Computational Reproducibility of R Code Supplements on OSF

Lorraine Saju, Tobias Holtdirk,
Meetkumar Pravinbhai Mangroliya, Arnim Bleier

GESIS – Leibniz Institute for the Social Sciences
Cologne, Germany
{lorraine.saju, tobias.holtdirk, meetkumar.mangroliya, arnim.bleier}@gesis.org

Abstract

Computational reproducibility is fundamental to scientific research, yet many published code supplements lack the necessary documentation to recreate their computational environments. While researchers increasingly share code alongside publications, the actual reproducibility of these materials remains poorly understood.

In this work, we assess the computational reproducibility of 296 R projects using the *StatCodeSearch* dataset. Of these, only 264 were still retrievable, and 98.8% lacked formal dependency descriptions required for successful execution. To address this, we developed an automated pipeline that reconstructs computational environments directly from project source code. Applying this pipeline, we executed the R scripts within custom Docker containers and found that 25.87% completed successfully without error.

We conducted a detailed analysis of execution failures, identifying reproducibility barriers such as undeclared dependencies, invalid file paths, and system-level issues. Our findings show that automated dependency inference and containerisation can support scalable verification of computational reproducibility and help identify practical obstacles to code reuse and transparency in scientific research.

Introduction

Ensuring computational reproducibility is increasingly recognized as a cornerstone of credible scientific research (Peng 2011; Seibold et al. 2021; National Academies of Sciences and Medicine 2019). Several works, beginning in the 1990s (Claerbout and Karrenbach 1992), have highlighted the importance of making research outputs reproducible and have proposed computational methodologies and best practices to achieve this goal.

A fundamental aspect in achieving computational reproducibility is the provision of all necessary materials, including *the raw data and the code* used to generate the results. Barba (2018) paraphrases Claerbout and Karrenbach (1992) with what is now widely considered the ideal of computationally reproducible research.

> An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.

Following this goal, Chung-hong Chan, Tim Schatto-Eckrodt, and Johannes Gruber (2024), among others, emphasise that sharing of all code and data is crucial for transparency and forms the basis for computational repro-

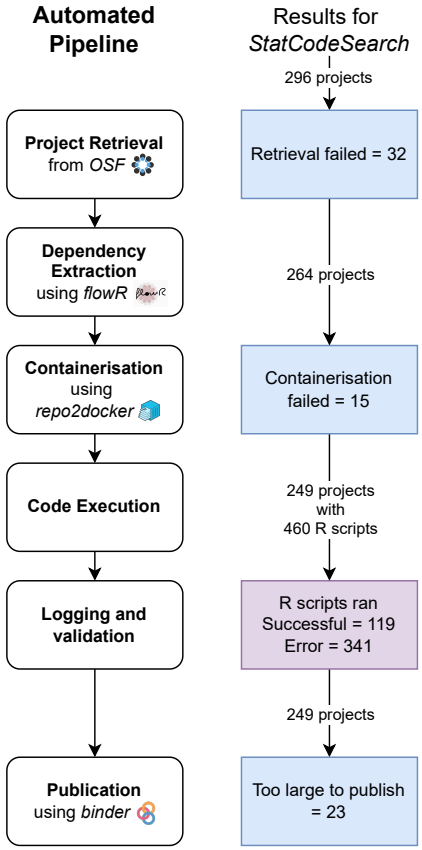

Figure 1: Our completely automated pipeline `osf-to-binder` (left) and the analysis results using this pipeline for the *StatCodeSearch* dataset (right).

ducibility. To facilitate the execution of their shared code, researchers must pay close attention to documenting their *computational environment* thoroughly. This includes explicitly listing all software dependencies and their versions, using files like `requirements.txt` or including detailed information about their used environment using commands such as `sessionInfo()` in R. Containerisation technologies such as Docker, which use a `Dockerfile` to create consistent and isolated environments across different systems (e.g., different operating systems), are another widely accepted approach in the literature on computational reproducibility (Boettiger 2015). Furthermore, Schoch et al. (2024) emphasise that external dependencies such as online *APIs* are also part of the potentially changing environment, which can undermine computational reproducibility. Achieving reproducibility requires a multifaceted, proactive approach that includes transparent sharing of materials and thorough documentation of the computational environment by the authors. Sandve et al. (2013) and Kohrs et al. (2023) condense these requirements into basic rules for reproducible computational research.

