Towards Long-term and Archivable Reproducibility

Mohammad Akhlaghi, Raúl Infante-Sainz, Boudewijn F. Roukema, Mohammadreza Khellat, David Valls-Gabaud, Roberto Baena-Gallé

***Abstract*—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 software. As a proof of concept, we introduce "Maneage" (Managing data lineage), enabling cheap archiving, provenance extraction, and peer verification that 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 paper is itself written with Maneage (project commit eeff5de).**

***Appendices* — Two comprehensive appendices that review the longevity of existing solutions; available after main body of paper (Appendices A and B).**

***Reproducibility* — All products in `zenodo.4291207`, Git history of source at `gitlab.com/makhlaghi/maneage-paper`, which is also archived in SoftwareHeritage.**

***Index Terms*—Data Lineage, Provenance, Reproducibility, Scientific Pipelines, Workflows**

I. Introduction

Reproducible research has been discussed in the sciences for at least 30 years [1], [2]. Many reproducible workflow solutions (hereafter, "solutions") have been proposed that 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 last 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 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.

II. 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 re-*use* 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, please see appendices A and B).

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

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

Generally, pre-built binary files (like Docker images) are large and expensive to maintain and archive. Because of this, in October 2020 Docker Hub (where many workflows are archived) announced that inactive images (more than 6 months) will be deleted in free accounts from mid 2021. 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 the host kernel, typically a decade).

Once the host OS is ready, PMs are used to install the software or environment. Usually the OS's PM, such as '`apt`' or '`yum`', is used first and higher-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 like 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 bit-wise 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 package managers, the exact version of

Manuscript received MM DD, YYYY; revised MM DD, YYYY.

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 workflow tools like Apache Taverna, GenePattern (deprecated), Kepler or VisTrails (deprecated), which were mostly introduced in the 2000s and used Java or Python 2 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 Virtual Machine (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 non-reproducibility, 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 popularity. However, because of their complex dependency trees, their build is vulnerable to the passage of time; e.g., see Figure 1 of [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].

III. 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 much 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). 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 longer *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 $\times 100$ 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, it will not be re-run). (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.

Criterion 5: Verifiable inputs and outputs. The project should automatically verify its inputs (software source code and data) *and* outputs, not 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" in [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: Non-free or non-open-source software typically cannot be distributed, inspected, or modified by others. They are reliant on a single supplier (even without payments) and prone to proprietary obsolescence. A project that is free software (as formally defined by GNU), allows others to run, learn from, distribute, build upon (modify), and publish their modified versions. When the software used by the project is itself 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 micro-arrays 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 as free formats.

IV. Proof of concept: Maneage

With the longevity problems of existing tools outlined above, 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 paper, it has also been used in zenodo.3951151 (on the COVID-19 pandemic) and zenodo.4062460. It was also awarded a Research Data Alliance (RDA) adoption grant for implementing the recommendations of the joint RDA and World Data System (WDS) 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 in [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 like 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]. Researchers using free software have also already had some exposure to it (most free research software 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 above, 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, [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 embedded numbers during the experimentation phase of a project *and* accurate post-publication 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 (C), for other examples, see [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 paper in the 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 dependency tracking and optimized execution (in parallel, no redundancies), for any level of complexity (e.g., Maneage builds Matplotlib if requested; see Figure 1 of [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 free software used in Maneage is archived in, and downloaded from, zenodo.3883409. Zenodo promises long-term archival and also provides a persistent identifier 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 (task 15390). Currently, TEXLive is also being added (task 15267), 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 hours (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` 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.

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 1C in [12]. Figure 1 (right) is 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 non-expert 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.

All projects first load `initialize.mk` and `download.mk`, and finish with `verify.mk` and `paper.mk` (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.

Listing 1
THIS PROJECT'S SIMPLIFIED `top-make.mk`, ALSO SEE FIGURE 1.
FOR FULL FILE, SEE SOFTWAREHERITAGE

```
# 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)
```

Just before reaching the ultimate target (`paper.pdf`), the lineage reaches a bottleneck in `verify.mk` to satisfy the verification criteria (this step was not available in [9]). 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 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 [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 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 co-authors 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

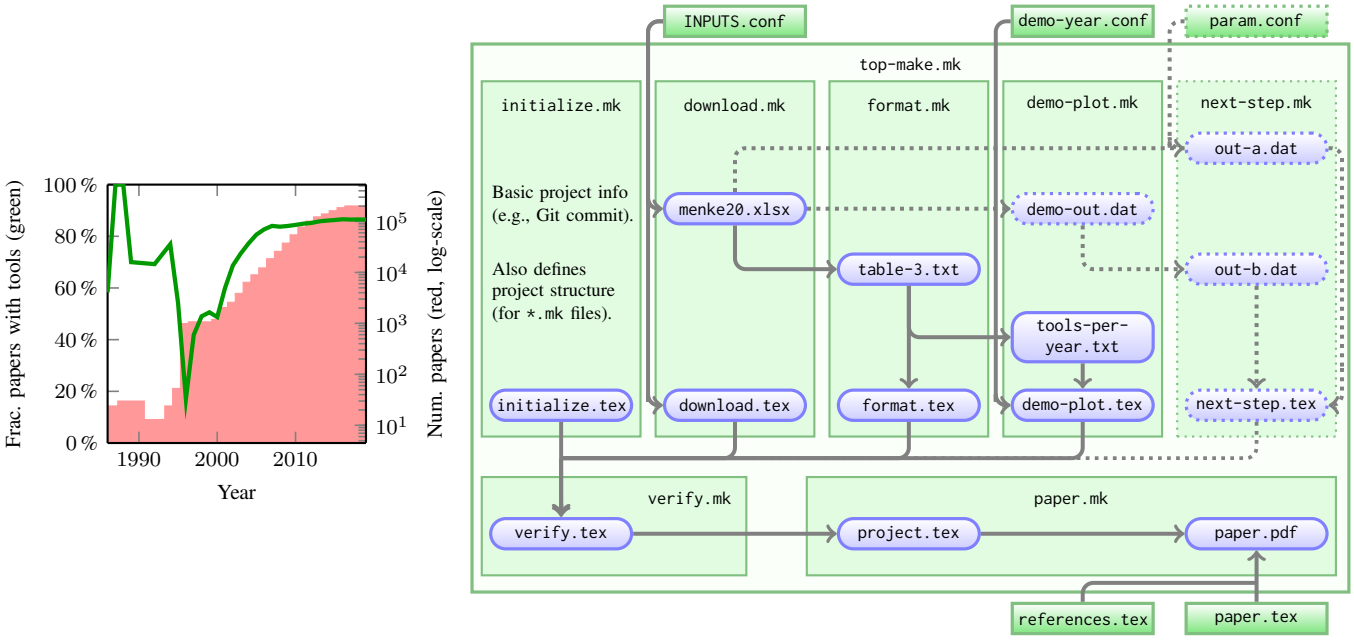

Fig. 1. Left: an enhanced replica of Figure 1C in [12], shown here for demonstrating Maneage. It shows the fraction of the number of papers mentioning software tools (green line, left vertical axis) in each year (red bars, right vertical axis on a log scale). Right: Schematic representation of the data 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 paper. 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 zenodo.4291207/tools-per-year.txt.

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 git.maneage.org/project.git (archived at Software Heritage). 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 at the end of the abstract), as well as the Git hash of the last commit in the Maneage branch (shown in the acknowledgments). These macros are created in `initialize.mk`, with other basic information from the running system like the CPU architecture, byte order or address sizes (shown in the acknowledgments).

Figure 2 shows how projects can re-import 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.

