content, directory, revision *and* snapshot identifiers, plus `--verify`. `pip install 'swh.model[cli]'`, then e.g. `swh identify --type snapshot repo.git/`. https://gitlab.softwareheritage.org/swh/devel/swh-model
Compute a SWHID inside your own program (libraries)		
`swhid` for OCaml	OCaml	A community library (OCamlPro) to parse, build, and compute SWHIDs from OCaml code. `opam install swhid`. https://github.com/OCamlPro/swhid
`swhid-go`	Go	An independent library port of the standard for the Go toolchain — computes SWHIDs in-process. https://github.com/andrew/swhid-go
`swhid` (Ruby)	Ruby	A Ruby gem to generate and parse SWHIDs in-process, with Git-compatible hashing for content, directory, revision, and release. https://github.com/andrew/swhid
Companion tools — built for other jobs, not for obtaining a SWHID		
`swh scanner`	Python	Walks a working tree and tells you which files and directories are *already archived* in Software Heritage; it hashes them to SWHIDs to look them up, but reporting coverage — not handing you an identifier to cite — is its job. `pip install swh.scanner`; `swh scanner scan ./repo`. https://gitlab.softwareheritage.org/swh/devel/swh-scanner
updateswh (Permalinks)	Browser extension	Point-and-click in the browser: a button on GitHub / GitLab / Bitbucket pages that triggers archival and then exposes the artifact's SWHID via the archive's *Permalinks* tab. *Fetches* one the archive computed; computes nothing locally. https://www.softwareheritage.org/browser-extensions/
Archive & `n2t.org` resolvers	Web service (no install)	The reverse direction — turn a SWHID *back* into code: paste it after `https://archive.softwareheritage.org/` to browse the bytes, or resolve via the `n2t.org` name-resolver (`https://n2t.net/`). https://archive.softwareheritage.org/

E

Policy reference: the instrument matrix

Policy is the part of an argument that ages fastest and scatters worst. Across Chapter 7 the instruments arrive one at a time, in the order the prose needs them — a UNESCO recommendation here, an EOSC report there, a French national plan a paragraph later — and that is the right way to make the case, but the wrong way to *act* on it. A reader who wants to write a clause, brief a funder, or check whether their own jurisdiction has already moved should not have to reassemble the list from the body text; and a future drafter of these notes should not have to rediscover, each time, which mandate said exactly what. And so, here, deduplicated into a single matrix, is every policy instrument surfaced in these notes and in the inventories behind them. Read each row as a pointer, not a paraphrase: the mandate column quotes the source *verbatim* where a quotable line exists, and otherwise gives the shortest faithful summary, with the citation and the live link carrying the rest. The list leans European, and French in particular — that is where the convergence happened first and hardest — but the international and national rows show that the direction of travel is the same everywhere: open the code, preserve it, and name it intrinsically.

One matrix, every instrument: jurisdiction, date, mandate, link — the deduplicated landing point for the policy that backs the practice.

Software policy did not appear from nowhere; it is the latest chapter of a twenty-year arc. Open access to the *literature* was declared in Budapest in 2002 [17] and Berlin in 2003 [62]; open access to research *data* was urged by the OECD in 2006 [69] and codified, a decade later, in the FAIR principles [95]; and the push to make open access immediate and mandatory arrived with Plan S in 2018 [22]. Source *code* is simply the next output to travel the same road — from voluntary good practice, to funder expectation, to law — and the matrix below is where it has reached so far.

Table E.1: Deduplicated matrix of policy instruments pointing research software at open licensing, preservation, and intrinsic identification. Mandates are quoted verbatim where a quotable line exists; links resolve the full text.

Instrument	Issuing body	Juris.	Date	Core mandate (verbatim where available)
Rec. on access to research data from public funding [69] https://legalinstruments.oecd.org/en/instruments/OECD-LEGAL-0347	OECD Council	Intl	2006 (rev. 2021)	Access to, and re-use of, publicly funded research data as a default principle.

continued on next page

Table E.1, continued from previous page

Instrument	Issuing body	Juris.	Date	Core mandate
Paris Call on Software Source Code [93] https://en.unesco.org/foss/paris-call-software-source-code	UNESCO, Inria, Software Heritage	Global	Feb 2019	"recognise software source code as a fundamental enabler in all aspects of human endeavour"; build shared infrastructures to preserve it.
Recommendation on Open Science [92] https://unesdoc.unesco.org/ark:/48223/pf0000379949	UNESCO	Global	2021	Open science including open-source code: "source code must be included in the software release".
Rec. 2018/790 on access to & preservation of scientific information [44] https://eur-lex.europa.eu/eli/reco/2018/790/oj	European Commission	EU	25 Apr 2018	Open access to, and long-term preservation of, scientific information and the outputs of publicly funded research.
Directive 2019/790 (Copyright in the DSM) [45] https://eur-lex.europa.eu/eli/dir/2019/790/oj	EU Parliament & Council	EU	2019	Copyright reform bearing on text-and-data mining and the legal status of source-code re-use.
EOSC SIRS report [38] https://data.europa.eu/doi/10.2777/28598	EOSC Executive Board (EC)	EU	Dec 2020	Archive in Software Heritage, use the SWHID; "all research software ... Open Source license by default ... deviations ... properly motivated".
Task Force on quality research software [43] https://eosc.eu/advisory-groups/infrastructures-quality-research-software/	EOSC Association	EU	2021	Recommendations for FAIR, open, and sustainable research software.
FAIRCORE-4EOSC [47] https://faircore4eosc.eu/	Horizon Europe consortium	EU	2022	Builds core EOSC persistent-identifier and metadata components, with Software Heritage among the partners.
Loi pour une République numérique (Loi Lemaire) [73] https://www.legifrance.gouv.fr/loda/id/JORFTEXT000033202746	République française	France	Oct 2016	Open public data and the source code of administrations as a default.
PM circular on data, algorithms & source code [72] https://www.legifrance.gouv.fr/circulaire/id/45162	Premier ministre	France	27 Apr 2021	Public data, algorithms and source code open by default across the State.

continued on next page

Table E.1, continued from previous page

Instrument	Issuing body	Juris.	Date	Core mandate
2nd National Plan for Open Science (PNSO2) [64] https://www.ouvrirlascience.fr/wp-content/uploads/2021/10/Second_French_Plan-for-Open-Science_web.pdf	Ministère de l'ESR	France	6 Jul 2021	Names Software Heritage and the SWHID explicitly; open-source research software by default.
Law 2022-217 (loi 3DS), art. 163 [74] https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000045197395	République française	France	21 Feb 2022	18-month report to Parliament on the production and valorization of public-research software (the 2023 *Rapport Boulet* [16]).
ANR open-science funding guidelines [3] https://anr.fr/en/anrs-role-in-research/commitments/open-science/	Agence nationale de la recherche	France	2023	Funded-project software released under a free license, its source archived in Software Heritage and described in HAL with the ANR grant reference.
Executive Order 14028, §4 [90] https://www.federalregister.gov/d/2021-10460	White House	US	12 May 2021	Software supply-chain security; mandates a software bill of materials (SBOM).
Scientific Information Policy (SPD-41a) [67] https://science.nasa.gov/researchers/science-information-policy/	NASA	US	Dec 2022	Open code, data and publications by default for NASA-funded research.
Model curriculum vitae [26] https://www.dfg.de/en/research_funding/announcements_proposals/2022/info_wissenschaft_22_61	DFG	Germany	Sept 2022	Recognizes research software as a citable scholarly output in evaluation and CVs.

A practical rule of thumb for reading the matrix: the *global* rows (OECD, the Paris Call, UNESCO) set the principle; the *European* rows turn it into operational guidance — SIRS is the one to quote when someone asks for chapter and verse, because it names Software Heritage and the SWHID outright; and the *national* rows are what a researcher or a funder points to in their own jurisdiction. Strip the acronyms from any of them and the same instruction remains, the one this whole note has been arguing for: open the code, preserve it, and identify it with an intrinsic, standard name.

F

Typesetting software citations in venue classes

These notes are set in the plain `report` class, where loading `biblatex-software` takes two lines. Submitting to a conference or a journal is less forgiving: the major venue classes ship their own bibliography machinery. The Hands-on box in Chapter 5 asks you to drop your software entries into a real venue template; this appendix is the part you would otherwise have to reverse-engineer.

One rule holds everywhere: build with `biber`, never with `bibtex`. Beyond that there are two situations, and telling them apart saves a great deal of confusion. Either the class knows nothing about software entries — in which case you load `biblatex` with `datamodel=software` and then `software-biblatex` yourself, as for `IEEEtran` below — or the class *already integrates* `biblatex-software`, in which case you must *not* load it again. `acmart` is now the second kind, which is why it needs less configuration than folklore suggests, not more.

