EDITORS: Lorena A. Barba, labarba@gwu.edu
Sandra Gesing, sandra.gesing@nd.edu

SPECIAL TRACK: REPRODUCIBLE RESEARCH

Toward Long-Term and Archivable Reproducibility

Mohammad Akhlaghi, *Instituto de Astrofísica de Canarias, La Laguna, Tenerife, 38205, Spain*
Raúl Infante-Sainz, *Universidad de La Laguna, La Laguna, Tenerife, 38205, Spain*
Boudewijn F. Roukema, *Nicolaus Copernicus University, Torun 87-100, Poland*
Mohammadreza Khellat, *Ideal-Information, PC 133 Al Khuwair, Muscat, Oman*
David Valls-Gabaud, *Paris Observatory, Paris 75014, France*
Roberto Baena-Gallé, *Universidad Internacional de La Rioja, Logroño 26006, Spain*

Analysis pipelines commonly use high-level technologies that are popular when created, but are unlikely to be readable, executable, or sustainable in the long term. A set of criteria is introduced to address this problem: completeness (no execution requirement beyond a minimal Unix-like operating system, no administrator privileges, no network connection, and storage primarily in plain text); modular design; minimal complexity; scalability; verifiable inputs and outputs; version control; linking analysis with narrative; and free and open-source software. As a proof of concept, we introduce "Maneage" (managing data lineage), enabling cheap archiving, provenance extraction, and peer verification that has been tested in several research publications. We show that longevity is a realistic requirement that does not sacrifice immediate or short-term reproducibility. The caveats (with proposed solutions) are then discussed and we conclude with the benefits for the various stakeholders. This article is itself a Maneage'd project (project commit 313db0b).
Appendices—*Two comprehensive appendices that review the longevity of existing solutions are available as supplementary "Web extras," which are available in the IEEE Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/MCSE.2021.3072860.* Reproducibility—*All products available in* `zenodo.4913277`*, the Git history of this paper's source is at* `git.maneage.org/paper-concept.git`*, which is also archived in Software Heritage: swh:1:dir:33fea87068c1612daf011f161b97787b9a0df39f. Clicking on the SWHIDs in the digital format will provide more "context" for same content.*

Reproducible research has been discussed in the sciences for at least 30 years.[1,2] Many reproducible workflow solutions (hereafter, "solutions") have been proposed, which mostly rely on the common technology of the day, starting with Make and Matlab libraries in the 1990s, Java in the 2000s, and mostly shifting to Python during the past decade.

However, these technologies develop fast, e.g., code written in Python 2 (which is no longer officially maintained) often cannot run with Python 3. The cost of staying up to date within this rapidly evolving landscape is high. Scientific projects, in particular, suffer the most: Scientists have to focus on their own research domain, but to some degree, they need to understand the technology of their tools because it determines their results

Digital Object Identifier 10.1109/MCSE.2021.3072860
Date of publication 13 April 2021; date of current version 15 June 2021.

and interpretations. Decades later, scientists are still held accountable for their results, and therefore, the evolving technology landscape creates generational gaps in the scientific community, preventing previous generations from sharing valuable experience.

LONGEVITY OF EXISTING TOOLS

Reproducibility is defined as "obtaining consistent results using the same input data; computational steps, methods, and code; and conditions of analysis."[2] Longevity is defined as the length of time that a project remains *functional* after its creation. Functionality is defined as *human readability* of the source and its *execution possibility* (when necessary). Many usage contexts of a project do not involve execution: for example, checking the configuration parameter of a single step of the analysis to *reuse* in another project, or checking the version of used software, or the source of the input data. Extracting these from execution outputs is not always possible. A basic review of the longevity of commonly used tools is provided here (for a more comprehensive review, see the supplementary appendices, available online).

To isolate the environment, virtual machines (VMs) have sometimes been used, e.g., in SHARE[a] (awarded second prize in the Elsevier Executable Paper Grand Challenge of 2011, discontinued in 2019). However, containers (e.g., Docker or Singularity) are currently more widely used. We will focus on Docker here because it is currently the most common.

It is possible to precisely identify the used Docker "images" with their checksums (or "digest") to recreate an identical operating system (OS) image later. However, that is rarely done. Usually images are imported with OS names; e.g., Mesnard and Barba[3] use "`FROM ubuntu:16.04`." The extracted tarball URL[b] is updated almost monthly, and only the most recent five are archived. Hence, if the image is built in different months, it will contain different OS components. In 2024, when this version's long-term support (LTS) expires (if not earlier, like CentOS 8, which will terminate 8 years early[c]), the image will not be available at the expected URL.

Generally, prebuilt binary files (such as Docker images) are large and expensive to maintain, distribute, and archive. Because of this, in October 2020, Docker Hub (where many workflows are archived) announced[d] a new consumption-based payment model. Furthermore, Docker requires root permissions, and only supports recent (LTS) versions of the host kernel. Hence, older Docker images may not be executable: Their longevity is determined by OS kernels, typically a decade.

Once the host OS is ready, package managers (PMs) are used to install the software or environment. Usually the PM of the OS, such as "`apt`" or "`yum`," is used first and high-level software are built with generic PMs. The former has the same longevity as the OS while some of the latter (such as Conda and Spack) are written in high-level languages such as Python; so, the PM itself depends on the host's Python installation with a typical longevity of a few years. Nix and GNU Guix produce bitwise identical programs with considerably better longevity; that of their supported CPU architectures. However, they need root permissions and are primarily targeted at the Linux kernel. Generally, in all the PMs, the exact version of each software (and its dependencies) is not precisely identified by default, although an advanced user can, indeed, fix them. Unless precise version identifiers of *every software package* are stored by project authors, a third-party PM will use the most recent version. Furthermore, because third-party PMs introduce their own language, framework, and version history (the PM itself may evolve) and are maintained by an external team, they increase a project's complexity.

