RESCIENCE C

Replication / Fault Tolerance

[Re] Storage Tradeoffs in a Collaborative Backup Service for Mobile Devices

Ludovic Courtès[1, ID]
[1]Inria, Bordeaux, France

Edited by
Tiziano Zito

Reviewed by
Sabino Maggi [ID]

Received
28 April 2020

Published
09 June 2020

DOI
10.5281/zenodo.3886739

This article reports on the effort to reproduce the results shown in *Storage Tradeoffs in a Collaborative Backup Service for Mobile Devices*[1], an article published in 2006, more than thirteen years ago. The article presented the design of the storage layer of such a backup service. It included an evaluation of the efficiency and performance of several storage pipelines, which is the experiment we replicate here.
Additionally, this article describes a way to capture the complete dependency graph of this article and the software and data it refers to, making it fully reproducible, end to end. Using GNU Guix[2], we bridge together code that deploys the software evaluated in the paper, scripts that run the evaluation and produce plots, and scripts that produce the final PDF file from LaTeXsource and plots. The end result—and the major contribution of this article—is approximately 400 lines of code that allow Guix to rebuild the whole article *and the experiment it depends on* with a well-specified, reproducible software environment.

1 Getting the Source Code

The first author's younger self, a PhD student, would follow good practices: the libchop library[3] benchmarked in the article, the article itself, and the benchmarking scripts were all under version control. Libchop was published as free software, but the other repositories had never been published, which is not-so-good practice. Thus, the first task in replicating this analysis was to find the initial repositories.
Luckily, the repositories were found on a dusty hard disk drive. However, they were repositories for the GNU Arch version control system, also known as `tla`[4]—one of the first free software distributed version control systems, which saw its last release in 2006, around the time Git started to get momentum.
Having deployed `tla`, the author was able to convert the following repositories, thanks to the `git archimport` command, still distributed with Git:

- https://gitlab.inria.fr/lcourtes-phd/edcc-2006 contains the source of the paper itself—*i.e.*, the text and figures, but neither the benchmarking scripts nor the source of libchop. It turned out to not be of any use for this replication.
- https://gitlab.inria.fr/lcourtes-phd/chop-eval contains the scripts used to run the benchmarks that led to Figure 5 of the paper[1].

The code of libchop itself was published as free software in 2007 and continued to evolve in the following years[3]. As of this writing, there have been no changes to its source code since 2016.

Correspondence should be addressed to Ludovic Courtès (ludovic.courtes@inria.fr)
The authors have declared that no competing interests exist.
Code is available at https://gitlab.inria.fr/lcourtes-phd/edcc-2006-redone. – SWH swh:1:rev:36fde7e5ba289c4c3e30d9afccebbe0cfe83853a;origin=https://gitlab.inria.fr/lcourtes-phd/edcc-2006-redone.
Open peer review is available at https://github.com/ReScience/submissions/issues/32.

2 Building the Source Code

Libchop is written in C and accessible from Scheme thanks to bindings for GNU Guile, an implementation of the Scheme programming language. The benchmarking scripts mentioned above rely on those Scheme bindings.

2.1 Dependencies

In addition to Guile, it has a number of dependencies, among which:

- the GNU DBM key/value database (*gdbm*);
- the GNU Libgcrypt cryptography library;
- the zlib, bzip2, and lzo compression libraries;
- support for ONC Remote Procedure Calls (RPC), formerly provided as part of the GNU C Library (*glibc*), but nowadays available separately as part of TI-RPC;
- G-Wrap, a now defunct binding generator for Guile.

Additionally, libchop uses the GNU "Autotools" as its build system: Autoconf, Automake, and Libtool.