Listing 2
STARTING A NEW PROJECT WITH MANEAGE, AND BUILDING IT

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

# 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            # Re-build to see effect.
$ git add -u && git commit  # Commit changes.
```

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

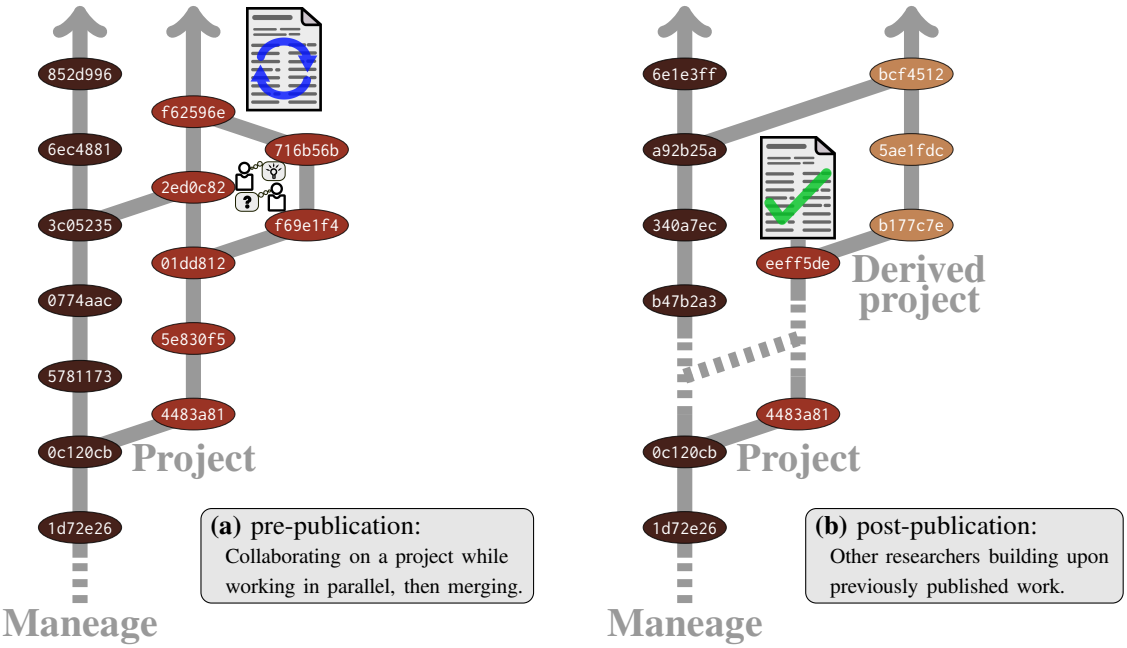

Fig. 2. Maneage is a Git branch. Projects using Maneage are branched off it and apply their customizations. (a) A hypothetical project's history before publication. The low-level structure (in Maneage, shared between all projects) can be updated by merging with Maneage. (b) A finished/published project 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 Appendix A-C. 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.

V. 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).

Firstly, 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 strategies. 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.

Secondly, 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.

Thirdly, Unix-like OSs are a very large and diverse group (mostly conforming with POSIX), so our completeness condition does not guarantee bit-wise 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 (CI) 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 Appendix B-Q). 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 SoftwareHeritage, 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 [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 software used in each project in a tarball, as deliverables. (4) "Linkages between documentation, code, data, and journal articles in an integrated environment", which effectively summarizes the whole purpose of these criteria.

ACKNOWLEDGMENT

The authors wish to thank (sorted alphabetically) Julia Aguilar-Cabello, Dylan Aïssi, Marjan Akbari, Alice Allen, Pedram Ashofteh Ardakani, Roland Bacon, Michael R. Crusoe, Antonio Díaz Díaz, Surena Fatemi, Fabrizio Gagliardi, Konrad Hinsen, Marios Karouzos, Johan Knapen, Tamara Kovazh, Terry Mahoney, Ryan O'Connor, Mervyn O'Luing, Simon Portegies Zwart, Idafen Santana-Pérez, Elham Saremi, Yahya Sefidbakht, Zahra Sharbaf, Nadia Tonello, Ignacio Trujillo and the AMIGA team at the Instituto de Astrofísica de Andalucía for their useful help, suggestions, and feedback on Maneage and this paper. The five referees and editors of CiSE (Lorena Barba and George Thiruvathukal) provided many points that greatly helped to clarify this paper.

This project (commit `eeff5de`) is maintained in Maneage (*Man*aging data lin*eage*). The latest merged Maneage branch commit was `a1a966a` (4 Jan 2021). This project was built on an `x86_64` machine with Little Endian byte-order and address sizes 39 bits physical, 48 bits virtual.

Work on Maneage, and this paper, has been partially funded/supported by the following institutions: The Japanese MEXT PhD scholarship to M.A and its Grant-in-Aid for Scientific Research (21244012, 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 grant agreement No 721463 to the SUNDIAL ITN. The State Research Agency (AEI) of the Spanish Ministry of Science, Innovation and Universities (MCIU) and the European Regional Development Fund (ERDF) under the grant AYA2016-76219-P. The IAC project P/300724, financed by the MCIU, 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 grant DIR/WK/2018/12. 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 Technical Program Expanded Abstracts*, vol. 1, pp. 601–604, DOI:10.1190/1.1822162, 1992.

[2] H. V. Fineberg, D. B. Allison, L. A. Barba et al., "Reproducibility and replicability in science," *The National Academies Press*, pp. 1–256, DOI:10.17226/25303, 2019.

[3] O. Mesnard and L. A. Barba, "Reproducible workflow on a public cloud for computational fluid dynamics," *Computing in Science & Engineering*, vol. 22, pp. 102–116, DOI:10.1109/MCSE.2019.2941702, 2020.

[4] A. Rule, A. Tabard, and J. Hollan, "Exploration and explanation in computational notebooks," *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, vol. 1, p. 30, DOI:10.1145/3173574.3173606, 2018.

[5] P. Alliez, R. Di Cosmo, B. Guedj, A. Girault, M.-S. Hacid, A. Legrand, and N. P. Rougier, "Attributing and Referencing (Research) Software: Best Practices and Outlook from Inria," *Computing in Science & Engineering*, vol. 22, pp. 39–52, arXiv:1905.11123, DOI:10.1109/MCSE.2019.2949413, May 2019.

[6] J. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of jupyter notebooks," *Proceedings of the 16th International Conference on Mining Software Repositories*, vol. 1, pp. 507–517, DOI:10.1109/MSR.2019.00077, 2019.

[7] C. Austin, T. Bloom, S. Dallmeier-Tiessen et al., "Key components of data publishing: using current best practices to develop a reference model for data publishing," *International Journal on Digital Libraries*, vol. 18, pp. 77–92, DOI:10.1007/s00799-016-0178-2, 2017.

[8] M. Akhlaghi, "Carving out the low surface brightness universe with NoiseChisel," *IAU Symposium 355*, vol. arXiv:1909.11230, Sep 2019.

[9] R. Infante-Sainz, I. Trujillo, and J. Román, "The Sloan Digital Sky Survey extended point spread functions," *Monthly Notices of the Royal Astronomical Society*, vol. 491, no. 4, pp. 5317–5329, arXiv:1911.01430, DOI:10.1093/mnras/stz3111, Feb 2020.

[10] M. Akhlaghi and T. Ichikawa, "Noise-based Detection and Segmentation of Nebulous Objects," *The Astrophysical Journal Supplement Series*, vol. 220, pp. 1–33, arXiv:1505.01664, DOI:10.1088/0067-0049/220/1/1 , Sep. 2015.

[11] M. Schwab, M. Karrenbach, and J. F. Claerbout, "Making scientific computations reproducible," *Computing in Science & Engineering*, vol. 2, p. 61, DOI:10.1109/5992.881708, 2000.

[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, p. 101698, DOI:10.1016/j.isci.2020.101698, 2020.

Mohammad Akhlaghi is a postdoctoral researcher at the Instituto de Astrofísica de Canarias (IAC), Spain. He has a PhD from Tohoku University (Japan) and was previously a CNRS postdoc in Lyon (France). Email: mohammad-AT-akhlaghi.org; Website: https://akhlaghi.org.

Raúl Infante-Sainz is a doctoral student at IAC, Spain. He has an M.Sc from the University of Granada (Spain). Email: infantesainz-AT-gmail.com; Website: https://infantesainz.org.

Boudewijn F. Roukema is a professor of cosmology at the Institute of Astronomy, Faculty of Physics, Astronomy and Informatics, Nicolaus Copernicus University in Toruń, Grudziadzka 5, Poland. He has a PhD from Australian National University. Email: boud-AT-astro.uni.torun.pl.

Mohammadreza Khellat is the backend technical services manager at Ideal-Information, Oman. He has an M.Sc in theoretical particle physics from Yazd University (Iran). Email: mkhellat-AT-ideal-information.com.

David Valls-Gabaud is a CNRS Research Director at LERMA, Observatoire de Paris, France. Educated at the universities of Madrid, Paris, and Cambridge, he obtained his PhD in 1991. Email: david.valls-gabaud-AT-obspm.fr.

Roberto Baena-Gallé held a postdoc position at IAC and obtained a degree in Telecommunication and Electronic at the University of Seville, with a PhD at the University of Barcelona. Email: rbaena-AT-iac.es

APPENDIX A
SURVEY OF EXISTING TOOLS FOR VARIOUS PHASES

Data analysis workflows (including those that aim for reproducibility) are commonly high-level frameworks that employ various lower-level components. To help in reviewing existing reproducible workflow solutions in light of the proposed criteria in Appendix B, we first need to survey the most commonly employed lower-level tools.

A. Independent environment

The lowest-level challenge of any reproducible solution is to avoid the differences between various run-time environments, to a desirable/certain level. For example different hardware, operating systems, versions of existing dependencies, etc. Therefore, any reasonable attempt at providing a reproducible workflow starts with isolating its running environment from the host environment. Three general technologies are used for this purpose and reviewed below: 1) Virtual machines, 2) Containers, 3) Independent build in the host's file system.

1) Virtual machines: Virtual machines (VMs) host a binary copy of a full operating system that can be run on other operating systems. This includes the lowest-level operating system component or the kernel. VMs thus provide the ultimate control one can have over the run-time environment of the analysis. However, the VM's kernel does not talk directly to the running hardware that is doing the analysis, it talks to a simulated hardware layer that is provided by the host's kernel. Therefore, a process that is run inside a virtual machine can be much slower than one that is run on a native kernel. An advantage of VMs is that they are a single file that can be copied from one computer to another, keeping the full environment within them if the format is recognized. VMs are used by cloud service providers, enabling fully independent operating systems on their large servers where the customer can have root access.

VMs were used in solutions like SHARE [13] (which was awarded second prize in the Elsevier Executable Paper Grand Challenge of 2011 [14]), or in suggested reproducible papers like [15]. However, due to their very large size, these are expensive to maintain, thus leading SHARE to discontinue its services in 2019. The URL to the VM file `provenance_machine.ova` that is mentioned in [15] is also not currently accessible (we suspect that this is due to size and archival costs).

2) Containers: Containers also host a binary copy of a running environment but do not have their own kernel. Through a thin layer of low-level system libraries, programs running within a container talk directly with the host operating system kernel. Otherwise, containers have their own independent software for everything else. Therefore, they have much less overhead in hardware/CPU access. Like VMs, users often choose an operating system for the container's independent operating system (most commonly GNU/Linux distributions which are free software).

We review some of the most common container solutions: Docker, Singularity, and Podman.

- **Docker containers:** Docker is one of the most popular tools nowadays for keeping an independent analysis environment. It is primarily driven by the need of software developers for reproducing a previous environment, where they have root access mostly on the "cloud" (which is usually a remote VM). A Docker container is composed of independent Docker "images" that are built with a `Dockerfile`. It is possible to precisely version/tag the images that are imported (to avoid downloading the latest/different version in a future build). To have a reproducible Docker image, it must be ensured that all the imported Docker images check their dependency tags down to the initial image which contains the C library. An important drawback of Docker for high-performance scientific needs is that it runs as a daemon (a program that is always running in the background) with root permissions. This is a major security flaw that discourages many high-performance computing (HPC) facilities from providing it.
- **Singularity:** Singularity [16] is a single-image container (unlike Docker, which is composed of modular/independent images). Although it needs root permissions to be installed on the system (once), it does not require root permissions every time it is run. Its main program is also not a daemon, but a normal program that can be stopped. These features make it much safer for HPC administrators to install compared to Docker. However, the fact that it requires root access for the initial install is still a hindrance for a typical project: if Singularity is not already present on the HPC, the user's science project cannot be run by a non-root user.
- **Podman:** Podman uses the Linux kernel containerization features to enable containers without a daemon, and without root permissions. It has a command-line interface very similar to Docker, but only works on GNU/Linux operating systems.

Generally, VMs or containers are good solutions to reproducibly run/repeating an analysis in the short term (a couple of years). However, their focus is to store the already-built (binary, non-human readable) software environment. Because of this, they will be large (many Gigabytes) and expensive to archive, download, or access. Recall the two examples above for VMs in Section A-A1. But this is also valid for Docker images, as is clear from Dockerhub's recent decision to delete images of free accounts that have not been used for more than 6 months. Meng & Thain [17] also give similar reasons on why Docker images were not suitable in their trials.

On a more fundamental level, VMs or containers do not store *how* the core environment was built. This information is usually in a third-party repository, and not necessarily inside the container or VM file, making it hard (if not impossible) to track for future users. This is a major problem in relation to the proposed completeness criteria and is also highlighted as an issue in terms of long term reproducibility by [18].

The example of `Dockerfile` of [3] was previously mentioned in in Section III. Another useful example is the `Dockerfile` of [19] (published in June 2015) which starts with `FROM rocker/verse:3.3.2`. When we tried to build

it (November 2020), we noticed that the core downloaded image (`rocker/verse:3.3.2`, with image "digest" `sha256:c136fb0dbab...`) was created in October 2018 (long after the publication of that paper). In principle, it is possible to investigate the difference between this new image and the old one that the authors used, but that would require a lot of effort and may not be possible when the changes are not available in a third public repository or not under version control. In Docker, it is possible to retrieve the precise Docker image with its digest, for example, `FROM ubuntu:16.04@sha256:XXXXXXX` (where `XXXXXXX` is the digest, uniquely identifying the core image to be used), but we have not seen this often done in existing examples of "reproducible" `Dockerfiles`.

The "digest" is specific to Docker repositories. A more generic/long-term approach to ensure identical core OS components at a later time is to construct the containers or VMs with fixed/archived versions of the operating system ISO files. ISO files are pre-built binary files with volumes of hundreds of megabytes and not containing their build instructions. For example, the archives of Debian[1] or Ubuntu[2] provide older ISO files.

The concept of containers (and the independent images that build them) can also be extended beyond just the software environment. For example, [20] propose a "data pallet" concept to containerize access to data and thus allow tracing data back to the application that produced them.

In summary, containers or VMs are just a built product themselves. If they are built properly (for example building a Maneage'd project inside a Docker container), they can be useful for immediate usage and fast-moving of the project from one system to another. With a robust building, the container or VM can also be exactly reproduced later. However, attempting to archive the actual binary container or VM files as a black box (not knowing the precise versions of the software in them, and *how* they were built) is expensive, and will not be able to answer the most fundamental questions.

3) Independent build in host's file system: The virtual machine and container solutions mentioned above, have their own independent file system. Another approach to having an isolated analysis environment is to use the same file system as the host, but installing the project's software in a non-standard, project-specific directory that does not interfere with the host. Because the environment in this approach can be built in any custom location on the host, this solution generally does not require root permissions or extra low-level layers like containers or VMs. However, "moving" the built product of such solutions from one computer to another is not generally as trivial as containers or VMs. Examples of such third-party package managers (that are detached from the host OS's package manager) include (but are not limited to) Nix, GNU Guix, Python's Virtualenv package, Conda. Because it is highly intertwined with the way software is built and installed, third party package managers are described in more detail as part of Section A-B.

[1] `https://cdimage.debian.org/mirror/cdimage/archive/`
[2] `http://old-releases.ubuntu.com/releases`