F.1 ACM: `acmart`

This used to be the awkward case, and much advice still found online says so: `acmart` preloads `natbib`, which collides with `biblatex`, and for years the remedy was a hand-rolled incantation with macros to unbind. That is no longer necessary. Since version 1.84 the class ships a rewritten `biblatex` layer and two bibliography styles of its own — `acmnumeric` and `acmauthoryear` — which reproduce the behaviour of the traditional `ACM-Reference-Format.bst`. Better still for our purposes, those styles *already integrate* `biblatex-software`: `acmdatamodel.dbx` pulls in the software data model, and `acmnumeric.bbx` pulls in the software bibliography driver and switches `swhid` on for you.

The consequence is that you do *less* here than in a plain document, not more. You do not pass `datamodel=software`, and you do not load `software-biblatex` yourself — the class does both. What you must do is turn `natbib` off and pick an ACM style:

```
\documentclass[sigconf,natbib=false]{acmart}

\RequirePackage[
  datamodel=acmdatamodel,   % pulls in the biblatex-software data model
  style=acmnumeric,         % or acmauthoryear, if the venue requires it
  ]{biblatex}

\addbibresource{yourbib.bib}
```

Then put `\printbibliography` where the references belong. If you are converting an existing ACM paper, delete any `\citestyle{acmauthoryear}` line as well: it belongs to the `natbib` world you have just left.

`biblatex-software` must still be installed — the class *uses* it rather than *replacing* it — but you no longer name it anywhere. All four entry types then render inside the ACM reference list, each carrying its SWHID: a project as `[SW]`, a release as `[SW Rel.]`, a module as

`[SW Mod.]`, and a code fragment as `[SW exc.]`, with the `crossref` relations printed as "*from*" and "*part of*" just as they are in these notes.

One caveat, from ACM itself

The class documentation is blunt about the status of this path: the `biblatex` styles are "highly experimental" and are *not* supported by TAPS, the ACM production system that builds the final camera-ready version. Read that as: excellent for preprints, for arXiv, and for your own archival copy; but check with the venue before relying on it for camera-ready, and be ready to fall back to `ACM-Reference-Format.bst` — losing the software entry types in the process — if production insists.

F.2 IEEE: `IEEEtran`

`IEEEtran` is the first kind of class: it knows nothing about software entries, but it also does not preload a competing citation package, so `biblatex` loads in the ordinary way and you supply the software support yourself.

```
\documentclass[conference]{IEEEtran}

\usepackage[backend=biber, style=ieee, datamodel=software]{biblatex}
\usepackage{software-biblatex}

\ExecuteBibliographyOptions{
  swhid=true, swlabels=true, vcs=true, license=true}

\addbibresource{yourbib.bib}
```

That is the whole of it. All four entry types render inside the IEEE numbered reference list, each carrying its SWHID, and the `crossref` chains behave as they do everywhere else — a code fragment prints as "… *from* [1]" under its parent release rather than repeating the project's authors, repository, and license.

F.3 Other classes

For any class not covered here, first ask which of the two kinds it is. If it already loads `natbib`, expect a collision and look for a `natbib=false` class option before reaching for macro surgery. If it ships its own `biblatex` styles, check whether they already input `software.dbx` and `software.bbx` — if they do, load neither the software data model nor the style yourself. And if the venue insists on `bibtex` and will not accept `biber`, the software entry types are simply not available: fall back to `@misc` and put the SWHID in the `howpublished` field, which is lossy, but at least keeps a verifiable identifier in the record.

G

A model artifact-evaluation section

Chapter 6 argues that artifact evaluation is where an intrinsic identifier earns its keep: a reviewer can check an artifact offline, in under a minute, without asking anyone's permission. This appendix is the operational form of that argument — text a programme chair can lift into a call for artifacts, and the short procedures behind it.

It is written to be *additive*. Every venue that runs artifact evaluation already has a deposit rule, a badge vocabulary, and a reviewing workload it is trying not to increase. Nothing below removes a DOI, displaces Zenodo, or asks a committee to retrain its reviewers. It fills a slot that most guidelines already contain and few can currently fill.

It states *what* must be provided and *what* is checked — not the incantations. Tools drift faster than principles, so the operational detail lives in two maintained pages outside this book. Software Heritage keeps the general checklist for archiving and referencing code — what a public repository should contain, every archival route, and how to pull an identifier off the *Permalinks* tab — at `https://www.softwareheritage.org/how-to-archive-reference-code/`; that is the address to put in a call for artifacts. What it does not cover, because it is specific to evaluation, is carried by a companion HOWTO versioned alongside these notes (`docs/HOWTO-artifact-swhid.md`, `https://github.com/rdicosmo/source-code-of-science/blob/main/docs/HOWTO-artifact-swhid.md`): the clean-tree invariant, the working-copy hazard, the self-checks, submodules and LFS, and what to do when two parties arrive at different numbers. Both are referenced by address rather than pinned to a revision — they are expected to move as tooling does, which is the entire point of keeping them out of this appendix.

G.1 The slot that is already there

The ACM Artifact Review and Badging policy asks, for the *Artifacts Available* badge, that authors provide "a DOI or link to this repository along with a unique identifier for the object". Two slots. The first is routinely filled with a DOI; the second is usually left as a bare repository URL — which is a location, not an identifier of an object. Many venues inherit this sentence verbatim, and a good number have quietly dropped its second half.

Filling that second slot with a SWHID gives the record something it has never had: a claim about the artifact that anyone can check rather than trust — and, as Section G.7 argues, that nobody at review time then needs to. Everything else in this appendix follows from that one substitution.

Most guidelines already ask for an identifier of the object. Few can name one.

G.2 The common case: archive, then name the version

Most venues run artifact evaluation *after* the paper has been accepted. The authors are known to the committee, and the artifact lives in a public repository on a public forge. In

that setting — which is the common one, and the one to describe first — the entire author-side procedure is three steps, and involves no local tooling at all.

1. **Make sure the version is archived.** The least error-prone route is the UPDATESWH browser extension (Firefox, Chrome, Edge; https://www.softwareheritage.org/browser-extensions/). On any repository page of a supported forge it shows a coloured tab — green if the archive is already up to date, yellow if it has fallen behind — and one click either opens the archived version or requests archival of the page you are looking at. Nothing is typed, so nothing can be mistyped, and you can see whether a save is needed at all. Failing that, paste the repository URL by hand at https://save.softwareheritage.org (no account needed), or wire a forge webhook to the archive — Software Heritage recommends firing it on branch, tag or release creation rather than on every push. All the routes, and what a well-formed repository should contain before you archive it, are on https://www.softwareheritage.org/how-to-archive-reference-code/.
2. **Take the identifier from the archive.** Browse to the exact version, open the *Permalinks* tab, enable the `origin`, `visit` and `anchor` qualifiers, and copy the *qualified* `dir` SWHID.
3. **Hand it over.** Put that string in the artifact appendix, or give it to the chair or editor with the submission.

That is all. Nothing was computed, nothing was installed, nothing was deposited a second time. The reviewer pastes the identifier into any browser and is looking at the exact tree the authors named, together with the repository it came from, the visit that captured it and the commit it belongs to (Figure G.1 a). The artifact has also, as a side effect, been preserved — which is the thing the *Available* badge was asking for in the first place.

G.3 When review is blind

In the common case the author computes nothing: the archive already holds the number.

A qualified identifier names the origin, so it cannot go into an anonymous submission as it stands. But the qualifiers are exactly that — a suffix:

`swh:1:dir:2dc0f462d191524530f5612d2935851505af41dd`
core — what goes into an anonymized submission

`swh:1:dir:2dc0f462d191524530f5612d2935851505af41dd;origin=…;anchor=…`
qualified — what goes into the camera-ready

The hash is the same in both. It is computed from the content of the tree and from nothing else: not from an author, a date, a repository name or a commit. A *core* identifier therefore discloses none of them, and resolving it in the archive shows the contents of the directory and nothing whatever besides (Figure G.1 b). The ordinary interfaces offer no route back either: the directory endpoint returns the tree's entries and nothing else, and the graph traversal that could answer "which origins contain this tree?" is not anonymously available. That is a practical obstacle rather than a guarantee — as with a submitted tarball, a reviewer who already suspects a particular repository can always fetch it and compare. What the identifier *itself* discloses is nothing, and that is what the venue is relying on.

So the blind procedure is the common one with its last step trimmed. Archive as usual, take the qualified identifier, and give the reviewers only the part before the first semicolon. On acceptance, restore the suffix in the camera-ready copy. The hash the committee checked and the hash in the published paper are then the same string — the difference between them is only how much context the reader is offered.

Qualifiers are a suffix. Cut them for review, restore them at camera-ready: same hash.

One residual disclosure is worth naming, because it is not quite zero. Requesting archival adds a row — repository URL, date, status — to the archive's publicly browsable list of save requests. That row says a repository was archived on a date; it says nothing about a paper, a venue or a submission, and a repository already public on a forge gives away no more by being archived than by existing.