2.2 Software Deployment as Code

It should come as no surprise that the author, who has been working on reproducible software deployment issue for several years now, felt the need to address the software deployment issue using GNU Guix[2].
GNU Guix allows users to deploy software in a way similar to popular "package managers" such as Fedora's RPM, Debian's APT, or CONDA. Unlike those, it follows a *functional deployment* paradigm, inherited from Nix[5]. "Functional" in this context means that Guix views software build processes as pure functions that take inputs—source code, build scripts, compilers, libraries—and produce output—libraries, programs. It arranges so that build processes run in well-defined environments that contain nothing but the declared inputs. Thus, given the same inputs, deterministic build processes always produce the same output, bit for bit. Consequently, Guix supports *reproducible* software deployment, which we consider a prerequisite for computational experiments—the digital counterpart of the pen-and-paper lab book.
Guix can be programmed in Scheme, a language of the Lisp family. It provides high-level interfaces that allow users to define software packages in a declarative fashion that does not require familiarity with Scheme[6]. For the purposes of this replication, the author wrote definitions of the required packages, as we will see below, as well as definitions of each of the stages leading to the final PDF file, as will be explained in Section 4.

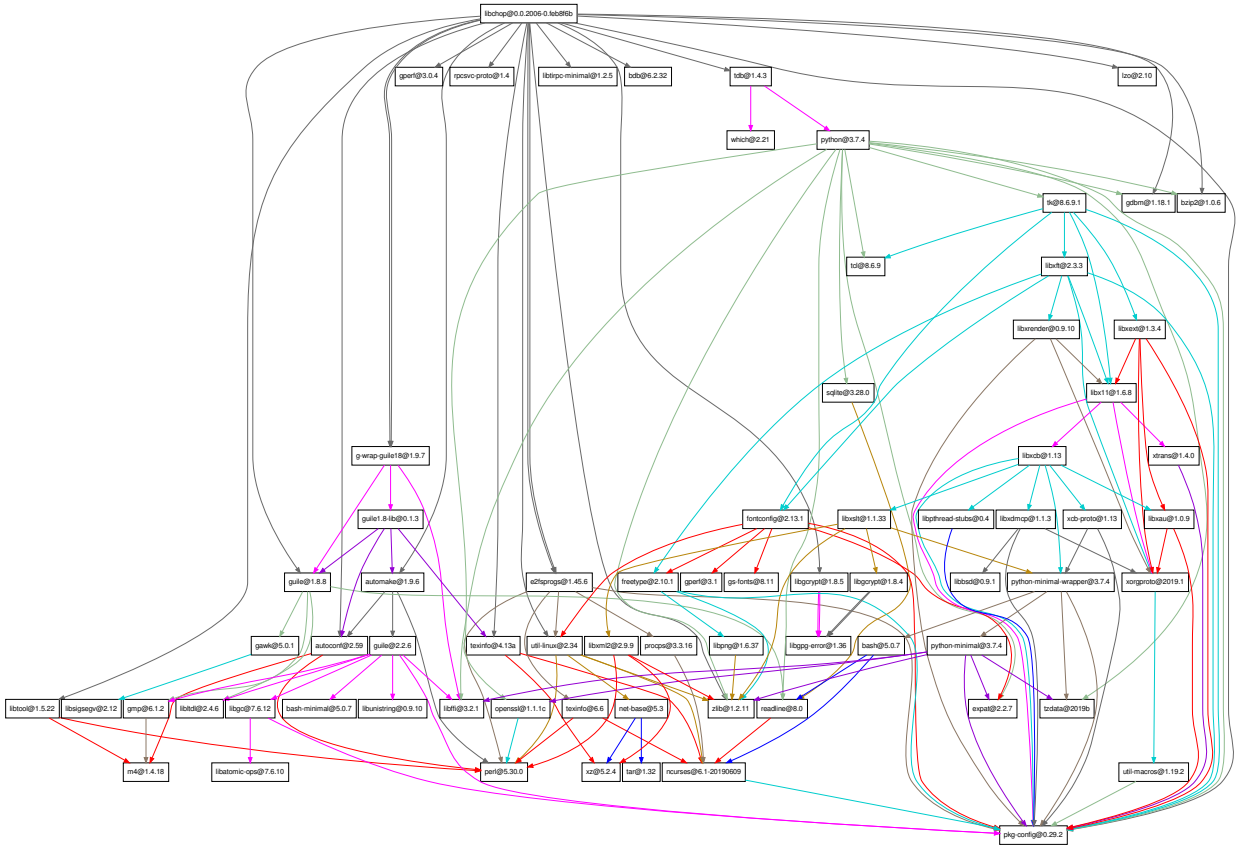
2.3 Choosing a Revision

Should we run the latest revision of libchop, dated 2016, or should we rather run the 2006 revision that was used at the time the paper was written? The latest libchop revision is available as a GNU Guix package. Unfortunately, the benchmarking scripts mentioned above are stuck in 2006–2007, so to speak: they require libchop programming interfaces that changed after that time, and they also require interfaces specific to Guile 1.8, the version that was current at the time (the latest version of Guile today is 3.0.2; it has seen *three* major versions since 2006).
The author chose to use libchop, Guile, and G-Wrap from 2006, but reusing as many as possible of today's software packages apart from these. Building from a *source tarball*—a

