Sharing and Preserving Computational Analyses for Posterity with encapsulator

Article *in* Computing in Science and Engineering · July 2018

DOI: 10.1109/MCSE.2018.042781334

9 authors, including:

Thomas Pasquier
University of Bristol
31 PUBLICATIONS **349** CITATIONS
SEE PROFILE

Xueyuan Han
Harvard University
10 PUBLICATIONS **28** CITATIONS
SEE PROFILE

Sharing and Preserving Computational Analyses for Posterity with `encapsulator`

Thomas Pasquier*†, Matthew K. Lau‡, Xueyuan Han†, Elizabeth Fong§,
Barbara S. Lerner§, Emery Boose‡, Mercè Crosas¶, Aaron Ellison‡, Margo Seltzer†
*Department of Computer Science and Technology
University of Cambridge, Cambridge, UK
†School of Engineering and Applied Sciences
Harvard University, Cambridge, USA
‡Harvard Forest,
Harvard University, Petersham, USA
§Department of Computer Science,
Mount Holyoke College, South Hadley, USA
¶ Institute for Quantitative Social Science,
Harvard University, Cambridge, USA

***Abstract*—Open data and open-source software may be part of the solution to sciences reproducibility crisis, but they are insufficient to guarantee reproducibility. Requiring minimal end-user expertise, `encapsulator` creates a *"time capsule"* with reproducible code in a self-contained computational environment. `encapsulator` provides end-users with a fully-featured desktop environment for reproducible research.**

1. Introduction

Reproducibility has become a recurring topic of discussion in many scientific disciplines [1]. Although it may be expected that some studies will be difficult to reproduce, recent conversations highlight important aspects of the scientific endeavor that could be improved to facilitate reproducibility. Open data and open-source software are two important parts of a con-certed effort to achieve reproducibility [2]. However, multiple publications point out the short-comings of these approaches [3, 4], such as the identification of dependencies, poor documentation of the installation processes, code rot, failure to capture dynamic inputs, and technical barriers.

In prior work [5], we pointed out that open data and open-source software alone are insufficient to ensure reproducibility, as they do not capture information about the computational execution, i.e., the "process" and context that produced the results using the data and code. In keeping with the "open" culture, we defined open-process as the practice of both sharing the source and the input data and providing a description of the entire computational environment, including the software, libraries, and operating system used for an analysis. We suggested the use of data provenance [6], formalized metadata representing the execution of a computational task and its context (e.g., dependencies, specific data versions, and random or pseudo-random values), which can be captured during computation.

We view data provenance as key to addressing these issues, yet still insufficient. We need tools that leverage provenance to put capabilities, not complex metadata, into scientists hands. We build on recent developments that address this need, such as executable papers [7] and experiment packaging systems, e.g., ReproZip [8]. We propose a solution for scientists running small-to-medium-scale computational experiments or analyses on commodity machines. Although tools exist to cover analyses done using spreadsheet programs (further discussed in the Challenges section), we intentionally

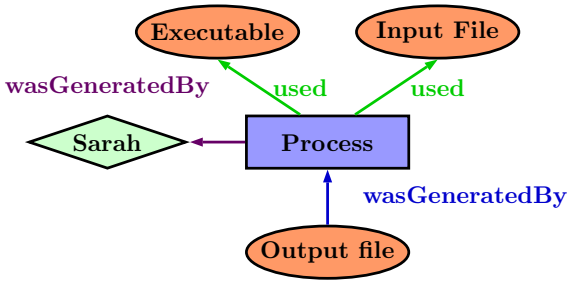

Figure 1. A simple W3C PROV-DM compliant provenance graph.

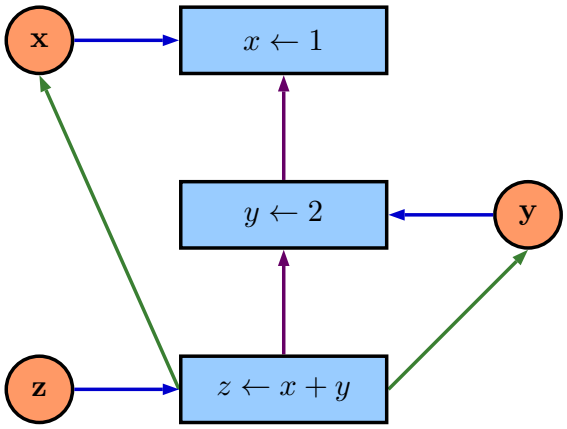

Figure 2. A simple provenance graph for an R script.

do not cover that space, as it has inherent barriers to transparency and identification of the source of errors [9, 10]. Similarly, we do not attempt to address the reproducibility of large-scale computational analysis.

