R4R: Reproducibility for R

Pierre Donat-Bouillud
pierre.donat.bouillud@fit.cvut.cz
Czech Technical University
Prague, Czech Republic
Northeastern University
Boston, United States

Filip Křikava
filip.krikava@fit.cvut.cz
Czech Technical University
Prague, Czech Republic
Northeastern University
Boston, United States

Sebastian Krynski
krynsseb@fit.cvut.cz
Czech Technical University
Prague, Czech Republic
Northeastern University
Boston, United States

Jan Vitek
vitekj@me.com
Czech Technical University
Prague, Czech Republic
Charles University
Prague, Czech Republic

Abstract

Ensuring reproducibility is a fundamental challenge in computational research. Reproducing results often requires reconstructing complex software environments involving data files, external tools, system libraries, and language-specific packages. While various tools aim to simplify this process, they often rely on user-provided metadata, overlook system dependencies, or produce unnecessarily large environments.

We present `r4r`, a tool that automates the creation of minimal, user-inspectable, self-contained execution environments through dynamic program analysis techniques. `r4r` captures all runtime dependencies of a data analysis pipeline and produces a Docker image capable of reproducing the original execution. Although designed with first-class support for the R programming language, `r4r` also includes a generic fallback mechanism applicable to other languages. We evaluate `r4r` on a collection of R Markdown notebooks from Kaggle and find that it achieves exact reproducibility for 97.5% of deterministic notebooks.

CCS Concepts

• **General and reference** → *Empirical studies*; • **Software and its engineering** → **Software version control**; **Software evolution**; **Dynamic analysis**; • **Social and professional topics** → **Software maintenance**; • **Security and privacy** → *Virtualization and security*; • **Applied computing** → *Bioinformatics.*

Keywords

reproducibility, record and replay, container, R language

ACM Reference Format:
Pierre Donat-Bouillud, Filip Křikava, Sebastian Krynski, and Jan Vitek. 2025. R4R: Reproducibility for R. In *ACM Conference on Reproducibility and Replicability (ACM REP '25), July 29–31, 2025, Vancouver, BC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3736731.3746156

1 Introduction

Reproducibility is one of the cornerstones of the scientific method. In computation-based research, this means that conclusions drawn from data need to be supported with the raw data and code that would allow rerunning the analysis from scratch. In other words, transparency is essential to support reproducibility. As experimental results in various fields of science have increasingly failed to be replicated, concerns about the reproducibility of research results have arisen [5]. Some have even claimed it to be a *replicability crisis* [4].

Results are often presented as virtual notebooks. The two most popular formats are Jupyter notebooks [19] for Python, and R Markdown [39] for R. Notebooks are ultimately pieces of software, and, as such, they bring with them all the software-related problems. But, at the same time, they aim for user-friendliness as they are intended to be used by researchers that often lack formal training in programming. Therefore, these languages do not generally impose rigorous package dependencies' specification, or environment, that are needed to run.

A study [38] of 2000 replication datasets with code in R, i.e. research code in their final, published version, showed that only 26% of the code successfully completed execution, and after applying automated fixes, only 56% did succeed. This is only an upper bound on the *reproducible* replications packages, as re-execution of code is just a necessary condition of *reproducibility.* Studies on Python's Jupyter notebooks [29, 43] have shown similar results. As another example, the study [34] shows poor success when attempting to reproduce notebooks from biomedical publications, even in cases where dependencies were declared in standard requirement files. Many of such dependencies did not even install successfully.

Here, we focus on the R language, and we are concerned with the computational reproducibility [27] of notebooks, sometimes called *methods reproducibility* [34], where the goal is to obtain consistent results using the same input data, computational steps, methods, and code, and conditions of analysis. The consistency of results will be checked by comparing the outputs of the analysis when run on different hosts. Note that we do not aim at achieving *result reproducibility* [13] nor *replicability* [1], where the goal is to find the same results but not necessarily using the same exact computational steps. We also want to create reproducible artifacts that are easily inspectable and modifiable by humans.

In R, creating reproducible artifacts has meant painstakingly collecting the versions of R packages but also system dependencies used in the code, making sure that all data sources are included, and carefully reviewing all possible sources of non-determinism. Retrieving the R package versions can now be automated, for instance, with a tool such as `renv` [40], but system dependencies often require manually building a Docker image. Computation reproducibility with the current R tooling indeed require manual or semi-manual efforts. This is further discussed in Section 2.

We introduce a tool, `r4r`, that can *automatically* create a reproducible artifact from an R notebook. It traces the execution of a notebook and detects the R packages, the system dependencies, the data sources, and even the operating system resources that have been used to generate the notebook output. It then generates a human-inspectable Dockerfile and packages it into a Docker image. We detail the design of `r4r` in Section 3 and its implementation in Section 4.

We then evaluate `r4r` on a dataset of R Markdown notebooks from the data science competition Kaggle to check whether `r4r` correctly creates reproducible artifacts in Section 5.

2 Background

Achieving computational reproducibility requires careful management of software environments, data inputs, and code dependencies. In the context of data science workflows, particularly those written in R, reproducibility is complicated by the dynamic nature of the language, the diversity of available packages, and their reliance on system-level libraries. Moreover, the increasing use of computational notebooks as a medium for scientific communication introduces additional challenges, as these documents often conflate code, data, and narrative in a format that lacks rigorous dependency specification. In this section, we provide the necessary background on the R language and its ecosystem, discuss the structure and execution model of computational notebooks, and review existing efforts to support reproducibility in both R and Python.

2.1 The R language

R is an open-source programming language and software environment widely used for statistical computing and data analysis. Originally developed for statisticians, it has become a de facto standard in many scientific disciplines including bioinformatics, data analysis, finance, or data mining [12]. R is particularly well-suited for users without formal training in computer science, as it was designed with statisticians and domain scientists in mind, offering a high-level syntax tailored to data analysis tasks.

The strength of R lies in its extensive ecosystem of packages, which significantly extend the core language's capabilities. These packages are primarily provided from curated repositories such as CRAN[1] and Bioconductor,[2] which together host tens of thousands of actively maintained packages: as of March 2025, CRAN hosts over 22K packages and Bioconductor contains over 2K packages.

[1]https://cran.r-project.org/

[2]https://www.bioconductor.org

Packages are also hosted on code sharing websites such as GitHub or Gitlab.

Many of these packages are written in R, but they can also include native code written in C, C++, Fortran, or even Rust. Consequently, installing such packages often requires a range of system libraries and build tools. Because R packages are distributed primarily as source code, installing them involves compiling code locally. This introduces often undocumented dependencies, not only on other R packages but also on external system-level components, such as compilers, development headers, and shared libraries. As a result, creating a fully reproducible R environment requires capturing both the language-level and system-level dependencies.

2.2 Computational notebooks

Computational notebooks integrate narrative text, executable code, and visualizations or results within a single, cohesive file. These notebooks embody the concept of *literate programming* [20], an approach to software development and research where documentation and source code coexist together, explaining how the code works in natural language.