`tar.gz` archive—produced by the Autotools (with `make dist`) is as simple as running `./configure; make`. The nice property here is that users do not need to install the Autotools to do that: all they need is a shell and `make`, along with the tools needed to build the software itself.

Unfortunately, no release of libchop had been published as a source tarball back then. Thus, we had to build it from a version-control checkout, which requires the Autotools so we can generate the `configure` script and related files. The author quickly found out that building the 2006 libchop would *also* require building versions of Autoconf, Automake, and Libtool that were current back then since today's versions are incompatible. Fortunately, the "downgrade cascade" stops here.

Figure 1. Dependency graph for the 2006 revision of libchop.



The Guix-Past channel for GNU Guix was developed to provide reproducible, unambiguous definitions for all these software packages: https://gitlab.inria.fr/guix-hpc/guix-past. It provides a 2006 revision of libchop, along with 2006 versions of the aforementioned software. This channel can be used with today's Guix, bringing software from the past to the present. The libchop revision was chosen as dating to right before the submission of the paper for the European Dependable Computing Conference (EDCC), where it was eventually presented.

The resulting dependency graph—packages needed to build this libchop revision—is of course more complex. It is shown in Figure 1 for reference (the reader is invited to zoom in or use a high-resolution printer). It is interesting to see that it is a unique blend of vintage 2006 packages with 2020 software. Section 4 will get back to this graph.

Table 1. File sets.

Name	Size	Files	Average Size
Lout (versions 3.20 to 3.29)	76 MiB	5,853	13 KiB
Ogg Vorbis files	32 MiB	10	3 MiB
mbox-formatted mailbox	8 MiB	1	8 MiB

Table 2. Storage pipeline configurations benchmarked.

Config.	Single Instance?	Chopping Algo.	Block Size	Input Zipped?	Blocks Zipped?
A1	no	—	—	yes	—
A2	yes	—	—	yes	—
B1	yes	Manber's	1024 B	no	no
B2	yes	Manber's	1024 B	no	yes
B3	yes	fixed-size	1024 B	no	yes
C	yes	fixed-size	1024 B	yes	no

3 Running the Benchmarks

Section 4.2 of the original paper[1] evaluates the efficiency and computational cost of several storage pipelines, on different file sets, each involving a variety of compression techniques.