Achieving computational reproducibility should not be assumed; it requires external verification. Hardwicke et al. (2020) manually examined 250 articles from the social science literature and found that fewer than 3% made their analysis scripts available. Rainey et al. (2025) found that only about 12% of quantitative research articles provided access to both the data and the code. Trisovic et al. (2022) executed R code from replication datasets hosted on the Harvard Dataverse repository in a clean runtime environment and found that 74% of R files failed to complete without error. Pimentel et al. (2019) and Samuel and Mietchen (2024) examined the reproducibility of Jupyter notebooks, mostly written in Python, and found that only 24% and 11.6% respectively ran without errors in a fully automated analysis. Chung-hong Chan, Tim Schatto-Eckrodt, and Johannes Gruber (2024) tested the reproducibility of 30 papers and found that, even after manual restoration of the code, at least 20% were only partially reproducible. Furthermore, both practitioners (Lasser 2020; Nüst and Eglen 2021) and guides (Arnold et al. 2019; Bleier 2025) emphasise the role of services like MyBinder in enabling authors to share their analysis scripts in a way that allows for easy verification by others.

In this work, we empirically test the computational reproducibility of 296 R code supplements published as projects on the Open Science Framework (OSF) repository. However, unlike earlier approaches (Trisovic et al. 2022) that used a clean runtime or manual intervention (Chung-hong Chan, Tim Schatto-Eckrodt, and Johannes Gruber 2024) to establish reproducibility, we apply an automated approach to infer and extract dependencies that are necessary for a successful execution. Our work is guided by the following research questions: At what rate are we able to verify the computational reproducibility of the submissions published on OSF? Can automatic dependency inference aid in successful re-execution? Can a statistical analysis of replication failure modes inform recommendations on best practices for the publication of code supplements, and if so, what are these best practices?

Methodology

The starting point for this study was the **StatCodeSearch dataset**, which is part of the GenCodeSearchNet benchmark suite (Diera et al. 2023). This dataset, available on HuggingFace[1], consists of code-comment pairs extracted from R scripts hosted on the Open Science Framework (OSF)[2]. It focuses specifically on R projects in the social sciences and psychology, particularly those involving statistical analysis.

The dataset contains 1,070 code-comment pairs drawn from 558 unique R scripts across 296 distinct OSF projects. While the dataset is organized at the level of individual code-comment pairs, our goal is to reconstruct interactive, reproducible computational environments at the project level.

To achieve this, we used the project identifiers provided in the dataset to retrieve the corresponding research materials. We then employed the OSFClient API[3] to download the full contents of each associated OSF repository.

An initial verification step revealed that, out of the 558 R code files referenced across 296 OSF projects in the `StatCodeSearch` dataset, 63 files from 32 distinct projects were no longer accessible through their original OSF directories. This outcome suggests that a portion of the dataset had become outdated, likely due to file deletions, renaming, or changes to project access permissions on the OSF platform following the initial data collection.

While OSF supports the creation of immutable, time-stamped project snapshots through its registration feature, our analysis found that only 58 out of 296 projects had used registrations, and only 49 of those preserved the files referenced in the dataset. Moreover, registered snapshots are not automatically created or mandatory, and their selective use makes it difficult to systematically recover the original state of all materials. The lack of widespread adoption of OSF registrations and the absence of robust version control systems (such as those provided by Git) make it challenging to replicate the computational environment used in these studies at the time of publication.

Following the identification and removal of unresolvable file references from the initial dataset, the remaining 264 projects were examined for files that could support the reproduction of the original analyses. The downloaded project contents were systematically searched for reproducibility-relevant files that document the computational R environment, including `renv.lock`, `sessionInfo.txt`, `sessionInfo.RData`, `.Rprofile`, `DESCRIPTION`, `dependencies.R`, `dependency.R`, `Dockerfile`, `environment.yml`, and `install.R`.