We present a *time capsule* for small-to-medium-scale computational analysis. This *time capsule* is a self-contained environment that allows other scientists to explore the results of a published paper, reproduce them, or build upon them with minimal effort. We automatically curate the scientist's code to extract only those elements pertinent to a particular figure, table, or dataset.

2. Data Provenance

Data provenance [6] has the potential to address some of the challenges related to reproducibility. Indeed, to assess the validity or quality of information, it is necessary to understand the context of its creation. Unfortunately, digital artifacts frequently omit or hide much of the context in which they were created. As an example, many of us have been guilty of sharing code we developed on our machines that our colleagues could not run, because we often work in the same environment for months or years, forgetting about software and libraries we have installed over time.

Meanwhile, small differences in a computational pipeline can lead to vastly different results. For example, different analyses of the same dataset of carbon flux in an Amazonian forest ecosystem differed in their estimates by up to 140% [11], amounting to differences of up to 7 tons of carbon in an area of the size of a football field. This example highlights the significant impact of small differences in code, especially when analyses or models contain user-defined or interactive (e.g., multiplicative) terms. Seemingly small changes to inputs or in the computational pipeline can lead to large differences in results, impeding their reproducibility and verification.

Data provenance is a formal representation of the context and execution of a computation. This information is represented as a directed acyclic graph (DAG), a structure amenable to computational analysis. We use the World Wide Web Consortium (W3C) standard for data provenance: PROV-DM. Fig. 1 shows a simple provenance graph. Vertices represent enti-ties (representing data), activities (representing actions or transformations), and agents (repre-senting users or organizations). In Fig. 1, a process, controlled by Scientist Sarah, uses an executable function (i.e., a program) and an input file (i.e., data) to generate an output.

Provenance can be captured at various levels of a system, such as in libraries explicitly called by a program, in a language interpreter, in system libraries, or in the operating system. The specific capture approach produces subtly different types of provenance: observed provenance is deduced by a system that monitors execution, whereas disclosed provenance is created explicitly by software

that understands the semantics of the computations performed [12]. `encapsulator` uses observed provenance capture, which reveals the inner workings of an analysis script by collecting fine-grained provenance.

When provenance is captured for a scripting or programming language, the provenance DAG represents relationships among inputs, outputs, transient data objects, and statements. For example, Fig. 2 illustrates a provenance graph of the following R script:

```
x <- 1
y <- 2
z <- x + y
```

In the figure, the blue rectangles correspond to statements in the language; the orange circles correspond to data items (i.e., inputs, outputs, or transient objects); the purple arrows show the control flow, representing the precise sequence of steps taken while executing the program; and the blue and green arrows show data dependencies (i.e., the data used by an operation, and the data generated by an operation respectively).

The provenance DAG illustrates data dependencies (i.e., what input generated a given output), software dependencies (i.e., on what libraries a script depends), and information about the structure of a program. We next discuss how we use provenance DAGs to generate a *time capsule*.

3. Creating a Time-Capsule

Provenance alone provides a "picture" of a computational context, yet we want to provide an active artifact that can reproduce a computational context: the time capsule. Fig. 3 illustrates the two phases involved in creating a time capsule from the provenance collected during execution: 1) curate the script to identify the precise lines of code and input data needed to produce a result; 2) build the time capsule containing the previously generated artifact and the environment necessary to reproduce it.

3.1. Curating the code

Science is, by its very nature, an iterative process. The task of cleaning and analyzing data is a stark example of this. The data obtained from scientific instruments or other measurements of the

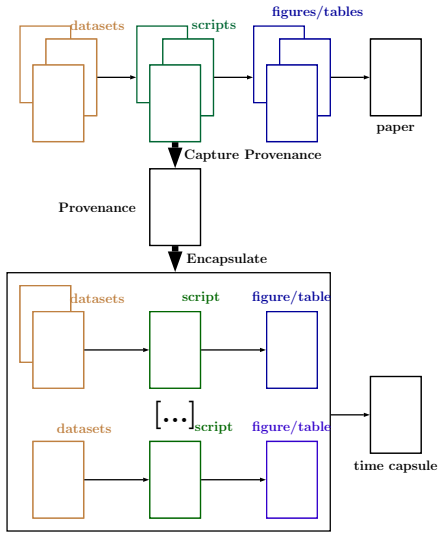

Figure 3. The encapsulation process.