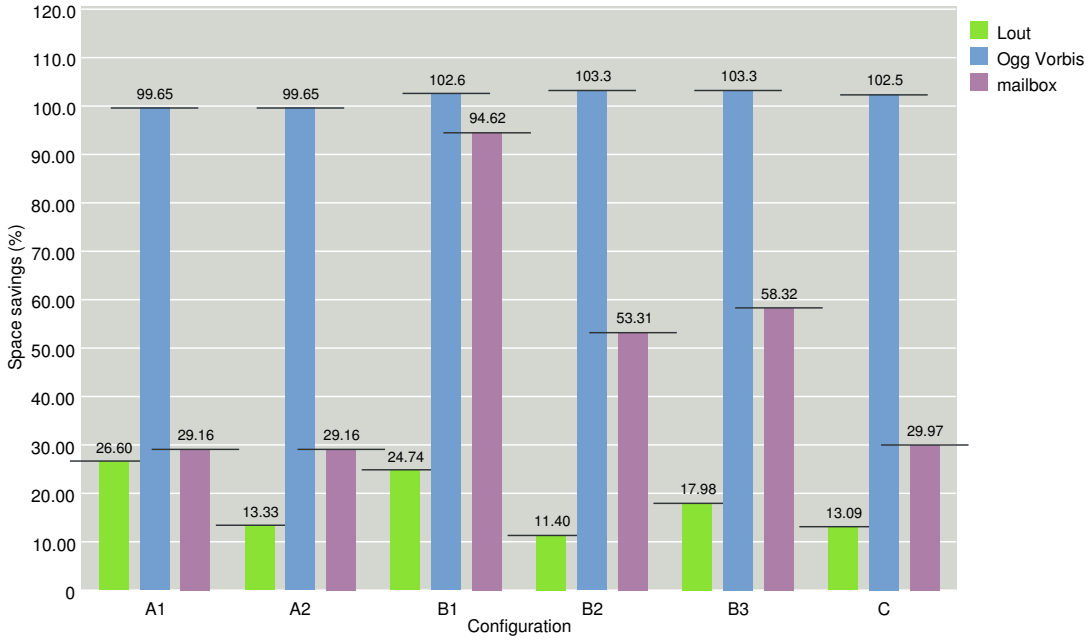
3.1 Input File Sets

Figure 3 of the original article describes the three file sets used as input of the evaluation. Of these three file sets, only the first one could be recovered precisely: it is source code publicly available from https://download.savannah.gnu.org/releases/lout and in the Software Heritage archive. The two other file sets were not publicly available. With the information given in the paper, we decided to use *similar* file sets, publicly available this time. For the "Ogg Vorbis" file set, we chose freely-redistributable files available from https://archive.org/download/nine_inch_nails_the_slip/. For the "mailbox" file set, we chose an mbox-formatted monthly archive of the `guix-devel@gnu.org` mailing list.
Table 1 summarizes the file sets used in this replication. This is an informal description, but rest assured: Section 4 will explain the "executable specification" of these file sets that accompanies this article.