Both Jupyter and R Markdown notebooks allow researchers to weave descriptive content with computational analysis, fostering reproducibility and transparency. R Markdown (Rmd) notebooks, as the name suggests, use Markdown [14], a lightweight markup language with an optional header specifying metadata and configuration options. The code snippets are organized into fenced blocks. An Rmd document can be rendered into various output formats, such as HTML or PDF. During rendering (the other often used term is *knitting*), the code blocks are sequentially executed, and their results are embedded in the document. This execution produces the final output.

Figure 1a shows an example of an Rmd document with three R code blocks. Rendering is done in two steps. First, the code in these blocks is sequentially executed, embedding the results into a plain markdown document which is then turned into an HTML document as specified in the header, using `pandoc`.[3] The result is shown in Figure 1b. Concretely, this example extracts the shape files of the world's countries and plots the ones that are part of the African continent.[4]

2.3 Related work

2.3.1 R. The R ecosystem has developed various solutions to manage dependencies for notebooks. The canonical solution for R scripts is to create an R package around the scripts and declare the R dependencies in the `DESCRIPTION` file of the package. However, this does not integrate well with R Markdown notebooks.

R dependencies. Some packages aim at capturing the precise versions of the R packages used for the notebook. The most popular one is `renv` [40]; it can analyze the R files in the project, looking for occurrences of `library(package)`, `package::method` and so on, to find out the R dependencies, and saves a snapshot of those that can be sent to another user. `groundhog` [35] supports individual R files, not only projects like `renv`, and makes it possible to load

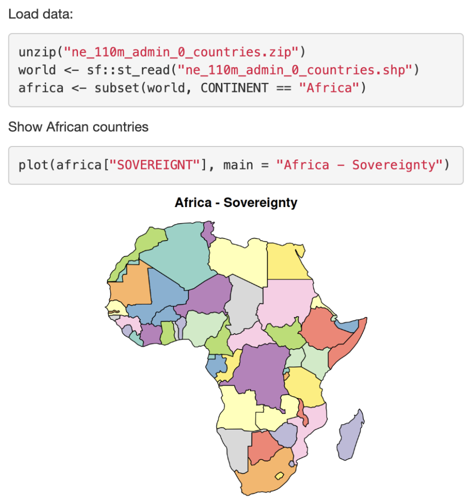
[3]https://pandoc.org/

[4]The archive comes from natural earth data*cf.* https://www.naturalearthdata.com/downloads/110m-cultural-vectors/110m-admin-0-countries/

~~~
---
output: html_document
---

```{r echo=FALSE, include=FALSE}
library(sf)
~~~
Load data:
```{r}
unzip("ne_110m_admin_0_countries.zip")
world <- sf::st_read("ne_110m_admin_0_countries.shp")
africa <- subset(world, CONTINENT == "Africa")
```
Show African countries
```{r}
plot(africa["SOVEREIGNT"], main = "Africa - Sovereignty")
```
```

(a) source (`notebook.Rmd`)



Load data:

```
unzip("ne_110m_admin_0_countries.zip")
world <- sf::st_read("ne_110m_admin_0_countries.shp")
africa <- subset(world, CONTINENT == "Africa")
```

Show African countries

```
plot(africa["SOVEREIGNT"], main = "Africa - Sovereignty")
```

(b) rendered document (`notebook.html`)

Figure 1: Example R markdown document

a specific version of a package at a given date. `miniCRAN` [8] can be used to build a smaller version of CRAN, only tailored to the packages required for a specific notebook. `Require` [23] is similar to `renv` but optimizes for speed.

System dependencies. Other aspects of the system can influence how a notebook runs, such as the operating system, the system libraries, and the compiler flags used to compile R and the packages. All those system and build dependencies can be set along the notebook using a containerized environment, such as Docker. The Rocker project [6] provides several Docker images specialized for various tasks involving R. If more system dependencies are required, a secondary Dockerfile has to be written. Some packages provide utilities to ease adding dependencies, including the system dependencies. `dockerfiler` [9] provides an R API to build Docker files. `pracprac` [24] combines `renv` and helpers to create a Dockerfile. `containerit` [26] scans the R session info and adds required R dependencies to a chosen base image from Rocker. It also adds some system dependencies from a manually curated repository that only supports them for some Linux distributions.[5] `liftr` [25] extends the YAML metadata section of a R Markdown document with directives to indicate the base Docker image and the R and system dependencies. `r2u`[6] installs CRAN packages as Ubuntu packages, which makes it possible to specify system dependencies. However, it does not handle all versions of the packages. `rang` [16] aims at reconstructing *a historical computation environment* by specifying a past date when the notebook used to run and queries various databases to find dependencies, including system ones. `rix` [32] leverages *Nix*, a Linux package manager focused on reproducible builds, to specify together the version of R, its packages, and all the system dependencies.

2.3.2 *Python.*

The problem of reproducibility is, unfortunately, not exclusive to R. Let's look at how packages and their dependencies are managed in the Python ecosystem.

The authoritative `pypi` [17] package index, used by tools such as `pip`[30], does not allow for specifying system dependencies such as C/C++ and Fortran compilers, or shared C libraries. As such, issues might occur when building self-contained native code on the end client.

`virtualenv` [31], a superset of the official `venv` [33], provides isolated development environments allowing for installing and managing dependencies independently for each project. Dependencies can be exported into a `requirements.txt` file with `pip` to recreate it later on a different host. This, however, does not consider system dependencies, or the Python version used to create the virtual environment. Moreover, the mentioned file could omit exact versions of dependencies or specify versions by constraints. Problems can arise if a dependency introduces breaking changes in a future release. The more recent [3] makes it possible to indicate the Python version but does not support system dependencies either.

`conda` [2] is an alternative package and environment manager with its own package index. It hosts several packages for data science, and it is not limited to Python. As opposed to `virtualenv`, it can manage different versions of Python as part of package dependencies. Exact versions of dependencies can be explicitly listed when exported, increasing reproducibility. This is done with `lock` files, which can pin exact versions and builds across platforms, even transitive dependencies. Furthermore, `conda` can package all system dependencies, making the package self-contained.

All those tools, when they support system dependencies, require some level of manual intervention to list the required system dependencies whereas `r4r` automatically detects those dependencies.