physical world are frequently a superset of the data a scientist wants to analyze. The first step in computation or analysis is often to "process" raw data to produce something that can be analyzed to answer a specific scientific question. This processing typically includes deciding how to handle missing data values, extracting parts of the data, computing new data from pieces of raw data, etc. A scientist typically performs many such operations, not all of which end up being useful. Additionally, code evolves and accretes over time as scientists try different ways to interpret or analyze the data. False starts and abandoned analyses frequently persist in the final scripts that scientists use. The result is that code often contains a complex and evolving story of what transpired, rather than a clear, straight-line path from data to discovery. Although this history may be interesting, it may lead to confusing and difficult-to-understand code.

The first phase of `encapsulator` takes as input the provenance of the computations execution, including all the false starts and abandoned attempts, and produces a curated script corresponding to the generation of a specific result. Such a curated script contains the minimum sufficient code to generate the output. Therefore, to understand a specific result, one can examine the curated version, rather than having to wade through potentially large amounts of irrelevant code.

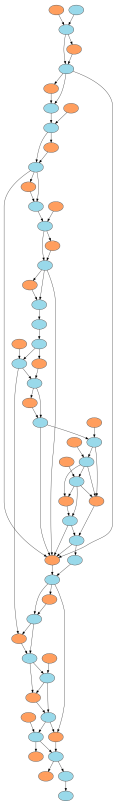

Figure 4. The provenance graph corresponding to a small R script (around 60 lines of code).

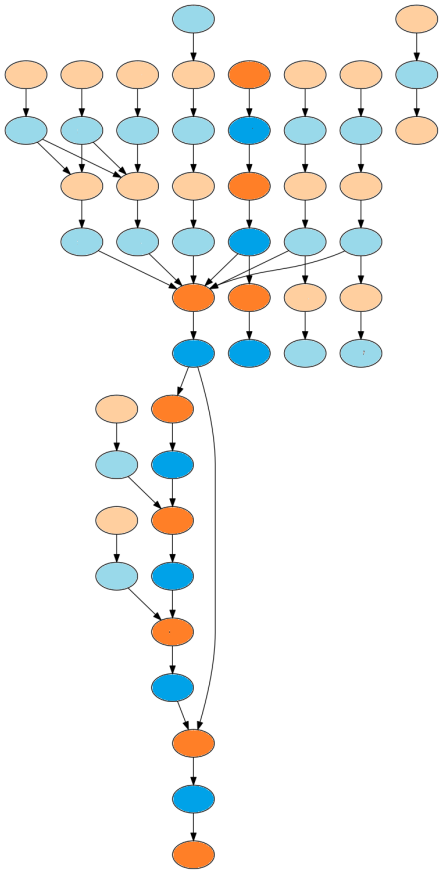

Figure 5. The data dependencies tranformation of the provenance graph shown in Fig. 4.

To generate the minimal "cleaned" code, we analyze the provenance graph. Intuitively, the operations relevant to the generation of a figure or table are those connected in the DAG through data dependencies to the output. First, we trim the provenance graph by deleting control flow, considering only data dependencies. For example, the provenance graph illustrated in Fig. 4 is transformed into a set of data dependency graphs shown in Fig. 5.

In a data dependency graph, orange nodes represent inputs, outputs, or transient data, and blue nodes represent operations on data items. As we examine data dependencies in Fig. 5, we alternate between data items and operations. The code necessary to generate an output (at the top of the figure) is the ordered set of operations present on all paths starting from the output in the original source code (more intense colored nodes in Fig. 5 show an example of such a path for a given output). Similarly, the inputs necessary to generate an output are those encountered while traversing those paths. We generate the final, curated code by retaining all the operations on the paths in the graph leading to the output of interest, and then perform a final pass over the provenance DAG to identify all the required libraries. Once the final code has been generated, we run a source-code formatting tool (`formatR` for R scripts) to bring the code closer to best practices. We repeat these steps for every output of interest until we have generated a curated script for each. The inputs used to generate the selected outputs are identified and saved as part of the time capsule. We have made available (see http://provtools.org/) a stan-

dalone R library (Rclean https://cran.r-project.org/web/packages/Rclean/) implementing the mechanism described here.