3.2 Evaluation Results

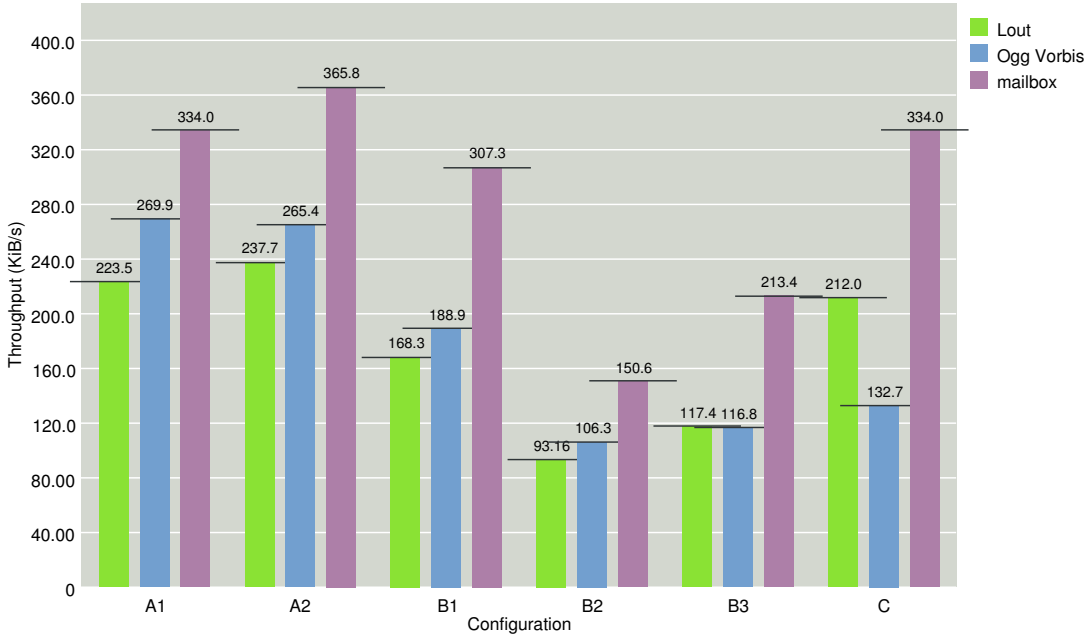
Like in the original article, we benchmarked the configurations listed in Table 2. Running the benchmarking scripts using the libchop revision packaged earlier revealed a crash for some of the configurations. Fortunately, that problem had been fixed in later revisions of libchop, and we were able to "backport" a small fix to our revision (most likely, the bug depended on other factors such as the CPU architecture and libc version and did not show up back in 2006).
The original benchmarks run on a PowerPC G4 machine running GNU/Linux. This time, we ran them on an x86_64 machine with an Intel i7 CPU at 2.6 GHz (the author playfully started looking for a G4 so that even the *hardware* setup could be replicated, but eventually gave up). The benchmarking results in Figure 5 of the original paper[1] were squashed in a single, hard-to-read chart. Here we present them as two separate figures: Figure 2 shows the space savings (ratio of the resulting data size to the input data size) and Figure 3 shows the throughput of each storage pipeline, for each file set.
The space savings in Figure 2 are about the same as in the original article, with one exception: the "mailbox" file set has noticeably better space savings in configurations A1

Figure 2. Ratio of the resulting data size to the input data size (lower is better).



and C this time. This could be due to the mailbox file chosen in this replication exhibiting more redundancy; or it could be due to today's zlib implementation having different defaults, such as a larger compression buffer, allowing it to achieve better compression.

Figure 3. Throughput for each storage pipeline and each file set (higher is better).



The throughput shown in Figure 3 is, not surprisingly, an order of magnitude higher than that measured on the 2006-era hardware. The CPU cost of configurations relative

to one another is close to that of the original paper, though less pronounced. For example, the throughput for B2 is only half that of A1 in this replication, whereas it was about a third in the original paper. There can be several factors explaining this, such as today's compiler producing better code for the implementation of the "chopper" based on Manber's algorithm in libchop, or very low input/output costs on today's hardware (using a solid-state device today compared to a spinning hard disk drive back then).
Overall, the analysis in Section 4.2.2 of the original paper remains valid today. The part of evaluation that relates to the CPU cost is, as we saw, sensitive to changes in the underlying hardware. Nevertheless, the main performance characteristics of the different configurations observed in 2006 remain valid today.

4 Reproducing this Article

We were able to replicate experimental results obtained thirteen years ago, observing non-significant variations. Yet, this replication work highlighted the weaknesses of the original work, which fall into three categories:

1. Lack of a properly referenced public archive of the input data.
2. Gaps in the document authoring pipeline: running the benchmarks was fully automated thanks to the scripts mentioned earlier, but the figure that appeared in the 2006 paper was made "by hand" from the output produced by the script.
3. Lack of a way to redeploy the software stack: the 2006 article did not contain references to software revisions and version numbers, let alone a way to automatically deploy the software stack.

This section explains how we addressed, in a rigorous and reproducible way, all these issues.