To enable the automated execution and validation of project files associated with the GenCodeSearchNet dataset, we developed an automated pipeline, `osf-to-binder`[4], which is publicly available on GitHub. The goal of this pipeline is to generate verifiably reproducible computational environments directly from the source code of scientific publications hosted on OSF.

[1] https://huggingface.co/datasets/drndr/statcodesearch

[2] https://osf.io

[3] https://github.com/osfclient/osfclient

[4] https://github.com/Code-Inspect/osf-to-binder

The `osf-to-binder` pipeline operates through the following steps (see Figure 1):

- **Project Retrieval**: Given one or more OSF project identifiers, the pipeline automatically downloads and unpacks the entire file storage associated with each project.
- **Dependency Extraction**: For projects containing R scripts, the pipeline employs `flowR` (Sihler and Tichy 2024), a static dataflow analyser and program slicer to automatically extract dependencies.
- **Docker Configuration**: The extracted R dependencies are used to generate a `DESCRIPTION` file, an R package metadata file that is essential for specifying dependencies in Docker-based environments.
- **Containerisation**: Using `repo2docker` (Forde et al. 2018), the pipeline builds a Docker container based on the project directory. It scans the repository for standard configuration files (e.g., `DESCRIPTION`) and creates a runnable Docker image accordingly.
- **Code Execution**: Within the built container, the pipeline executes all identified R scripts in a fully isolated and dependency-managed environment.
- **Logging and open validation**: Execution results and logs are recorded to ensure transparency and support both internal and external validation.
- **Publication**: To support open reproducibility, the resulting Docker image is published to a container registry (DockerHub)[5], and the project code is made available via a version control system (GitHub)[6]. Additionally, we generate a MyBinder (Ragan-Kelley et al. 2018) launch link, enabling users to run the environment in a remote RStudio instance without any local setup.

By automating dependency extraction, environment configuration, containerisation, and execution, the `osf-to-binder` pipeline offers a scalable and transparent approach to enhancing computational reproducibility for OSF-hosted research projects. It thereby supports broader efforts toward open and verifiable science.

Results

An analysis of the remaining 264 OSF projects, identified after excluding those for which the referenced R scripts could not be located during initial verification, revealed a limited presence of files commonly associated with computational reproducibility. The results are detailed in Table 1.

These findings highlight the current state of explicit reproducibility provisions within the examined subset of OSF R Projects. The scarcity of these files suggests that many projects may lack readily available instructions or specifications for recreating computational environments.

[5]All Docker images are published at https://hub.docker.com/u/meet261.

[6]All GitHub repositories are maintained under https://github.com/code-inspect-binder.

Table 1: Presence of reproducibility-related files in the 264 analysed OSF projects.

Dependency File	Projects	
	Total	Percentage
`DESCRIPTION`	2	0.8%
`Dockerfile`	1	0.4%
`renv.lock`	0	0.0%
`sessionInfo.txt`	0	0.0%
`sessionInfo.RData`	0	0.0%
`.Rprofile`	0	0.0%
`dependencies.R`	0	0.0%
`dependency.R`	0	0.0%
`environment.yml`	0	0.0%
`install.R`	0	0.0%
No dependency file	261	98.8%

Containerisation success and failures

Of the 264 projects processed by the pipeline, containerisation failed for 15, comprising 35 R scripts (around 5% of the total). The main reasons included:

- **Malformed or incomplete `DESCRIPTION` files:** Generated from `flowR` outputs, these often lacked required fields or had formatting errors, rendering them invalid.
- **Incorrectly extracted dependencies:** In several cases, `flowR` misidentified variables, file paths, or internal objects as package names, resulting in invalid entries such as `unknown`, `NULL`, or numeric values like `'0'`, `'1'`.
- **Invalid scoped package references:** Calls like `knitr::opts_chunk` were mistakenly treated as standalone packages, which are not installable.
- **Unavailable or incompatible packages:** Some projects listed packages such as `crsh/papaja`, `DF_network`, and `swfscMisc`, which were either not available for the R version in the container or required system-level libraries that were not included.
- **Failed package installation:** These issues caused `devtools::install_local(getwd())` to fail, stopping container creation during the `repo2docker` build.