3.2. Building the time capsule

Having shown how we produce curated scripts, we next explain how to construct a *time capsule*, leveraging freely-available tools wherever possible. Our goal is to generate a self-contained environment that most scientists can use. This leads to the following requirements:

- The environment should present a user interface familiar to scientists;
- Encapsulation and use (de-encapsulation) of time capsules must require minimal technical expertise;
- The installation process itself must also require minimum intervention and technical knowledge;
- Time capsules, their installation, and re-execution must be platform-independent.

We demonstrate through a practical scenario how well we meet those requirements in the next section.

Based on those criteria, we selected virtual machines (VMs) as the self-contained environment for our time capsules (i.e., their behavior and content is independent of the guest machine, and will remain identical over time). As one of the main barriers to reproducibility is technical, we want to avoid introducing additional technical complexity. Software, such as `VirtualBox` (see https://www.virtualbox.org/), has made VMs an easy-to-use, "push button" technology, and it is possible to use a user-friendly interface to run a virtualized desktop with almost no technical knowledge. To most scientists, a VM will appear as a desktop environment similar to the one they use every day. To facilitate ease of adoption, we make sure that the time capsule contains all the tools scientists need to usefully interact with the computational process.

We use `Vagrant` (see https://www.vagrantup.com/) infrastructure and software to build, share, and distribute time capsules. Its VM provisioning is akin to that of `Docker` for containers. To provision a VM, one simply writes a script specifying the base VM (a pre-configured image), additional software, and files that should be installed. This is completely transparent to scientists: `encapsulator` generates a Vagrant file based on the information extracted from the provenance data in the previous phase. Although users can (optionally) customize the provision script, such customization should never be necessary. In the current prototype, the time capsule is Linux-based, as we leverage its package manager; other operating systems present licensing challenges (discussed in § 5). However, the creation of the time capsule itself can be done from experiments running on Windows, Mac, or any Linux distributions.

The provenance capture is achieved through program introspection using ProvR (see http://provtools.org/). This presents some restrictions regarding the amount of system details that can be captured. In the current proof of concept implementation, we rely on the package manager of the Fedora Linux distribution (see https://fedoraproject.org/wiki/dnf) to install the system dependencies required by a specific version of an R library. We are exploring the possibility of complementing our provenance source using CamFlow [13] (see http://camflow.org/) to capture system level provenance in the Linux operating system. However, it must be noted that system-level provenance capture in closed-source operating systems remains a challenge.

During encapsulation, the scripts created in the first phase run in the time-capsule environment. Their outputs are compared to those from the original script (i.e., the one run on the host machine) to ensure that they are identical. Once the encapsulation is finished, the VM is packaged, ready to be shared. This VM contains individual R scripts for each selected figure, along with the datasets used as inputs. The current prototype relies on `Vagrant`s cloud platform to host the VM.

4. Using Encapsulator

Consider the following scenario: Sarah is a young and brilliant scientist who would like to make her research results available to the community, allow reviewers to easily verify her results, and encourage others to build on them. Prof. O'Brien is a reviewer, interested in verifying Sarah's findings. John is a scientist from a near future who wishes to use Sarah's results.

Alternatives to `encapsulator`

Some systems are designed to reproduce complex workflows running on grid or cloud infrastructures (e.g., `Kepler` [14]), and fill a related, but distinct niche. Indeed, `encapsulator` is intended to support research run on single commodity machines, which accounts for a significant proportion of research results in a number of fields. Systems designed for particular domains already exist (e.g., `GenePattern` [15], and `Galaxy` [16]), but the role of encapsulator is to provide a general approach.

`ReproZip` [8] and `CDE` [17] are directly comparable to `encapsulator`. However, they use system calls to identify dependencies and package experiments. Therefore, computations first must be run in Linux before they can be packaged. This may prove problematic for many scientists who do not use Linux. `encapsulator` relies on language-level observed provenance and is not subject to such limitations.

The main difference between `encapsulator` and alternative tools is the focus on ease of use. Modifying packaged computations generated by the alternatives may require a relatively high level of technical skill. `encapsulator` builds a fully functional, self-contained environment that is easy for scientists to navigate. The list presented here is succinct, but we maintain online a list of open-source provenance tools including some designed for reproducibility and replication purposes (see https://projects.iq.harvard.edu/provenance-at-harvard/tools).

The "messycode" examples (see https://github.com/ProvTools/encapsulator) illustrate several "lazy coding practices" that scientists, including Sarah, often use when writing code for models and analyses:

- near "stream-of-consciousness" coding that follows a train of thought in script development;
- output to console that is not written to disk;
- intermediate objects that are abandoned;
- library and new data calls throughout the script;
- output written to disk but not used in final documents;
- code is not modularized;
- code that is syntactically correct but not particularly comprehensible.