Maneage (the solution proposed in this paper) also follows a similar approach of building and installing its own software environment within the host's file system, but without depending on it beyond the kernel. However, unlike the third-party package manager mentioned above, Maneage'd software management is not detached from the specific research/analysis project: the instructions to build the full isolated software environment is maintained with the high-level analysis steps of the project, and the narrative paper/report of the project. This is fundamental to achieve the Completeness criteria.

B. Package management

Package management is the process of automating the build and installation of a software environment. A package manager thus contains the following information on each software package that can be run automatically: the URL of the software's tarball, the other software that it possibly depends on, and how to configure and build it. Package managers can be tied to specific operating systems at a very low level (like `apt` in Debian-based OSs). Alternatively, there are third-party package managers that can be installed on many OSs. Both are discussed in more detail below.

Package managers are the second component in any workflow that relies on containers or VMs for an independent environment, and the starting point in others that use the host's file system (as discussed above in Section A-A). In this section, some common package managers are reviewed, in particular those that are most used by the reviewed reproducibility solutions of Appendix B. For a more comprehensive list of existing package managers, see Wikipedia. Note that we are not including package managers that are specific to one language, for example `pip` (for Python) or `tlmgr` (for LaTeX).

1) Operating system's package manager: The most commonly used package managers are those of the host operating system, for example, `apt`, `yum` or `pkg` which are respectively used in Debian-based, Red Hat-based and FreeBSD-based OSs (among many other OSs).

These package managers are tightly intertwined with the operating system: they also include the building and updating of the core kernel and the C library. Because they are part of the OS, they also commonly require root permissions. Also, it is usually only possible to have one version/configuration of the software at any moment and downgrading versions for one project, may conflict with other projects, or even cause problems in the OS. Hence if two projects need different versions of the software, it is not possible to work on them at the same time in the OS.

When a container or virtual machine (see Appendix A-A) is used for each project, it is common for projects to use the containerized operating system's package manager. However, it is important to remember that operating system package managers are not static: software is updated on their servers. Hence, simply running `apt install gcc`, will install different versions of the GNU Compiler Collection (GCC) based on the version of the OS and when it has been run. Requesting a special version of that special software does not fully address the problem because the package managers also download

and install its dependencies. Hence a fixed version of the dependencies must also be specified.

In robust package managers like Debian's `apt` it is possible to fully control (and later reproduce) the built environment of a high-level software. Debian also archives all packaged high-level software in its Snapshot[3] service since 2005 which can be used to build the higher-level software environment on an older OS [21]. Therefore it is indeed theoretically possible to reproduce the software environment only using archived operating systems and their own package managers, but unfortunately, we have not seen it practiced in (reproducible) scientific papers/projects.

In summary, the host OS package managers are primarily meant for the low-level operating system components. Hence, many robust reproducible analysis workflows (reviewed in Appendix B) do not use the host's package manager, but an independent package manager, like the ones discussed below.

2) Blind packaging of already built software: An already-built software contains links to the system libraries it uses. Therefore one way of packaging a software is to look into the binary file for the libraries it uses and bring them into a file with the executable so on different systems, the same set of dependencies are moved around with the desired software. Tools like AppImage[4], Flatpak[5] or Snap[6] are designed for this purpose: the software's binary product and all its dependencies (not including the core C library) are packaged into one file. This makes it very easy to move that single software's built product and already built dependencies to different systems. However, because the C library is not included, it can fail on newer/older systems (depending on the system it was built on). We call this method "blind" packaging because it is agnostic to *how* the software and its dependencies were built (which is important in a scientific context). Moreover, these types of packagers are designed for the Linux kernel (using its containerization and unique mounting features). They can therefore only be run on GNU/Linux operating systems.

3) Nix or GNU Guix: Nix[7] [22] and GNU Guix[8] [23] are independent package managers that can be installed and used on GNU/Linux operating systems, and macOS (only for Nix, prior to macOS Catalina). Both also have a fully functioning operating system based on their packages: NixOS and "Guix System". GNU Guix is based on the same principles of Nix but implemented differently, so we focus the review here on Nix.

The Nix approach to package management is unique in that it allows exact dependency tracking of all the dependencies, and allows for multiple versions of software, for more details see [22]. In summary, a unique hash is created from all the components that go into the building of the package (including the instructions on how to build the software). That hash is then prefixed to the software's installation directory. As an example from [22]: if a certain build of GNU C Library 2.3.2 has a hash of `8d013ea878d0`, then it is installed under `/nix/store/8d013ea878d0-glibc-2.3.2` and all software that is compiled with it (and thus need it to run) will link to this unique address. This allows for multiple versions of the software to co-exist on the system, while keeping an accurate dependency tree.

As mentioned in [23], one major caveat with using these package managers is that they require a daemon with root privileges (failing our completeness criteria). This is necessary "to use the Linux kernel container facilities that allow it to isolate build processes and maximize build reproducibility". This is because the focus in Nix or Guix is to create bit-wise reproducible software binaries and this is necessary for the security or development perspectives. However, in a non-computer-science analysis (for example natural sciences), the main aim is reproducible *results* that can also be created with the same software version that may not be bit-wise identical (for example when they are installed in other locations, because the installation location is hard-coded in the software binary or for a different CPU architecture).

Finally, while Guix and Nix do allow precisely reproducible environments, it requires extra effort on the user's side to ensure that the built environment is reproducible later. For example, simply running `guix install gcc` (the most common way to install a new software) will install the most recent version of GCC, that can be different at different times. Hence, similar to the discussion in host operating system package managers, it is up to the user to ensure that their created environment is recorded properly for reproducibility in the future. It is not a complex operation, but like the Docker digest codes mentioned in Appendix A-A2, many will probably not know, forget or ignore it. Generally, this is an issue with projects that rely on detached (third party) package managers for building their software, including the other tools mentioned below. We solved this problem in Maneage by including the package manager and analysis steps into one project: it is simply not possible to forget to record the exact versions of the software used.

4) Conda/Anaconda: Conda is an independent package manager that can be used on GNU/Linux, macOS, or Windows operating systems, although all software packages are not available in all operating systems. Conda is able to maintain an approximately independent environment on an operating system without requiring root access.

Conda tracks the dependencies of a package/environment through a YAML formatted file, where the necessary software and their acceptable versions are listed. However, it is not possible to fix the versions of the dependencies through the YAML files alone. This is thoroughly discussed under issue 787 (in May 2019) of `conda-forge`[9]. In that discussion, the authors of [24] report that the half-life of their environment (defined in a YAML file) is 3 months, and that at least one of their dependencies breaks shortly after this period. The main reply they got in the discussion is to build the Conda environment in a container, which is also the suggested solution by [25]. However, as described in Appendix A-A,

[3] https://snapshot.debian.org/
[4] https://appimage.org
[5] https://flatpak.org
[6] https://snapcraft.io
[7] https://nixos.org
[8] https://guix.gnu.org

[9] https://github.com/conda-forge/conda-forge.github.io/issues/787

containers just hide the reproducibility problem, they do not fix it: containers are not static and need to evolve (i.e., get rebuilt) with the project. Given these limitations, [24] are forced to host their conda-packaged software as tarballs on a separate repository.

Conda installs with a shell script that contains a binary-blob (+500 megabytes, embedded in the shell script). This is the first major issue with Conda: from the shell script, it is not clear what is in this binary blob and what it does. After installing Conda in any location, users can easily activate that environment by loading a special shell script. However, the resulting environment is not fully independent of the host operating system as described below:

- The Conda installation directory is present at the start of environment variables like `PATH` (which is used to find programs to run) and other such environment variables. However, the host operating system's directories are also appended afterward. Therefore, a user or script may not notice that the software that is being used is actually coming from the operating system, and not from the controlled Conda installation.
- Generally, by default, Conda relies heavily on the operating system and does not include core commands like `mkdir` (to make a directory), `ls` (to list files) or `cp` (to copy). Although a minimal functionality is defined for them in POSIX and generally behave similarly for basic operations on different Unix-like operating systems, they have their differences. For example, `mkdir -p` is a common way to build directories, but this option is only available with the `mkdir` of GNU Coreutils (default on GNU/Linux systems and installable in almost all Unix-like OSs). Running the same command within a Conda environment that does not include GNU Coreutils on a macOS would crash. Important packages like GNU Coreutils are available in channels like conda-forge, but they are not the default. Therefore, many users may not recognize this, and failing to account for it, will cause unexpected crashes when the project is run on a new system.
- Many major Conda packaging "channels" (for example the core Anaconda channel, or very popular conda-forge channel) do not include the C library, that a package was built with, as a dependency. They rely on the host operating system's C library. C is the core language of modern operating systems and even higher-level languages like Python or R are written in it, and need it to run. Therefore if the host operating system's C library is different from the C library that a package was built with, a Conda-packaged program will crash and the project will not be executable. Theoretically, it is possible to define a new Conda "channel" which includes the C library as a dependency of its software packages, but it will take too much time for any individual team to practically implement all their necessary packages, up to their high-level science software.
- Conda does allow a package to depend on a special build of its prerequisites (specified by a checksum, fixing its version and the version of its dependencies). However, this is rarely practiced in the main Git repositories of channels like Anaconda and conda-forge: only the name of the high-level prerequisite packages is listed in a package's `meta.yaml` file, which is version-controlled. Therefore two builds of the package from the same Git repository will result in different tarballs (depending on what prerequisites were present at build time). In Conda's downloaded tarball (that contains the built binaries and is not under version control) the exact versions of most build-time dependencies are listed. However, because the different software of one project may have been built at different times, if they depend on different versions of a single software there will be a conflict and the tarball cannot be rebuilt, or the project cannot be run.

As reviewed above, the low-level dependence of Conda on the host operating system's components and build-time conditions, is the primary reason that it is very fast to install (thus making it an attractive tool to software developers who just need to reproduce a bug in a few minutes). However, these same factors are major caveats in a scientific scenario, where long-term archivability, readability, or usability are important.

5) Spack: Spack is a package manager that is also influenced by Nix (similar to GNU Guix), see [26]. But unlike Nix or GNU Guix, it does not aim for full, bit-wise reproducibility and can be built without root access in any generic location. It relies on the host operating system for the C library.

Spack is fully written in Python, where each software package is an instance of a class, which defines how it should be downloaded, configured, built, and installed. Therefore if the proper version of Python is not present, Spack cannot be used and when incompatibilities arise in future versions of Python (similar to how Python 3 is not compatible with Python 2), software building recipes, or the whole system, have to be upgraded. Because of such bootstrapping problems (for example how Spack needs Python to build Python and other software), it is generally a good practice to use simpler, lower-level languages/systems for a low-level operation like package management.

In conclusion for all package managers, there are two common issues regarding generic package managers that hinder their usage for high-level scientific projects:

- **Pre-compiled/binary downloads:** Most package managers primarily download the software in a binary (pre-compiled) format. This allows users to download it very fast and almost instantaneously be able to run it. However, to provide for this, servers need to keep binary files for each build of the software on different operating systems (for example Conda needs to keep binaries for Windows, macOS and GNU/Linux operating systems). It is also necessary for them to store binaries for each build, which includes different versions of its dependencies. Maintaining such a large binary library is expensive, therefore once the shelf-life of a binary has expired, it will be removed, causing problems for projects that depend on them.
- **Adding high-level software:** Packaging new software is not trivial and needs a good level of knowledge/experience

with that package manager. For example, each one has its own special syntax/standards/languages, with pre-defined variables that must already be known before someone can package new software for them. However, in many research projects, the most high-level analysis software is written by the team that is doing the research, and they are its primary/only users, even when the software is distributed with free licenses on open repositories.

Although active package manager members are commonly very supportive in helping to package new software, many teams may not be able to make that extra effort and time investment to package their most high-level (i.e., relevant) software in it. As a result, they manually install their high-level software in an uncontrolled, or non-standard way, thus jeopardizing the reproducibility of the whole work. This is another consequence of the detachment of the package manager from the project doing the analysis.