4.1 Deploying Software

The original paper lacked references to the software. Figure 1 here provides much information, but how useful is it to someone trying to redeploy this software stack? Sure it contains version and dependency information, but it says nothing about configuration and build flags, about patches that were applied, and so on. It also lacks information about dependencies that are considered implicit such as the compiler tool chain. This calls for a *formal and executable specification* of the software stack.
As mentioned in Section 2, we defined all the software stack as Guix packages: most of them pre-existed in the main Guix channel, and old versions that were needed were added to the new Guix-Past channel. By specifying the commits of Guix and Guix-Past of interest, one can build the complete software stack of this article. For example, the instructions below build the 2006 revision of libchop along with its dependencies, downloading pre-built binaries if they are available:

```
git clone https://gitlab.inria.fr/lcourtes-phd/edcc-2006-redone
cd edcc-2006-redone
guix time-machine -C channels.scm -- build libchop@0.0
```

The file `channels.scm` above lists the commits of Guix and Guix-Past to be used. Thus, recording the commit of `edcc-2006-redone` that was used *is all it takes to refer unambiguously to this whole software stack*.
The key differences compared to a "container image" are *provenance tracking* and *reproducibility*. Guix has a complete view of the package dependency graph; for example, Figure 1 is the result of running:

```
guix time-machine -C channels.scm -- graph libchop@0.0 \
  | dot -Tpdf > graph.pdf
```

Furthermore, almost all the packages Guix provides are bit-reproducible: building a package at different times or on different machines gives the exact same binaries (there is a small minority of exceptions, often packages that record build timestamps).
Last, each package's source code is automatically looked up in Software Heritage should its nominal upstream location become unreachable.

4.2 Reproducible Computations

Often enough, software deployment is treated as an activity of its own, separate from computations and from document authoring. But really, this separation is arbitrary: a software build process *is* a computation, benchmarks like those discussed in this paper *are* computations, and in fact, the process that produced the PDF file you are reading is yet another computation.
The author set out to describe this whole pipeline as a single dependency graph whose sink is the LaTeX build process that produces this PDF. The end result is that, from a checkout of the `edcc-2006-redone` repository, this PDF, *and everything it depends on* (software, data sets, benchmarking results, plots) can be produced by running:

```
guix time-machine -C channels.scm -- build -f article/guix.scm
```

The files `guix.scm` and `article/guix.scm` describe the dependency graph above libchop. Conceptually, they are similar to a makefile and in fact, part of `article/guix.scm` is a translation of the makefile of the ReScience article template. Using the Scheme programming interfaces of Guix and its support for *code staging*, which allows users to write code staged for eventual execution[6], these files describe the dependency graph and, for each node, its associated build process.
For the purposes of this article, we had to bridge the gap from the benchmarking scripts to the actual plots by implementing a parser of the script's standard output that would then feed it to Guile-Charting, the library used to produce the plots. They are chained together in the top-level `guix.scm` file. The graph in Figure 1 is also produced automatically as part of the build process, using the channels specified in `channels.scm`. Thus, it is guaranteed to describe precisely to the software stack used to produce the benchmark results in this document.
What about the input data? Guix `origin` records allow us to declare data that is to be downloaded, along with the cryptographic hash of its content—a form of *content addressing*, which is the most precise way to refer to data, independently of its storage location and transport. The three file sets in Figure 1 are encoded as `origin`s and downloaded if they are not already available locally.

Listing 1. Representation of a content-addressed Git checkout.

```
(define rescience-template
  (origin
    (method git-fetch)
    (uri (git-reference
           (url "https://github.com/rescience/template")
           (commit "93ead8f348925aa2c649e2a55c6e16e8f3ab64a5")))
    (sha256
     (base32 "10xrflbkrv6bq92nd169y5jpsv36dk4i6h765026wln7kpyfwk8j"))))
```

As an example, Listing 1 shows the definition of a Git checkout. The `origin` form specifies the expected SHA256 content hash of the checkout; thus, should the upstream repository be modified in place, Guix reports it and stops. Guix transparently fetches the specified commit from the Software Heritage archive if the upstream repository is unavailable and, of course, assuming it has been archived.