(a) the qualified identifier: the tree, in its context



(b) the core identifier: the same tree, and nothing else

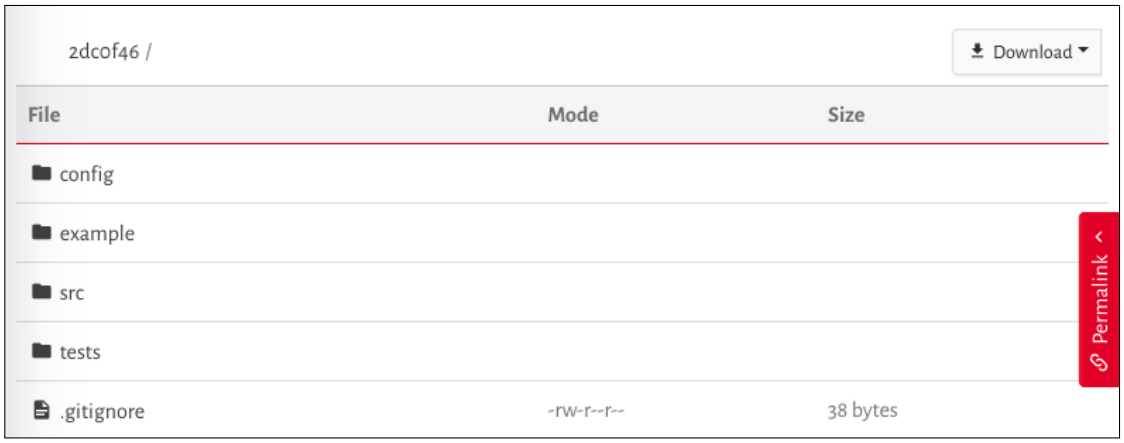


Figure G.1: What the archive displays for the two forms of one identifier. (**a**) the qualified SWHID resolves in context: the origin `github.com/rdicosmo/parmap`, the visit that captured it, the revision, its author, its date and its commit message. (**b**) the same hash without qualifiers resolves to the directory itself — no origin, no author, no date, no message. The breadcrumb `2dc0f46 /` is identical in both, because it is the same tree; everything around it is what the qualifiers add. This is what makes a core SWHID usable in a double-blind submission.

Where a venue's threat model is stricter than that, invert the order: leave the artifact unarchived until acceptance and have authors compute the core identifier locally, as in Section G.4.

G.4 If you compute it yourself

Not every artifact sits in a public git repository, and some venues will want a number before anything is archived. Then the author computes it, and one invariant carries the whole workflow.

The identifier always names a *clean tree*: for a git-managed artifact, the committed tree of a precise commit; for anything else, the content of the exact bundle submitted, unpacked into an empty directory. Nobody — author, reviewer or committee — ever hashes a live working copy: a checkout contains `.git/`, build output and editor droppings, so its hash matches neither the archive nor the reviewer's tarball, and is not even stable over time on

an unchanged commit. For a git artifact the clean tree's `dir` SWHID is, bit for bit, the hash that `git rev-parse <commit>^{tree}` already prints — which is also exactly what Software Heritage stores.

Hence one recipe, current at press time (maintained version, variants and troubleshooting in the HOWTO; the local tools in Appendix A):

```
d=$(mktemp -d) && git archive <commit> | tar -x -C "$d"
swh identify --no-filename --type directory "$d"
git rev-parse <commit>^{tree}   # self-check: the two hashes must be equal
```

If the self-check fails, the tree is not clean: diagnose (the HOWTO has the checklist) rather than submit the number. Where a locally computed value and an archived one both exist they must coincide, and that coincidence is itself the end-to-end check of the procedure.

G.5 Model text for a call for artifacts

Artifact identification and archival — text to adapt

Identifying the artifact. Authors must identify the artifact by its `dir` SWHID (*SoftWare Hash IDentifier*, ISO/IEC 18670): an identifier derived from the content of the artifact tree alone, which changes whenever the artifact changes. Request archival of the repository — history included — and take the identifier of the exact version from the archive's *Permalinks* tab. The procedure is on Software Heritage's own page, https://www.softwareheritage.org/how-to-archive-reference-code/: in short, one click on the UPDATESWH browser extension, or the repository URL pasted at https://save.softwareheritage.org; no account is needed for either. Report the identifier with the submission. A revised artifact gets a new SWHID, reported with the revision; earlier identifiers remain valid names for the earlier states.

Under double-blind review. Report only the *core* identifier — the part before the first semicolon, `swh:1:dir:…` — omitting the `origin` and `anchor` qualifiers. The core identifier carries no author, origin, date or commit metadata, and resolves in the archive to the artifact's contents alone, so it is safe in an anonymized submission. At camera-ready, add the qualifiers back around the identical core hash.

In the camera-ready paper. Cite the *qualified* SWHID of the evaluated version alongside the DOI of any deposit this venue already requires (for example Zenodo). Its core hash must equal the identifier recorded at the close of evaluation; verify that equality before submitting. If the artifact was improved after evaluation, list both identifiers, labelled "as evaluated" and "as published".

Artifacts not in a public repository. Where there is no repository to archive, compute the identifier from the exact bundle submitted, unpacked into an empty directory, following the HOWTO accompanying this call — never from a live working copy. The HOWTO includes a one-line self-check; if it fails, diagnose before reporting a number.

What reviewers do. Reviewers resolve the identifier, download the artifact from the archive and evaluate it. *There is no hash for them to check*: the archive retrieves the object by its hash and cannot return a different one, so re-deriving the identifier they used to fetch the artifact would test nothing. Where authors supply a bundle rather than an identifier, reviewers compute the identifier from that bundle per the HOWTO, so that what was evaluated acquires a name. Should two numbers ever disagree, the HOWTO's diagnostic checklist comes first — the commonest cause is procedural, not a changed artifact — and only what survives diagnosis is reported.

What the committee does, once. At camera-ready, confirm that the core hash of the qualified SWHID in the paper equals the identifier recorded at the close of evaluation. This is the check that makes the badge mean something, and it is one line of work per paper.

Scope. This covers **source code**, including build recipes — Dockerfiles, lockfiles, provisioning and experiment scripts, pinned under the same identifier as the artifact tree. Datasets, container *images* and large binaries follow this venue's existing instructions; Software Heritage is a source-code archive, not a data archive.

G.6 The author's checklist

1. Develop wherever you already develop; nothing changes about your forge, your CI or your packaging.
2. Archive the version you are submitting — one click in UPDATESWH, or Save Code Now by hand, or a forge webhook for good; see https://www.softwareheritage.org/how-to-archive-reference-code/ — and copy the qualified SWHID from the *Permalinks* tab.
3. Report it with the submission: the whole string normally, the core hash alone if the venue is double-blind.
4. At each revision during evaluation, report the new one in a line: "revised tree: `swh:1:dir:…`".
5. In the camera-ready, cite the qualified SWHID beside the DOI of any deposit, and check its core hash against the one recorded at the close of evaluation.

G.7 For reviewers: there is nothing to check

A reviewer handed a SWHID does not verify it. They paste it into the archive, press *Download* — the button visible in both panels of Figure G.1 — and evaluate the code that comes back. That is the whole of it. Their time goes where it should: into running the artifact.

It is worth saying plainly why no check is needed, because committees keep writing one into their instructions. The archive retrieves the object *by* its hash. Given `swh:1:dir:2dc0f462…` it can return that tree or nothing at all; there is no way for it to return some other tree under that name. Asking a reviewer to download an artifact and then confirm that it hashes to the identifier they used to fetch it tests their `tar` command, not the artifact. The guarantee was spent the moment the identifier resolved.

So what does the identifier buy a reviewer? Not a check they perform — a check they no longer have to. What they evaluated now has a name, and *later* anyone can confirm that the version in the published paper is that same thing. The verification is real; it simply does not happen at the reviewer's desk.

Resolve, download, evaluate. The archive cannot hand back a tree other than the one you asked for.

The two cases that do involve a computation. Neither is routine, and neither belongs in a reviewing form as a default step.

- **The authors supplied a bundle instead of an identifier.** Then nothing has a name yet, and hashing is how one is made: unpack into a fresh, empty directory and compute the identifier per the HOWTO. The same path serves a reviewer who would rather not consult a service at all — it needs no account and no network.
- **The chair, at camera-ready.** One line, once: check that the core hash of the qualified SWHID printed in the paper equals the identifier recorded at the close of evaluation. That compares an archive-held object against an independent committee record, so unlike the reviewer's non-check it really is one — and it is the step that makes "the artifact we badged" and "the artifact a reader can download" the same claim.

If two numbers ever do disagree, the inequality proves exactly one thing: two different trees were hashed. It says nothing about how different they are, or on whose side the error lies, and in practice the commonest cause is procedural — someone hashed a live working copy. Run the HOWTO's diagnostics first and report only what survives them, the way you

would report a build failure: the identifier obtained and the procedure used, never an accusation.