With the software environment built, job management is the next component of a workflow. Visual/GUI tools (written in Java or Python 2) Taverna (deprecated), GenePattern (deprecated), Kepler, or VisTrails (deprecated) were mostly introduced in the 2000s and encourage modularity and robust job management. However, a GUI environment is tailored to specific applications and is hard to generalize while being hard to reproduce once the required Java VM (JVM) is deprecated. These tools' data formats are complex (designed for computers to read) and hard to read by humans without the GUI. The more recent solutions (mostly non-GUI, written in Python) leave this to the authors of the project.

Designing a robust project needs to be encouraged and facilitated because scientists (who are not usually trained in project or data management) will rarely apply best practices. This includes automatic verification, which is possible in many solutions, but is rarely practiced. Besides nonreproducibility, weak project management leads to many inefficiencies in project cost and/or scientific accuracy (reusing, expanding, or validating will be expensive).

Finally, to blend narrative and analysis, computational notebooks,[4] such as Jupyter, are currently gaining

[a]https://is.ieis.tue.nl/staff/pvgorp/share
[b]https://partner-images.canonical.com/core/xenial
[c]https://blog.centos.org/2020/12/future-is-centos-stream
[d]https://www.docker.com/blog/docker-hub-image-retention-policy-delayed-and-subscription-updates

popularity. However, because of their complex dependence trees, their build is vulnerable to the passage of time; e.g., see Figure 1 in the work of Alliez *et al.*[5] for the dependencies of Matplotlib, one of the simpler Jupyter dependencies. It is important to remember that the longevity of a project is determined by its shortest lived dependency. Furthermore, as with job management, computational notebooks do not actively encourage good practices in programming or project management. The "cells" in a Jupyter notebook can either be run sequentially (from top to bottom, one after the other) or by manually selecting the cell to run. By default, cell dependencies are not included (e.g., automatically running some cells only after certain others), parallel execution, or usage of more than one language. There are third-party add-ons like `sos` or `extension's` (both written in Python) for some of these. However, since they are not part of the core, a shorter longevity can be assumed. The core Jupyter framework has few options for project management, especially as the project grows beyond a small test or tutorial. Notebooks can, therefore, rarely deliver their promised potential[4] and may even hamper reproducibility.[6]

PROPOSED CRITERIA FOR LONGEVITY

The main premise here is that starting a project with a robust data management strategy (or tools that provide it) is more effective, for researchers and the community, than imposing it just before publication.[2,7] In this context, researchers play a critical role[7] in making their research more Findable, Accessible, Interoperable, and Reusable (the FAIR principles[e]). Simply archiving a project workflow in a repository after the project is finished is, on its own, insufficient, and maintaining it by repository staff is often either practically unfeasible or unscalable. We argue and propose that workflows satisfying the following criteria can not only improve researcher flexibility during a research project, but can also increase the FAIRness of the deliverables for future researchers.

Criterion 1: Completeness. A project that is complete (self-contained) has the following properties. (1) No *execution requirements* apart from a minimal Unix-like operating system. Fewer explicit execution requirements would mean larger *execution possibility* and consequently better *longevity*. (2) Primarily stored as plain text (encoded in ASCII/Unicode), not needing specialized software to open, parse, or execute. (3) No impact on the host OS libraries, programs, and environment variables. (4) No root privileges to run (during development or post-publication). (5) Builds its own controlled software with independent environment variables. (6) Can run locally (without an internet connection). (7) Contains the full project's analysis, visualization, *and* narrative: including instructions to automatically access/download raw inputs, build necessary software, do the analysis, produce final data products, *and* final published report with figures *as output*, e.g., PDF or HTML. (8) It can run automatically, without human interaction.

Criterion 2: Modularity. A modular project enables and encourages independent modules with well-defined inputs/outputs and minimal side effects. In terms of file management, a modular project will *only* contain the hand-written project source of that particular high-level project: no automatically generated files (e.g., software binaries or figures), software source code, or data should be included. The latter two (developing low-level software, collecting data, or the publishing and archival of both) are separate projects in themselves because they can be used in other independent projects. This optimizes the storage, archival/mirroring, and publication costs (which are critical to longevity): A snapshot of a project's hand-written source will usually be on the scale of ~100 kilobytes, and the version controlled history may become a few megabytes.

In terms of the analysis workflow, explicit communication between various modules enables optimizations on many levels: (1) Modular analysis components can be executed in parallel and avoid redundancies (when a dependency of a module has not changed, the latter will not be rerun). (2) Usage in other projects. (3) Debugging and adding improvements (possibly by future researchers). (4) Citation of specific parts. (5) Provenance extraction.

Criterion 3: Minimal complexity. Minimal complexity can be interpreted as: (1) Avoiding the language or framework that is currently in vogue (for the workflow, not necessarily the high-level analysis). A popular framework typically falls out of fashion and requires significant resources to translate or rewrite every few years (for example Python 2, which is no longer supported). More stable/basic tools can be used with less long-term maintenance costs. (2) Avoiding too many different languages and frameworks; e.g., when the workflow's PM and analysis are orchestrated in the same framework, it becomes easier to maintain in the long term.