4.3 Discussion

The techniques described above to encode the complete document authoring pipeline as a fully-specified, executable and reproducible computation, could certainly be applied to a wide range of scientific articles. We think that, at least conceptually, it could very much represent the "gold standard" of reproducible scientific articles. Nevertheless, there are three points that deserve further discussion: handling input data, dealing with non-deterministic computations, and dealing with expensive computations.

Our input file sets were easily handled using the standard Guix `origin` mechanism because they are relatively small and easily downloaded. This data is copied as content-addressed items in the "store", which would be unsuitable or at least inconvenient for large data sets. Probably some "out-of-band" mechanism would need to be sought for those data sets—similar to how Git-Annex provides "out-of-band" data storage integrated with Git. As an example, the developers of the Guix Workflow Language[7] (GWL), which is used for bioinformatics workflows over large data sets, chose to treat each process and its data outside standard Guix mechanisms.

The second issue is non-deterministic byproducts like the performance data of Figure 3. That information is inherently non-deterministic: the actual throughput varies from run to run and from machine to machine. The functional model implemented in Guix[5] is designed for deterministic build processes. While it is entirely possible to include non-deterministic build processes in the dependency graph without any practical issues, there is some sort of an "impedance mismatch". It would be interesting to see whether explicit support for non-deterministic processes would be useful.

Last, the approach does not mesh with long-running computations that require high-performance computing (HPC) resources. Again, some mechanism is needed to bridge between these necessarily out-of-band computations and the rest of the framework. The GWL provides preliminary answers to this question.

5 Related Work

Software engineering around "reproducible research" in a broad sense is a fast-moving field. Researchers interested in reproducibility these days are often familiar with tools such as Docker, Jupyter, and Org-Mode. This section explains how Guix and the technique described in Section 4 relates to these other tools and approaches.

First, it is worth noting that these tools are not concerned with supporting reproducible computations in general: Docker focuses on software deployment whereas Jupyter Notebook focuses on document authoring. Conversely, our work in this article is about achieving reproducibility and provenance tracking *end to end*.

Docker and similar "container tools", such as Singularity, really combine two tools: one to build "application bundles" (or "container images"), and one to run the software contained in such bundles. The latter is a thin layer above virtualization mechanisms built into the kernel Linux (in particular "namespaces"), which provides much welcome flexibility to users. The former is about provisioning those container images, and we think it hinders provenance tracking and reproducibility.

As an example, the "source code" of a container image built with Docker is a "Docker file". Docker files start by importing an existing container image, which contains pre-built software. This starting point already loses the connection to source code. Docker files go on by listing commands to run to install additional software in the image. Those commands typically download additional pre-built binaries from external servers. Consequently, the result of those commands depends on external state; it may vary over time, or even fail. In other words, Docker files describe *non-reproducible computations* and are *opaque*.

At the other end of the spectrum, Jupyter Notebook and Jupyter Lab support literate programming, like Org-Mode. Users can write documents that interleave a narrative

and code snippets; Jupyter takes care of evaluating those code snippets and presenting their result in the document. Jupyter focuses on document authoring, leaving software deployment as an exercise for the user. For example, to evaluate a Jupyter notebook that contains Python code using the NumPy library, the user must install the right version of Python and NumPy. A common approach is to ship Docker containers that contain Jupyter Notebook and all the necessary dependencies, often delegating it to services such as https://mybinder.org/. With Guix-Jupyter, we proposed a different approach where users annotate notebooks with information about their software dependencies, which Guix automatically deploys in a reproducible fashion[8].