At this stage, we assume that Sarah has finished her computations, built the figures and tables for her paper, and has the paper ready for submission. She is aware of open-data repositories, such as `Dataverse` (see https://dataverse.org/), and source-code repositories, such as `GitHub` (see https://github.com/), but she knows they may not be sufficient to make her code truly re-usable. In the past, when she tried to re-use code written by other scientists, she often discovered that it was poorly documented and hard to use. She also constantly found herself baffled by questions such as what external packages the computation depends on, where to obtain those dependent files and libraries, and what parameters were used to obtain the published results. Trying to figure out these details resulted in her wasting countless hours. She would like to save other scientists from these challenges, so that they can more easily build upon her work.

Sarah wants a "picture" of the context of her computations that allows anyone to reproduce them. Provenance captured by tools such as `provR` (see http://provtools.org/) for R scripts contains the following information, represented as nodes or node attributes in a DAG:

- inputs;
- outputs;
- transient data objects and their values;
- operations;
- library dependencies.

This information facilitates depiction of the development environment, accurately capturing, for example, random seeds used and the version of a library that was required by the system. Although this picture is important, it could prove difficult for John or Prof. O'Brien to use it to create an

environment in which Sarah's computations can be reproduced. They may not have the required expertise or the required version of a library has become unavailable. Thus, Sarah wants her experiments to be preserved in a time capsule.

Sarah decides to use `encapsulator`. She needs to install it and its dependencies: `VirtualBox` and `Vagrant`. On her Mac laptop, she can do this:

```
brew install ruby
gem install encapsulator
encapsulator --install mac
```

Listing 1. Installing `encapsulator` and its dependencies.

The next step is to examine her R script and determine what outputs she wants to include in her time capsule. She can find out what the possibilities are using `encapsulator`'s info capability:

```
encapsulator --info sarah.R
```

Listing 2. Obtaining a summary of an R script.

This generates the following output:

```
Files
-----
Input july_biomass_survey.csv
Input dataset_v2_june_from_collaborator1.
    csv
Output save1.csv
Output fig1_biplot.png
Output fig1_biplot_v2.png
Output fig2_biplot.png

Packages
--------
base v3.4.0
gdata v2.18.0
lattice v0.20-35
permute v0.9-4
txtplot v1.0-3
vegan v2.4-3
```

Listing 3. Installing `encapsulator` and its dependencies.

Sarah included only `fig1_biplot_v2.png` and `fig2_biplot.png` in her article, so she wants to generate a *time capsule* containing only the code (see supplementary material) needed to generate those two images:

```
encapsulator --encapsulate sarah/
    experiment sarah.R fig1_biplot_v2.
    png fig2_biplot.png
```

Listing 4. Creating the time capsule.

Once `encapsulator` has finished building the time capsule, all that is left to do is to upload it to Sarah's `Vagrant` cloud account.

t

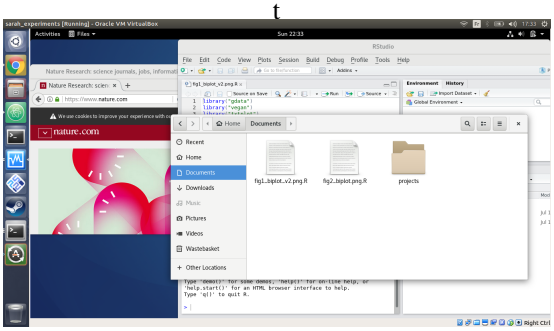

Figure 6. The *time-capsule* running on Prof. O'Brien's machine.

A few months later, Prof. O'Brien is reviewing Sarah's paper and wants to understand her analysis. He sees that Sarah has used `encapsulator` to share her work. As Sarah did in her workflow to produce the published results, he can easily install it on a Linux machine:

```
sudo apt install ruby
gem install encapsulator
encapsulator --install ubuntu
```

Listing 5. Installing `encapsulator` and its dependencies

Once it is installed, he retrieves Sarah's work by running:

```
encapsulator --decapsulate sarah/
    experiment
```

Listing 6. Decapsulating a shared environment.

`encapsulator` manages the VM download and startup transparently. After a short time, a window appears on Prof. O'Brien's desktop presenting him with the virtual desktop shown in Fig. 6. In this environment, he has access to familiar tools and can work without difficulty. Further, the code that he examines for each figure is about a dozen lines of clean code, not Sarah's original 60 lines of messy code. Naturally, `encapsulator` can handle longer and more complex scripts.