Criterion 4: Scalability: A scalable project can easily be used in arbitrarily large and/or complex projects. On a small scale, the criteria here are trivial to implement, but can rapidly become unsustainable.

[e]FAIR originally targeted data. Work is ongoing to adopt it for software through initiatives like FAIR4RS (FAIR for Research Software).

Criterion 5: Verifiable inputs and outputs: The project should automatically verify its inputs (software source code and data) *and* outputs, without needing any expert knowledge.

Criterion 6: Recorded history: No exploratory research is done in a single, first attempt. Projects evolve as they are being completed. Naturally, earlier phases of a project are redesigned/optimized only after later phases have been completed. Research papers often report this with statements such as *"we [first] tried method [or parameter] X, but Y is used here because it gave lower random error."* The derivation "history" of a result is, thus, not any the less valuable as itself.

Criterion 7: Including narrative that is linked to analysis. A project is not just its computational analysis. A raw plot, figure, or table is hardly meaningful alone, even when accompanied by the code that generated it. A narrative description is also a deliverable (defined as "data article"[7]): describing the purpose of the computations, interpretations of the result, and the context in relation to other projects/papers. This is related to longevity, because if a workflow contains only the steps to do the analysis or generate the plots, in time it may get separated from its accompanying published paper.

Criterion 8: Free and open-source software (FOSS): Non-FOSS software typically cannot be distributed, inspected, or modified by others. They are, thus, reliant on a single supplier (even without payments) and prone to *proprietary obsolescence*.[f] A project that is *free software* (as formally defined by GNU[g]), allows others to run, learn from, distribute, build upon (modify), and publish their modified versions. When the software used by the high-level project is also free, the lineage can be traced to the core algorithms, possibly enabling optimizations on that level and it can be modified for future hardware.

Proprietary software may be necessary to read proprietary data formats produced by data collection hardware (for example microarrays in genetics). In such cases, it is best to immediately convert the data to free formats upon collection and safely use or archive the data in free formats.

PROOF OF CONCEPT: MANEAGE

With the longevity problems of existing tools outlined earlier, a proof-of-concept solution is presented here via an implementation that has been tested in published papers.[8,9] Since the initial submission of this article, it has also been used in zenodo.3951151 (on the COVID-19 pandemic) and zenodo.4062460 (on galaxy evolution). It was also awarded a Research Data Alliance (RDA) adoption grant for implementing the recommendations of the joint RDA and World Data System working group on Publishing Data Workflows,[7] from the researchers' perspective.

It is called Maneage, for *Man*aging data Lin*eage* (the ending is pronounced as in "lineage"), hosted at `https://maneage.org`. It was developed as a parallel research project over five years of publishing reproducible workflows of our research. Its primordial implementation was used by Akhlaghi and Ichikawa,[10] which evolved in zenodo.1163746 and zenodo.1164774.

Technically, the hardest criterion to implement was the first (completeness); in particular, restricting execution requirements to only a minimal Unix-like operating system. One solution we considered was GNU Guix and Guix Workflow Language (GWL). However, because Guix requires root access to install, and only works with the Linux kernel, it failed the completeness criterion. Inspired by GWL+Guix, a single job management tool was implemented for both installing software *and* the analysis workflow: Make.

Make is not an analysis language, it is a job manager. Make decides when and how to call analysis steps/programs (in any language such as Python, R, Julia, Shell, or C). Make has been available since 1977, it is still heavily used in almost all components of modern Unix-like OSs and is standardized in POSIX. It is thus mature, actively maintained, highly optimized, efficient in managing provenance, and recommended by the pioneers of reproducible research.[1,11] Moreover, researchers using FOSS have already had some exposure to Make (most FOSS are built with Make).

Linking the analysis and narrative (criterion 7) was historically our first design element. To avoid the problems with computational notebooks mentioned before, we adopt a more abstract linkage, providing a more direct and traceable connection. Assuming that the narrative is typeset in LaTeX, the connection between the analysis and narrative (usually as numbers) is through automatically created LaTeX macros, during the analysis. For example, Akhlaghi[8] writes "*... detect the outer wings of M51 down to S/N of 0.25....*" The LaTeX source of the quote above is: "`detect the outer wings of M51 down to S/N of $\demosfoptimizedsn$`." The macro "`\demosfoptimizedsn`" is automatically generated after the analysis and expands to the value "`0.25`" upon creation of the PDF. Since values like this depend on the analysis, they should *also* be reproducible, along with figures and tables.

These macros act as a quantifiable link between the narrative and analysis, with the granularity of a word in a sentence and a particular analysis command. This allows automatic updates to the

[f]https://www.gnu.org/proprietary/proprietary-obsolescence.html
[g]https://www.gnu.org/philosophy/free-sw.en.html

embedded numbers during the experimentation phase of a project *and* accurate postpublication provenance. Through the former, manual updates by authors (which are prone to errors and discourage improvements or experimentation after writing the first draft) are by-passed.

Acting as a link, the macro files build the core skeleton of Maneage. For example, during the software building phase, each software package is identified by a LaTeX file, containing its official name, version, and possible citation. These are combined at the end to generate precise software acknowledgment and citation that is shown in the appendices, available online, other examples have also been published.[8,9] Furthermore, the machine-related specifications of the running system (including CPU architecture and byte-order) are also collected to report in the paper (they are reported for this article in the section "Acknowledgments"). These can help in *root cause analysis* of observed differences/issues in the execution of the workflow on different machines.