End-to-end documentation authoring pipelines have previously been explored from different angles notably with ActivePapers framework[9], Maneage[10,11], by combining literate programming with Org-Mode and version control with Git[12], and by combining scientific pipelines and a LaTeXpipeline in Docker images[13]. Maneage is one of the few efforts to consider software deployment as part of the broader scientific authoring pipeline. However, software deployed with Maneage relies on host software such as a compilation tool chain, making it non-self-contained; it also lacks the provenance tracking and reproducibility benefits that come with the functional deployment model implemented in Guix. Reconciler[13] connects the scientific software workflow to the document authoring pipeline through two distinct Docker images. It provides a way to check that the end result (the PDF) is bit-for-bit reproducible. Guix can check for the reproducibility of *each* computation—package builds, benchmark runs, LaTeXpipeline—through its `--check` command-line option.

6 Conclusion

We are glad to report that we were able to replicate the experimental results that appear in our thirteen-year-old article and that its conclusions in this area still hold[1]. But really, truth be told, the replication was also an excuse to prototype an *end-to-end reproducible scientific pipeline*—from source code to PDF.

We hope our work could serve as the basis of a template for reproducible papers in the spirit of Maneage. We are aware that, in its current form, our reproducible pipeline requires a relatively high level of Guix expertise—although, to be fair, it should be compared with the wide variety of programming languages and tools conventionally used for similar purposes. We think that, with more experience, common build processes and idioms could be factorized as libraries and high-level programming constructs, making it more approachable.

This article was built from commit `cf110733aa03c2cf9c1051bb6a2c1ae8562c35c2` of the `edcc-2006-redone` repository. It is interesting to see that this single Git commit identifier, which can be looked up on Software Heritage, is enough to refer to whole pipeline leading to this article! We look forward to a future where reproducible scientific pipelines become commonplace.

References

1. L. Courtès, M.-O. Killijian, and D. Powell. “Storage Tradeoffs in a Collaborative Backup Service for Mobile Devices.” In: **Proceedings of the Sixth European Dependable Computing Conference**. Coimbra, Portugal: IEEE CS Press, Oct. 2006, pp. 129–138.
2. L. Courtès and R. Wurmus. “Reproducible and User-Controlled Software Environments in HPC with Guix.” In: **2nd International Workshop on Reproducibility in Parallel Computing (RepPar)**. Vienna, Austria, Aug. 2015.
3. L. Courtès. **Libchop**. https://nongnu.org/libchop. Accessed 2020/04/28.
4. T. Lord et al. **GNU Arch**. https://www.gnu.org/software/gnu-arch/. Accessed 2020/04/28.
5. E. Dolstra, M. de Jonge, and E. Visser. “Nix: A Safe and Policy-Free System for Software Deployment.” In: **Proceedings of the 18th Large Installation System Administration Conference (LISA)**. Atlanta, Georgia, USA: USENIX, Nov. 2004, pp. 79–92.

6. L. Courtès. “Code Staging in GNU Guix.” In: **16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17)**. Vancouver, Canada, Oct. 2017.
7. R. Janssen, R. Wurmus, et al. **GNU Guix Workflow Language**. https://www.guixwl.org/. Accessed 2020/04/28.
8. L. Courtès. **Towards reproducible Jupyter notebooks**. https://hpc.guix.info/blog/2019/10/towards-reproducible-jupyter-notebooks/. Accessed 2020/05/29.
9. K. Hinsen. “A Data and Code Model for Reproducible Research and Executable Papers.” In: **Procedia Computer Science** 4 (2011), pp. 579–588.
10. M. Akhlaghi and T. Ichikawa. “Noise-Based Detection and Segmentation of Nebulous Objects.” In: **The Astrophysical Journal Supplement Series** 220.1 (Aug. 2015), p. 1.
11. M. Akhlaghi. **Maneage: Managing Data Lineage**. http://maneage.org/. Accessed 2020/05/29.
12. L. Stanisic and A. Legrand. “Effective Reproducible Research with Org-Mode and Git.” In: **1st International Workshop on Reproducibility in Parallel Computing**. Porto, Portugal, Aug. 2014.
13. P. Bizopoulos and D. Bizopoulos. **Reconciler: A Workflow for Certifying Computational Research Reproducibility**. 2020. arXiv: 2005.12660 [cs.SE].