The remaining 249 projects were successfully containerised, yielding 460 R scripts, and demonstrating the pipeline's ability to automatically generate and build Docker images for a majority of the analysed OSF R projects. A key constraint during the publication step was GitHub's 100MB per-file size limit, which prevented 23 of these projects (51 scripts) from being pushed to the repository. While this limited their accessibility via Git-based platforms such as Binder, the scripts were still containerised and included in the execution analysis, maintaining the total number of executed scripts at 460.

Code execution within containerised environments

Following successful containerisation, 460 R scripts from 249 OSF projects were executed. Of these, 119

scripts (25.87%) completed successfully without critical errors, while the remaining 341 scripts (74.13%) failed—highlighting persistent challenges in computational reproducibility. Among the successful scripts, 51 came from 40 projects (16.06% of all 249 projects) in which all scripts executed without failure, indicating full project-level reproducibility. The other 68 successful scripts were from 34 projects (13.65%) that also included at least one failed script, reflecting partial reproducibility. The remaining 175 projects (70.28%) had no successfully executed scripts.

To analyse and interpret script execution failures, a two-level classification approach was implemented. At the first level, regular expression patterns were used to extract common error messages and map them to initial error types, such as missing objects, function call issues, invalid paths, or package loading problems. In the second level, semantically similar errors were grouped into broader categories.

Errors such as "object not found," "could not find function," and "unexported object access" were grouped under *Missing Object or Function* (18.2%), which also includes references to undefined variables or function calls made without properly loaded packages. Errors involving non-existent file or folder paths, hardcoded directories, or failed attempts to change the working directory were classified as *Invalid File or Directory Path* (19.1%). Typical cases include "cannot open file," "failed to search directory," or "directory already exists," reflecting file system access issues. Many errors stemmed from scripts referencing missing datasets or using absolute paths, both of which hinder reproducibility in containerised or remote environments.

Package-related issues were among the most frequent failure sources, with *Missing Package* errors accounting for 26.1% of failed scripts. This category includes unmet dependencies due to missing, outdated, or deprecated packages, or libraries implicitly required but not declared. Separately, errors during installation, such as broken dependencies, compilation failures, or loading issues, were categorized as *Package Installation Failure* (8.2%), often signaled by messages like unable to install packages," lazy loading failed," or "package or namespace load failed."

System-level problems related to shared object files or display devices were categorized under *Shared Library Load Error* (8.5%). These include failures such as "unable to load shared object" or GUI-dependent functions failing in headless environments (e.g., "unable to start data viewer"). Direct file access errors, such as failures when trying to read a file, were grouped under *File Read Error* (7.9%). These typically manifested as errors in reading `.rds`, `.csv`, or other files due to missing or incorrect paths.

The *Other Errors* category, accounting for 12% of all failed scripts, includes less frequent but collectively significant error types. A substantial portion involved *RStudio Environment Errors* and *Compressed File Not Found* errors, often caused by assumptions about interactive sessions (e.g., active RStudio) or attempts to load missing `.rds` or compressed files. Less common issues included *Syntax or Argument Errors*, *Encoding and String Handling Problems*, *Missing Arguments in* `setwd()`, and *Data Structure Mismatches*. Though individually rare, these represent a long tail of reproducibility challenges in real-world R scripts.

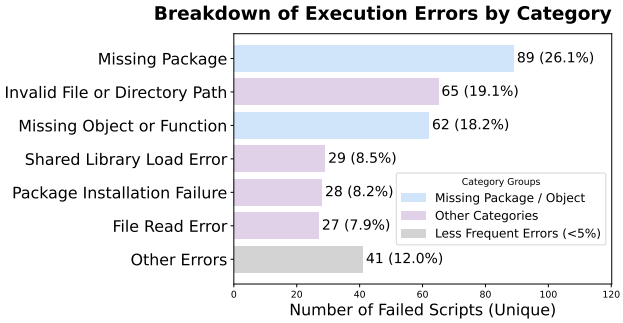

Figure 2: Breakdown of execution errors by category.