The macro files also act as Make *targets* and *prerequisites* to allow accurate dependence tracking and optimized execution (in parallel, no redundancies), for any level of complexity (e.g., Maneage builds Matplotlib if requested; see Figure 1 in the work by Alliez *et al.*[5]). All software dependencies are built down to precise versions of every tool, including the shell, important low-level application programs (e.g., GNU Coreutils) and of course, the high-level science software. The source code of all the FOSS software used in Maneage is archived in, and downloaded from, zenodo.3883409. Zenodo promises long-term archival and also provides persistent identifiers for the files, which are sometimes unavailable at a software package's web page.

On GNU/Linux distributions, even the GNU Compiler Collection (GCC) and GNU Binutils are built from source and the GNU C library (glibc) is being added[h]. Currently, TEXLive is also being added,[i] but that is only for building the final PDF, not affecting the analysis or verification.

Building the core Maneage software environment on an 8-core CPU takes about 1.5 h (GCC consumes more than half of the time). However, this is only necessary once in a project: the analysis (which usually takes months to write/mature for a normal project) will only use the built environment. Hence, the few hours of initial software building is negligible compared to a project's life span. To facilitate moving to another computer in the short term, Maneage'd projects can be built in a container or VM. The `README.md`[j] file has thorough instructions on building in Docker. Through containers or VMs, users on non-Unix-like OSs (like Microsoft Windows) can use Maneage. For Windows-native software that can be run in batch-mode, evolving technologies like Windows Subsystem for Linux may be usable.

```
# Default target/goal of project.
all: paper.pdf

# Define subMakefiles to load in order.
makesrc = initialize \          # General
          download \            # General
          format \              # Project-specific
          demo-plot \           # Project-specific
          verify \              # General
          paper                 # General

# Load all the configuration files.
include reproduce/analysis/config/*.conf

# Load the subMakefiles in the defined order.
include $(foreach s,$(makesrc), \
            reproduce/analysis/make/$(s).mk)
```

LISTING 1. This project's simplified `top-make.mk`, see Figure 1. `swh:1:cnt:27c1b5b7e103d2e35f0ba76b3a59add8edf52110`.

The analysis phase of the project, however, is naturally different from one project to another at a low level. It was, thus, necessary to design a generic framework to comfortably host any project while still satisfying the criteria of modularity, scalability, and minimal complexity. This design is demonstrated with the example of Figure 1 (left) which is an enhanced replication of the "tool" curve of Figure 1 C in the work by Menke *et al.*[12] Figure 1 (right) shows the data lineage that produced it.

The analysis is orchestrated through a single point of entry (`top-make.mk`, which is a Makefile; see Listing 1). It is only responsible for `include`-ing the modular *subMakefiles* of the analysis, in the desired order, without doing any analysis itself. This is visualized in Figure 1 (right) where no built (blue) file is placed directly over `top-make.mk`. A visual inspection of this file is sufficient for a nonexpert to understand the high-level steps of the project (irrespective of the low-level implementation details), provided that the subMakefile names are descriptive (thus encouraging good practice). A human-friendly design that is also optimized for execution is a critical component for the FAIRness of reproducible research.

[h]http://savannah.nongnu.org/task/?15390
[i]http://savannah.nongnu.org/task/?15267

[j]`swh:1:cnt:66c1d53b2860a40aa9d350048f6b02c73c3b46c8`
Software Heritage identifiers (SWHIDs) can be used with resolvers like `http://n2t.net/` (`e.g., http://n2t.net/swh:1:...`).