2.3.3 *Generic reproducibility tools.*

Binder [18] can analyze a git repository that follows the Reproducible Execution Environment Specification and generate a Docker image from it. It supports

[5] *SysReqs* https://github.com/r-hub/sysreqs, now superseded by *SystemRequirements* https://github.com/rstudio/r-system-requirements
[6] https://eddelbuettel.github.io/r2u/
```

Python, R and Julia. For R, the repository must contain a `runtime.txt` file with the date that represents the snapshot of CRAN hosted on the Posit Public Package Manager, and an `install.R` file that explicitly installs the required R packages, and so requires manual intervention.

CDE [15] uses `ptrace` to detect the files require to run the program and then to change the paths when replaying the execution. Provenance-to-use [28] uses CDE and enrich it with provenance information about the detected files and processes. Provenance-to-use is included in SciUnit [10]. SciUnit creates Reusable Research Objects and makes it possible to visualize which process created which file or spawned which process. It does not indicate from which system or R package the process or file comes from.

Reprozip [7] also uses `ptrace` to detect files and processes and creates a `rpz` archive, than can be unpacked to a Docker Image, a VM with Vagrant, a directory, or a chroot environment. It can optionally resolve Debian and RPM packages. Its Debian package resolver uses `dpkg-query` and so it much slower than the Debian package resolver of `r4r`. It does not support R packages.

## 3 Requirements of the `r4r` tool

Our tool aims to create a reproducible environment (a sandbox) for a given program execution. For an environment to be reproducible, it should satisfy two conditions: it should confine all sources of non-determinism so that the program can run unimpacted, *i.e.* producing the same result as it did while running on the host machine, and it should be transferable to other hosts.

By producing the same result, we mean that the output of the program is byte-by-byte (or line-by-line) identical when run on different hosts. The task to decide on the similarity of the output is delegated to a diff program.

One such environment is the current environment of the host machine, which, by definition, has all the dependencies the program needs. However, such an environment would not be very practical. Instead, we want to construct the smallest possible one that can still run the program and produce the same result.

There are three problems that we need to address: What are the sources of non-determinism (what can cause the program to have a different output), how can we find them (what the program needs to produce the same output), and how do we use this information to create a reproducible environment.

### 3.1 Sources of non-determinism

There are many sources of non-determinism. For example, at the hardware level, there are cache contention, CPU throttling, and random rounding. At the operating system level, there is possible non-determinism introduced by concurrency, scheduling, or memory allocation and address space layout. Finally, at the application level, there is time, randomness, environment variables, program input, configuration, libraries, and external commands.

They all can impact program results. For example, Vila *et al.* [42] has shown an observable effect of hardware perturbations in neuroimaging. In this work, we focus on the last application level. Concretely, if a given program is reproducible on the host machine, *i.e.*, multiple runs produce the same output, it should produce the same output in the sandbox.

The primary source of non-determinism is, therefore, program dependencies and environment variables. By dependencies, we mean both the program input and the transitive closure of all the resources that the program needs to run, including libraries, external commands, and configuration files that are used during the execution.

*A note on randomness.* We do not address randomness as it could interfere with the data analysis in the notebook. Instead, we leave it up to the user to re-seed the pseudorandom number generator in a controlled way. Taking into account the variability due to randomness would require the tool to decide on the statistical similarity of, for instance, numbers output by the program on several runs. This would be the task of the diff tool. Capturing the time/date of execution is also out of scope.

### 3.2 Dependency tracing

To find program dependencies, we use dynamic program analysis. Any file that is interacted with can be considered a dependency. For example, running `less notebook.Rmd` on Ubuntu 24.04 accesses eight files. Next to the binary itself and the Rmd file, the command uses a dynamic linker, which accesses its cache (`ld.so.cache`) to load two shared libraries, `libc` and `libtinfo`, to access terminal information capabilities. The `libc` accesses the compiled database of locale information (`locale-archive`) to learn how to do locale-aware formatting for the current locale. The `libtinfo` loads the terminal capabilities from a terminfo file (`xterm-256color`) for the current terminal. Finally, the `less` command also has a configuration file (`~/.less`) and a history file (`~/.lesshst`). These files belong to different categories, direct program dependencies, such as shared libraries, or configuration files. Some often do not change (*e.g.* the `xterm-256color`), but for the reproducible environment, we need to consider all of them.

Next to file dependencies, we must also capture environment variables as they affect the behavior of many programs and libraries. For example, the `LC_*` variable family defines locale-dependent behavior, including text encoding, sorting, date, and number formatting. Finally, we need to capture information about access control, *i.e.* which users and groups are related to the traced files. We use dynamic analysis rather than static, as we are not looking for an exhaustive list but for dependencies used by a concrete program execution.

### 3.3 Reproducible environment

The last problem is turning these traced dependencies into a reproducible environment. We need a way to package all the dependencies together in a transferable and runnable format on different machines.

One could use two main technologies: virtual machines and containers. Virtual machines (VMs), such as those provided by VirtualBox,[7] emulate complete hardware systems, with their CPU architecture, including operating systems and software dependencies, enabling full isolation from host environments. Container

[7]https://www.virtualbox.org/

technologies, such as Docker[8] and Singularity[9], offer a lighter-weight alternative by sharing the host's kernel and isolating only application-level dependencies.

Virtual machines provide superior long-term preservation and archiving capabilities due to their comprehensive encapsulation of both software and hardware. Conversely, containers are practical for reproducible research due to their portability, lower overhead, and ease of use across platforms.

## 4 Design and implementation

Based on the above design, we have implemented the `r4r` tool. Currently, it supports Ubuntu-based distributions – we tested it on Ubuntu 24.04. We have chosen Docker containers as the target for the reproducible environment primarily due to their popularity and ease of use. The tool is written in C++ and contains 5.5K lines of code and 1.8K of tests.

The tool is a command-line application that executes a target program (the *tracee*) while monitoring its runtime dependencies. Upon successful execution, it analyzes the resulting trace data to generate a Docker image specification and a corresponding Makefile capable of building the image and re-instantiating the program. Execution is structured as a pipeline of tasks (*cf.* Figure 2), divided into three main components. The *front-end* captures low-level information about file accesses during execution. The *middle-end* abstracts these file-level traces into higher-level, user-understandable units, such as system or language-specific packages, and encodes them into a manifest that serves as an intermediate representation. This manifest is then passed to the *back-end* that uses it to synthesize a reproducible target environment. In the following sections, we illustrate each stage of the pipeline using the example R Markdown file from Figure 1,[10] rendered and traced by `r4r` as follows:

```
$ r4r R -e 'rmarkdown::render("notebook.Rmd")'
```

4.1 Trace files

A file interaction generally happens via system calls. Therefore, to trace file usage in the front-end, we need to trace system calls. In Linux, instrumenting running processes can be done using kernel interfaces such as eBPF [36] or the `ptrace` system call.[11] eBPF is a recent and efficient mechanism that allows user-defined programs to execute within the kernel in response to various events—including system calls. It is a sandboxed virtual machine inside the kernel that allows one to safely and efficiently run custom code in response to kernel events: essentially, programmable kernel hooks without the need to do kernel modules. In contrast, `ptrace` is a longstanding system call interface traditionally used for debugging and process control used by system tracers and debuggers. It operates by intercepting system calls and allowing a tracing process to observe and manipulate the state of a target process, but with significantly higher performance costs due to extensive context switching and synchronization. While `ptrace` has a higher overhead, it is simpler to use and at the time of writing `r4r`, eBPF had a longstanding issue of disallowing reading of arbitrary long strings.[12]

Another way to trace system calls is via system call interposition. However, this approach is inherently prone to race conditions since system calls may be executed in parallel [11, 45].

When `r4r` starts, it forks the tracee and attaches `ptrace` to it. Until the tracee ceases to exist, we capture all system calls and record those related to file access. Concretely, we record paths on `openat`, `execve`, `readlinkat` and `newfstatat` system calls. Additionally, the system monitors `fork`, `vfork`, and `clone`, making sure that children's processes are traced as well.

Using `ptrace` adds a significant overhead. To mitigate it, we first use the relatively new `PTRACE_GET_SYSCALL_INFO` (introduced in Linux 5.3) API to extract information about system call parameters. Next, `ptrace` API allows one to read only one word at a time, which would be too slow for reading longer paths (effectively executing a system call for every 8 bytes of data). Instead, we use `process_vm_readv`, directly accessing the tracee memory. This has to be done with care in order not to trigger a page fault while reading past a page boundary looking for the end of the null-terminated path string.

Running the example Rmd file spawned 18 processes that together triggered over 14K system calls. Out of these, 1.5K were related to file interaction recording 539 unique files and 50 symbolic links.

There were more opened files, but we already ignored certain files while tracing (*e.g.* paths starting with `/dev`, `/proc` or `/sys`), as well as all files that are part of the target Docker base image that will be used as the base image for the environment, as long as they have the same checksum. Together, the tool ignored 178 files.

4.2 Resolve files

The file resolver in the middle-end processes the captured file paths, resolving relative paths to absolute ones and handling symbolic links. It also determines the nature of each file (e.g., system library, user file, or package resource) and categorizes it accordingly. Each file is analyzed to determine its source. Currently, we support the following sources:

- *Ubuntu Packages*: files belonging to system packages. We use the `dpkg` database to get the list of installed packages and the `dpkg` info list files to build the reverse index of the package file map, as querying `dpkg-query` is just too slow. Optionally, additional package archives (such as PPAs on Ubuntu) are detected.[13]
- *R Packages*: files associated with R libraries (on CRAN and GitHub). We use the R API, with `installed_packages`, to get the list of installed packages and their locations.
- *Copy files*: Files not resolved by any other resolver are treated as user files. They can be either inputs, in which case they are marked for direct inclusion in the environment, or outputs, in which case they will be copied from the environment upon rerun. If the file existed before, it is considered input, and if it has been created and still exists after the tracee exists, it will

[8]https://www.docker.com/
[9]https://docs.sylabs.io/guides/latest/user-guide/
[10]https://cran.r-project.org/web/packages/rmarkdown/index.html
[11]https://man7.org/linux/man-pages/man2/ptrace.2.html

[12]The effective limit was 240 bytes, *cf.* https://github.com/bpftrace/bpftrace/issues/305
[13]The detection fails if the source list file is purged before the tracing, which sometimes happens on automated setups. In that case, the user must manually add the other source lists to be copied in the Manifest.

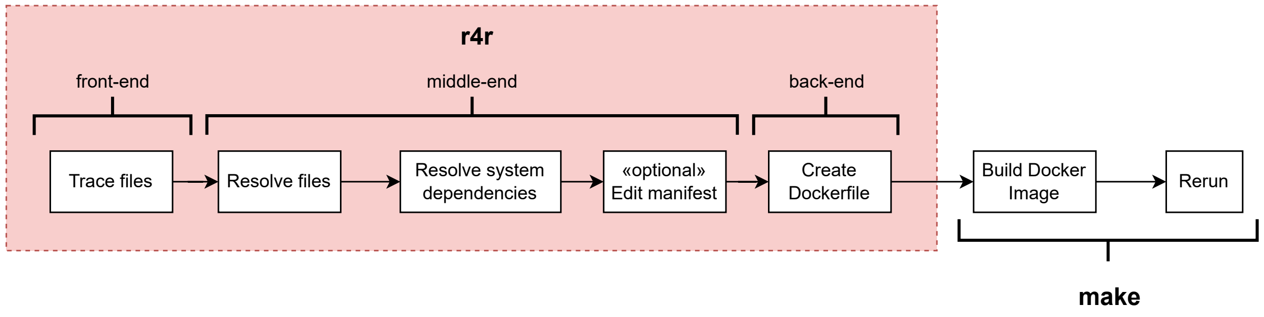


Figure 2: The `r4r` pipeline.

belong to the results file set. Both sets are configurable using command line parameters and a manifest (*cf.* Section 4.4).

Rendering the example notebook uses 129 Ubuntu packages and 25 R packages. Two files were considered as inputs, the notebook itself and the archive containing the shape file, and eight files as output, the rendered HTML page and seven files extracted from the archive.

4.3 Resolve system dependencies

R packages can contain a mixture of R code and native code (usually in C/C++ or Fortran). On most platforms (Linux distributions), they are distributed in source form and thus need to be built first at installation time. Unfortunately, there is no standardized way for a package to indicate what system dependency it needs. Worse, R packages can use arbitrary code during its build [46], through a `configure` script that runs as the first step of the package build. We could trace the build process of the packages but they usually not installed directly by the traced notebook and can be time-consuming. Fortunately, the Posit package manager[14] maintains a database of such system dependencies and provides a web-based API to query it. We query every package that needs compilation and include it in the existing set of resolved system packages.

In our example, the `sf` package depends on five additional Ubuntu packages. Note that just installing the package would not reveal this, and the user would only get a compiler error.

4.4 Edit manifest

After the file resolution is done, the tool prepares a manifest, a description of the environment that will be built. This description can be saved to a disk and edited by the user before the pipeline processes the next step. For example, the manifest created for the example notebook is shown in Figure 3 (showing just the copy section provided by the copy resolver). The manifest serves as an intermediate representation of the program dependencies.

4.5 Create Dockerfile

In the backend, the generator takes the manifest and translates it into a Dockerfile. This includes specifying the base image, copying the required files, and installing the necessary packages to replicate the runtime environment. We also create a username and all necessary groups, install the proper locale, and set the timezone.

The process is straightforward except for installing R packages. In R, there is no built-in support for version pinning. Therefore, if we want to install a specific package version, we have to do it manually. We achieve this by topologically sorting the package tree and installing packages in batches to leverage some parallelism.

Files are copied using `tar`. We first create an archive with all accessed files and symbolic links, and then we generate a script that sets proper permissions and ownership for all affected directories. If a file is allowed to be accessed because of a user membership in a particular group, the very same condition will be recovered in the sandbox.