G.8 How it maps onto existing badges

Badge	What changes
Artifacts Available	The identifier slot is filled with a verifiable identifier. The DOI keeps its role: citation metadata, discovery, a landing page.
Artifacts Evaluated	The committee can state *which* artifact it evaluated, and the published version can be shown to be that one.
Results Reproduced	Unaffected in substance, but the reproduction now names the exact tree it reproduced from.

G.9 The questions a chair will ask

We already require a DOI. Why a second identifier? They answer different questions. A DOI names a deposit and certifies a landing page; the deposit behind it can be replaced without the DOI changing, and nothing in the identifier tells you it did. A SWHID *is* the content. That is why both belong in the paper, not one instead of the other.

Is Software Heritage durable enough to rely on? The identifier does not depend on the archive's survival: it can be recomputed from any copy of the content, by anyone, indefinitely. This is the practical difference between an intrinsic identifier and a registry entry — the registry can fail; arithmetic cannot.

Does this cover datasets, containers and models? No. It identifies source code, and build recipes pinned with it. Datasets and images keep the arrangements your venue already has. A proposal that claimed otherwise would be easy to refute and would deserve to be.

What does it cost the committee? One sentence in the call for artifacts and a link to Software Heritage's how-to. No new infrastructure, no accounts and no service dependency. It costs the reviewers nothing at all — it removes a step rather than adding one, since the artifact now arrives from the archive under the name it will keep. The only new work is the chair's one-line check at camera-ready.

H

Archie, and how these illustrations were made

A note that argues for archiving, referencing and reproducing research artefacts ought to say where its own artefacts came from. Ten drawings punctuate these pages: a frontispiece and nine vignettes, one at the head of each chapter and one over the appendices. This appendix records what the character is, why it is an elephant, how the images were produced, and what went wrong on the way.

Disclosure

All ten illustrations are **AI-generated**, using Google Gemini's image generation, directed and selected by the author, with light local post-processing (greyscale conversion and white-point normalisation) in ImageMagick. No human illustrator was involved. The print copies are in `figures/illustrations/` in the source tree of these notes, and are therefore archived and named by the same SWHID as the text.

H.1 Why a mascot

The model is Knuth's *TeXbook* and *METAFONTbook*, whose whimsical pen-and-ink vignettes, a friendly recurring lion among them, punctuate a dense technical text without cartoonifying it. The aim here is the same: a small warm drawing at each chapter head, and one recurring character, so that the set reads as a series rather than as a pile of clip art.

The visual style was specified by *attributes*: line weight, cross-hatching, vignette framing, whimsy. It was deliberately *not* specified by naming a living illustrator, for two reasons. Image models reproduce an aesthetic more reliably from attributes than from a name, and naming a working artist to imitate their style is a thing one should not do.

H.2 Archie, the elephant archivist

The character. A gentle, rounded elephant who works as the archive's librarian: small round spectacles, a simple archivist's apron with one breast pocket, no lettering anywhere on the figure. Warm, unhurried, dignified. He appears in every plate, sometimes alone, once multiplied into a network of equal peers.

Why an elephant. The choice is an argument in itself, and it is the book's argument.

An elephant never forgets. The thesis of these notes is long memory: a universal archive that keeps the world's source code, with its full development history, for the long term. The proverb does the work of a paragraph.

Large, calm, dependable. The archive is a patient, non-profit, multi-decade commitment, not a platform racing to a funding round. A big unhurried animal reads as durable custodianship rather than as move-fast development.

Expressive, and easy to pose. Trunk, ears and spectacles allow a wide range of gentle business while staying dignified enough for a scholarly text. A mascot that cannot act out the chapter's mechanism is decoration, and decoration was not the point.

Why "Archie". The name reads instantly as *archivist* and *archive*, which is the character's job and the book's subject in five letters. It also carries a quiet nod to computing history: Archie, built at McGill University in 1990, was the first internet search engine, the first tool made to *find* what had been stored. It works unchanged in English, French and Italian, which matters for a note that travels, and it is short enough for a caption. Above all it names a *character* rather than a concept: a recurring mascot needs a name a reader can use.

Three other names were considered and rejected, and the reasons are instructive.

Table H.1: Names considered for the mascot, and why each was set aside.

Name	For	Against
Merkle	The tree he tends *is* a Merkle DAG; the name would teach the structure on every appearance.	Eponymises a living researcher, and sits oddly on a cartoon animal.
Trunk	A triple pun: the elephant's trunk, the tree trunk, the version-control trunk. The wittiest option.	Names an object rather than a person, and reads awkwardly as a character name.
Mnemo	From Mnemosyne; leans directly on the long-memory idea.	Abstract, and less pronounceable across the three languages.

A mascot earns its place when it carries the argument, not when it decorates it.

One standing caution. A rounded elephant in spectacles and an apron sits visually close to Babar. The art direction avoids names and costumes in that register, so that an incidental family resemblance does not harden into an implied reference to a character still under copyright. The waistcoat and necktie in the Chapter 4 plate are the one departure from the apron, kept deliberately after review.

A small supporting cast appears where multiplicity is the point, and nowhere else: worker mice with satchels in Chapter 3, the listers and loaders that fill the archive; two identical round creatures at a shared drawer in the same plate, whose *sameness* is the whole joke, because identical content is stored once; and in Chapter 7 several small keepers, all the same size as each other, each tending an archive of their own.

H.3 How the images were made

The tool: Google Gemini image generation. It was chosen over the alternatives for one specific property, *character consistency across a series*. Its reference-image editing, "here is the character, now draw him doing this", is what keeps Archie recognisably the same elephant across ten drawings. The trade-off is weaker literal adherence to dense compositional prose, which is exactly where the failures in the next section came from.

The consistency chain. The images were not generated independently. They were chained:

1. Generate one vignette first with no reference, and regenerate it until the design is right. It establishes the canonical Archie, and becomes the reference sheet.
2. For every later image, *attach that reference*, instruct the model to keep the character identical, and describe only the new scene. Attaching the image is what carries the character; words alone do not.
3. On drift, re-attach the best prior image: "match this elephant exactly, same face, spectacles, apron; change only the scene."

4. State the aspect ratio in words ("landscape 3:2 composition"), which the model honours.
5. Regenerate on stray lettering rather than keeping it. Garbled characters read as an error in a scholarly book.

A fixed style anchor was prepended to every scene prompt, and never varied:

> **Style:** black-and-white **pen-and-ink line drawing**, confident hand-drawn linework of varying weight, light **cross-hatching** for shading, generous white space, no fill tones or grey washes beyond hatching. A small, warm **vignette**, a single self-contained little scene that fades softly at its edges into the white of the page, *not* a full bordered illustration and *not* a busy full scene. The mood is whimsical, gentle and good-humoured, in the tradition of a mid-twentieth-century technical-book chapter-head drawing. **No colour. No photorealism, no digital gloss, no gradients, no 3D rendering. No text, no lettering, no numbers, no logos anywhere in the image.** Recurring character: a gentle, rounded **elephant archivist** with small round spectacles and a simple archivist's apron, drawn the same way each time.

H.4 What each plate says

Each vignette states its chapter's mechanism, not its mood.

Frontispiece. A tree rooted in an open book, its leaves made of files, with Archie steadying a branch: the science grows out of its source.

Front matter. Three pedestals bearing a bound book, a specimen jar and a tree-shaped bundle of files, the trunk gesturing to the third as to an old friend left out of things. Publications, data and software are three research outputs of equal standing.

Chapter 1. The journal is shelved on solid ground while the code tumbles off a crumbling ledge into loose leaves, just past the reach of the outstretched trunk. One output is preserved by an apparatus built for it; the other is not.

Chapter 2. A sealed jar with a pinned specimen, held up against a potted sapling that sprouts, reaches tendrils to its neighbours and overgrows its pot. Data holds still; software does not.

Chapter 3. A tree built of shelves and drawers, files at the tips, gathered into branches, gathered into one trunk, with one branch growing *into* the branch above it, because the archive is a directed acyclic graph and a subtree can have several parents.

Chapter 4. The same bundle named twice: a paper luggage tag on a string running off to a distant registry desk, eyed with scepticism, against a fingerprint pressed into the object's own surface and read on the spot through a lens.

Chapter 5. The right hand writes in the open citation catalogue; the left holds a lens flat over a completely blank book, and the fingerprint appears only inside the glass. The identifier is computed from the content and then transcribed, never stamped on.

Chapter 6. A fair examiner at a desk, matching a submitted bundle's mark against the claimed mark on a slip, a blank ribboned rosette ready to award, a folding screen hiding who sent it. Verification needs no authority, and survives double-blind review.