Addressing these issues has been the basic reason behind the proposed solution: based on the completeness criteria, instructions to download and build the packages are included within the actual science project, and no special/new syntax/language is used. Software download, built and installation is done with the same language/syntax that researchers manage their research: using the shell (by default GNU Bash in Maneage) for low-level steps and Make (by default, GNU Make in Maneage) for job management.

C. Version control

A scientific project is not written in a day; it usually takes more than a year. During this time, the project evolves significantly from its first starting date, and components are added or updated constantly as it approaches completion. Added with the complexity of modern computational projects, is not trivial to manually track this evolution, and the evolution's affect of on the final output: files produced in one stage of the project can mistakenly be used by an evolved analysis environment in later stages (where the project has evolved).

Furthermore, scientific projects do not progress linearly: earlier stages of the analysis are often modified after later stages are written. This is a natural consequence of the scientific method; where progress is defined by experimentation and modification of hypotheses (results from earlier phases).

It is thus very important for the integrity of a scientific project that the state/version of its processing is recorded as the project evolves. For example, better methods are found or more data arrive. Any intermediate dataset that is produced should also be tagged with the version of the project at the time it was created. In this way, later processing stages can make sure that they can safely be used, i.e., no change has been made in their processing steps.

Solutions to keep track of a project's history have existed since the early days of software engineering in the 1970s and they have constantly improved over the last decades. Today the distributed model of "version control" is the most common, where the full history of the project is stored locally on different systems and can easily be integrated. There are many existing version control solutions, for example, CVS, SVN, Mercurial, GNU Bazaar, or GNU Arch. However, currently, Git is by far the most commonly used in individual projects. Git is also the foundation upon which this paper's proof of concept (Maneage) is built. Archival systems aiming for long term preservation of software like Software Heritage [27] are also modeled on Git. Hence we will just review Git here, but the general concept of version control is the same in all implementations.

1) Git: With Git, changes in a project's contents are accurately identified by comparing them with their previous version in the archived Git repository. When the user decides the changes are significant compared to the archived state, they can be "committed" into the history/repository. The commit involves copying the changed files into the repository and calculating a 40 character checksum/hash that is calculated from the files, an accompanying "message" (a narrative description of the purpose/goals of the changes), and the previous commit (thus creating a "chain" of commits that are strongly connected to each other like Figure 2). For example `f4953ccf1ca8a33616ad602ddf4cd189c2eff97b` is a commit identifier in the Git history of this project. Commits are is commonly summarized by the checksum's first few characters, for example, `f4953cc` of the example above.

With Git, making parallel "branches" (in the project's history) is very easy and its distributed nature greatly helps in the parallel development of a project by a team. The team can host the Git history on a web page and collaborate through that. There are several Git hosting services for example codeberg.org, gitlab.com, bitbucket.org or github.com (among many others). Storing the changes in binary files is also possible in Git, however it is most useful for human-readable plain-text sources.

D. Job management

Any analysis will involve more than one logical step. For example, it is first necessary to download a dataset and do some preparations on it before applying the research software on it, and finally to make visualizations/tables that can be imported into the final report. Each one of these is a logically independent step, which needs to be run before/after the others in a specific order.

Hence job management is a critical component of a research project. There are many tools for managing the sequence of jobs, below we review the most common ones that are also used the existing reproducibility solutions of Appendix B and Maneage.

1) Manual operation with narrative: The most commonly used workflow system for many researchers is to run the commands, experiment on them, and keep the output when they are happy with it (therefore loosing the actual command that produced it). As an improvement, some researchers also keep a narrative description in a text file, and keep a copy of the command they ran. At least in our personal experience with colleagues, this method is still being heavily practiced by many researchers. Given that many researchers do not get trained well in computational methods, this is not surprising.

As discussed in Section V, based on this observation we believe that improved literacy in computational methods is the single most important factor for the integrity/reproducibility of modern science.

2) Scripts: Scripts (in any language, for example GNU Bash, or Python) are the most common ways of organizing a series of steps. They are primarily designed to execute each step sequentially (one after another), making them also very intuitive. However, as the series of operations become complex and large, managing the workflow in a script will become highly complex.

For example, if 90% of a long project is already done and a researcher wants to add a followup step, a script will go through all the previous steps every time it is run (which can take significant time). In other scenarios, when a small step in the middle of the analysis has to be changed, the full analysis needs to be re-run from the start. Scripts have no concept of dependencies, forcing authors to "temporarily" comment parts that they do not want to be re-run. Therefore forgetting to un-comment them afterwards is the most common cause of frustration.

This discourages experimentation, which is a critical component of the scientific method. It is possible to manually add conditionals all over the script, thus manually defining dependencies, or only run certain steps at certain times, but they just make it harder to read, add logical complexity and introduce many bugs themselves. Parallelization is another drawback of using scripts. While it is not impossible, because of the high-level nature of scripts, it is not trivial and parallelization can also be very inefficient or buggy.

3) Make: Make was originally designed to address the problems mentioned above for scripts [28]. In particular, it was originally designed in the context of managing the compilation of software source code that are distributed in many files. With Make, the source files of a program that have not been changed are not recompiled. Moreover, when two source files do not depend on each other, and both need to be rebuilt, they can be built in parallel. This was found to greatly help in debugging software projects, and in speeding up test builds, giving Make a core place in software development over the last 40 years.

The most common implementation of Make, since the early 1990s, is GNU Make. Make was also the framework used in the first attempts at reproducible scientific papers [1], [11]. Our proof-of-concept (Maneage) also uses Make to organize its workflow. Here, we complement that section with more technical details on Make.

Usually, the top-level Make instructions are placed in a file called Makefile, but it is also common to use the `.mk` suffix for custom file names. Each stage/step in the analysis is defined through a *rule*. Rules define *recipes* to build *targets* from *prerequisites*. In Unix-like operating systems, everything is a file, even directories and devices. Therefore all three components in a rule must be files on the running filesystem.

To decide which operation should be re-done when executed, Make compares the timestamp of the targets and prerequisites. When any of the prerequisite(s) is newer than a target, the recipe is re-run to re-build the target. When all the prerequisites are older than the target, that target does not need to be rebuilt. A recipe is just a bundle or shell commands that are executed if necessary. Going deeper into the syntax of Make is beyond the scope of this paper, but we recommend interested readers to consult the GNU Make manual for a very good introduction[10].

4) Snakemake: Snakemake is a Python-based workflow management system, inspired by GNU Make (discussed above). It is aimed at reproducible and scalable data analysis [29][11]. It defines its own language to implement the "rule" concept of Make within Python. Technically, calling command-line programs within Python is very slow, and using complex shell scripts in each step will involve a lot of quotations that make the code hard to read.

Currently, Snakemake requires Python 3.5 (released in September 2015) and above, while Snakemake was originally introduced in 2012. Hence it is not clear if older Snakemake source files can be executed today. As reviewed in many tools here, depending on high-level systems for low-level project components causes a major bootstrapping problem that reduces the longevity of a project.

5) Bazel: Bazel[12] is a high-level job organizer that depends on Java and Python and is primarily tailored to software developers (with features like facilitating linking of libraries through its high-level constructs).

6) SCons: Scons[13] is a Python package for managing operations outside of Python (in contrast to CGAT-core, discussed below, which only organizes Python functions). In many aspects it is similar to Make, for example, it is managed through a 'SConstruct' file. Like a Makefile, SConstruct is also declarative: the running order is not necessarily the top-to-bottom order of the written operations within the file (unlike the imperative paradigm which is common in languages like C, Python, or FORTRAN). However, unlike Make, SCons does not use the file modification date to decide if it should be remade. SCons keeps the MD5 hash of all the files in a hidden binary file and checks them to see if it is necessary to re-run.

SCons thus attempts to work on a declarative file with an imperative language (Python). It also goes beyond raw job management and attempts to extract information from within the files (for example to identify the libraries that must be linked while compiling a program). SCons is, therefore, more complex than Make and its manual is almost double that of GNU Make. Besides added complexity, all these "smart" features decrease its performance, especially as files get larger and more numerous: on every call, every file's checksum has to be calculated, and a Python system call has to be made (which is computationally expensive).

Finally, it has the same drawback as any other tool that uses high-level languages, see Section A-F. We encountered such a problem while testing SCons: on the Debian-10 testing system, the `python` program pointed to Python 2. However, since Python 2 is now obsolete, SCons was built with Python 3 and our first run crashed. To fix it, we had to either manually change the core operating system path, or the SCons source

[10] http://www.gnu.org/software/make/manual/make.pdf
[11] https://snakemake.readthedocs.io/en/stable
[12] https://bazel.build
[13] https://scons.org

hashbang. The former will conflict with other system tools that assume `python` points to Python-2, the latter may need root permissions for some systems. This can also be problematic when a Python analysis library, may require a Python version that conflicts with the running SCons.

7) CGAT-core: CGAT-Core is a Python package for managing workflows, see [30]. It wraps analysis steps in Python functions and uses Python decorators to track the dependencies between tasks. It is used in papers like [31]. However, as mentioned in [31] it is good for managing individual outputs (for example separate figures/tables in the paper, when they are fully created within Python). Because it is primarily designed for Python tasks, managing a full workflow (which includes many more components, written in other languages) is not trivial. Another drawback with this workflow manager is that Python is a very high-level language where future versions of the language may no longer be compatible with Python 3, that CGAT-core is implemented in (similar to how Python 2 programs are not compatible with Python 3).

8) Guix Workflow Language (GWL): GWL is based on the declarative language that GNU Guix uses for package management (see Appendix A-B), which is itself based on the general purpose Scheme language. It is closely linked with GNU Guix and can even install the necessary software needed for each individual process. Hence in the GWL paradigm, software installation and usage does not have to be separated. GWL has two high-level concepts called "processes" and "workflows" where the latter defines how multiple processes should be executed together.

9) Nextflow (2013): Nextflow[14] [32] workflow language with a command-line interface that is written in Java.

10) Generic workflow specifications (CWL and WDL): Due to the variety of custom workflows used in existing reproducibility solution (like those of Appendix B), some attempts have been made to define common workflow standards like the Common workflow language (CWL[15], with roots in Make, formatted in YAML or JSON) and Workflow Description Language (WDL[16], formatted in JSON). These are primarily specifications/standards rather than software. With these standards, ideally, translators can be written between the various workflow systems to make them more interoperable.

In conclusion, shell scripts and Make are very common and extensively used by users of Unix-based OSs (which are most commonly used for computations). They have also existed for several decades and are robust and mature. Many researchers that use heavy computations are also already familiar with them and have already used them already (to different levels). As we demonstrated above in this appendix, the list of necessary tools for the various stages of a research project (an independent environment, package managers, job organizers, analysis languages, writing formats, editors, etc) is already very large. Each software/tool/paradigm has its own learning curve, which is not easy for a natural or social scientist for example (who need to put their primary focus on their own scientific domain). Most workflow management tools and the reproducible workflow solutions that depend on them are, yet another language/paradigm that has to be mastered by researchers and thus a heavy burden.

Furthermore as shown above (and below) high-level tools will evolve very fast causing disruptions in the reproducible framework. A good example is Popper [33] which initially organized its workflow through the HashiCorp configuration language (HCL) because it was the default in GitHub. However, in September 2019, GitHub dropped HCL as its default configuration language, so Popper is now using its own custom YAML-based workflow language, see Appendix B-W for more on Popper.

E. Editing steps and viewing results

In order to reproduce a project, the analysis steps must be stored in files. For example Shell, Python, R scripts, Makefiles, Dockerfiles, or even the source files of compiled languages like C or FORTRAN. Given that a scientific project does not evolve linearly and many edits are needed as it evolves, it is important to be able to actively test the analysis steps while writing the project's source files. Here we review some common methods that are currently used.