```
# This is the manifest file generated by R4R.
copy:
  # # - ignore file.
  # C - mark file to be copied into the image.
  # R - mark file as a result file.
  C /r4r/notebook.Rmd
  R /r4r/notebook.html
  R /r4r/ne_110m_admin_0_countries.README.html
  R /r4r/ne_110m_admin_0_countries.VERSION.txt
  R /r4r/ne_110m_admin_0_countries.cpg
  R /r4r/ne_110m_admin_0_countries.dbf
  R /r4r/ne_110m_admin_0_countries.prj
  R /r4r/ne_110m_admin_0_countries.shp
  R /r4r/ne_110m_admin_0_countries.shx
  C /r4r/ne_110m_admin_0_countries.zip
```

Figure 3: Manifest for the example Rmd notebook with the copy section

4.6 Build Docker image and rerun

Next to the `Dockerfile`, we also generate a `Makefile` which contains targets for building the image, running the target program, and copying result files from the container. This is to make it easier for users to use. It simply calls `docker build`, `docker run`, and `docker cp`.

[14]https://packagemanager.posit.co/

4.7 Discussion

Copying files vs package installation. The current implementation focuses on creating a reproducible environment for scientific notebooks written in R. It has a first class support for R which is encapsulated in the R package resolver that understands how R manages its packages. It knows where packages are loaded from, how they are installed and what dependencies they have. Without the R resolver, `r4r` will simply copy the package content into the Docker image.

This is what the tool does for reproducible environment of an IPython notebook, which currently does not have first-class support. However, it presents two drawbacks. First, it complicates the inspection of artifacts and obscures high-level understanding of dependencies. Second, it creates a fragile environment that may be incomplete, limiting future experimentation. For example, in IPython, only compiled Python files (or pycache files for Python 3.12 and above) used during notebook execution are copied. If additional functions are used later, they might fail because their implementations reside in modules not included in the Docker image, as they were not initially needed.

Adding support for a new language is a matter of adding a new resolver. Concretely, this means implementing a new C++ class with one virtual method and extending the manifest with whatever information the resolver needs to recreate the state in a Docker image.

Support for other distributions and operating systems. Currently, `r4r` runs on Ubuntu Linux on x86_64. Porting it to another Linux distribution requires changing the middle-end to resolve system dependencies with another package manager. For instance, RPM-based distribution makes it possible to query the package database, which uses `sqlite`, a native implementation called `ndb`, or a Berkeley DB, with the `rpm` command. The database can also be directly accessed to perform custom requests. Porting it to other operating systems requires changing the front-end. For Windows, next to WSL (Windows Subsystem for Linux)[15] which `r4r` directly supports, we could use ETW (Event Tracing for Windows),[16] a kernel-level tracing system allowing to attach callbacks to both kernel-level and user-level events, or DLL injection to intercept file I/O operations (with the same drawbacks as system call interposition mentioned in 4.2). On MacOS, the simplest would be to use DTrace. It is similar to Linux's bpftrace, a dynamic tracing framework for both user-level and kernel-level operations. It requires disabling the System Integrity Protection, a security feature restricting the root user and system processes from modifying protected parts of the system but it is not justifiable for end users to compromise malware protection to use `r4r`. An alternative approach is either to use dynamic library interception via `DYLD_INSERT_LIBRARIES` (again, with the same problems, *cf.* 4.2) or the Endpoint Security Framework,[17] a C API for monitoring system events.

Support for other back-ends. We have chosen Docker for the back-end, but there are other options. The use of a manifest as intermediate representation makes it flexible for adding alternative back-ends. For example, instead of turning the manifest into a `Dockerfile`, we could generate a `Vagrantfile`[18] and create a complete virtual machine.

Security. Sensitive files such as SSH keys, secrets, or command line history files are copied into the Docker image if detected when tracing. The user can still edit them out of the manifest but we could show a warning if they are present, or even show them as not being copied in the manifest.

Shortcomings. Package file resolvers—such as those for Ubuntu and R—elevate the level of abstraction in environment specification by resolving dependencies based on package names and versions. However, this approach introduces a reliance on external services. For instance, an Ubuntu package may receive a security update that renders the previously specified version unavailable, or an R package might be removed from CRAN entirely. There are two primary strategies to address this limitation: (1) manually modify the manifest to relax version constraints (e.g., specifying `1.2.*` to allow updates within the `1.2` series), or (2) avoid using resolvers altogether by directly copying the necessary package files into the environment.

5 Evaluation

To check whether `r4r` correctly builds reproducible artifacts, we need to compare the outputs of several runs of the same program. Figure 4 shows the conceptual overview of how the evaluation works:

(1) For a given `program`, we first run it on the host machine where we expect it to be executed and produce a result (`output 1`).
(2) Next, we run it again on the host machine, expecting it to produce the same result as in the first run. If the `output 2` does not match `output 1` then it means that there are sources of non-determinism in the `program` execution that we cannot control. Note that if the 2 outputs do match, there could still be sources of non-determinism that could manifest for other runs.
(3) The third invocation executes the program with the `r4r` tool that tracks the `program` dependencies. The result of this run (`output 3`) should be the same as `output 2`, *i.e.* `r4r` should not alter the `program` behavior in any way. If it is not, it means that `r4r` impacted the run.
(4) Finally, we run the `program` in the guest environment, *i.e.* the environment created by the tool. If the `output 4` is different from `output 3` then the tool created an environment that does not account for all sources of non-determinism. Otherwise, the tool succeeded and we have a reproducible environment.

[15]https://learn.microsoft.com/en-us/windows/wsl/about
[16]https://learn.microsoft.com/en-us/windows/win32/etw/about-event-tracing
[17]https://developer.apple.com/documentation/endpointsecurity
[18]https://www.vagrantup.com/

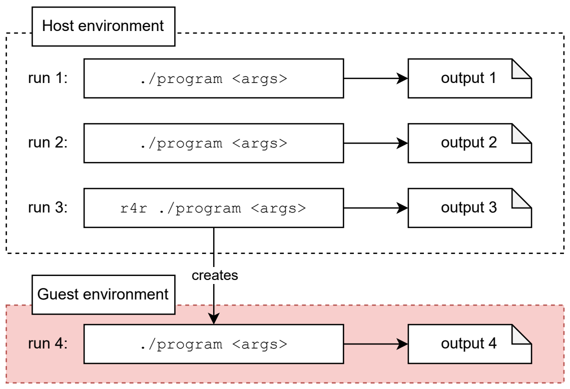


Figure 4: Overview of the `r4r` evaluation.

5.1 Experimental setup

The Kaggle dataset. Kaggle[19] is a competitive data science platform. Users can participate to competition or just host their notebooks there and use the compute offered by the platform. Kaggle allows users to use Python, R or Julia.