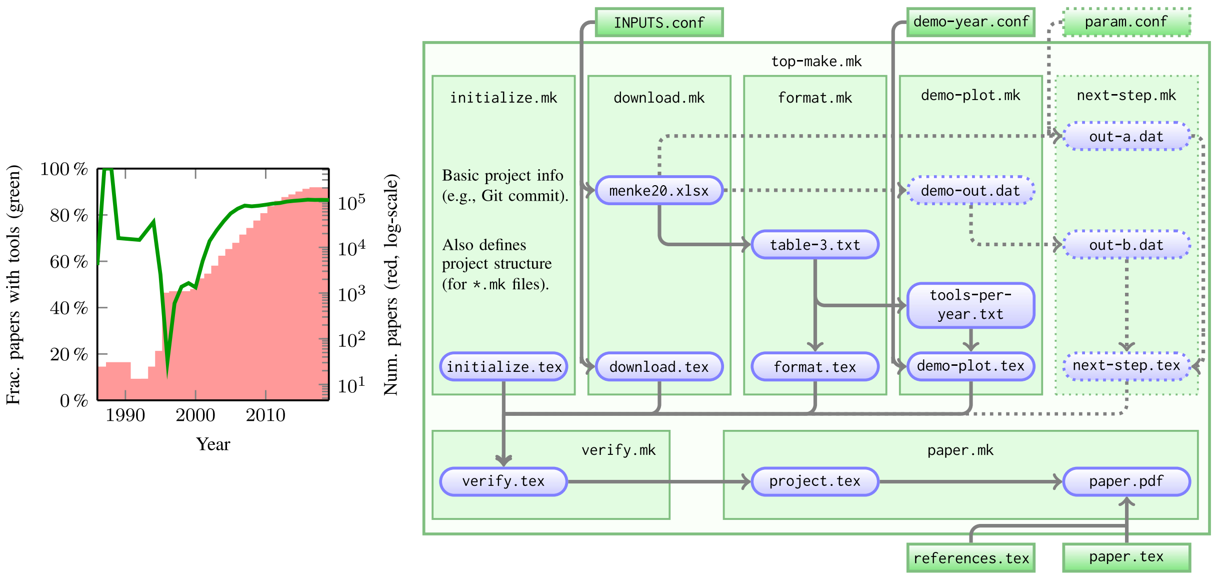

FIGURE 1. Left: An enhanced replica of Figure 1 C in the work by Menke *et al.*,[12] shown here for demonstrating Maneage. It shows the fraction of the number of papers mentioning software tools (green line, left vertical axis) in each year (red bars, right vertical axis on a log scale). Right: Schematic representation of the data lineage, or workflow, to generate the plot on the left. Each colored box is a file in the project and arrows show the operation of various software: linking input file(s) to the output file(s). Green files/boxes are plain-text files that are under version control and in the project source directory. Blue files/boxes are output files in the build directory, shown within the Makefile (`*.mk`) where they are defined as a *target*. For example, `paper.pdf` is created by running LaTeX on `project.tex` (in the build directory; generated automatically) and `paper.tex` (in the source directory; written manually). Other software is used in other steps. The solid arrows and full-opacity built boxes correspond to the lineage of this article. The dotted arrows and built boxes show the scalability of Maneage (ease of adding hypothetical steps to the project as it evolves). The underlying data of the left plot is available at https://zenodo.org/record/4913277/files/tools-per-year.txt

All projects first load `initialize.mk` and `download.mk`, and finish with `verify.mk` and `paper.mk` (see Listing 1). Project authors add their modular subMakefiles in between. Except for `paper.mk` (which builds the ultimate target: `paper.pdf`), all subMakefiles build a macro file with the same base-name (the `.tex` file at the bottom of each subMakefile in Figure 1). Other built files ("targets" in intermediate analysis steps) cascade down in the lineage to one of these macro files, possibly through other files.

Just before reaching the ultimate target (`paper.pdf`), the lineage reaches a bottleneck in `verify.mk` to satisfy the verification criteria. All project deliverables (macro files, plot or table data, and other datasets) are verified at this stage, with their checksums, to automatically ensure exact reproducibility. Where exact reproducibility is not possible (for example, due to parallelization), values can be verified by the project authors. For example, see `verify-parameter-statistically.sh`[k] of zenodo.4062460.

To further minimize complexity, the low-level implementation can be further separated from the high-level execution through configuration files. By convention in Maneage, the subMakefiles (and the programs they call for number crunching) do not contain any fixed numbers, settings, or parameters. Parameters are set as Make variables in "configuration files" (with a `.conf` suffix) and passed to the respective program by Make. For example, in Figure 1 (bottom), `INPUTS.conf` contains URLs and checksums for all imported datasets, thereby enabling exact verification before usage. To illustrate this, we report that Menke *et al.*[12] studied 53 papers in 1996 (which is not in their original plot). The number `1996` is stored in `demo-year.conf` and the result (`53`) was calculated after generating `tools-per-year.txt`. Both numbers are expanded as LaTeX macros when creating this PDF file. An interested reader can change the value in `demo-year.conf` to automatically update the result in the PDF, without knowing the underlying low-level implementation. Furthermore, the configuration files are a prerequisite of