Chapter 7. A globe of plain line-work continents, several equal keepers tending their own patch, their sites linked by faint dotted arcs, one of them offering a sapling toward the reader. A mirror network has no single point of failure and no hero.

Appendices. Archie in a cook's apron, spoon behind one ear, pulling a card from a recipe box: what follows is a cookbook, recipes to follow rather than arguments to weigh. It is the one image that deliberately contains lettering, because the words are the metaphor.

H.5 What went wrong, and what fixed it

Three failures are worth recording, because they generalise beyond drawings.

Name the delta; never re-describe the scene. A prompt that restates a scene is read as a request to regenerate it, not to edit it. Every style clause repeated for safety is re-interpreted from scratch, and that, rather than poor taste, produced every drift observed here: sepia backgrounds, an engraving style, four elephants where one was wanted. The reliable form is a preservation clause and one sentence of change, naming the region by landmark: "keep this exact image unchanged; change only …".

A drawing can assert the opposite of the text. An early version of the Chapter 5 plate had Archie *stamping* an identifier onto a citation slip. That is the extrinsic gesture Chapter 4 argues against, drawn at the head of the chapter that depends on the distinction, and it was discarded for that reason rather than retouched. In the same plate the model kept printing the fingerprint on the book cover as well as in the lens, which quietly restores the very claim the picture exists to refute. An illustration is part of the argument and has to be reviewed as such.

Judge the artefact, not its description. The Chapter 4 plate carried a small hanging tag reading `LUGGAGE LABL`: legible, unintended, misspelled, and forbidden by every prompt in the set. It survived an author-side review conducted from the prompts, the thumbnails and the file names, and was caught only when a reviewer opened each image at full resolution with no expectation of what it should contain. The lettering was then removed by a local edit of a few thousand pixels rather than by regeneration, which would have risked character drift for a defect confined to one tag.

That last failure is this book's own thesis turned back on its own production. An identifier, a description, a record of intent: none of them is the object. The reason to name source code by a hash of its content is precisely that the hash is computed from what is there, and not from what someone meant to put there.

A name, a prompt or a filename describes an artefact. Only the artefact is evidence.

How to cite these notes

These notes are revised: chapters gain material, the evidence is refreshed, and the version number on the title page moves. A reference to them therefore has to answer the same question that Chapter 5 puts to any research artefact — does it name *the work*, which will keep changing, or *one exact version*, which will not? Both are legitimate. What is not legitimate is a reference that leaves the reader unable to tell which was meant.

The short answer

Cite the HAL record **without** a version number:

https://hal.science/hal-05734828

That address always resolves to the most recent version. A reader who follows it in five years gets whatever these notes have become by then, which is what a reference to a living course note should do.

I.1 Citing the work

This is the form to use in a reading list, a syllabus, a policy document, or a paper that recommends the notes as a whole. In plain text:

> Roberto Di Cosmo. *The Source Code of Science*. Doctoral. Pise (Italia), Italy. 2026. https://hal.science/hal-05734828

and in BibTEX, adapted from the entry HAL generates, with the version-specific fields left out on purpose:

```
@unpublished{dicosmo:hal-05734828,
  TITLE   = {{The Source Code of Science}},
  AUTHOR  = {Di Cosmo, Roberto},
  URL     = {https://hal.science/hal-05734828},
  TYPE    = {Doctoral},
  ADDRESS = {Pise (Italia), Italy},
  YEAR    = {2026},
  HAL_ID  = {hal-05734828},
}
```

The `URL` carries no `v` suffix, and that is the whole point: HAL keeps every version, and the unsuffixed identifier is the one that follows the work forward.

I.2 Citing one exact version

This is the form to use when the reference supports a claim — when you quote a page, rely on a figure, or say that these notes recommend some specific thing. A later version may say something else, and a reader checking your claim must land on the text you actually read.

Three identifiers name an exact version, at three different levels:

The HAL record, versioned. Append the version suffix HAL assigns, as in `https://hal.science/hal-05734828v1`. Each deposit is a fixed document with a fixed cover page. HAL's own citation block for that version carries the page count and the version note, so take it from the record rather than retyping it.

The release. Every version is tagged in the source repository and published with its PDF attached: `https://github.com/rdicosmo/source-code-of-science/releases`. The tag (`v1.4`, `v1.5`, ...) is the human-readable name of the version.

The source tree. The LaTeX sources, figures and illustrations of each released version are archived in Software Heritage and named by a qualified SWHID. This is the identifier to use when what you need is the material these notes are built from, rather than a rendering of them. Version 1.5, for example, is named by

```
swh:1:dir:769f7afe6cc52fe86bd63d35cb68e7513874e2fd;origin=https://github.c
om/rdicosmo/source-code-of-science;visit=swh:1:snp:b8715018e8f7ffab19b3f8a8
8fcd036fd2020ca8;anchor=swh:1:rev:7f22576697b3ee89eb96ac11839f724fe7db91b0
```

The part before the first semicolon, `swh:1:dir:769f7afe…`, is the identifier proper: it is computed from the content, so it can be recomputed and verified offline by the method of Chapter 4. What follows it is context that helps a reader find the object again: where it came from, which visit saw it, and which revision it hangs from. Each released version has its own such identifier, recorded in that version's HAL record under related resources and in the release notes.

In BibTeX, the versioned form differs from the one above in two fields:

```
URL         = {https://hal.science/hal-05734828v1},
HAL_VERSION = {v1},
```

I.3 Which document is which

One caution, and it is the note's own lesson turned on itself. The PDF served by HAL is *not* byte-identical to the PDF attached to the corresponding release: HAL prepends a cover page, which makes it a different digital object even though it is the same document to a reader. A byte-stable copy of each release, exactly as built, is kept at `https://dicosmo.org/Articles/2026-source-code-of-science.pdf`.

The SWHID recorded with each version names the *source tree*, not any of those PDFs. This is not an oversight. A rendered document has no canonical byte form — it can be rebuilt, converted, or reprinted with a cover page, and remain the same work — whereas its source does, which is precisely why intrinsic identifiers fit code and fit sources, and sit awkwardly on a printed artefact.

The example above names version 1.5. The version you are holding names itself as well: Appendix J prints the identifier of the sources this copy was built from, and explains how the build derives it. The distinction matters. A document cannot carry an intrinsic identifier of *itself*, since writing the identifier in changes the bytes it names. It can carry the identifier of the *sources it was built from*, which is a different object, and the one you want in a citation anyway.

Cite the unsuffixed identifier for the work, the versioned one for a claim. A reference that does not say which it means is not doing its job.

J

How these notes name their own sources

Three facts printed in this copy were derived by the build rather than typed in by anyone: the version on the title page, the date beside it, and the intrinsic identifier of the sources the copy was built from. Each is read out of the repository at build time. This appendix explains how, because the mechanism transfers to any document whose sources live in a repository.

J.1 A key observation

An intrinsic identifier is a hash of content. Write a document's identifier into that document, and the content changes, so the hash changes, so the identifier just written is wrong. You cannot embed a digest of bytes that do not exist until the digest is embedded.

That argument holds, and it rules out a file that names itself. Two properties put the mechanism used here outside its reach.

The identifier names the sources. What is named is the directory of LaTeX files, figures and illustrations that the build consumes. The PDF is what the build produces from them. Recording the identity of your input in your output is ordinary provenance.

The stamp stays out of the sources. The identifier is written to `swhid.tex`, which is listed in `.gitignore` and therefore absent from `git archive HEAD`. That is the tree the identifier is computed over, so stamping the document leaves it untouched.

Drop either property and the problem comes back.

J.2 The mechanism

A SWHID is computed from content, so the build derives it locally and never asks the archive for it. Two commands in `.latexmkrc` do the work:

```
git archive HEAD | tar -x -C $tmp       # the exact tree that gets archived
swh identify --type directory $tmp      # -> swh:1:dir:...
```

The result, with the origin and the anchoring revision appended, goes into the git-ignored `swhid.tex`, which the note reads and prints. Two things follow. Anyone who checks out this revision and runs `make` derives the same identifier and builds the same page. And the printed identifier is falsifiable: compare it with what the archive reports for this revision, and a mismatch tells you something is wrong without requiring you to trust the author.

The version and the date come from the same place. The version is read from the annotated release tag with `git describe`, so the tag is the single source of truth and no file has to be kept in sync with it by hand. The date is the date of the commit, not the day the build happened, so two people building this revision a year apart still get the same title

page. Between releases the derived version says so, as in `1.5-3-g0a1b2c3d`, and a tree with uncommitted changes adds a trailing `+`.

One ordering constraint falls out of this, and it is worth stating because it is easy to get wrong: the tag has to exist before the release is built. Tagging does not touch the tree, so the identifier is unaffected, but a build made before the tag cannot know the version.

The build stamps a clean working tree only. Uncommitted changes correspond to no commit and to nothing archived, so a note built from them carries no identifier and says so on this page.