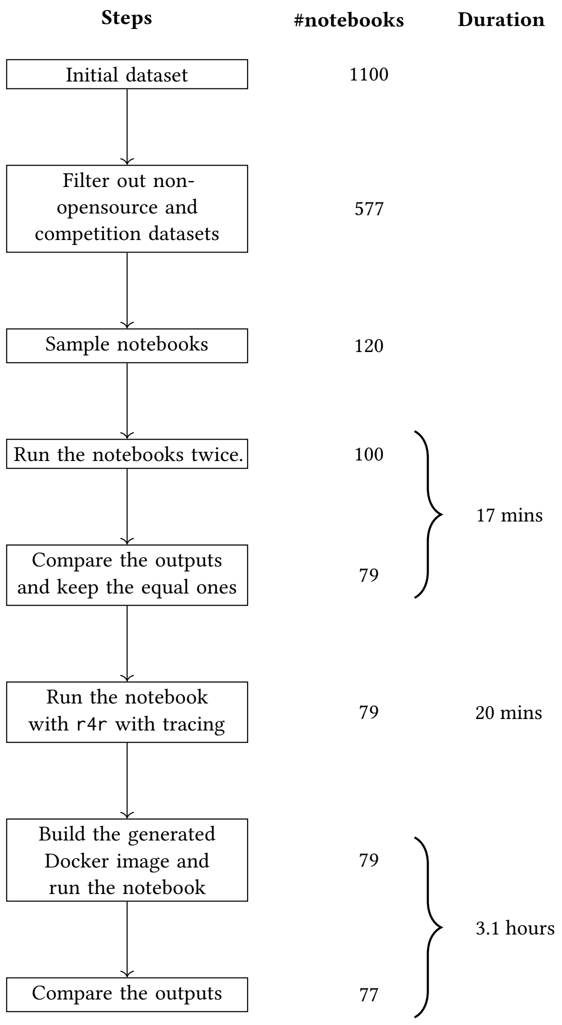
Using the Kaggle public API, we retrieve 1100 R Markdown notebooks and keep the ones that use datasets with open source licenses,[20] and which are not linked to competitions, as competitions required manual acceptance of the rules on the website. After that filtering step, we are left with 523 notebooks.

We randomly sample 120 notebooks among those notebooks for further experiments. All of them render to self-contained HTML documents,[21] where images are base64-encoded and set in the `src` attribute of `<img>` tags.

Running the notebooks. Notebooks on Kaggle expect a particular folder hierarchy: the notebook itself is located in `/kaggle/code/`, while the inputs are in `/kaggle/input/`. Kaggle also provides a directory to store persistent data between runs in `/kaggle/working/`. Besides, the notebooks we collected do not come with their R and system dependencies. To approximate those, we used a custom tool to parse the 120 R Markdown files to get package loading statements (`library(package)`), yielding 118 packages from CRAN and GitHub, and then `pak::pkg_sysreqs` to get the system dependencies.[22] We build a Docker image based on Ubuntu 24.04 with the correct folder structure and dependencies for the 120 notebooks.

Hardware. We run the experiments on an Ubuntu 24.04 server, with 189 GB of RAM and 72 cores.

Experiment pipeline. Figure 5 shows the successive steps of the experiment pipeline, with the durations of the non-negligible steps. After sampling the notebooks, we run them twice, with a 15-minute timeout, filter out the ones that do not complete execution, and then compare their outputs. The ones which do not have the same



Steps
#notebooks
Duration
Initial dataset
1100
Filter out non-opensource and competition datasets
577
Sample notebooks
120
Run the notebooks twice.
100
Compare the outputs and keep the equal ones
79
17 mins
Run the notebook with r4r with tracing
79
20 mins
Build the generated Docker image and run the notebook
79
Compare the outputs
77
3.1 hours

Figure 5: Experiment pipeline. Numbers on the right indicate how many notebooks are left after each step. After a run step, we remove notebooks that failed to execute to the end and produce an output.

outputs are *non-deterministic*. The remaining ones are said to be *deterministic*, at least observably on those runs. We only keep the *deterministic* ones to be traced with `r4r` and then we generate the Docker image with `make` and run the notebook a third time in that Docker image. We then compare the output of the execution of the notebook in the Docker image with the output of the non-traced notebook. The running and tracing steps, and the steps to generate the Docker image and run the notebook in it, are performed in parallel with `parallel` [37] using 25 cores.

Artifact. We make an artifact available with the data and the scripts to rerun the pipeline along with the submission.

5.2 Running and tracing errors.

Just running the 120 notebooks without `r4r` results in 20 erroring notebooks, including 2 timing-out notebooks. For 8 notebooks,

[19]https://www.kaggle.com/
[20]On Kaggle, notebooks themselves are licensed under the Apache 2 license.
[21]Kaggle makes it nearly impossible to output anything else than html.
[22]pak is a R package manager. It can look for system dependencies in an online database at https://github.com/rstudio/r-system-requirements

errors are due to missing input files, because they refer to datasets that were removed from Kaggle or whose paths were updated after the notebook creation. The remaining 10 notebooks have errors due to deprecated libraries (for instance, the `scales` library), duplicate chunk labels, or even unbalanced chunk fences. `r4r` does not add more errors: notebooks which successfully run without `r4r` do not fail when traced with `r4r`.

5.3 Reproducibility

Exact reproducibility. We compare the HTML outputs of the notebooks line by line, with the `diff` utility [22] in *unified* diff mode. If there are no differences, we say the notebooks are *exactly* reproducible. 97.5% of the artifacts `r4r` creates from the *deterministic* notebooks successfully reproduce, which represent 79% of the notebooks that successfully completed execution. 2 notebooks do not have the same outputs as the non-`r4r`-package version: one explicitly triggers the garbage collector at the beginning of the notebook after removing all bindings, which prints the used and cleaned memory of R, and that differs by a few MB across runs. Removing the call to `gc()` makes it reproducible. The other one fits a statistical model which triggers errors and does not show the errors exactly the same way.

Approximate reproducibility. To get a better understanding of the differences in notebooks with different outputs, including both *non-deterministic* notebooks and the 2 ones that fail to reproduce with `r4r`, we process the differences we get from `diff` to detect cases involving dates, images, or numbers. Images with differences come from plotting instructions. Manual visual inspections suggested that images are actually very similar, or that the numbers are close.

- Images. We compute the structural similarity [44] (SSIM) between the 2 images: 1 means that the images are the same, and −1 that they are totally different. Across all the notebooks, there are 34 changes involving images. None of them has a SSIM lower than 0.99, suggesting that the differences are not visually perceptible.
- Numbers. We detect when changes only touch numbers and then compare the numbers. Only 4 have such changes. Two notebooks perform a parallel model estimation which prints progress numbers. The interleaving of the progress messages is not the same on the outputs. One notebook performs sampling and exhibits only one difference, in which numbers only differ by less than 0.1. The last notebook is the one that performs explicit garbage collection and has only one number differing by more than 0.1.
- Dates. We use a regular expression to detect changes with only dates. The differences only appear if we run the various steps of the pipeline across at least 2 days.
- Other. We compute the Levenshtein distance on characters between the 2 versions using the R package `stringdist` [41]. The mean distance is 222.6 and the median 93, suggesting that each of the changes is large. Only 6 notebooks have such changes. It includes the notebook that triggers errors not shown the same way.

Overall, the notebooks that were not *deterministic* have little differences in their output, with 73.7% of the non-reproducible ones with fewer than 10 differences. Most changes touch only a small part of the output: differences between the outputs are more than 1% for only 3 notebooks, as shown on Figure 6.

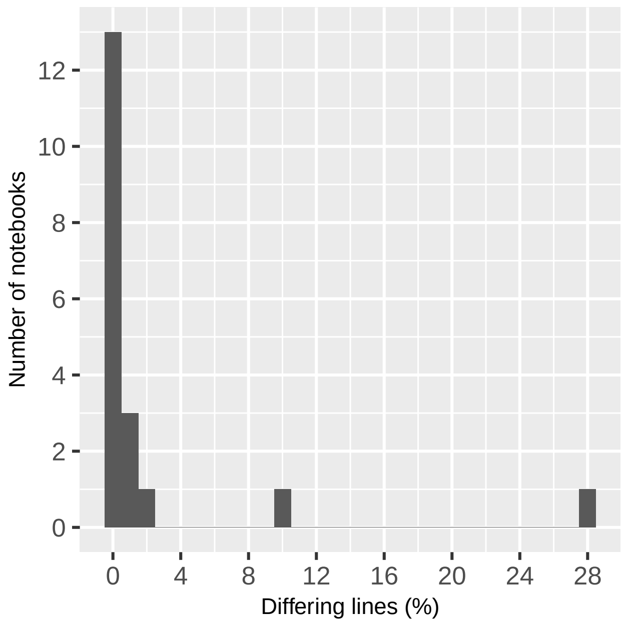


Figure 6: Percent of number of lines changed (n_1^c, n_2^c) over the number of lines (n_1, n_2) of the outputs, more formally, $\frac{2\max(n_1^c, n_2^c)}{n_1+n_2} \times 100$