[k] `swh:1:cnt:dae4e6de5399a061ab4df01ea51f4757fd7e293a`

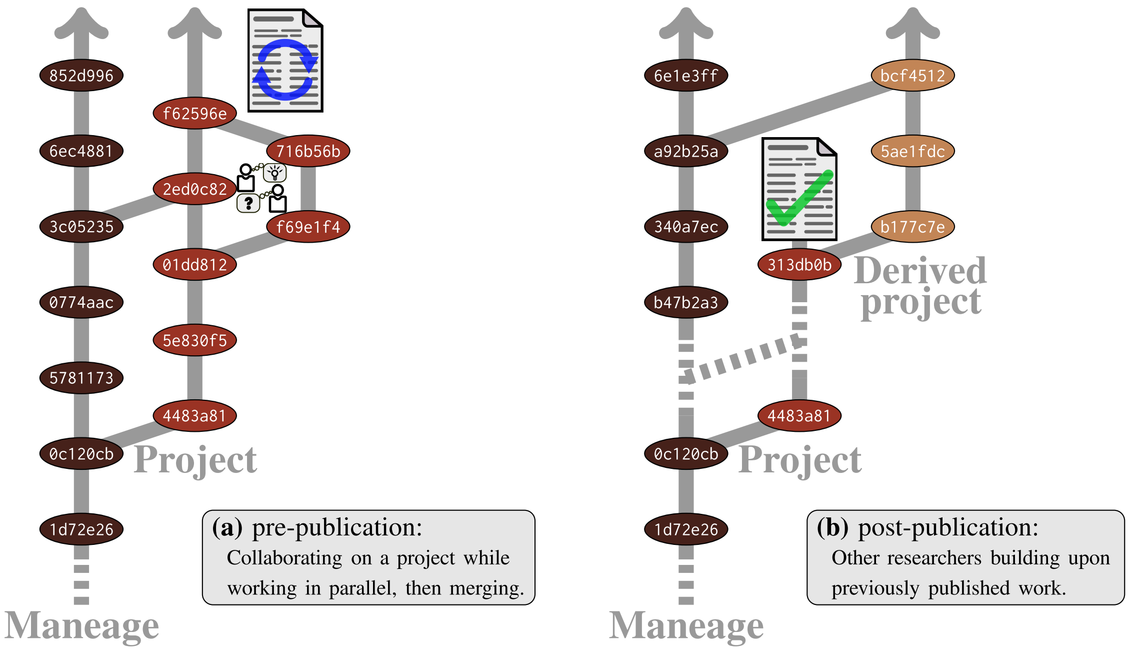

FIGURE 2. Maneage is a Git branch. Projects using Maneage are branched off it and apply their customizations. (a) Hypothetical project's history before publication. The low-level structure (in Maneage, shared between all projects) can be updated by merging with Maneage. (b) Finished/published project can be revitalized for new technologies by merging with the core branch. Each Git "commit" is shown on its branch as a colored ellipse, with its commit hash shown and colored to identify the team that is/was working on the branch. Briefly, Git is a version control system, allowing a structured backup of project files, for more see supplementary appendices, available online (section on version control). Each Git "commit" effectively contains a copy of all the project's files at the moment it was made. The upward arrows at the branch-tops are, therefore, in the direction of time.

the targets that use them. If changed, Make will *only* re-execute the dependent recipe and all its descendants, with no modification to the project's source or other built products. This fast and cheap testing encourages experimentation (without necessarily knowing the implementation details; e.g., by coauthors or future readers), and ensures self-consistency.

In contrast to notebooks like Jupyter, the analysis scripts, configuration parameters, and paper's narrative are, therefore, not blended into in a single file, and do not require a unique editor. To satisfy the modularity criterion, the analysis steps and narrative are written and run in their own files (in different languages) and the files can be viewed or manipulated with any text editor that the authors prefer. The analysis can benefit from the powerful and portable job management features of Make and communicates with the narrative text through LaTeX macros, enabling much better-formatted output that blends analysis outputs in the narrative sentences and enables direct provenance tracking.

To satisfy the recorded history criterion, version control (currently implemented in Git) is another component of Maneage (see Figure 2). Maneage is a Git branch that contains the shared components (infrastructure) of all projects (e.g., software tarball URLs, build recipes, common subMakefiles, and interface script). The core Maneage git repository is hosted at `http://git.maneage.org/project.git` (archived at Software Heritage[i]). Derived projects start by creating a branch and customizing it (e.g., adding a title, data links, narrative, and subMakefiles for its particular analysis, see Listing 2). There is a thoroughly elaborated customization checklist in `README-hacking.md`.

The current project's Git hash is provided to the authors as a LaTeX macro (shown here in the sections "Abstract" and "Acknowledgments"), as well as the Git hash of the last commit in the Maneage branch (shown here in the section "Acknowledgments").

[i]`swh:1:dir:c80868c9b46bda92f5989cc9fc3e8597a198880d`

```
# Cloning Maneage and branching off of it.
$ git clone https://git.maneage.org/project.git
$ cd project
$ git remote rename origin origin-maneage
$ git checkout -b main

# Build the raw Maneage skeleton in two phases.
$ ./project configure     # Build software environment.
$ ./project make          # Do analysis, build PDF paper.

# Start editing, test-building and committing.
$ emacs paper.tex            # Set your name as author.
$ ./project make             # Rebuild to see effect.
$ git add -u && git commit   # Commit changes.
```

LISTING 2. Starting a new project with Maneage, and building it.

These macros are created in `initialize.mk`, with other basic information from the running system like the CPU details (shown in the section "Acknowledgments"). As opposed to Git "tag"s, the hash is a core concept in the Git paradigm and is immutable and always present in a given history, which is why it is the recommended version identifier.

Figure 2 shows how projects can reimport Maneage at a later time (technically: *merge*), thus improving their low-level infrastructure: in (a), authors do the merge during an ongoing project; in (b), readers do it after publication; e.g., the project remains reproducible but the infrastructure is outdated, or a bug is fixed in Maneage. Generally, any Git flow (branching strategy) can be used by the high-level project authors or future readers. Low-level improvements in Maneage can, thus, propagate to all projects, greatly reducing the cost of project curation and maintenance, before *and* after publication.

Finally, a snapshot of the complete project source is usually $\sim$100 kilobytes. It can, thus, easily be published or archived in many servers; for example, it can be uploaded to arXiv (with the LaTeX source[8–10]), published on Zenodo and archived in Software Heritage.

DISCUSSION

We have shown that it is possible to build workflows satisfying all the proposed criteria. Here, we comment on our experience in testing them through Maneage and its increasing user-base (thanks to the support of RDA).

First, while most researchers are generally familiar with them, the necessary low-level tools (e.g., Git, LaTeX, the command-line, and Make) are not widely used. Fortunately, we have noticed that after witnessing the improvements in their research, many, especially early-career researchers, have started mastering these tools. Scientists are rarely trained sufficiently in data management or software development, and the plethora of high-level tools that change every few years discourages them. Indeed, the fast-evolving tools are primarily targeted at software developers, who are paid to learn and use them effectively for short-term projects before moving on to the next technology.

Scientists, on the other hand, need to focus on their own research fields and need to consider longevity. Hence, arguably the most important feature of these criteria (as implemented in Maneage) is that they provide a fully working template or bundle that works immediately out of the box by producing a paper with an example calculation that they just need to start customizing, using mature and time-tested tools, for blending version control, the research paper's narrative, the software management, *and* a robust data management strategy. We have noticed that providing a clear checklist of the initial customizations is much more effective in encouraging mastery of these core analysis tools than having abstract, isolated tutorials on each tool individually.

Second, to satisfy the completeness criterion, all the required software of the project must be built on various Unix-like OSs (Maneage is actively tested on different GNU/Linux distributions, macOS, and is being ported to FreeBSD also). This requires maintenance by our core team and consumes time and energy. However, because the PM and analysis components share the same job manager (Make) and design principles, we have already noticed some early users adding, or fixing, their required software alone. They later share their low-level commits on the core branch, thus propagating it to all derived projects.