John reads Sarah's article five years after its publication. Using the same sequence of commands that Prof. O'Brien used, he is able to get the time capsule running on his laptop, and the environment in the VM is identical to what it was at the time of publication. John can get to work easily without worrying about the problem of outdated dependencies (e.g., old library versions that are no longer available for download).

5. Challenges

Domain specific environment: Our time capsule comes with a generic environment, including some tools generally used for data analysis to provide an easy-to-use, familiar interface. In future versions, based on domain-scientist feedback, we will provide platforms containing standard toolsets specific to domains (e.g., ecology, genetics, chemistry, etc.).

Time-capsule OS: The current version of `encapsulator` uses Linux, in particular the package management system, to build a time capsule. Although a large number of tools used by scientists are available on Windows, Mac, and Linux, some tools may be available only on specific platforms. Furthermore, distributing Mac and Windows capsules introduces licensing issues (proprietary software in research is a complex topic [18]). At this stage, one can build a capsule on any platform, but the capsule itself is Linux-based. This may not pose a major obstacle for domain scientists whose analytical workflows occur almost entirely within an IDE, such as RStudio, since these IDEs are supported on all major operating systems and appear nearly identical across platforms.

Language support: Our current prototype supports only the R programming language. We intend to incorporate support for additional languages used in data analysis, including Python and provenance capture libraries such as `noWorkflow` [19]. Because `encapsulator` uses the PROV-JSON standard for data provenance, any provenance capture tool with a statement-level granularity for any language could be used to generate a capsule. Furthermore, it should be possible to support individual workflows that use multiple languages, which are becoming more common in some domains.

Integration with IDEs: Although they are relatively simple to use, a command-line interfaces are daunting to some users. We are investigating integrating `encapsulator` into existing, commonly-used IDEs, such as an encapsulator add-in for `RStudio`, a common IDE for `R` (see https://rstudio.github.io/rstudioaddins/). Many researchers use spreadsheet programs for their data management and analysis. Although the feasibility and sufficiency of capturing provenance for such workflows has been demonstrated [20], and encapsulation is therefore also theoretically possible, we argue that these methods are inherently unstable since they typically rely on proprietary software with complex underlying data structures. Additionally, best practices for data science typically conflict with spreadsheet-based workflows that tend to lead to informal, and often inaccurate, data management and analysis.

Out-of-tree libraries: Many obscure libraries may not be available through the package management system, either a specific Linux distribution or a programming language, such as `CRAN` (see https://cran.r-project.org/) for `R` packages. We are investigating ways to handle such library dependencies. Those that do not have dependencies are relatively easy to handle by building and installing the package during the encapsulation process. Others that use alternative package managers, such as Bioconductor (see https://www.bioconductor.org/), are also relatively easy to handle. However, those with complex third-party dependencies without formal definitions are more difficult to support.

Non-deterministic processes: Some scripts use pseudo-random-number generators and two runs may not produce identical results. We plan to incorporate the ability to reproduce such results in a future release once the provenance capture system records random values; however, a more serious issue is non-determinism introduced by concurrency. This could be ameliorated during the curation phase by producing scripts that enforce ordering. It might be preferable to enhance how we assess whether a given result produced within the time capsule is correct. In the current proof of concept, the results must be identical to those produced on the host machine. However, it might be reasonable to verify that the results meet some statistical property instead (e.g., within δ of the original results). We recognize that this is not a trivial task and that significant investigation is required to determine a suitable path forward.

Long-term archival: There are two major assumptions that `encapsulator` makes about availability of a time capsule for long-term archival: 1) the continued existence of the Vagrant cloud; and 2) x86-64 virtualization. The first issue can be addressed by replicating the time capsule in a trusted archival repository. One option that we plan to explore in future work is to publish the time capsule in a `Dataverse` repository as

a "replication dataset", assigning automatically a DOI and minimal citation metadata and generating a formal persistent data citation for the time capsule. The second issue is more complex, so the answer is speculative. Virtualization depends on the remaining life span of the x86-64 architecture and whether the concerned time capsule will have any relevance after that. This last point is an interesting issue to ponder, as preservation of our digital world is an issue [21] that goes beyond science and reproducibility. Artifacts of our modern culture are already disappearing (e.g., video games and digital publications), which is an important socio-cultural issue beyond the scope of our current project.