For numeric changes, we say that two outputs are approximately reproducible if given the same inputs, their differences are no larger than some user-provided threshold. Indeed, the 0.1 threshold for numeric changes is arbitrary but gives an intuition. Overall, with the reproducible deterministic notebooks, the 10 notebooks with only changes in images, and taking into account the one notebook with numeric changes less than 0.1, at least 89% of the notebooks are approximately reproducible.

5.4 Complexity of reproducibility artifacts

To estimate how much effort creating the Docker image would have been if done *manually*, in Table 1, we show the number of deb packages, R packages from CRAN and from GitHub, and files to copy that we need to specify to get a reproducible environment with the exact same package versions. This means that all dependencies must be explicitly specified with their version.

Even the minimum of Ubuntu packages, 59, or CRAN packages, 26, is already quite high. For each CRAN package, a user manually creating the Dockerfile would have to check which system dependencies it requires and add them to the Dockerfile. The files to copy include the notebook itself, all the data inputs it needs, but also system files that could not be resolved to Deb or R packages.

5.5 Performance and size

`r4r` makes running a notebook 4.7 times slower, on average, on the successfully traced notebooks. More than half of the notebooks

	mean	median	min	max
files	30.8	24	20	178
deb	101.1	90	59	151
cran	227.1	113	26	496
github	1	0	0	4

Table 1: Mean, median, min, and max of files to copy (files), deb packages (deb), CRAN packages (cran), GithHub packages (github) per package, among those that do not error when generating the Dockerfile.

even have a smaller slowdown, as the median slowdown is 3.2 x. Part of the slowdown is due to `r4r` resolving packages and creating an archive of the input files. The bigger the input data, the slower it is. Another part is the overhead due to `ptrace`: tracing a notebook execution is only 1.4 slower than just running it. The maximum slowdown is 26 but the tracing slowdown for this notebook is 1.91; the notebook uses 12 large datasets that have to be added into the archive.

Building the generated images can take a lot of time, on average, 32.3 mins per notebook. Most of that time is spent building the required R packages, which we already decreased by parallelizing the installation of R packages (see Section 4.5). The longest image build time is 75.6 mins.

The uncompressed Docker images are rather minimal: the mean size is 2.57 GB and the median size, 1.64 GB. The largest one is 6.06 GB. As a comparison, the Docker image used by Kaggle[23], which provides a general-purpose R image with RStudio, all CRAN packages, and some more utilities, is 38 GB uncompressed.

5.6 Limitations

The filtering step to remove *non-deterministic* notebooks assumes that if a notebook is *non-deterministic*, then it will have a different output at each run. This might not be always the case. In practice, we detected the same *non-deterministic* notebooks when running the experiment pipeline several times. It suggests that the other notebooks have sources of non-determinism that do not depend on the dependencies but rather at the hardware layer (as discussed in Section 3.1). This is the case for the notebook that explicitly calls the garbage collector (*cf.* Section 5.3).

For the evaluation, we use a dataset of notebooks from Kaggle and exclude the non-competition and non-open source ones. Although that dataset might not be representative of R Markdown notebooks in general, they use a large number of diverse packages and of system dependencies, which was already enough to discover bugs hindering reproducibility during the development of `r4r`. The R packages used in the dataset, such as `dplyr` [47] or `tidymodels` [21] from the `tidyverse` set of packages, are packages commonly used in R.

[23] gcr.io/kaggle-gpu-images/rstats:latest

6 Conclusion

Reproducibility is a cornerstone of scientific progress. However, recreating the execution environment necessary to reproduce computational experiments remains a significant challenge, particularly for researchers lacking expertise in system administration. To address this issue, we developed `r4r`, a tool that automates environment reconstruction. `r4r` employs dynamic program analysis to trace all runtime dependencies and assembles a self-contained, user-inspectable, environment in which the program can execute without external interference. Currently, r4r targets Ubuntu-based Linux distributions and produces Docker images as the reproducible environment. While it provides first-class support for the R programming language, it also includes a language-agnostic fallback mechanism based on file system tracing, enabling broader applicability. In an evaluation on a dataset of R Markdown notebooks from Kaggle, r4r successfully reproduced 97.5% of deterministic notebooks.

Future work. The `r4r` tool is designed with extensibility in mind. We plan to expand the front-end to support additional Linux distributions, and additional operating systems beyond Linux. In the middle-end, our next goal is to introduce first-class support for the Python programming language. On the back-end, in addition to generating Docker images, we aim to support Vagrant to enable the creation of full virtual machines, which would be particularly useful for long-term archival of computational environments.

Another direction involves improving the tracing mechanism on Linux. Specifically, we plan to replace `ptrace` with eBPF, which now supports reading arbitrary string arguments from system calls. This change is expected to significantly reduce tracing overhead, especially for data analysis workflows with intensive I/O.

Finally, we aim to improve the overall user experience. Currently, `r4r` is a command-line tool, which may pose a barrier for users unfamiliar with terminal-based workflows. To make the tool more accessible, we plan to develop a plugin for RStudio Desktop,[24] a widely used integrated development environment among data scientists.

Acknowledgments

We thank Petr Adamek for developing a first prototype of `r4r`. We thank the reviewers for their valuable feedback and suggestions that helped improve the paper. The research was supported by Czech Science Foundation Grant No. 23-07580X.

References

[1] ACM. 2020. Artifact Review and Badging - Current. https://www.acm.org/publications/policies/artifact-review-and-badging-current accessed 2025-01-14.

[2] Continuum Analytics. 2012. *Conda project.* https://docs.conda.io/en/latest/

[3] Astral. 2025. *uv project.* https://docs.astral.sh/uv/

[4] M Baker. 2016. 1,500 scientists lift. *Nature* 533 (2016), 452–454.

[5] Emery D Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the impact of programming languages on code quality: A reproduction study. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 4 (2019), 1–24.

[6] Carl Boettiger and Dirk Eddelbuettel. 2017. An Introduction to Rocker: Docker Containers for R. *The R Journal* 9, 2 (2017), 527–536. doi:10.32614/RJ-2017-065