This structured classification enabled the distillation of hundreds of distinct error messages into a concise set of meaningful categories, as visualized in the final breakdown of execution errors (see Figure 2).

Conclusion

This study examined the reproducibility of statistical R scripts hosted on the Open Science Framework (OSF) using the `StatCodeSearch` dataset and a new automated container-based pipeline, `osf-to-binder`. A total of 558 R scripts from 296 OSF projects were analysed, of which 460 scripts were successfully containerised and executed. Execution outcomes were categorized and analysed to uncover prevalent failure patterns. Our findings corroborate earlier results by Hardwicke et al. (2020); Trisovic et al. (2022); Pimentel et al. (2019) and highlight common barriers to reproducibility, including unresolved dependencies, non-portable file paths, and assumptions about graphical execution environments.

The analysis highlights several best practices to improve the reproducibility of R scripts. First, all required packages should be explicitly declared and sourced from reliable repositories such as CRAN, Bioconductor, or GitHub, with version information when relevant. Providing installation scripts (e.g., `install.packages()` or `renv::restore()`) simplifies environment setup. Second, scripts should use relative paths (e.g., `"./data/data.csv"`) instead of hardcoded absolute paths, and input data should be included or linked via persistent references. Scripts should be verified for required files at runtime and return informative errors if missing. Finally, to support execution in automated or containerised environments, scripts should run non-interactively and avoid GUI-based functions in favor of programmatic alternatives, enabling full execution without manual intervention.

Cross-domain comparisons suggest that the reproducibility challenges observed in R are not unique. For instance, Samuel and Mietchen (2024) analysed 27,271 Jupyter notebooks associated with biomedical publications. Among the 10,388 Python notebooks for which all declared dependencies could be successfully installed and execution was attempted, 1,203 notebooks (11.6%) ran without any errors.

In comparison, our study found that 119 out of 460 R scripts (25.87%) executed without errors. Although drawn from different research domains and computational ecosystems, both studies point to the persistence of reproducibility barriers even when execution environments are successfully constructed. These findings reinforce the importance of language-agnostic best practices such as explicit dependency declaration, non-interactive design, and environment encapsulation for reproducible research.

However, this work is subject to certain limitations:

- **Dependency extraction with `flowR`**: The `flowR` static dataflow analyser is limited in detecting all import patterns in R, especially dynamic loading methods (e.g., via `lapply`)[7]. Some required packages may remain undetected, leading to incomplete environment specifications and potential containerisation failures.
- **Single-error reporting:** Execution halts at the first critical error, preventing analysis of subsequent issues in the same script.
- **Limited GitHub compatibility:** Projects with large files cannot be pushed to GitHub without Git LFS, restricting their availability via platforms like Binder, though they remain executable in the pipeline.

Future work could enhance dependency detection through dynamic code analysis or AI-assisted parsing. Expanding the pipeline to support multi-error detection and fault-tolerant recovery could offer a fuller picture of reproducibility challenges. Addressing GitHub's size constraints through automated Git LFS integration or alternative hosting strategies (e.g., Zenodo, OSF storage links) would improve accessibility and enable broader sharing of data-intensive projects. Encouraging broader adoption of OSF registrations and promoting best practices for file versioning at the time of publication (e.g., linking analysis scripts explicitly to registration DOIs) could also strengthen long-term reproducibility. Finally, integrating large language models and intelligent debugging systems may provide promising avenues for reducing execution failures and streamlining reproducibility assessments in computational research.

This work was funded by the German Research Foundation (DFG) under project No. 504226141.

References

Arnold, B.; Bowler, L.; Gibson, S.; Herterich, P.; Higman, R.; Krystalli, A.; Morley, A.; O'Reilly, M.; Whitaker, K.; et al. 2019. The turing way: a handbook for reproducible data science. *Zenodo*.

Barba, L. A. 2018. Terminologies for Reproducible Research. ArXiv:1802.03311 [cs].

Bleier, A. 2025. *What is Computational Reproducibility?* GESIS – Leibniz Institute for the Social Sciences, Cologne.

Boettiger, C. 2015. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1): 71–79.