1) Text editors: The most basic way to edit text files is through simple text editors which just allow viewing and editing such files, for example, `gedit` on the GNOME graphic user interface. However, working with simple plain text editors like `gedit` can be very frustrating since its necessary to save the file, then go to a terminal emulator and execute the source files. To solve this problem there are advanced text editors like GNU Emacs that allow direct execution of the script, or access to a terminal within the text editor. However, editors that can execute or debug the source (like GNU Emacs), just run external programs for these jobs (for example GNU GCC, or GNU GDB), just as if those programs was called from outside the editor.

With text editors, the final edited file is independent of the actual editor and can be further edited with another editor, or executed without it. This is a very important feature and corresponds to the modularity criteria of this paper. This type of modularity is not commonly present for other solutions mentioned below (the source can only be edited/run in a specific browser). Another very important advantage of advanced text editors like GNU Emacs or Vi(m) is that they can also be run without a graphic user interface, directly on the command-line. This feature is critical when working on remote systems, in particular high performance computing (HPC) facilities that do not provide a graphic user interface. Also, the commonly used minimalistic containers do not include a graphic user interface. Hence by default all Maneage'd projects also build the simple GNU Nano plain-text editor as part of the project (to be able to edit the source directly within a minimal environment). Maneage can also also optinally build GNU Emacs or Vim, but its up to the user to build them (same as their high-level science software).

2) Integrated Development Environments (IDEs): To facilitate the development of source code in special programming

[14] https://www.nextflow.io
[15] https://www.commonwl.org
[16] https://openwdl.org

languages, IDEs add software building and running environments as well as debugging tools to a plain text editor. Many IDEs have their own compilers and debuggers, hence source files that are maintained in IDEs are not necessarily usable/portable on other systems. Furthermore, they usually require a graphic user interface to run. In summary, IDEs are generally very specialized tools, for special projects and are not a good solution when portability (the ability to run on different systems and at different times) is required.

3) Jupyter: Jupyter (initially IPython) [34] is an implementation of Literate Programming [35]. Jupyter's name is a combination of the three main languages it was designed for: Julia, Python, and R. The main user interface is a web-based "notebook" that contains blobs of executable code and narrative. Jupyter uses the custom built `.ipynb` format[17]. The `.ipynb` format, is a simple, human-readable format that can be opened in a plain-text editor) and formatted in JavaScript Object Notation (JSON). It contains various kinds of "cells", or blobs, that can contain narrative description, code, or multimedia visualizations (for example images/plots), that are all stored in one file. The cells can have any order, allowing the creation of a literal programming style graphical implementation, where narrative descriptions and executable patches of code can be intertwined. For example to have a paragraph of text about a patch of code, and run that patch immediately on the same page.

The `.ipynb` format does theoretically allow dependency tracking between cells, see IPython mailing list (discussion started by Gabriel Becker from July 2013[18]). Defining dependencies between the cells can allow non-linear execution which is critical for large scale (thousands of files) and complex (many dependencies between the cells) operations. It allows automation, run-time optimization (deciding not to run a cell if it is not necessary), and parallelization. However, Jupyter currently only supports a linear run of the cells: always from the start to the end. It is possible to manually execute only one cell, but the previous/next cells that may depend on it, also have to be manually run (a common source of human error, and frustration for complex operations). Integration of directional graph features (dependencies between the cells) into Jupyter has been discussed, but as of this publication, there is no plan to implement it (see Jupyter's GitHub issue 1175[19]).

The fact that the `.ipynb` format stores narrative text, code, and multi-media visualization of the outputs in one file, is another major hurdle and against the modularity criteria proposed here. The files can easily become very large (in volume/bytes) and hard to read when the Jupyter web-interface is not accessible. Both are critical for scientific processing, especially the latter: when a web browser with proper JavaScript features is not available (can happen in a few years). This is further exacerbated by the fact that binary data (for example images) are not directly supported in JSON and have to be converted into a much less memory-efficient textual encoding.

Finally, Jupyter has an extremely complex dependency graph: on a clean Debian 10 system, Pip (a Python package manager that is necessary for installing Jupyter) required 19 dependencies to install, and installing Jupyter within Pip needed 41 dependencies. [36] reported such conflicts when building Jupyter into the Active Papers framework (see Appendix B-K). However, the dependencies above are only on the server-side. Since Jupyter is a web-based system, it requires many dependencies on the viewing/running browser also (for example special JavaScript or HTML5 features, which evolve very fast). As discussed in Appendix A-F having so many dependencies is a major caveat for any system regarding scientific/long-term reproducibility. In summary, Jupyter is most useful in manual, interactive, and graphical operations for temporary operations (for example educational tutorials).

F. Project management in high-level languages

Currently, the most popular high-level data analysis language is Python. R is closely tracking it and has superseded Python in some fields, while Julia [37] is quickly gaining ground. These languages have themselves superseded previously popular languages for data analysis of the previous decades, for example, Java, Perl, or C++. All are part of the C-family programming languages. In many cases, this means that the language's execution environment are themselves written in C, which is the language of modern operating systems.

Scientists, or data analysts, mostly use these higher-level languages. Therefore they are naturally drawn to also apply the higher-level languages for lower-level project management, or designing the various stages of their workflow. For example Conda or Spack (Appendix A-B), CGAT-core (Appendix A-D), Jupyter (Appendix A-E) or Popper (Appendix B-W) are written in Python. The discussion below applies to both the actual analysis software and project management software. In this context, it is more focused on the latter.

Because of their nature, higher-level languages evolve very fast, creating incompatibilities on the way. The most prominent example is the transition from Python 2 (released in 2000) to Python 3 (released in 2008). Python 3 was incompatible with Python 2 and it was decided to abandon the former by 2015. However, due to community pressure, this was delayed to January 1st, 2020. The end-of-life of Python 2 caused many problems for projects that had invested heavily in Python 2: all their previous work had to be translated, for example, see [38] or Appendix B-R. Some projects could not make this investment and their developers decided to stop maintaining it, for example VisTrails (see Appendix B-G).

The problems were not just limited to translation. Python 2 was still being actively used during the transition period (and is still being used by some, after its end-of-life). Therefore, developers had to maintain (for example fix bugs in) both versions in one package. This is not particular to Python, a similar evolution occurred in Perl: in 2000 it was decided to improve Perl 5, but the proposed Perl 6 was incompatible with it. However, the Perl community decided not to abandon Perl 5, and Perl 6 was eventually defined as a new language that is now officially called "Raku" (https://raku.org).

It is unreasonably optimistic to assume that high-level languages will not undergo similar incompatible evolutions in the

[17] https://nbformat.readthedocs.io/en/latest
[18] https://mail.python.org/pipermail/ipython-dev/2013-July/010725.html
[19] https://github.com/jupyter/notebook/issues/1175

(not too distant) future. For industial software developers, this is not a major problem: non-scientific software, and the general population's usage of them, has a similarly fast evolution and shelf-life. Hence, it is rarely (if ever) necessary to look into industrial/business codes that are more than a couple of years old. However, in the sciences (which are commonly funded by public money) this is a major caveat for the longer-term usability of solutions.

In summary, in this section we are discussing the bootstrapping problem as regards scientific projects: the workflow/pipeline can reproduce the analysis and its dependencies. However, the dependencies of the workflow itself should not be ignored. Beyond technical, low-level, problems for the developers mentioned above, this causes major problems for scientific project management as listed below:

1) Dependency hell: The evolution of high-level languages is extremely fast, even within one version. For example, packages that are written in Python 3 often only work with a special interval of Python 3 versions. For example Snakemake and Occam which can only be run on Python versions 3.4 and 3.5 or newer respectively, see Appendices A-D4 and B-Y. This is not just limited to the core language, much faster changes occur in their higher-level libraries. For example version 1.9 of Numpy (Python's numerical analysis module) discontinued support for Numpy's predecessor (called Numeric), causing many problems for scientific users [36].

On the other hand, the dependency graph of tools written in high-level languages is often extremely complex. For example, see Figure 1 of [5], it shows the dependencies and their inter-dependencies for Matplotlib (a popular plotting module in Python). Acceptable version intervals between the dependencies will cause incompatibilities in a year or two, when a robust package manager is not used (see Appendix A-B).

Since a domain scientist does not always have the resources/knowledge to modify the conflicting part(s), many are forced to create complex environments with different versions of Python and pass the data between them (for example just to use the work of a previous PhD student in the team). This greatly increases the complexity of the project, even for the principal author. A well-designed reproducible workflow like Maneage that has no dependencies beyond a C compiler in a Unix-like operating system can account for this. However, when the actual workflow system (not the analysis software) is written in a high-level language like the examples above.

Another relevant example of the dependency hell is mentioned here: merely installing the Python installer (`pip`) on a Debian system (with `apt install pip2` for Python 2 packages), required 32 other packages as dependencies. `pip` is necessary to install Popper and Sciunit (Appendices B-W and B-R). As of this writing, the `pip3 install popper` and `pip2 install sciunit2` commands for installing each, required 17 and 26 Python modules as dependencies. It is impossible to run either of these solutions if there is a single conflict in this very complex dependency graph. This problem actually occurred while we were testing Sciunit: even though it was installed, it could not run because of conflicts (its last commit was only 1.5 years old), for more see Appendix B-R. [36] also report a similar problem when attempting to install Jupyter (see Appendix A-E). Of course, this also applies to tools that these systems use, for example Conda (which is also written in Python, see Appendix A-B).

2) Generational gap: This occurs primarily for domain scientists (for example astronomers, biologists, or social sciences). Once they have mastered one version of a language (mostly in the early stages of their career), they tend to ignore newer versions/languages. The inertia of programming languages is very strong. This is natural because they have their own science field to focus on, and re-writing their high-level analysis toolkits (which they have curated over their career and is often only readable/usable by themselves) in newer languages every few years is not practically possible.

When this investment is not possible, either the mentee has to use the mentor's old method (and miss out on all the newly fashionable tools that many are talking about), or the mentor has to avoid implementation details in discussions with the mentee because they do not share a common language. The authors of this paper have personal experiences in both mentor/mentee relational scenarios. This failure to communicate in the details is a very serious problem, leading to the loss of valuable inter-generational experience.

APPENDIX B
SURVEY OF COMMON EXISTING REPRODUCIBLE WORKFLOWS

The problem of reproducibility has received considerable attention over the last three decades and various solutions have already been proposed. The core principles that many of the existing solutions (including Maneage) aim to achieve are nicely summarized by the FAIR principles [39]. In this appendix, *some* of the solutions are reviewed. We are not just reviewing solutions that can be used today. The main focus of this paper is longevity, therefore we also spent considerable time on finding and inspecting solutions that have been aborted, discontinued or abandoned.

The solutions are based on an evolving software landscape, therefore they are ordered by date: when the project has a web page, the year of its first release is used for the sorting. Otherwise their paper's publication year is used. For each solution, we summarize its methodology and discuss how it relates to the criteria proposed in this paper. Freedom of the software/method is a core concept behind scientific reproducibility, as opposed to industrial reproducibility where a black box is acceptable/desirable. Therefore proprietary solutions like Code Ocean[20] or Nextjournal[21] will not be reviewed here. Other studies have also attempted to review existing reproducible solutions, for example, see [40].

We have tried our best to test and read through the documentation of almost all reviewed solutions to a sufficient level. However, due to time constraints, it is inevitable that we may have missed some aspects the solutions, or incorrectly interpreted their behavior and outputs. In this case, please let us know and we will correct it in the text on the paper's Git repository and publish the updated PDF on zenodo.3872247

[20] https://codeocean.com
[21] https://nextjournal.com

(this is the version-independent DOI, that always points to the most recent Zenodo upload).

A. Suggested rules, checklists, or criteria

Before going into the various implementations, it is useful to review some existing suggested rules, checklists, or criteria for computationally reproducible research.

Sandve et al. [41] propose "ten simple rules for reproducible computational research" that can be applied in any project. Generally, these are very similar to the criteria proposed here and follow a similar spirit, but they do not provide any actual research papers following up all those points, nor do they provide a proof of concept. The Popper convention [33] also provides a set of principles that are indeed generally useful, among which some are common to the criteria here (for example, automatic validation, and, as in Maneage, the authors suggest providing a template for new users), but the authors do not include completeness as a criterion nor pay attention to longevity: Popper has already changed its core workflow language once and is written in Python with many dependencies that evolve fast, see A-F. For more on Popper, please see Section B-W.