Third, Unix-like OSs are a very large and diverse group (mostly conforming with POSIX), so our completeness condition does not guarantee bitwise reproducibility of the software, even when built on the same hardware. However, our focus is on reproducing results (output of software), not the software itself. Well-written software internally corrects for differences in OS or hardware that may affect its output (through tools like the GNU Portability Library or Gnulib).

On GNU/Linux hosts, Maneage builds precise versions of the compilation tool chain. However, glibc is not installable on some Unix-like OSs (e.g., macOS) and all programs link with the C library. This may hypothetically hinder the exact reproducibility *of results* on non-GNU/Linux systems, but we have not encountered this in our research so far. With everything else under precise control in Maneage, the effect of differing hardware, Kernel, and C libraries on high-level science can now be systematically studied in follow-up research (including floating-point arithmetic or optimization differences). Using continuous integration is one way to precisely identify breaking points on multiple systems.

Other implementations of the criteria, or future improvements in Maneage, may solve some of the caveats, but this proof of concept already shows many advantages. For example, the publication of projects meeting these criteria on a wide scale will allow automatic workflow generation, optimized for desired characteristics of the results (e.g., via machine learning). The completeness criterion implies that algorithms and data selection can be included in the optimizations.

Furthermore, through elements like the macros, natural language processing can also be included, automatically analyzing the connection between an analysis with the resulting narrative *and* the history of that analysis +narrative. Parsers can be written over projects for meta-research and provenance studies, e.g., to generate Research Objects (see Supplement Appendix B, available online) or allow interoperability with Common Workflow Language or high-level concepts like Canonical Workflow Framework for Research, or CWFR (see Supplement Appendix A, available online).

Likewise, when a bug is found in one science software, affected projects can be detected and the scale of the effect can be measured. Combined with Software-Heritage, precise high-level science components of the analysis can be accurately cited (e.g., even failed/abandoned tests at any historical point). Many components of "machine-actionable" data management plans can also be automatically completed as a byproduct, useful for project PIs and grant funders.

From the data repository perspective, these criteria can also be useful, e.g., the challenges mentioned in the work by Austin *et al.*[7]: (1) The burden of curation is shared among all project authors and readers (the latter may find a bug and fix it), not just by database curators, thereby improving sustainability. (2) Automated and persistent bidirectional linking of data and publication can be established through the published *and complete* data lineage that is under version control. (3) Software management: With these criteria, each project comes with its unique and complete software management. It does not use a third-party PM that needs to be maintained by the data center (and the many versions of the PM), hence enabling robust software management, preservation, publishing, and citation. For example, see `zenodo.1163746`, `zenodo.3408481`, `zenodo.3524937`, `zenodo.3951151`, or `zenodo.4062460`, where we distribute the source code of all (FOSS) software used in each project, as deliverables. (4) "Linkages between documentation, code, data, and journal articles in an integrated environment," which effectively summarizes the whole purpose of these criteria.

ACKNOWLEDGMENTS

This project (commit 313db0b) is maintained in Maneage (Managing data lineage). The latest merged Maneage branch commit was 6e4ec9a (17 Apr 2021). This project was built on an x86 64 machine with Little Endian byte-order and address sizes 39 bits physical, 48 bits virtual. The authors would like to thank (sorted alphabetically) Julia Aguilar-Cabello, Dylan Aïssi, Marjan Akbari, Alice Allen, Pedram Ashofteh Ardakani, Roland Bacon, Michael R. Crusoe, Roberto Di Cosmo, Antonio Díaz Díaz, Surena Fatemi, Fabrizio Gagliardi, Konrad Hinsen, Marios Karouzos, Johan Knapen, Tamara Kovazh, Sebastian Luna Valero, Terry Mahoney, Javier Moldón, Ryan O'Connor, Mervyn O'Luing, Simon Portegies Zwart, Rémi Rampin, Vicky Rampin, Susana Sanchez Exposito, Idafen Santana-Pérez, Elham Saremi, Yahya Sefidbakht, Zahra Sharbaf, Nadia Tonello, Ignacio Trujillo, Lourdes Verdes-Montenegro, and Peter Wittenburg for their useful help, suggestions, and feedback on Maneage and this article. The five referees and editors of CiSE (Lorena Barba and George Thiruvathukal) provided many points that greatly helped to clarify this article. The work on Maneage, and this article, has been supported by the following institutions: The Japanese MEXT PhD Scholarship to Mohammad Akhlaghi and its Grant-in-Aid for Scientific Research under Grant 21244012 and Grant 24253003; The European Research Council (ERC) Advanced Grant 339659-MUSICOS; The European Union (EU) Horizon 2020 (H2020) Research and Innovation Programmes No 777388 under RDA EU 4.0 Project, and Marie Skłodowska-Curie under Grant 721463 to the SUNDIAL ITN; The State Research Agency (AEI-MCINN) of the Spanish Ministry of Science and Innovation (SMSI) under the grant "The structure and evolution of galaxies and their central regions" PID2019-105602GBI00/10.13039/501100011033; the IAC Project P/300724, financed by the SMSI, through the Canary Islands Department of Economy, Knowledge and Employment; the "A next-generation worldwide quantum sensor network with optical atomic clocks" project of the TEAM IV Programme of the Foundation for Polish Science co-financed by the EU under ERDF; the Polish MNiSW under Grant DIR/WK/2018/12; and the Poznań Supercomputing and Networking Center (PSNC) Computational Grant 314.