Chung-hong Chan; Tim Schatto-Eckrodt; and Johannes Gruber. 2024. What makes computational communication science (ir)reproducible? *Computational Communication Research*, 6(1): 1.

Claerbout, J. F.; and Karrenbach, M. 1992. Electronic documents give reproducible research a new meaning. In *SEG technical program expanded abstracts 1992*, 601–604. Society of Exploration Geophysicists.

Diera, A.; Dahou, A.; Galke, L.; Karl, F.; Sihler, F.; and Scherp, A. 2023. GenCodeSearchNet: A Benchmark Test Suite for Evaluating Generalization in Programming Language Understanding. ArXiv:2311.09707 [cs].

Forde, J.; Head, T.; Holdgraf, C.; Panda, Y.; Nalvarete, G.; Ragan-Kelley, B.; and Sundell, E. 2018. Reproducible research environments with repo2docker.

Hardwicke, T. E.; Wallach, J. D.; Kidwell, M. C.; Bendixen, T.; Crüwell, S.; and Ioannidis, J. P. 2020. An empirical assessment of transparency and reproducibility-related research practices in the social sciences (2014–2017). *Royal Society open science*, 7(2): 190806.

Kohrs, F. E.; Auer, S.; Bannach-Brown, A.; Fiedler, S.; Haven, T. L.; Heise, V.; Holman, C.; Azevedo, F.; Bernard, R.; Bleier, A.; et al. 2023. Eleven strategies for making reproducible research and open science training the norm at research institutions. *Elife*, 12: e89736.

Lasser, J. 2020. Creating an executable paper is a journey through Open Science. *Communications Physics*, 3(1): 143.

National Academies of Sciences, E.; and Medicine. 2019. *Reproducibility and Replicability in Science*. Washington, DC: The National Academies Press. ISBN 978-0-309-48616-3.

Nüst, D.; and Eglen, S. J. 2021. CODECHECK: an Open Science initiative for the independent execution of computations underlying research articles during peer review to improve reproducibility. *F1000Research*, 10: 253.

Peng, R. D. 2011. Reproducible research in computational science. *Science*, 334(6060): 1226–1227.

Pimentel, J. F.; Murta, L.; Braganholo, V.; and Freire, J. 2019. A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*, 507–517. IEEE.

Ragan-Kelley, B.; Willing, C.; Akici, F.; Lippa, D.; Niederhut, D.; and Pacer, M. 2018. Binder 2.0-Reproducible, interactive, sharable environments for science at scale. In *Proceedings of the 17th python in science conference*, 113–120. F. Akici, D. Lippa, D. Niederhut, and M. Pacer, eds.

Rainey, C.; Roe, H.; Wang, Q.; and Zhou, H. 2025. Data and Code Availability in Political Science Publications from 1995 to 2022. *Political Science & Politics*, 1–7.

Samuel, S.; and Mietchen, D. 2024. Computational reproducibility of Jupyter notebooks from biomedical publications. *GigaScience*, 13: giad113.

Sandve, G. K.; Nekrutenko, A.; Taylor, J.; and Hovig, E. 2013. Ten simple rules for reproducible computational research. *PLoS computational biology*, 9(10): e1003285.

[7] https://github.com/flowr-analysis/flowr/issues/1259

Schoch, D.; Chan, C.-h.; Wagner, C.; and Bleier, A. 2024. Computational reproducibility in computational social science. *EPJ Data Science*, 13(1): 75.

Seibold, H.; Czerny, S.; Decke, S.; Dieterle, R.; Eder, T.; Fohr, S.; Hahn, N.; Hartmann, R.; Heindl, C.; Kopper, P.; et al. 2021. A computational reproducibility study of PLOS ONE articles featuring longitudinal data analyses. *PLoS One*, 16(6): e0251194.

Sihler, F.; and Tichy, M. 2024. flowR: A Static Program Slicer for R. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, 2390–2393. New York, NY, USA: Association for Computing Machinery. ISBN 979-8-4007-1248-7.

Trisovic, A.; Lau, M. K.; Pasquier, T.; and Crosas, M. 2022. A large-scale study on research code quality and execution. *Scientific Data*, 9(1): 60.