For improved reproducibility Jupyter notebooks, [42] propose ten rules and also provide links to example implementations. These can be very useful for users of Jupyter but are not generic for non-Jupyter-based computational projects. Some criteria (which are indeed very good in a more general context) do not directly relate to reproducibility, for example their Rule 1: "Tell a Story for an Audience". Generally, as reviewed in Section IIand Section A-E3 (below), Jupyter itself has many issues regarding reproducibility. To create Docker images, Nüst et al. propose "ten simple rules" in [43]. They recommend some issues that can indeed help increase the quality of Docker images and their production/usage, such as their rule 7 to "mount datasets [only] at run time" to separate the computational environment from the data. However, the long-term reproducibility of the images is not included as a criterion by these authors. For example, they recommend using base operating systems, with version identification limited to a single brief identifier such as `ubuntu:18.04`, which has a serious problem with longevity issues (Section II). Furthermore, in their proof-of-concept Dockerfile (listing 1), `rocker` is used with a tag (not a digest), which can be problematic due to the high risk of ambiguity (as discussed in Section A-A2).

Previous criteria are thus primarily targeted to immediate reproducibility and do not consider longevity. Therefore, they lack a strong/clear completeness criterion (they mainly only suggest, rather than require, the recording of versions, and their ultimate suggestion of storing the full binary OS in a binary VM or container is problematic (as mentioned in A-A and [18]).

B. Reproducible Electronic Documents, RED (1992)

RED[22] is the first attempt that we could find on doing reproducible research, see [1], [11]. It was developed within the Stanford Exploration Project (SEP) for Geophysics publications. Their introductions on the importance of reproducibility, resonate a lot with today's environment in computational sciences. In particular, the heavy investment one has to make in order to re-do another scientist's work, even in the same team. RED also influenced other early reproducible works, for example [44].

To orchestrate the various figures/results of a project, from 1990, they used "Cake" [45], a dialect of Make, for more on Make, see Appendix A-D. As described in [11], in the latter half of that decade, they moved to GNU Make, which was much more commonly used, better maintained, and came with a complete and up-to-date manual. The basic idea behind RED's solution was to organize the analysis as independent steps, including the generation of plots, and organizing the steps through a Makefile. This enabled all the results to be re-executed with a single command. Several basic low-level Makefiles were included in the high-level/central Makefile. The reader/user of a project had to manually edit the central Makefile and set the variable `RESDIR` (result directory), this is the directory where built files are kept. The reader could later select which figures/parts of the project to reproduce by manually adding its name in the central Makefile, and running Make.

At the time, Make was already practiced by individual researchers and projects as a job orchestration tool, but SEP's innovation was to standardize it as an internal policy, and define conventions for the Makefiles to be consistent across projects. This enabled new members to benefit from the already existing work of previous team members (who had graduated or moved to other jobs). However, RED only used the existing software of the host system, it had no means to control them. Therefore, with wider adoption, they confronted a "versioning problem" where the host's analysis software had different versions on different hosts, creating different results, or crashing [46]. Hence in 2006 SEP moved to a new Python-based framework called Madagascar, see Appendix B-D.

C. Apache Taverna (2003)

Apache Taverna[23] [47] is a workflow management system written in Java with a graphical user interface which is still being used and developed. A workflow is defined as a directed graph, where nodes are called "processors". Each Processor transforms a set of inputs into a set of outputs and they are defined in the Scufl language (an XML-based language, where each step is an atomic task). Other components of the workflow are "Data links" and "Coordination constraints". The main user interface is graphical, where users move processors in the given space and define links between their inputs and outputs (manually constructing a lineage like Figure 1). Taverna is only a workflow manager and is not integrated with a package manager, hence the versions of the used software can be different in different runs. [48] have studied the problem of workflow decays in Taverna.

[22] `http://sep.stanford.edu/doku.php?id=sep:research:reproducible` [23] `https://taverna.incubator.apache.org`

D. Madagascar (2003)

Madagascar[24] [49] is a set of extensions to the SCons job management tool (reviewed in A-D6). Madagascar is a continuation of the Reproducible Electronic Documents (RED) project that was discussed in Appendix B-B. Madagascar has been used in the production of hundreds of research papers or book chapters[25], 120 prior to [49].

Madagascar does include project management tools in the form of SCons extensions. However, it is not just a reproducible project management tool. The Regularly Sampled File (RSF) file format[26] is a custom plain-text file that points to the location of the actual data files on the file system and acts as the intermediary between Madagascar's analysis programs. Therefore, Madagascar is primarily a collection of analysis programs and tools to interact with RSF files and plotting facilities. For example in our test of Madagascar 3.0.1, it installed 855 Madagascar-specific analysis programs (`PREFIX/bin/sf*`). The analysis programs mostly target geophysical data analysis, including various project-specific tools: more than half of the total built tools are under the `build/user` directory which includes names of Madagascar users.

Besides the location or contents of the data, RSF also contains name/value pairs that can be used as options to Madagascar programs, which are built with inputs and outputs of this format. Since RSF contains program options also, the inputs and outputs of Madagascar's analysis programs are read from, and written to, standard input and standard output.

In terms of completeness, as long as the user only uses Madagascar's own analysis programs, it is fairly complete at a high level (not lower-level OS libraries). However, this comes at the expense of a large amount of bloatware (programs that one project may never need, but is forced to build), thus adding complexity. Also, the linking between the analysis programs (of a certain user at a certain time) and future versions of that program (that is updated in time) is not immediately obvious. Furthermore, the blending of the workflow component with the low-level analysis components fails the modularity criteria.

E. GenePattern (2004)

GenePattern[27] [50] (first released in 2004) is a client-server software containing many common analysis functions/modules, primarily focused for Gene studies. Although it is highly focused to a special research field, it is reviewed here because its concepts/methods are generic.

Its server-side software is installed with fixed software packages that are wrapped into GenePattern modules. The modules are used through a web interface, the modern implementation is GenePattern Notebook [51]. It is an extension of the Jupyter notebook (see Appendix A-E), which also has a special "GenePattern" cell that will connect to GenePattern servers for doing the analysis. However, the wrapper modules just call an existing tool on the running system. Given that each server may have its own set of installed software, the analysis may differ (or crash) when run on different GenePattern servers, hampering reproducibility.

The primary GenePattern server was active since 2008 and had 40,000 registered users with 2000 to 5000 jobs running every week [51]. However, it was shut down on November 15th 2019 due to the end of funding. All processing with this sever has stopped, and any archived data on it has been deleted. Since GenePattern is free software, there are alternative public servers to use, so hopefully, work on it will continue. However, funding is limited and those servers may face similar funding problems.

This is a very nice example of the fragility of solutions that depend on archiving and running the research codes with high-level research products (including data and binary/compiled codes that are expensive to keep in one place). The data and software may have backups in other places, but the high-level project-specific workflows that researchers spent most time on, have been lost due to the deletion (unless they were backed up privately by the authors!).

F. Kepler (2005)

Kepler[28] [52] is a Java-based Graphic User Interface workflow management tool. Users drag-and-drop analysis components, called "actors", into a visual, directional graph, which is the workflow (similar to Figure 1). Each actor is connected to others through Ptolemy II[29] [53]. In many aspects, the usage of Kepler and its issues for long-term reproducibility is like Apache Taverna (see Section B-C).

G. VisTrails (2005)

VisTrails[30] [54] was a graphical workflow managing system. According to its web page, VisTrails maintenance has stopped since May 2016, its last Git commit, as of this writing, was in November 2017. However, given that it was well maintained for over 10 years is an achievement.

VisTrails (or "visualization trails") was initially designed for managing visualizations, but later grew into a generic workflow system with meta-data and provenance features. Each analysis step, or module, is recorded in an XML schema, which defines the operations and their dependencies. The XML attributes of each module can be used in any XML query language to find certain steps (for example those that used a certain command). Since the main goal was visualization (as images), apparently its primary output is in the form of image spreadsheets. Its design is based on a change-based provenance model using a custom VisTrails provenance query language (vtPQL), for more see [55]. Since XML is a plain text format, as the user inspects the data and makes changes to the analysis, the changes are recorded as "trails" in the project's VisTrails repository that operates very much like common version control systems (see Appendix A-C). . However, even though XML is in plain text, it is very hard to read/edit without the VisTrails software (which is no

[24] http://ahay.org
[25] http://www.ahay.org/wiki/Reproducible_Documents
[26] http://www.ahay.org/wiki/Guide_to_RSF_file_format
[27] https://www.genepattern.org
[28] https://kepler-project.org
[29] https://ptolemy.berkeley.edu
[30] https://www.vistrails.org

longer maintained). VisTrails, therefore, provides a graphic user interface with a visual representation of the project's inter-dependent steps (similar to Figure 1). Besides the fact that it is no longer maintained, VisTrails did not control the software that is run, it only controlled the sequence of steps that they are run in.

H. Galaxy (2010)

Galaxy[31] is a web-based Genomics workbench [56]. The main user interface is the "Galaxy Pages", which does not require any programming: users graphically manipulate abstract "tools" which are wrappers over command-line programs. Therefore the actual running version of the program can be hard to control across different Galaxy servers. Besides the automatically generated metadata of a project (which include version control, or its history), users can also tag/annotate each analysis step, describing its intent/purpose. Besides some small differences, Galaxy seems very similar to GenePattern (Appendix B-E), so most of the same points there apply here too. For example the very large cost of maintaining such a system, being based on a graphic environment and blending hand-written code with automatically generated (large) files.

I. Image Processing On Line journal, IPOL (2010)

The IPOL journal[32] [57] (first published article in July 2010) publishes papers on image processing algorithms as well as the the full code of the proposed algorithm. An IPOL paper is a traditional research paper, but with a focus on implementation. The published narrative description of the algorithm must be detailed to a level that any specialist can implement it in their own programming language (extremely detailed). The author's own implementation of the algorithm is also published with the paper (in C, C++, or MATLAB), the code must be commented well enough and link each part of it with the relevant part of the paper. The authors must also submit several example of datasets that show the applicability of their proposed algorithm. The referee is expected to inspect the code and narrative, confirming that they match with each other, and with the stated conclusions of the published paper. After publication, each paper also has a "demo" button on its web page, allowing readers to try the algorithm on a web-interface and even provide their own input.

IPOL has grown steadily over the last 10 years, publishing 23 research articles in 2019. We encourage the reader to visit its web page and see some of its recent papers and their demos. The reason it can be so thorough and complete is its very narrow scope (low-level image processing algorithms), where the published algorithms are highly atomic, not needing significant dependencies (beyond input/output of well-known formats), allowing the referees and readers to go deeply into each implemented algorithm. In fact, high-level languages like Perl, Python, or Java are not acceptable in IPOL precisely because of the additional complexities, such as the dependencies that they require. However, many data-intensive projects commonly involve dozens of high-level dependencies, with large and complex data formats and analysis, so while it is modular (a single module, doing a very specific thing) this solution is not scalable.

Furthermore, by not publishing/archiving each paper's version control history or directly linking the analysis and produced paper, it fails criteria 6 and 7. Note that on the web page, it is possible to change parameters, but that will not affect the produced PDF. A paper written in Maneage (the proof-of-concept solution presented in this paper) could be scrutinized at a similar detailed level to IPOL, but for much more complex research scenarios, involving hundreds of dependencies and complex processing of the data.

J. WINGS (2010)

WINGS[33] [58] is an automatic workflow generation algorithm. It runs on a centralized web server, requiring many dependencies (such that it is recommended to download Docker images). It allows users to define various workflow components (for example datasets, analysis components, etc), with high-level goals. It then uses selection and rejection algorithms to find the best components using a pool of analysis components that can satisfy the requested high-level constraints.

K. Active Papers (2011)

Active Papers[34] attempts to package the code and data of a project into one file (in HDF5 format). It was initially written in Java because its compiled byte-code outputs in JVM are portable on any machine [59]. However, Java is not a commonly used platform today, hence it was later implemented in Python [36].

In the Python version, all processing steps and input data (or references to them) are stored in an HDF5 file. When the Python module contains a component written in other languages (mostly C or C++), it needs to be an external dependency to the Active Paper.

As mentioned in [36], the fact that it relies on HDF5 is a caveat of Active Papers, because many tools are necessary to merely open it. Downloading the pre-built "HDF View" binaries (a GUI browser of HDF5 files that is provided by the HDF group) is not possible anonymously/automatically (login is required). Installing it using the Debian or Arch Linux package managers also failed due to dependencies in our trials. Furthermore, as a high-level data format HDF5 evolves very fast, for example HDF5 1.12.0 (February 29th, 2020) is not usable with older libraries provided by the HDF5 team.

While data and code are indeed fundamentally similar concepts technically [60], they are used by humans differently. The hand-written code of a large project involving Terabytes of data can be 100 kilo bytes. When the two are bundled together, merely seeing one line of the code, requires downloading Terabytes volume that is not needed, this was also acknowledged in [36]. It may also happen that the data are proprietary (for

[31] https://galaxyproject.org
[32] https://www.ipol.im
[33] https://wings-workflows.org
[34] http://www.activepapers.org