Container support: Although we claim that tools such as Docker are not ideal to reduce the technical barriers to reproducibility for scientists, they are useful for automating the repetition of results. As `Vagrant` supports container provisioning, `encapsulator` could handle such targets. However, one should also remember that while containers are lighter, they are not as self-contained as virtual machines. Indeed, containers run over the kernel of their host machine; if change to the kernel were to affect results then reproducibility could not be guaranteed.

6. Conclusion

We introduce `encapsulator`, a sophisticated yet simple toolbox that uses the provenance of computational data analysis to produce a *time capsule* in which computational workflows can be re-run and modified. This tool is designed to require minimal overhead for integration into a user's workflow and limited technical expertise. When viewed within the context of increasing computational demands of all disciplines, `encapsulator` provides a key tool for facilitating transparent research at a crucial time for science.

Acknowledgment

This work was supported by the US National Science Foundation grant SSI-1450277 *End-to-End Provenance* and grant ACI-1448123 *Citation++*. More details about those projects is available at https://projects.iq.harvard.edu/provenance-at-harvard.

Our reviewers were Prof. Lorena Barba (School of Engineering and Applied Science, George Washington University) and Prof. Carl Boettiger (Department of Environmental Science, Policy and Management, University of California Berkeley). They both helped to clarify the terminology used around reproducibility. Prof Boettiger helped us to clarify the extent of the provenance captured.

Software Engineering Practices

All software presented in this paper is open-source under GPL v3, and available at http://provtools.org/ or directly through GitHub (https://github.com/ProvTools). The latest version (at the time of submission) can be referenced with the DOI: *10.5281/zenodo.1199232* and is distributed via the RubyGems service (https://rubygems.org/gems/encapsulator). The software presented in this paper remains under development and is subject to change. Matthew K. Lau should be contacted for any additional information about the ProvTools ecosystem. Further details about continuous integration and engineering practices are available in the README.md files of the individual components.

References

[1] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature*, vol. 533, no. 7604, pp. 452–454, 2016.

[2] J. D. Gezelter, "Open source and open data should be standard practices," *Journal of Physical Chemistry*, 2015.

[3] D. Garijo, S. Kinnings, L. Xie, L. Xie, Y. Zhang, P. E. Bourne, and Y. Gil, "Quantifying reproducibility in computational biology: the case of the tuberculosis drugome," *PloS one*, vol. 8, no. 11, p. e80278, 2013.

[4] L. N. Joppa, G. McInerny, R. Harper, L. Salido, K. Takeda, K. O'hara, D. Gavaghan, and S. Emmott, "Troubling trends in scientific software use," *Science*, vol. 340, no. 6134, pp. 814–815, 2013.

[5] T. Pasquier, M. Lau, A. Trisovic, E. Boose, B. Couturier, M. Crosas, A. Ellison, V. Gibson, C. Jones, and M. Seltzer, "If these data could talk," *Scientific Data*, 2017, accepted.

[6] L. Carata, S. Akoush, N. Balakrishnan, T. Bytheway, R. Sohan, M. Seltzer, and A. Hopper, "A primer on provenance," *Communications of the ACM*, vol. 57, no. 5, pp. 52–60, 2014.

[7] R. Strijkers, R. Cushing, D. Vasyunin, C. de Laat, A. S. Belloum, and R. Meijer, "Toward executable scientific publications," *Procedia Computer Science*, vol. 4, pp. 707–715, 2011.

[8] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "Reprozip: Computational reproducibility with ease," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 2085–2088.

[9] J. Cunha, J. P. Fernandes, H. Ribeiro, and J. Saraiva, "Towards a catalog of spreadsheet smells," in *International Conference on Computational Science and Its Applications*. Springer, 2012, pp. 202–216.

[10] M. Ziemann, Y. Eren, and A. El-Osta, "Gene name errors are widespread in the scientific literature," *Genome biology*, vol. 17, no. 1, p. 177, 2016.

[11] A. M. Ellison, L. J. Osterweil, L. Clarke, J. L. Hadley, A. Wise, E. Boose, D. R. Foster, A. Hanson, D. Jensen, P. Kuzeja, E. Riseman, and H. Schultz, "An analytic web to support the analysis and synthesis of ecological data," *Ecology*, vol. 87, no. 6, pp. 1345–1358, 2006.

[12] U. Braun, S. Garfinkel, D. Holland, K.-K. Muniswamy-Reddy, and M. Seltzer, "Issues in automatic provenance collection," in *Provenance and annotation of data*. Springer, 2006, pp. 171–183.