J.3 This build

The version on the title page, **1.6**, was read from the annotated release tag by `git describe`. The date beside it, **September 24, 2026**, is the date of that tagged commit, recovered with `git log -1`. And the sources are named by:

```
swh:1:dir:8245d19766173d59c2847826ade1128f4f4fd8b1
;origin=https://github.com/rdicosmo/source-code-of-science
;anchor=swh:1:rev:5b7778c7bd64fc167fa85c0c54de9c571a91a8cf
```

The first line is the identifier, computed from the content. The two qualifiers say where these sources are published and which revision they were taken from. The link resolves to the directory in the archive.

J.4 Checking it yourself

Clone the repository, check out the tag matching the version on the title page, and run:

```
make swhid
```

It prints the identifier derived on your own machine, offline. Three numbers should agree: the one printed above, the one your checkout produces, and the one the archive reports for that revision.

A difference usually means you are on a later commit. The identifier names the tree you have, so a tree that has moved on hashes differently; check out the release tag to reproduce the number printed here. An earlier version of this machinery told readers to compute from `HEAD`, which drifts past the release, and that produced exactly this confusion.

J.5 What the stamp leaves out

The stamp carries the identifier, the origin and the anchor. It omits the `visit` qualifier, which names the archive crawl that observed these sources. A visit records when a crawler passed, so it cannot be derived from your bytes, and including it would cost the property this appendix rests on: that you can recompute everything printed here without asking anyone. The fully qualified form, `visit` included, is recorded in the HAL deposit and in the release notes, both written after the archiving has happened.

All three derived values are read from the repository, so they need the repository. A plain download of the archived source directory carries the files without their history, and a build from it reports no version, no commit date and no identifier. Recover the history first, by cloning the origin or by asking the archive's vault for this revision as a git bundle, and the three values come back.

A document can carry the identifier of the sources it was built from, provided the build derives the stamp and keeps it out of the tree it names.

References

[1] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. "Building the Universal Archive of Source Code". In: *Communications of the ACM* 61.10 (2018), pp. 29–31. DOI: 10.1145/3183558.

[2] ACM. *Artifact Review and Badging — Version 1.1*. Association for Computing Machinery. Aug. 2020. URL: https://www.acm.org/publications/policies/artifact-review-and-badging-current.

[3] Agence Nationale de la Recherche. *Open Science*. 2023. URL: https://anr.fr/en/anrs-role-in-research/commitments/open-science/.

[4] Pierre Alliez et al. "Attributing and Referencing (Research) Software: Best Practices and Outlook From Inria". In: *Computing in Science & Engineering* 22.1 (2020), pp. 39–52. DOI: 10.1109/MCSE.2019.2949413.

[5] American Economic Association. *Data and Code Availability Policy and the Office of the AEA Data Editor*. Mandatory pre-publication code-and-data reproducibility check for all AEA journals since the July 2019 policy; Data Editor Lars Vilhuber, Cornell University. 2019. URL: https://aeadataeditor.github.io/ (visited on 06/27/2026).

[6] David Anderson. "The Digital Dark Age". In: *Communications of the ACM* 58.12 (2015), pp. 20–23. DOI: 10.1145/2842646.

[7] William Y. Arms. "Uniform resource names: handles, PURLs, and digital object identifiers". In: *Communications of the ACM* 44.5 (2001), p. 68. DOI: 10.1145/374308.375358.

[8] Atlassian. *Sunsetting Mercurial Support in Bitbucket*. Mercurial repositories set read-only on 1 July 2020 and disabled thereafter. 2019. URL: https://bitbucket.org/blog/sunsetting-mercurial-support-in-bitbucket.

[9] Ben Balter. *Open Source License Usage on GitHub.com*. GitHub. 2015. URL: https://github.blog/open-source/open-source-license-usage-on-github-com/.

[10] Olivier Barais et al. "Software Identification for Cybersecurity: Survey and Recommendations for Regulators". HAL: hal-05009757. 2025. URL: https://hal.science/hal-05009757.

[11] Michelle Barker et al. "Introducing the FAIR Principles for research software". In: *Scientific Data* 9 (2022), p. 622. DOI: 10.1038/s41597-022-01710-x.

[12] Tim Berners-Lee, Larry Masinter, and Mark McCahill. *Uniform Resource Locators (URL)*. RFC 1738. RFC Editor, Dec. 1994. DOI: 10.17487/RFC1738.

[13] Black Duck Software. *2025 Open Source Security and Risk Analysis (OSSRA) Report*. Tech. rep. 965 commercial codebases: 97% contained open source; 86% contained open-source vulnerabilities. Black Duck Software, Feb. 2025. URL: https://www.blackduck.com/resources/analyst-reports/open-source-security-risk-analysis.html (visited on 06/27/2026).

[14] Isabelle Blanc Catala et al. "Establishing a national research software award". In: *Open Research Europe* 3 (2023), p. 185. DOI: 10.12688/openreseurope.16069.1.

[15] Carl Boettiger et al. *CodeMeta: an exchange schema for software metadata*. Version 2.0; JSON-LD crosswalk vocabulary for software metadata. 2017. DOI: 10.5063/SCHEMA/CODEMETA-2.0. URL: https://codemeta.github.io/.

[16] Pierre Boulet. *État des lieux de la production et de la valorisation des logiciels issus de la recherche publique française*. Rapport au Parlement (art. 163, loi n° 2022-217 dite "3DS"). Survey of 1331 research-software entries; the "Rapport Boulet". Ministère de l'Enseignement supérieur et de la Recherche (DGRI), Sept. 2023. URL: https://hal-lara.archives-ouvertes.fr/hal-04844037 (visited on 06/27/2026).

[17] Budapest Open Access Initiative. *Budapest Open Access Initiative*. Feb. 2002. URL: https://www.budapestopenaccessinitiative.org/read/ (visited on 06/28/2026).

[18] Jeffrey C. Carver et al. "A survey of the state of the practice for research software in the United States". In: *PeerJ Computer Science* 8 (2022), e963. DOI: 10.7717/peerj-cs.963.

[19] cascad. *cascad: Certification Agency for Scientific Code and Data*. Non-profit reproducibility certification (CNRS UAR2007; founders CNRS, HEC Paris and Université d'Orléans); 170+ certifications to date. 2026. URL: https://www.cascad.tech/ (visited on 06/27/2026).

[20] Vinton G. Cerf. "Avoiding "Bit Rot": Long-Term Preservation of Digital Information". In: *Proceedings of the IEEE* 99.6 (2011), pp. 915–916. DOI: 10.1109/JPROC.2011.2124190.

[21] Chan Zuckerberg Initiative. *New data reveals the hidden impact of open source in science*. CZ Software Mentions dataset; software mentions extracted from the biomedical literature with a SciBERT model fine-tuned on the Softcite dataset. Sept. 2022. URL: https://medium.com/czi-technology/new-data-reveals-the-hidden-impact-of-open-source-in-science-11cc4a16fea2 (visited on 06/27/2026).

[22] cOAlition S. *Plan S: Making full and immediate Open Access a reality*. Sept. 2018. URL: https://www.coalition-s.org/ (visited on 06/28/2026).

[23] Christian Collberg and Todd A. Proebsting. "Repeatability in Computer Systems Research". In: *Communications of the ACM* 59.3 (2016), pp. 62–69. DOI: 10.1145/2812803.

[24] Comité pour la science ouverte. *Building an Open Science Monitoring Framework with open technologies — UNESCO workshop*. Convened by France and UNESCO, UNESCO HQ, Paris, 19 December 2023. Dec. 2023. URL: https://www.ouvrirlascience.fr/building-an-open-science-monitoring-framework-with-open-technologies-unesco-workshop-19-12-23/ (visited on 06/27/2026).

[25] Crossref and DataCite. *Why Metadata Matters for Research Integrity: A Joint Guide from Crossref and DataCite*. 2026. URL: https://datacite.org/blog/why-metadata-matters-for-research-integrity-a-new-joint-guide-from-crossref-and-datacite/ (visited on 06/24/2026).

[26] Deutsche Forschungsgemeinschaft. *Package of Measures to Support a Shift in the Culture of Research Assessment*. DFG Information für die Wissenschaft Nr. 61; research software added to the model academic CV (template mandatory from 1 March 2023). Sept. 2022. URL: https://www.dfg.de/en/research_funding/announcements_proposals/2022/info_wissenschaft_22_61 (visited on 06/27/2026).

[27] Roberto Di Cosmo. "Announcing biblatex-software". In: *ACM SIGSOFT Software Engineering Notes* 45.4 (2020). HAL: hal-02977711, pp. 22–23. DOI: 10.1145/3417564.3417570.

[28] Roberto Di Cosmo. "Archiving and Referencing Source Code with Software Heritage". In: *Mathematical Software – ICMS 2020*. Vol. 12097. Lecture Notes in Computer Science. Springer, 2020, pp. 362–373. DOI: 10.1007/978-3-030-52200-1_36.