example medical patient data). In such cases, the data must not be publicly released, but the methods that were applied to them can. Furthermore, since all reading and writing is done in the HDF5 file, it can easily bloat the file to very large sizes due to temporary files. These files can later be removed as part of the analysis, but this makes the code more complicated and hard to read/maintain. For example the Active Papers HDF5 file of [61, in zenodo.2549987] is 1.8 giga-bytes.

In many scenarios, peers just want to inspect the processing by reading the code and checking a very special part of it (one or two lines, just to see the option values to one step for example). They do not necessarily need to run it or obtain the output the datasets (which may be published elsewhere). Hence the extra volume for data and obscure HDF5 format that needs special tools for reading its plain-text internals is an issue.

L. Collage Authoring Environment (2011)

The Collage Authoring Environment [62] was the winner of Elsevier Executable Paper Grand Challenge [14]. It is based on the GridSpace2[35] distributed computing environment, which has a web-based graphic user interface. Through its web-based interface, viewers of a paper can actively experiment with the parameters of a published paper's displayed outputs (for example figures) through a web interface. In their Figure 3, they nicely vizualize how the "Executable Paper" of Collage operates through two servers and a computing backend.

Unfortunately in the paper no webpage has been provided follow up on the work and find its current status. A web search also only pointed us to its main paper ([62]). In the paper they do not discuss the major issue of software versioning and its verification to ensure that future updates to the backend do not affect the result; apparently it just assumes the software exist on the "Computing backend". Since we could not access or test it, from the descriptions in the paper, it seems to be very similar to the modern day Jupyter notebook concept (see A-E3), which had not yet been created in its current form in 2011. So we expect similar longevity issues with Collage.

M. SHARE (2011)

SHARE[36] [13] is a web portal that hosts virtual machines (VMs) for storing the environment of a research project. SHARE was recognized as the second position in the Elsevier Executable Paper Grand Challenge [14]. Simply put, SHARE was just a VM library that users could download or connect to, and run. The limitations of VMs for reproducibility were discussed in Appendix A-A1, and the SHARE system does not specify any requirements or standards on making the VM itself reproducible, or enforcing common internals for its supported projects. As of January 2021, the top SHARE web page still works. However, upon selecting any operation, a notice is printed that "SHARE is offline" since 2019 and the reason is not mentioned.

[35] http://dice.cyfronet.pl

[36] https://is.ieis.tue.nl/staff/pvgorp/share

N. Verifiable Computational Result, VCR (2011)

A "verifiable computational result"[37] is an output (table, figure, etc) that is associated with a "verifiable result identifier" (VRI), see [63]. It was awarded the third prize in the Elsevier Executable Paper Grand Challenge [14].

A VRI is a hash that is created using tags within the programming source that produced that output, also recording its version control or history. This enables the exact identification and citation of results. The VRIs are automatically generated web-URLs that link to public VCR repositories containing the data, inputs, and scripts, that may be re-executed. According to [63], the VRI generation routine has been implemented in MATLAB, R, and Python, although only the MATLAB version was available on the webpage in January 2021. VCR also has special LaTeX macros for loading the respective VRI into the generated PDF. In effect this is very similar to what have done at the end of the caption of Figure 1, where you can click on the given Zenodo link and be taken to the raw data that created the plot. However, instead of a long and hard to read hash, we simply point to the plotted file's source as a Zenodo DOI (which has long term funding for logevity).

Unfortunately, most parts of the web page are not complete as of January 2021. The VCR web page contains an example PDF[38] that is generated with this system, but the linked VCR repository[39] did not exist (again, as of January 2021). Finally, the date of the files in the MATLAB extension tarball is set to May 2011, hinting that probably VCR has been abandoned soon after the publication of [63].

O. SOLE (2012)

SOLE (Science Object Linking and Embedding) defines "science objects" (SOs) that can be manually linked with phrases of the published paper [64], [65]. An SO is any code/-content that is wrapped in begin/end tags with an associated type and name. For example, special commented lines in a Python, R, or C program. The SOLE command-line program parses the tagged file, generating metadata elements unique to the SO (including its URI). SOLE also supports workflows as Galaxy tools [56].

For reproducibility, [64] suggest building a SOLE-based project in a virtual machine, using any custom package manager that is hosted on a private server to obtain a usable URI. However, as described in Appendices A-A and A-B, unless virtual machines are built with robust package managers, this is not a sustainable solution (the virtual machine itself is not reproducible). Also, hosting a large virtual machine server with fixed IP on a hosting service like Amazon (as suggested there) for every project in perpetuity will be very expensive.

The manual/artificial definition of tags to connect parts of the paper with the analysis scripts is also a caveat due to human error and incompleteness (the authors may not consider tags as important things, but they may be useful later). In Maneage, instead of using artificial/commented tags, the analysis inputs and outputs are automatically linked into the

[37] http://vcr.stanford.edu

[38] http://vcr.stanford.edu/paper.pdf

[39] http://vcr-stat.stanford.edu

paper's text through LaTeX macros that are the backbone of the whole system (are not artifical/extra features).

P. Sumatra (2012)

Sumatra[40] [66] attempts to capture the environment information of a running project. It is written in Python and is a command-line wrapper over the analysis script. By controlling a project at running-time, Sumatra is able to capture the environment it was run in. The captured environment can be viewed in plain text or a web interface. Sumatra also provides LaTeX/Sphinx features, which will link the paper with the project's Sumatra database. This enables researchers to use a fixed version of a project's figures in the paper, even at later times (while the project is being developed).

The actual code that Sumatra wraps around, must itself be under version control, and it does not run if there are non-committed changes (although it is not clear what happens if a commit is amended). Since information on the environment has been captured, Sumatra is able to identify if it has changed since a previous run of the project. Therefore Sumatra makes no attempt at storing the environment of the analysis as in Sciunit (see Appendix B-R), but its information. Sumatra thus needs to know the language of the running program and is not generic. It just captures the environment, it does not store *how* that environment was built.

Q. Research Object (2013)

The Research object[41] is collection of meta-data ontologies, to describe aggregation of resources, or workflows, see [67] and [68]. It thus provides resources to link various workflow/-analysis components (see Appendix A) into a final workflow.

[67] describes how a workflow in Apache Taverna (Appendix B-C) can be translated into research objects. The important thing is that the research object concept is not specific to any special workflow, it is just a metadata bundle/standard which is only as robust in reproducing the result as the running workflow. Therefore if implemented over a complete workflow like Maneage, it can be very useful in analysing/optimizing the workflow, finding common components between many Maneage'd workflows, or translating to other complete workflows.

R. Sciunit (2015)

Sciunit[42] [69] defines "sciunit"s that keep the executed commands for an analysis and all the necessary programs and libraries that are used in those commands. It automatically parses all the executable files in the script and copies them, and their dependency libraries (down to the C library), into the sciunit. Because the sciunit contains all the programs and necessary libraries, it is possible to run it readily on other systems that have a similar CPU architecture. Sciunit was originally written in Python 2 (which reached its end-of-life on January 1st, 2020). Therefore Sciunit2 is a new implementation in Python 3.

The main issue with Sciunit's approach is that the copied binaries are just black boxes: it is not possible to see how the used binaries from the initial system were built. This is a major problem for scientific projects: in principle (not knowing how the programs were built) and in practice (archiving a large volume sciunit for every step of the analysis requires a lot of storage space and archival cost).

S. Umbrella (2015)

Umbrella [70] is a high-level wrapper script for isolating the environment of the analysis. The user specifies the necessary operating system, and necessary packages for the analysis steps in various JSON files. Umbrella will then study the host operating system and the various necessary inputs (including data and software) through a process similar to Sciunits mentioned above to find the best environment isolator (maybe using Linux containerization, containers, or VMs). We could not find a URL to the source software of Umbrella (no source code repository is mentioned in the papers we reviewed above), but from the descriptions in [17], it is written in Python 2.6 (which is now deprecated).

T. ReproZip (2016)

ReproZip[43] [71] is a Python package that is designed to automatically track all the necessary data files, libraries, and environment variables into a single bundle. The tracking is done at the kernel system-call level, so any file that is accessed during the running of the project is identified. The tracked files can be packaged into a `.rpz` bundle that can then be unpacked into another system.

ReproZip is therefore very good to take a "snapshot" of the running environment into a single file. However, the bundle can become very large when many/large datasets are involved, or if the software environment is complex (many dependencies). Since it copies the binary software libraries, it can only be run on systems with a similar CPU architecture to the original. Furthermore, ReproZip just copies the binary/compiled files used in a project, it has no way to know how the software was built. As mentioned in this paper, and also [18] the question of "how" the environment was built is critical for understanding the results, and simply having the binaries cannot necessarily be useful.

For the data, it is similarly not possible to extract which data server they came from. Hence two projects that each use a 1-terabyte dataset will need a full copy of that same 1-terabyte file in their bundle, making long-term preservation extremely expensive.

U. Binder (2017)

Binder[44] is used to containerize already existing Jupyter based processing steps. Users simply add a set of Binder-recognized configuration files to their repository and Binder will build a Docker image and install all the dependencies inside of it with Conda (the list of necessary packages comes

[40] http://neuralensemble.org/sumatra
[41] http://www.researchobject.org
[42] https://sciunit.run
[43] https://www.reprozip.org
[44] https://mybinder.org

from Conda). One good feature of Binder is that the imported Docker image must be tagged, although as mentioned in Appendix A-A2, tags do not ensure reproducibility. However, it does not make sure that the Dockerfile used by the imported Docker image follows a similar convention also. So users can simply use generic operating system names. Binder is used by [31].

V. Gigantum (2017)

Gigantum[45] is a client/server system, in which the client is a web-based (graphical) interface that is installed as "Gigantum Desktop" within a Docker image. Gigantum uses Docker containers for an independent environment, Conda (or Pip) to install packages, Jupyter notebooks to edit and run code, and Git to store its history. The reproducibility issues with these tools has been thoroughly discussed in A.

Simply put, it is a high-level wrapper for combining these components. Internally, a Gigantum project is organized as files in a directory that can be opened without their own client. The file structure (which is under version control) includes codes, input data, and output data. As acknowledged on their own web page, this greatly reduces the speed of Git operations, transmitting, or archiving the project. Therefore there are size limits on the dataset/code sizes. However, there is one directory that can be used to store files that must not be tracked.

W. Popper (2017)

Popper[46] is a software implementation of the Popper Convention [33]. The Popper team's own solution is through a command-line program called `popper`. The `popper` program itself is written in Python. However, job management was initially based on the HashiCorp configuration language (HCL) because HCL was used by "GitHub Actions" to manage workflows at that time. However, from October 2019 GitHub changed to a custom YAML-based language, so Popper also deprecated HCL. This is an important issue when low-level choices are based on service providers (see Appendix A-F).

To start a project, the `popper` command-line program builds a template, or "scaffold", which is a minimal set of files that can be run. By default, Popper runs in a Docker image (so root permissions are necessary and reproducible issues with Docker images have been discussed above), but Singularity is also supported. See Appendix A-A for more on containers, and Appendix A-F for using high-level languages in the workflow.

Popper does not comply with the completeness, minimal complexity, and including the narrative criteria. Moreover, the scaffold that is provided by Popper is an output of the program that is not directly under version control. Hence, tracking future low-level changes in Popper and how they relate to the high-level projects that depend on it through the scaffold will be very hard. In Maneage, users start their projects by branching-off of the core `maneage` git branch. Hence any future change in the low level features will directly propagated to all derived projects (and be clear as Git conflicts if the user has customized them).

X. Whole Tale (2017)

Whole Tale[47] is a web-based platform for managing a project and organizing data provenance, see [72]. It uses online editors like Jupyter or RStudio (see Appendix A-E) that are encapsulated in a Docker container (see Appendix A-A).

The web-based nature of Whole Tale's approach and its dependency on many tools (which have many dependencies themselves) is a major limitation for future reproducibility. For example, when following their own tutorial on "Creating a new tale", the provided Jupyter notebook could not be executed because of a dependency problem. This was reported to the authors as issue 113[48] and fixed. But as all the second-order dependencies evolve, it is not hard to envisage such dependency incompatibilities being the primary issue for older projects on Whole Tale. Furthermore, the fact that a Tale is stored as a binary Docker container causes two important problems: 1) it requires a very large storage capacity for every project that is hosted there, making it very expensive to scale if demand expands. 2) It is not possible to see how the environment was built accurately (when the Dockerfile uses operating system package managers like `apt`). This issue with Whole Tale (and generally all other solutions that only rely on preserving a container/VM) was also mentioned in [18], for more on this, please see Appendix A-B.

Y. Occam (2018)

Occam[49] [18] is a web-based application to preserve software and its execution. To achieve long-term reproducibility, Occam includes its own package manager (instructions to build software and their dependencies) to be in full control of the software build instructions, similar to Maneage. Besides Nix or Guix (which are primarily a package manager that can also do job management), Occam has been the only solution in our survey here that attempts to be complete in this aspect.