[13] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Symposium on Cloud Computing (SoCC'17)*. ACM, 2017, pp. 405–418.

[14] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*. IEEE, 2004, pp. 423–424.

[15] M. Reich, T. Liefeld, J. Gould, J. Lerner, P. Tamayo, and J. P. Mesirov, "Genepattern 2.0," *Nature genetics*, vol. 38, no. 5, pp. 500–501, 2006.

[16] B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor *et al.*, "Galaxy: a platform for interactive large-scale genome analysis," *Genome research*, vol. 15, no. 10, pp. 1451–1455, 2005.

[17] B. Howe, "Cde: A tool for creating portable experimental software packages," *Computing in Science & Engineering*, vol. 14, no. 4, pp. 32–35, 2012.

[18] A. Gambardella and B. H. Hall, "Proprietary versus public domain licensing of software and research products," *Research Policy*, vol. 35, no. 6, pp. 875–892, 2006.

[19] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire, "noWorkflow: Capturing and Analyzing Provenance of Scripts," in *International Provenance and Annotation Workshop*, 2014, pp. 71–83.

[20] H. U. Asuncion, "In situ data provenance capture in spreadsheets," in *International Conference on eScience*. IEEE, 12 2011, pp. 240–247.

[21] K.-H. Lee, O. Slattery, R. Lu, X. Tang, and V. McCrary, "The state of the art and practice in digital preservation," *Journal of research of the National institute of standards and technology*, vol. 107, no. 1, p. 93, 2002.

Appendix

```
### Messy code is a fabricated example
### intended to capture the essentials
### of a typical, lazy scripter's R code.
### It is, however, tremendously more
### organized than the vast majority of
### scripts.

### Depencies are loaded throughout the
    script.

### Also, some depencies that are loaded
    are often
### not used anymore but are still
    present.
library('gdata')

### Read data from some random file path
### Here, a relative path is being used,
    but
### typically, file paths are given from
    root.
data.16 <- read.csv("projects/2016/july_
    biomass_survey.csv")

### Some datasets are loaded and no
    longer used.
### Like this one
data.16.2 <- read.csv('projects/data_
    forestplot/dataset_v2_june_from_
    collaborator1.csv')

### Create a bunch of intermediate
    objects
data.v1.1to4 <- data.16[,1:4]
data.v1.1to4. <- data.v1.1to4
data.v1.1to4 <- data.v1.1to4 * 2
data.v1.1to4.2 <- data.v1.1to4 * 2
data.16[,1:4] <- data.v1.1to4.2
library('vegan')
d1 <- vegdist(data.16[,1:2])
d2 <- vegdist(data.16[,2:3])

### Conduct some analyses
mant1 <- mantel(d1,d2)
mant2 <- mantel(d2,d1)
mant11 <- mantel(d1,d1)
fit1 <- lm(Sepal.Length~Sepal.Width,data=
    data.16)
lm.summary.1 <- summary(fit1)

### write some data to file
write.csv(data.v1.1to4,'projects/data_
    forestplot/save1.csv',row.names = F)

### write some analyses to file
capture.output(lm.summary.1, file="
    analysis_forest/anova_table_1.txt")

### write some figures to file
### Here's another random, unused package
library('txtplot')

png('figures_1/fig1_biplot.png')
plot(data.16[,1:2])
dev.off()

png('figures_1/fig1_biplot_t2.png')
plot(data.16[,1:2]*2)
dev.off()

png('figures_2/fig2_biplot.png')
plot(data.16[,2:3])
dev.off()
```

Listing 7. Original “messy” code.

```
data.16 <- read.csv("projects/2016/july_
    biomass_survey.csv")
data.v1.1to4 <- data.16[, 1:4]
data.v1.1to4 <- data.v1.1to4 * 2
data.v1.1to4.2 <- data.v1.1to4 * 2
data.16[, 1:4] <- data.v1.1to4.2
png("figures_1/fig1_biplot_v2.png")
plot(data.16[, 1:2] * 2)
dev.off()
```

Listing 8. Curated code for figure 1.

```
data.16 <- read.csv("projects/2016/july_
    biomass_survey.csv")
data.v1.1to4 <- data.16[, 1:4]
data.v1.1to4 <- data.v1.1to4 * 2
data.v1.1to4.2 <- data.v1.1to4 * 2
data.16[, 1:4] <- data.v1.1to4.2
png("figures_2/fig2_biplot.png")
plot(data.16[, 2:3])
dev.off()
```

Listing 9. Curated code for figure 2.