REFERENCES

1. J. F. Claerbout and M. Karrenbach, "Electronic documents give reproducible research a new meaning," *SEG Tech. Prog. Expanded Abstr.*, vol. 1, pp. 601–604, 1992, doi: 10.1190/1.1822162.

2. H. V. Fineberg *et al.*, Reproducibility and Replicability in Science. Washington, DC, USA: Nat. Acad. Press, 2019, pp. 1–256, doi: 10.17226/25303.
3. O. Mesnard and L. A. Barba, "Reproducible workflow on a public cloud for computational fluid dynamics," *Comput. Sci. Eng.*, vol. 22, no. 1, pp. 102–116, Jan./Feb. 2020, doi: 10.1109/MCSE.2019.2941702.
4. A. Rule, A. Tabard, and J. Hollan, "Exploration and explanation in computational notebooks," *Proc. CHI Conf. Hum. Factors Comput. Syst.*, vol. 1, pp. 1–12, 2018, doi: 10.1145/3173574.3173606.
5. P. Alliez *et al.*, "Attributing and referencing (research) software: Best practices and outlook from Inria," *Comput. Sci. Eng.*, vol. 22, no. 3, pp. 39–52, May/Jun. 2019, doi: 10.1109/MCSE.2019.2949413.
6. J. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of Jupyter notebooks," *Proc. 16th Int. Conf. Mining Softw. Repositories*, vol. 1, pp. 507–517, 2019, doi: 10.1109/MSR.2019.00077.
7. C. Austin *et al.* "Key components of data publishing: Using current best practices to develop a reference model for data publishing," *Int. J. Digit. Libraries*, vol. 18, pp. 77–92, 2017, doi: 10.1007/s00799-016-0178-2.
8. M. Akhlaghi, "Carving out the low surface brightness universe with NoiseChisel," *in Proc. Int. Astron. Union Symp.*355, Sep. 2019, *arXiv:1909.11230*, doi: 10.5281/zenodo.3408480.
9. R. Infante-Sainz, I. Trujillo, and J. Román, "The Sloan Digital Sky Survey extended point spread functions," *Monthly Notices R. Astron. Soc.*, vol. 491, no. 4, pp. 5317–5329, Feb. 2020, doi: 10.1093/mnras/stz3111.
10. M. Akhlaghi and T. Ichikawa, "Noise-based detection and segmentation of nebulous objects," *Astrophys. J. Suppl. Ser.*, vol. 220, pp. 1–33, Sep. 2015, doi: 10.1088/0067-0049/220/1/1.
11. M. Schwab, M. Karrenbach, and J. F. Claerbout, "Making scientific computations reproducible," *Comput. Sci. Eng.*, vol. 2, no. 6, pp. 61–67, Nov./Dec. 2000, doi: 10.1109/5992.881708.
12. J. Menke, M. Roelandse, B. Ozyurt, M. Martone, and A. Bandrowski, "Rigor and transparency index, a new metric of quality for assessing biological and medical science methods," *iScience*, vol. 23, 2020, Art no. 101698, doi: 10.1016/j.isci.2020.101698.

MOHAMMAD AKHLAGHI is currently a postdoctoral researcher with the Instituto de Astrofísica de Canarias, Tenerife, Spain. He is also with Facultad de Física, Universidad de La Laguna, Tenerife, Spain, and Univ Lyon, Ens de Lyon, Univ Lyon1, CNRS, Centre de Recherche Astrophysique de Lyon UMR5574, Lyon. He received the PhD degree from Tohoku University, Sendai, Japan. He is the corresponding author of this article. Contact him at mohammad@akhlaghi.org.

RAÚL INFANTE-SAINZ is currently a doctoral student at the Instituto de Astrofísica de Canarias, Tenerife, Spain. He is also with Facultad de Física, Universidad de La Laguna, Tenerife, Spain. He received the MSc degree from the University of Granada, Granada, Spain. Contact him at infantesainz@gmail.com.

BOUDEWIJN F. ROUKEMA is a professor of cosmology with the Institute of Astronomy, Faculty of Physics, Astronomy and Informatics, Nicolaus Copernicus University, Torun, Poland. He is also with Univ Lyon, Ens de Lyon, Univ Lyon1, CNRS, Centre de Recherche Astrophysique de Lyon UMR5574, Lyon, France. He received the PhD degree from Australian National University, Canberra, ACT, Australia. Contact him at boud@astro.uni.torun.pl.

MOHAMMADREZA KHELLAT is currently the backend technical services manager at Ideal-Information, Muscat, Oman. He received the MSc degree in theoretical particle physics from Yazd University, Yazd, Iran. Contact him at mkhellat@ideal-information.com.

DAVID VALLS-GABAUD is a CNRS Research Director at Paris Observatory, Paris, France. He studied at the Universities of Madrid, Paris and Cambridge, and obtained his Ph.D. degree in 1991. Contact him at david.vallsgabaud@observatoiredeparis.psl.eu.

ROBERTO BAENA-GALLÉ is a professor at the Universidad Internacional de La Rioja (UNIR), Gran Vía Rey Juan Carlos I, La Rioja, Spain. He was a postdoctoral researcher with Instituto de Astrofísica de Canarias (IAC), Spain. He received the degree from the University of Seville, Seville, Spain, and the PhD degree from the University of Barcelona, Barcelona, Spain. Contact him at roberto.baena@unir.net.