However it is incomplete from the perspective of requirements: it works within a Docker image (that requires root permissions) and currently only runs on Debian-based, Red Hat based, and Arch-based GNU/Linux operating systems that respectively use the `apt`, `pacman` or `yum` package managers. It is also itself written in Python (version 3.4 or above).

Furthermore, it does not account for the minimal complexity criteria because the instructions to build the software and their versions are not immediately viewable or modifiable by the user. Occam contains its own JSON database that should be parsed with its own custom program. The analysis phase of Occam is also through a drag-and-drop interface (similar to Taverna, Appendix B-C) that is a web-based graphic user interface. All the connections between various phases of the analysis need to be pre-defined in a JSON file and manually linked in the GUI. Hence for complex data analysis operations that involve thousands of steps, it is not scalable.

[45] https://gigantum.com
[46] https://falsifiable.us
[47] https://wholetale.org
[48] https://github.com/whole-tale/wt-design-docs/issues/113
[49] https://occam.cs.pitt.edu

APPENDIX C
SOFTWARE ACKNOWLEDGEMENT

This research was done with the following free software programs and libraries: Bzip2 1.0.6, CMake 3.18.1, cURL 7.71.1, Dash 0.5.10.2, Discoteq flock 0.2.3, Expat 2.2.9, File 5.39, Fontconfig 2.13.1, FreeType 2.10.2, Git 2.28.0, GNU Autoconf 2.69.200-babc, GNU Automake 1.16.2, GNU AWK 5.1.0, GNU Bash 5.0.18, GNU Binutils 2.35, GNU Compiler Collection (GCC) 10.2.0, GNU Coreutils 8.32, GNU Diffutils 3.7, GNU Findutils 4.7.0, GNU gettext 0.21, GNU gperf 3.1, GNU Grep 3.4, GNU Gzip 1.10, GNU Integer Set Library 0.18, GNU libiconv 1.16, GNU Libtool 2.4.6, GNU libunistring 0.9.10, GNU M4 1.4.18-patched, GNU Make 4.3, GNU Multiple Precision Arithmetic Library 6.2.0, GNU Multiple Precision Complex library, GNU Multiple Precision Floating-Point Reliably 4.0.2, GNU Nano 5.2, GNU NCURSES 6.2, GNU Readline 8.0, GNU Sed 4.8, GNU Tar 1.32, GNU Texinfo 6.7, GNU Wget 1.20.3, GNU Which 2.21, GPL Ghostscript 9.52, Less 563, Libbsd 0.10.0, Libffi 3.2.1, libICE 1.0.10, Libidn 1.36, Libjpeg v9b, Libpaper 1.1.28, Libpng 1.6.37, libpthread-stubs (Xorg) 0.4, libSM 1.2.3, Libtiff 4.0.10, libXau (Xorg) 1.0.9, libxcb (Xorg) 1.14, libXdmcp (Xorg) 1.1.3, libXext 1.3.4, Libxml2 2.9.9, libXt 1.2.0, Lzip 1.22-rc2, Metastore (forked) 1.1.2-23-fa9170b, Minizip 1.2.11, OpenSSL 1.1.1a, PatchELF 0.10, Perl 5.32.0, pkg-config 0.29.2, Python 3.8.5, Unzip 6.0, util-Linux 2.35, util-macros (Xorg) 1.19.2, X11 library 1.6.9, XCB-proto (Xorg) 1.14, XLSX I/O 0.2.21, xorgproto 2020.1, xtrans (Xorg) 1.4.0, XZ Utils 5.2.5, Zip 3.0 and Zlib 1.2.11. The LaTeX source of the paper was compiled to make the PDF using the following packages: cite 5.5, courier 35058 (revision), etoolbox 2.5k, IEEEtran 1.8b, inconsolata 1.121, listings 1.8d, multibib 1.4, pgfplots 1.17, ps2eps 1.68, times 35058 (revision), ulem 53365 (revision), xcolor 2.12 and xkeyval 2.8. We are very grateful to all their creators for freely providing this necessary infrastructure. This research (and many other projects) would not be possible without them.

BIBLIOGRAPHY

[13] P. Van Gorp and S. Mazanek, "Share: a web portal for creating and sharing executable research," *Procedia Computer Science*, vol. 4, p. 589, DOI:10.1016/j.procs.2011.04.062, 2011.

[14] A. Gabriel and R. Capone, "Executable paper grand challenge workshop," *Procedia Computer Science*, vol. 4, p. 577, DOI:10.1016/j.procs.2011.04.060, 2011.

[15] M. Dolfi, J. Gukelberger, A. Hehn, J. Imriška, K. Pakrouski, T. F. Rønnow, M. Troyer, I. Zintchenko, F. Chirigati, J. Freire, and D. Shasha, "A model project for reproducible papers: critical temperature for the Ising model on a square lattice," *arXiv*, p. arXiv:1401.2000, arXiv:1401.2000, Jan. 2014.

[16] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLoS ONE*, vol. 12, p. e0177459, DOI:10.1371/journal.pone.0177459, 2017.

[17] H. Meng and D. Thain, "Facilitating the reproducibility of scientific workflows with execution environment specifications," *Procedia Computer Science*, vol. 108, p. 705, DOI:10.1016/j.procs.2017.05.116, 2017.

[18] L. Oliveira, D. Wilkinson, D. Mossé, and B. R. Childers, "Supporting long-term reproducible software execution," *Proceedings of the First International Workshop on Practical Reproducible Evaluation of Computer Systems (P-RECS'18)*, vol. 1, p. 6, DOI:10.1145/3214239.3214245, 2018.

[19] C. Clarkson, M. Smith, B. Marwick, R. Fullagar, L. A. Wallis, P. Faulkner, T. Manne, E. Hayes, R. G. Roberts, Z. Jacobs, X. Carah, K. M. Lowe, J. Matthews, and S. A. Florin, "The archaeology, chronology and stratigraphy of Madjedbebe (Malakunanja II): A site in northern Australia with early occupation," *Journal of Human Evolution*, vol. 83, p. 46, DOI:10.1016/j.jhevol.2015.03.014, 2015.

[20] J. Lofstead, J. Baker, and A. Younge, "Data pallets: Containerizing storage for reproducibility and traceability," *High Performance Computing. ISC High Performance 2019*, vol. 11887, p. 1, DOI:10.1007/978-3-030-34356-9_4, 2019.

[21] D. Aïssi, "Review for Towards Long-term and Archivable Reproducibility," *Authorea*, vol. n/a, p. n/a, DOI:10.22541/au.159724632.29528907, 2020.

[22] E. Dolstra, M. de Jonge, and E. Visser, "Nix: A safe and policy-free system for software deployment," *Large Installation System Administration Conference*, vol. 18, pp. 79, PDF in LISA04 webpage, 2004.

[23] L. Courtés and R. Wurmus, "Reproducible and user-controlled software environments in hpc with guix," *Euro-Par*, vol. 9523, p. arXiv:1506.02822, arXiv:1506.02822, DOI:10.1007/978-3-319-27308-2_47, Jun 2015.

[24] S. Uhse, F. G. Pflug, A. von Haeseler, and A. Djamei, "Insertion pool sequencing for insertional mutant analysis in complex host-microbe interactions," *Current Protocols in Plant Biology*, vol. 4, p. e20097, DOI:10.1002/cppb.20097, July 2019.

[25] B. Grüning, J. Chilton, J. Köster, R. Dale, N. Soranzo, M. van den Beek, J. Goecks, R. Backofen, A. Nekrutenko, and J. Taylor, "Practical computational reproducibility in the life sciences," *Cell Systems*, vol. 6, p. 631. bioRxiv:200683, DOI:10.1016/j.cels.2018.03.014, 2018.

[26] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The spack package manager: bringing order to hpc software chaos," *IEEE SC15*, vol. 1, p. 1, DOI:10.1145/2807591.2807623 , 2015.

[27] R. Di Cosmo, M. Gruenpeter, and S. Zacchiroli, "Identifiers for digital objects: The case of software source code preservation," *Proceedings of iPRES 2018*, p. 204.4, DOI:10.17605/osf.io/kde56, 2018.

[28] S. I. Feldman, "Make – a program for maintaining computer programs," *Journal of Software: Practice and Experience*, vol. 9, p. 255, DOI:10.1002/spe.4380090402, 1979.

[29] J. Köster and S. Rahmann, "Snakemake—a scalable bioinformatics workflow engine," *Bioinformatics*, vol. 28, p. 2520, DOI:10.1093/bioinformatics/bts480, 2012.

[30] A. Cribbs, S. Luna-Valero, C. George, I. Sudbery, A. Berlanga-Taylor, S. Sansom, T. Smith, N. Ilott, J. Johnson, J. Scaber, K. Brown, D. Sims, and A. Heger, "Cgat-core: a python framework for building scalable, reproducible computational biology workflows [version 2; peer review: 1 approved, 1 approved with reservations]," *F1000Research*, vol. 8, p. 377, DOI:10.12688/f1000research.18674.2, 2019.

[31] M. G. Jones, L. Verdes-Montenegro, A. Damas-Segovia, S. Borthakur, M. Yun, A. del Olmo, J. Perea, J. Román, S. Luna, D. Lopez Gutierrez, B. Williams, F. P. A. Vogt, J. Garrido, S. Sanchez, J. Cannon, and P. Ramírez-Moreta, "Evolution of compact groups from intermediate to final stages. A case study of the H I content of HCG 16," *Astronomy & Astrophysics*, vol. 632, p. A78, arXiv:1910.03420 , DOI:10.1051/0004-6361/201936349, Dec 2019.

[32] P. D. Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature Biotechnology*, vol. 35, p. 316, DOI:10.1038/nbt.3820, 2017.

[33] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "The popper convention: Making reproducible systems evaluation practical," *IEEE IPDPSW*, p. 1561, DOI:10.1109/IPDPSW.2017.157, 2017.

[34] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter Notebooks – a publishing format for reproducible computational workflows," *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, p. 87, DOI:10.3233/978-1-61499-649-1-87, 2016.

[35] D. Knuth, "Literate programming," *The Computer Journal*, vol. 27, p. 97, DOI:10.1093/comjnl/27.2.97, 1984.

[36] K. Hinsen, "Activepapers: a platform for publishing and archiving computer-aided research," *F1000Research*, vol. 3, p. 289, DOI:10.12688/f1000research.5773.3, 2015.

[37] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, p. 65, arXiv:1411.1607, DOI:10.1137/141000671, 2017.

[38] T. Jenness, "Modern Python at the Large Synoptic Survey Telescope," *ADASS 27*, p. arXiv:1712.00461, arXiv:1712.00461, Dec 2017.

[39] M. Wilkinson, M. Dumontier, I. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J. Boiten, L. da Silva Santos, P. Bourne, J. Bouwman, A. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C. Evelo, R. Finkers, A. Gonzalez-Beltran, A. Gray, P. Groth, C. Goble, J. S. Grethe, J. Heringa, P. 't Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S. Lusher, M. Martone, A. Mons, A. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, and B. Mons, "The FAIR Guiding Principles for scientific data management and stewardship," *Scientific Data*, vol. 3, p. 160018, DOI:10.1038/sdata.2016.18, Mar. 2016.

[40] M. Konkol, D. Nüst, and L. Goulier, "Publishing computational research – A review of infrastructures for reproducible and transparent scholarly communication," *arXiv*, p. 2001.00484, arXiv:2001.00484, Jan. 2020.

[41] G. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig, "Ten simple rules for reproducible computational research," *PLoS Computational Biology*, vol. 9, p. e1003285, DOI:10.1371/journal.pcbi.1003285, Oct. 2013.

[42] A. Rule, A. Birmingham, C. Zuniga, I. Altintas, S.-C. Huang, R. Knight, N. Moshiri, M. H. Nguyen, S. Brin Rosenthal, F. Pérez, and P. W. Rose, "Ten simple rules for reproducible research in jupyter notebooks," *PLOS Computational Biology*, vol. 15, p. e1007007, arXiv:1810.08055, DOI:10.1371/journal.pcbi.1007007, Jul. 2019.

[43] D. Nüst, V. Sochat, B. Marwick, S. J. Eglen, T. Head, T. Hirst, and B. D. Evans, "Ten simple rules for writing Dockerfiles for reproducible data science," *PLOS Computational Biology*, vol. 16, p. e1008316, DOI:10.1371/journal.pcbi.1008316, 2020.

[44] J. B. Buckheit and D. L. Donoho, "WaveLab and Reproducible Research," *Wavelets and Statistics*, vol. 1, p. 55, DOI:10.1007/978-1-4612-2544-7_5, 1995.

[45] Z. Somogyi, "Cake: a fifth generation version of make," *University of Melbourne*, p. Corpus ID: 107669553, 1987.

[46] S. Fomel and J. F. Claerbout, "Reproducible research," *Computing in Science Engineering*, vol. 11, p. 5, DOI:10.1109/MCSE.2009.14, 2009.

[47] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, p. 3045, DOI:10.1093/bioinformatics/bth361, 2004.

[48] J. Zhao, J. M. Gomez-Perez, K. Belhajjame, G. Klyne, E. Garcia-Cuesta, A. Garrido, K. Hettne, M. Roos, D. De Roure, and C. Goble, "Why workflows break — understanding and combating decay in taverna workflows," *IEEE 8th International Conference on E-Science*, p. 1, DOI:10.1109/eScience.2012.6404482, 2012.

[49] S. Fomel, P. Sava, I. Vlad, Y. Liu, and V. Bashkardin, "Madagascar: open-source software project for multidimensional data analysis and reproducible computational experiments," *Journal of open research software*, vol. 1, p. e8, DOI:10.5334/jors.ag , 2013.

[50] M. Reich, T. Liefeld, J. Gould, J. Lerner, P. Tamayo, and J. P. Mesirov, "Genepattern 2.0," *Nature Genetics*, vol. 38, p. 500, DOI:10.1038/ng0506-500, 2006.

[51] M. Reich, T. Tabor, T. Liefeld, H. Thorvaldsdóttir, B. Hill, P. Tamayo, and J. P. Mesirov, "The genepattern notebook environment," *Cell Systems*, vol. 5, p. 149, DOI:10.1016/j.cels.2017.07.003, 2017.

[52] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the Kepler system," *Concurrency Computation: Practice and Experiment*, vol. 18, p. 1039, DOI:10.1002/cpe.994, 2006.

[53] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, and S. Neuendorffer, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, p. 127, DOI:10.1109/JPROC.2002.805829, 2003.

[54] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "VisTrails: Enabling Interactive Multiple-View Visualizations," *VIS 05. IEEE Visualization*, p. 135, DOI:10.1109/VISUAL.2005.1532788, 2005.

[55] C. Scheidegger, D. Koop, E. Santos, H. Vo, S. Callahan, J. Freire, and C. Silva, "Tackling the provenance challenge one layer at a time," *Concurrency Computation: Practice and Experiment*, vol. 20, p. 473, DOI:10.1002/cpe.1237, 2008.

[56] J. Goecks, A. Nekrutenko, and J. Taylor, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome Biology*, vol. 11, p. R86, DOI:10.1186/gb-2010-11-8-r86, 2010.

[57] N. Limare and J.-M. Morel, "The ipol initiative: Publishing and testing algorithms on line for reproducible research in image processing," *Procedia Computer Science*, vol. 4, p. 716, DOI:10.1016/j.procs.2011.04.075, 2011.

[58] Y. Gil, P. A. González-Calero, J. Kim, J. Moody, and V. Ratnakar, "A semantic framework for automatic generation of computational workflows using distributed data and component catalogues," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 23, p. 389, DOI:10.1080/0952813X.2010.490962, 2010.

[59] K. Hinsen, "A data and code model for reproducible research and executable papers," *Procedia Computer Science*, vol. 4, p. 579, DOI:10.1016/j.procs.2011.04.061, 2011.

[60] K. Hinsen, "Scientific notations for the digital era," *The Self Journal of Science*, p. 1: arXiv:1605.02960, 2016.

[61] G. R. Kneller and K. Hinsen, "Memory effects in a random walk description of protein structure ensembles," *The Journal of Chemical Physics*, vol. 150, p. 064911, DOI:10.1063/1.5054887, 2019.

[62] P. Nowakowski, E. Ciepiela, D. Harężlak, J. Kocot, M. Kasztelnik, T. Bartyński, J. Meizner, G. Dyk, and M. Malawski, "The collage authoring environment," *Procedia Computer Science*, vol. 4, p. 608, DOI:j.procs.2011.04.064, 2011.

[63] M. Gavish and D. L. Donoho, "A universal identifier for computational results," *Procedia Computer Science*, vol. 4, p. 637, DOI:10.1016/j.procs.2011.04.067, 2011.

[64] Q. Pham, T. Malik, I. Foster, R. Di Lauro, and R. Montella, "Sole: Linking research papers with science objects," *Provenance and Annotation of Data and Processes (IPAW)*, p. 203, DOI:10.1007/978-3-642-34222-6_16, 2012.

[65] T. Malik, Q. Pham, and I. Foster, "Sole: Towards descriptive and interactive publications," *Implementing Reproducible Research*, vol. Chapter 2, p. 1. URL: https://osf.io/ns2m3, 2013.

[66] A. Davison, "Automated capture of experiment context for easier reproducibility in computational research," *Computing in Science & Engineering*, vol. 14, p. 48, DOI:10.1109/MCSE.2012.41, 2012.

[67] S. Bechhofer, I. Buchan, D. De Roure, P. Missier, J. Ainsworth, J. Bhagat, P. Couch, D. Cruickshank, M. Delderfield, I. Dunlop, M. Gamble, D. Michaelides, S. Owen, D. Newman, S. Sufi, and C. Goble, "Why linked data is not enough for scientists," *Future Generation Computer Systems*, vol. 29, p. 599, DOI:10.1016/j.future.2011.08.004, 2013.

[68] K. Belhajjame, Z. Zhao, D. Garijo, M. Gamble, K. Hettne, R. Palma, E. Mina, O. Corcho, J. Gómez-Pérez, S. Bechhofer, G. Klyne, and C. Goble, "Using a suite of ontologies for preserving workflow-centric research objects," *Journal of Web Semantics*, vol. 32, p. 16, DOI:10.1016/j.websem.2015.01.003, 2015.

[69] H. Meng, R. Kommineni, Q. Pham, R. Gardner, T. Malik, and D. Thain, "An invariant framework for conducting reproducible computational science," *Journal of Computational Science*, vol. 9, p. 137, DOI:10.1016/j.jocs.2015.04.012, 2015.

[70] H. Meng and D. Thain, "Umbrella: A portable environment creator for reproducible computing on clusters, clouds, and grids," *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing (VTDC 15)*, vol. 1, p. 23, DOI:10.1145/2755979.2755982, 2015.

[71] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "Reprozip: Computational reproducibility with ease," *Proceedings of the 2016 International Conference on Management of Data (SIGMOD 16)*, vol. 2, p. 2085, DOI:10.1145/2882903.2899401, 2016.

[72] A. Brinckman, K. Chard, N. Gaffney, M. Hategan, M. B. Jones, K. Kowalik, S. Kulasekaran, B. Ludäscher, B. D. Mecum, J. Nabrzyski, V. Stodden, I. J. Taylor, M. J. Turk, and K. Turner, "Computing environments for reproducibility: Capturing the "whole tale"," *Future Generation Computer Systems*, vol. 94, p. 854, DOI:10.1016/j.future.2017.12.029, 2017.