[7] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. 2016. ReproZip: Computational Reproducibility With Ease. In *Proceedings of the 2016*

[24] https://posit.co/download/rstudio-desktop/

International Conference on Management of Data (San Francisco, California, USA) *(SIGMOD '16)*. ACM, 2085–2088. doi:10.1145/2882903.2899401

[8] Andrie de Vries. 2024. *miniCRAN: Create a Mini Version of CRAN Containing Only Selected Packages.* https://github.com/andrie/miniCRAN R package version 0.3.0.9000.

[9] Colin Fay, Vincent Guyader, Josiah Parry, and Sébastien Rochette. 2022. *dockerfiler: Easy Dockerfile Creation from R.* https://github.com/ThinkR-open/dockerfiler R package version 0.1.5.0002.

[10] Gabriel Fils, Zhihao Yuan, Tanu Malik, et al. 2017. Sciunits: Reusable research objects. In *2017 IEEE 13th International Conference on e-Science (e-Science)*. IEEE, 374–383.

[11] Tal Garfinkel. 2003. Traps and pitfalls: Practical problems in system call interposition based security tools. In *In Proc. Network and Distributed Systems Security Symposium*.

[12] Federico M Giorgi, Carmine Ceraolo, and Daniele Mercatelli. 2022. The R language: an engine for bioinformatics and data science. *Life* 12, 5 (2022), 648.

[13] Steven N Goodman, Daniele Fanelli, and John PA Ioannidis. 2016. What does research reproducibility mean? *Science translational medicine* 8, 341 (2016), 341ps12–341ps12.

[14] John Gruber. 2004. Markdown. https://daringfireball.net/projects/markdown/. Accessed: 2025-04-02.

[15] Philip J Guo and Dawson Engler. 2011. {CDE}: Using system call interposition to automatically create portable software packages. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*.

[16] Chung hong Chan and David Schoch. 2023. rang: Reconstructing reproducible R computational environments. *PLOS ONE* (2023). doi:10.1371/journal.pone.0286761

[17] Richard Jones. 2003. *The Python Package Index.* https://pypi.org/

[18] Jupyter. 2025. *uv project.* https://mybinder.readthedocs.io

[19] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks—a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. IOS Press, 87–90. doi:10.3233/978-1-61499-649-1-87

[20] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111. doi:10.1093/comjnl/27.2.97

[21] Max Kuhn and Hadley Wickham. 2020. *Tidymodels: a collection of packages for modeling and machine learning using tidyverse principles.* https://www.tidymodels.org

[22] David MacKenzie, Paul Eggert, and Richard Stallman. 2002. Comparing and Merging Files with GNU diff and patch. *Network Theory Ltd* 4 (2002), 23–25.

[23] Eliot J B McIntire. 2024. *Require: Installing and Loading R Packages for Reproducible Workflows.* https://Require.predictiveecology.org R package version 1.0.1, https://github.com/PredictiveEcology/Require.

[24] V. P. Nagraj and Stephen D. Turner. 2023. pracpac: Practical R Packaging with Docker. arXiv:2303.07876 [q-bio.QM] https://arxiv.org/abs/2303.07876

[25] Daniel Nüst, Dirk Eddelbuettel, Dom Bennett, Robrecht Cannoodt, Dav Clark, Gergely Daróczi, Mark Edmondson, Colin Fay, Ellis Hughes, Lars Kjeldgaard, Sean Lopp, Ben Marwick, Heather Nolis, Jacqueline Nolis, Hong Ooi, Karthik Ram, Noam Ross, Lori Shepherd, Péter Sólymos, Tyson Lee Swetnam, Nitesh Turaga, Charlotte Van Petegem, Jason Williams, Craig Willis, and Nan Xiao. 2020. The Rockerverse: Packages and Applications for Containerisation with R. *The R Journal* 12, 1 (2020), 437–461. doi:10.32614/RJ-2020-007

[26] Daniel Nüst and Matthias Hinz. 2019. containerit: Generating Dockerfiles for reproducible research with R . *Journal of Open Source Software* 4, 40 (8 2019), 1603. doi:10.21105/joss.01603

[27] National Academies of Sciences, Medicine, Policy, Global Affairs, Board on Research Data, Information, Division on Engineering, Physical Sciences, Committee on Applied, Theoretical Statistics, et al. 2019. *Reproducibility and replicability in science.* National Academies Press.

[28] Quan Pham, Tanu Malik, and Ian Foster. 2013. Using provenance for repeatability. In *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13)*.

[29] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*. IEEE, 507–517.

[30] The pip developers. 2008. *pip project.* https://pypi.org/project/pip/

[31] PyPA. 2007. *virtualenv project.* https://virtualenv.pypa.io/en/latest/

[32] Bruno Rodrigues and Philipp Baumann. 2025. *rix: Reproducible Data Science Environments with 'Nix'.* https://docs.ropensci.org/rix/ R package version 0.14.3.

[33] Vinay Sajip. 2012. *venv project.* https://docs.python.org/3/library/venv.html

[34] Sheeba Samuel and Daniel Mietchen. 2024. Computational reproducibility of Jupyter notebooks from biomedical publications. *GigaScience* 13 (2024), giad113.

[35] Uri Simonsohn and Hugo Gruson. 2024. *groundhog: Version-Control for CRAN, GitHub, and GitLab Packages.* https://groundhogr.com R package version 3.2.1.

[36] Gavin D. Stark and Brendan Gregg. 2019. BPF Performance Tools: Linux System and Application Observability. In *USENIX Annual Technical Conference*. USENIX. O'Reilly Media, Inc..

[37] Ole Tange. 2023. *GNU Parallel 2023.* Ole Tange. doi:10.5281/zenodo.10199085

[38] Ana Trisovic, Matthew K Lau, Thomas Pasquier, and Mercè Crosas. 2022. A large-scale study on research code quality and execution. *Scientific Data* 9, 1 (2022), 60.

[39] Dana Udwin and Ben Baumer. 2015. R Markdown. *Wiley Interdisciplinary Reviews: Computational Statistics* 7 (01 2015). doi:10.1002/wics.1348

[40] Kevin Ushey and Hadley Wickham. 2024. renv: Project Environments. https://rstudio.github.io/renv/ R package version 1.0.11, https://github.com/rstudio/renv.

[41] Mark P.J. van der Loo. 2014. The stringdist Package for Approximate String Matching. *The R Journal* 6, 1 (2014), 111–122. doi:10.32614/RJ-2014-011

[42] Gael Vila, Emmanuel Medernach, Ines Gonzalez Pepe, Axel Bonnet, Yohan Chatelain, Michael Sdika, Tristan Glatard, and Sorina Camarasu Pop. 2024. The Impact of Hardware Variability on Applications Packaged with Docker and Guix: a Case Study in Neuroimaging. In *Proceedings of the 2nd ACM Conference on Reproducibility and Replicability* (Rennes, France) *(ACM REP '24)*. Association for Computing Machinery, New York, NY, USA, 75–84. doi:10.1145/3641525.3663626

[43] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. 2020. Restoring reproducibility of Jupyter notebooks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 288–289. doi:10.1145/3377812.3390803

[44] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.

[45] Robert N. M. Watson. 2007. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *Proceedings of the First USENIX Workshop on Offensive Technologies (WOOT'07)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/woot-07/exploiting-concurrency-vulnerabilities-system-call-wrappers

[46] Hadley Wickham and Jennifer Bryan. 2023. *R packages.* " O'Reilly Media, Inc.".

[47] Hadley Wickham, Romain François, Lionel Henry, Kirill Müller, and Davis Vaughan. 2023. *dplyr: A Grammar of Data Manipulation.* https://dplyr.tidyverse.org R package version 1.1.4, https://github.com/tidyverse/dplyr.