[29] [SW] Roberto Di Cosmo, *biblatex-software* 2020. LIC: LPPL-1.3c. URL: https://ctan.org/pkg/biblatex-software, VCS: https://gitlab.inria.fr/gt-sw-citation/bibtex-sw-entry.

[30] Roberto Di Cosmo. "Construire le pilier logiciel de la Science Ouverte". In: *Open Science European Conference (OSEC 2022)*. OpenEdition Press, 2022, pp. 183–193. DOI: 10.4000/books.oep.15822.

[31] Roberto Di Cosmo. "Should We Preserve the World's Software History, And Can We?" In: *Linking Theory and Practice of Digital Libraries (TPDL 2022)*. Vol. 13541. Lecture Notes in Computer Science. Springer, 2022, pp. 3–7. DOI: 10.1007/978-3-031-16802-4_1.

[32] Roberto Di Cosmo, Pierre Alliez, et al. *Archiving, Referencing and Citing Software Artefacts Made Easy*. ACM SIGSOFT blog. 2022. URL: https://sigsoft.medium.com/archiving-referencing-and-citing-software-artefacts-made-easy-a5f00b8952d8 (visited on 06/24/2026).

[33] [SW Rel.] Roberto Di Cosmo and Marco Danelutto, *The Parmap library* version 1.1.1, 2020. Inria, University of Paris, and University of Pisa. LIC: LGPL-2.0. URL: https://rdicosmo.github.io/parmap/, VCS: https://github.com/rdicosmo/parmap, SWHID: ⟨swh:1:rel:373e2604d96de4ab1d505190b654c5c4045db773;origin=https://github.com/rdicosmo/parmap;visit=swh:1:snp:2a6c348c53eb77d458f24c9cbcecaf92e3c45615⟩.

[34] [SW exc.] "Core mapping routine", 2020 from [33], SWHID: ⟨swh:1:cnt:43a6b232768017b03da934ba22d9cc3f2726a6c5;origin=https://github.com/rdicosmo/parmap;visit=swh:1:snp:2a6c348c53eb77d458f24c9cbcecaf92e3c45615;anchor=swh:1:rel:373e2604d96de4ab1d505190b654c5c4045db773;path=/src/parmap.ml;lines=192-228⟩.

[35] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. "Identifiers for Digital Objects: the Case of Software Source Code Preservation". In: *Proceedings of the 15th International Conference on Digital Preservation (iPRES 2018), Boston, USA*. HAL: hal-01865790. 2018. DOI: 10.17605/OSF.IO/KDE56.

[36] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. "Referencing Source Code Artifacts: a Separate Concern in Software Citation". In: *Computing in Science & Engineering* 22.2 (2020), pp. 33–43. DOI: 10.1109/MCSE.2019.2963148.

[37] Roberto Di Cosmo and Stefano Zacchiroli. "The Software Heritage Open Science Ecosystem". In: *Software Ecosystems: Tooling and Analytics*. Ed. by Tom Mens, Coen De Roover, and Anthony Cleve. arXiv:2310.10295. Springer, 2023, pp. 33–61. DOI: 10.1007/978-3-031-36060-2_2.

[38] Roberto Di Cosmo et al. *Scholarly Infrastructures for Research Software. Report from the EOSC Executive Board Working Group (Architecture) SIRS Task Force*. European Commission, Directorate-General for Research and Innovation, 2020. DOI: 10.2777/28598. URL: https://data.europa.eu/doi/10.2777/28598.

[39] Roberto Di Cosmo et al. "CODE beyond FAIR: A Roadmap for Reusable Research Software". In: *Scientific Data* (2026). DOI: 10.1038/s41597-026-06705-6.

[40] [SW] Stephan Druskat et al., *Citation File Format* version 1.2.0, 2021. DOI: 10.5281/zenodo.5171937.

[41] Stephen J. Eglen and Daniel Nüst. *CODECHECK: Independent Execution of Computations Underlying Research Articles*. Partner venues include GigaScience and Reproducible AGILE. 2026. URL: https://codecheck.org.uk/ (visited on 06/27/2026).

[42] eLife and Software Heritage. *eLife archives software in Software Heritage*. 2020. URL: https://www.softwareheritage.org/2020/11/12/elife-archives-in-software-heritage-2/ (visited on 06/24/2026).

[43] EOSC Association. *Task Force on Infrastructures for Quality Research Software*. EOSC Association advisory group (term 2021–2023); fosters archiving, referencing, describing and quality of research software. 2021. URL: https://eosc.eu/advisory-groups/infrastructures-quality-research-software/ (visited on 06/27/2026).

[44] European Commission. *Commission Recommendation (EU) 2018/790 of 25 April 2018 on access to and preservation of scientific information*. Non-binding Recommendation; OJ L 134, 31.5.2018, p. 12–18. Apr. 2018. URL: https://eur-lex.europa.eu/eli/reco/2018/790/oj (visited on 06/27/2026).

[45] European Parliament and Council of the European Union. *Directive (EU) 2019/790 of 17 April 2019 on copyright and related rights in the Digital Single Market*. Dated 17 April 2019 (EP vote 26 March 2019); OJ L 130, 17.5.2019. Apr. 2019. URL: https://eur-lex.europa.eu/eli/dir/2019/790/oj (visited on 06/27/2026).

[46] European Parliament and Council of the European Union. *Regulation (EU) 2024/2847 on horizontal cybersecurity requirements for products with digital elements (Cyber Resilience Act)*. Official Journal of the European Union. Oct. 2024. URL: https://eur-lex.europa.eu/eli/reg/2024/2847/oj.

[47] FAIRCORE4EOSC Consortium. *FAIRCORE4EOSC: Core Components Supporting a FAIR EOSC*. Horizon Europe project 101057264, started 1 June 2022; includes an EOSC Software Heritage Mirror component. 2022. URL: https://faircore4eosc.eu/ (visited on 06/27/2026).

[48] GitLab. *GitLab Acquires Gitorious to Bolster Its On-Premises Code Collaboration Platform*. Mar. 2015. URL: https://about.gitlab.com/2015/03/03/gitlab-acquires-gitorious/.

[49] Carole Goble et al. "FAIR Computational Workflows". In: *Data Intelligence* 2.1-2 (2020), pp. 108–121. DOI: 10.1162/dint_a_00033.

[50] Google Open Source. *Bidding Farewell to Google Code*. Mar. 2015. URL: https://opensource.googleblog.com/2015/03/farewell-to-google-code.html.

[51] Graphics Replicability Stamp Initiative. *The Graphics Replicability Stamp Initiative*. 478 papers with code awarded the stamp since 2017. 2026. URL: http://www.replicabilitystamp.org/ (visited on 06/27/2026).

[52] International DOI Foundation. *DOI Registration Agencies*. 2026. URL: https://www.doi.org/the-community/existing-registration-agencies/ (visited on 06/24/2026).

[53] IPOL Journal. *IPOL: Image Processing On Line*. Each article ships a peer-reviewed reference implementation and a runnable online demo. 2026. URL: https://www.ipol.im/ (visited on 06/27/2026).

[54] ISO/IEC. *ISO/IEC 18670:2025 Information technology — SoftWare Hash IDentifier (SWHID) Specification V1.2*. 1st ed. International Organization for Standardization. 2025. URL: https://www.iso.org/standard/89985.html.

[55] [SW Mod.] Menelaos Karavelas, "2D Voronoi Diagram Adaptor", part of [89]. LIC: GPL. URL: https://doc.cgal.org/5.0.2/Manual/packages.html#PkgVoronoiDiagram2.

[56] Anna-Lena Lamprecht et al. “Towards FAIR principles for research software”. In: *Data Science* 3.1 (2020), pp. 37–59. DOI: 10.3233/DS-190026.

[57] Steve Lawrence et al. “Persistence of Web References in Scientific Research”. In: *Computer* 34.2 (2001), pp. 26–31. DOI: 10.1109/2.901164.

[58] Randall J. Leveque. “Top Ten Reasons To Not Share Your Code (and why you should anyway)”. In: *SIAM News* 46.3 (2013). URL: https://sinews.siam.org/Details-Page/top-ten-reasons-to-not-share-your-code-and-why-you-should-anyway.

[59] Linux Foundation Research. *The 2024 State of OSPOs and Open Source Management*. Tech. rep. The Linux Foundation, 2024. URL: https://www.linuxfoundation.org/research/ospo-2024 (visited on 06/27/2026).

[60] Anton Lozhkov, Raymond Li, Loubna Ben Allal, et al. *StarCoder 2 and The Stack v2: The Next Generation*. 2024. arXiv: 2402.19173. URL: https://arxiv.org/abs/2402.19173.

[61] Brian Matthews et al. “A Framework for Software Preservation”. In: *International Journal of Digital Curation* 5.1 (2010), pp. 91–105. DOI: 10.2218/ijdc.v5i1.145.

[62] Max Planck Society. *Berlin Declaration on Open Access to Knowledge in the Sciences and Humanities*. Oct. 2003. URL: https://openaccess.mpg.de/Berlin-Declaration (visited on 06/28/2026).

[63] Ralph C. Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Advances in Cryptology — CRYPTO ’87*. Vol. 293. Lecture Notes in Computer Science. Springer, 1988, pp. 369–378. DOI: 10.1007/3-540-48184-2_32.

[64] Ministère de l’Enseignement supérieur et de la Recherche. *Second French Plan for Open Science: Generalising Open Science in France 2021–2024*. Tech. rep. Ministère de l’Enseignement supérieur et de la Recherche, France, July 2021. URL: https://www.ouvrirlascience.fr/wp-content/uploads/2021/10/Second_French_Plan-for-Open-Science_web.pdf.

[65] Ministère de l’Enseignement supérieur et de la Recherche. *Prix science ouverte du logiciel libre de la recherche*. Second (2023) edition: 66 submissions across four categories. 2023. URL: https://www.ouvrirlascience.fr/category/openscience-prices/open-source-software-prices/ (visited on 06/27/2026).

[66] Saleha Muzammil et al. “Source Code Hotspots: A Diagnostic Method for Quality Issues”. In: *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR)*. arXiv:2602.13170. Pinpoints code passages with qualified SWHIDs. ACM, 2026.

[67] NASA Science Mission Directorate. *Scientific Information Policy for the Science Mission Directorate (SPD-41a)*. Effective 2 December 2022; open sharing of publications, data and software from SMD-funded research. Dec. 2022. URL: https://science.nasa.gov/researchers/science-information-policy/ (visited on 06/27/2026).

[68] NISO. *NISO RP-31-2021: Reproducibility Badging and Definitions*. National Information Standards Organization. 2021. URL: https://www.niso.org/publications/rp-31-2021-badging.

[69] OECD. *Recommendation of the Council concerning Access to Research Data from Public Funding*. OECD/LEGAL/0347; adopted 2006, revised 20 January 2021 to cover algorithms, workflows, models and software/code. 2021. URL: https://legalinstruments.oecd.org/en/instruments/OECD-LEGAL-0347 (visited on 06/27/2026).

[70] Norman Paskin. “Digital Object Identifier (DOI) System”. In: *Encyclopedia of Library and Information Sciences*. 3rd ed. Taylor & Francis, 2010, pp. 1586–1592.

[71] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. "The Software Heritage Graph Dataset: Public Software Development under One Roof". In: *Proceedings of the 16th International Conference on Mining Software Repositories* (*MSR*). IEEE, 2019, pp. 138–142. DOI: 10.1109/MSR.2019.00030.

[72] Premier ministre (France). *Circulaire n° 6264/SG du 27 avril 2021 relative à la politique publique de la donnée, des algorithmes et des codes sources*. Opens public data, algorithms and source code by default; creates the AMDAC ministerial-officer network and routes public code to code.gouv.fr. Apr. 2021. URL: https://www.legifrance.gouv.fr/circulaire/id/45162 (visited on 06/27/2026).

[73] République française. *Loi n° 2016-1321 du 7 octobre 2016 pour une République numérique*. "Loi Lemaire"; makes administrations' source code a communicable administrative document, open by default. Oct. 2016. URL: https://www.legifrance.gouv.fr/loda/id/JORFTEXT000033202746 (visited on 06/27/2026).

[74] République française. *Loi n° 2022-217 du 21 février 2022 (loi 3DS), article 163*. Loi 3DS, JO 22.2.2022. Art. 163 requires the Government to deliver to Parliament, within 18 months, a report on the production and valorization of free and proprietary software issued from French public research; fulfilled by the 2023 Rapport Boulet. Feb. 2022. URL: https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000045197395 (visited on 06/27/2026).

[75] ReScience C. *ReScience C: A Journal of Reproducible Replications in Computational Science*. Publishes independent re-implementations that reproduce a paper's findings from scratch. 2026. URL: https://rescience.github.io/ (visited on 06/27/2026).

[76] Arfon M. Smith, Daniel S. Katz, and Kyle E. Niemeyer. "Software citation principles". In: *PeerJ Computer Science* 2 (2016), e86. DOI: 10.7717/peerj-cs.86.

[77] [SW] Software Heritage, *Software Heritage Archive* 2016. Inria and UNESCO. URL: https://www.softwareheritage.org.

[78] Software Heritage. *Connecting publications and software with SoFAIR*. SoFAIR (a CHIST-ERA project) detects software mentions in the scholarly literature and links them to the archived code via the SWHID. Dec. 2023. URL: https://www.softwareheritage.org/2023/12/08/connecting-publications-software-sofair/ (visited on 06/27/2026).

[79] Software Heritage. *Software Heritage and Zenodo integrate to safeguard research*. Companion announcement: Zenodo, "Zenodo launches integration with Software Heritage", 21 October 2024, https://blog.zenodo.org/2024/10/21/2024-10-21-swh/. Nov. 2024. URL: https://www.softwareheritage.org/2024/11/13/software-heritage-zenodo-integration/ (visited on 06/24/2026).

[80] Software Heritage. *Software Heritage FAQ: Which type of SWHID should I use in my article/documentation?* Recommends printing the core directory SWHID with the full qualified one as a hyperlink; the `dir` core id is computed locally from content and stays the same regardless of which release or commit the directory belongs to. 2024. URL: https://www.softwareheritage.org/software-heritage-faq/ (visited on 06/28/2026).

[81] Software Heritage. *Episciences links article code through Software Heritage*. 2025. URL: https://www.softwareheritage.org/2025/08/21/episciences-links-article-code-software-heritage/ (visited on 06/25/2026).

[82] Diomidis Spinellis. "The Decay and Failures of URL References". In: *Communications of the ACM* 46.1 (2003), pp. 71–77. DOI: 10.1145/602421.602422.

[83] Stack Overflow. *Stack Overflow Annual Developer Survey 2022*. 93.87% of professional developers reported using Git. 2022. URL: https://survey.stackoverflow.co/2022/ (visited on 06/28/2026).

[84] Marilyn Strathern. "'Improving ratings': audit in the British University system". In: *European Review* 5.3 (1997), pp. 305–321.

[85] Sam Sun, Larry Lannom, and Brian Boesch. *Handle System Overview*. RFC 3650. RFC Editor, Nov. 2003. DOI: 10.17487/RFC3650.

[86] SWHID Working Group. *SWHID Specification, Version 1.2*. Open companion to ISO/IEC 18670:2025. 2025. URL: https://www.swhid.org/specification/v1.2/ (visited on 06/25/2026).

[87] Synopsys Cybersecurity Research Center. *2024 Open Source Security and Risk Analysis (OSSRA) Report*. Tech. rep. Ninth edition; 1 067 commercial codebases: 96% contained open source, 77% of the code was open source, average 526 open-source components per application. Synopsys, Feb. 2024. URL: https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html (visited on 06/27/2026).

[88] [SW] The CGAL Project, *The Computational Geometry Algorithms Library* (Coord.by CGAL Editorial Board), 1996. URL: https://cgal.org/.

[89] [SW Rel.] Version 5.0.2, 2020 of [88], SWHID: ⟨swh:1:rel:636541bbf6c77863908eae744610a3d91fa58855;origin=https://github.com/CGAL/cgal/⟩. URL: https://docs.cgal.org/5.02.

[90] The White House. *Executive Order 14028: Improving the Nation's Cybersecurity*. Signed 12 May 2021; Federal Register 86 FR 26633, 17 May 2021. May 2021. URL: https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity (visited on 06/27/2026).

[91] TODO Group. *Open Source Program Office (OSPO) Definition and Guide*. 2024. URL: https://ospodefinition.org (visited on 06/27/2026).

[92] UNESCO. *UNESCO Recommendation on Open Science*. Tech. rep. Paris: United Nations Educational, Scientific and Cultural Organization, Nov. 2021. DOI: 10.54677/MNMH8546. URL: https://unesdoc.unesco.org/ark:/48223/pf0000379949.

[93] UNESCO, Inria, and Software Heritage. *Paris Call on Software Source Code as Heritage for Sustainable Development*. UNESCO expert group report, Paris. Feb. 2019. URL: https://en.unesco.org/foss/paris-call-software-source-code.

[94] USENIX Association. *USENIX Security '25 Call for Artifacts*. Phase-1 artifact-availability evaluation made mandatory for all accepted papers. 2025. URL: https://www.usenix.org/conference/usenixsecurity25/call-for-artifacts (visited on 06/27/2026).

[95] Mark D. Wilkinson et al. "The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific Data* 3 (2016), p. 160018. DOI: 10.1038/sdata.2016.18.