"Works on My Machine": A Case Study of Replicability Challenges in Computational Humanities Research

Viktor J. Illmer[1]

[1] EXC 2020 *Temporal Communities*, Freie Universität Berlin, Germany

Abstract

The replicability of computational research is often stated as a key concern in digital humanities scholarship, yet its practical realisation frequently encounters limitations. This contribution analyses some technical conditions for the replicability of articles in the 2024 *Computational Humanities Research* conference proceedings. A survey was conducted to determine the stated availability of source code, the programming languages employed, and whether and in which way dependencies were declared. The results show that a majority of contributions were not able to make their source code available. Among those that did provide code, many supplied insufficient information on software dependencies to reproduce their computational environments.

This circumstance sheds light on infrastructural challenges that complicate the replication of results and, by extension, the review and reuse of these works.

Keywords: replicability, reproducibility, open science

1 Introduction

The replicability of computational research has become a central concern in digital humanities scholarship, yet its practical realisation often encounters significant limitations. This issue is part of the broader so-called *reproducibility crisis* that gained significant attention in academia around 2015 [10] and in public discourse in 2019 [9]. While this crisis typically refers to the inability to corroborate published research findings based on newly collected data, the present contribution focuses on the narrower concept of *replicability* and its technical requirements in computational humanities research.

To investigate replicability practices, this paper examines research contributions published in the proceedings of the 2024 *Computational Humanities Research* (CHR) conference, one of the largest venues dedicated to computational work in the humanities. It asks whether code artefacts are made available, and whether the computational environments used are specified in sufficient detail to allow for their reproduction. In doing so, it aims to assess the state of replicability practices in the field and to identify both common patterns and persistent shortcomings in current workflows. The study investigates some of the key preconditions to replicability, namely source code availability and the reproducibility of the computational environment's software dependencies. Through a systematic analysis of 78 CHR 2024 contributions, this work provides insight into the current state of replicability practices in computational humanities research.

2 Replicability and Reproducibility

The terms reproducibility and replicability are used differently across and within fields, and sometimes even interchangeably. As Patil, Peng, and Leek put it, "Everyone agrees that scientific studies

Viktor J. Illmer. ""Works on My Machine": A Case Study of Replicability Challenges in Computational Humanities Research." In: *Computational Humanities Research 2025*, ed. by Taylor Arnold, Margherita Fantoli, and Ruben Ros. Vol. 3. Anthology of Computers and the Humanities. 2025, 142–148. https://doi.org/10.63744/iAqEoznkfKuz.

should be reproducible and replicable. The problem is almost no one agrees upon what those terms mean" [7]. In his terminological work on *repetitive research*, Schöch proposes a set of definitions from the perspective of the digital humanities along three core dimensions: the research question, the dataset, and the method [10]. In this conceptual space, the case where all of these are (virtually) identical to the original study is termed *replication*, and the degree to which this can be achieved *replicability*.

Replication as a practice is characterised as quality control, as it ensures that research workflows produce the expected results and that no manipulation has taken place. As Schöch points out, replication is explicitly not concerned with generating new knowledge, but rather with its validation. In the case of computational research, the availability of source code has been positioned as "one of the most critical factors" in aiding replicability [1], with the added effect that replicable computational research artefacts also lend themselves more easily to re-use and adaptation. However, as Arvan et al. point out in their in-depth survey of contributions to major computational linguistics conferences, while source code availability is a necessary condition for the replication of computational research, it is by no means sufficient [1].

Achieving replicability in computational research comes with a set of technical requirements, or, in other words, projects must be developed with replicability in mind. In addition, these projects must necessarily concern themselves with the question of sustainability, as software, and software libraries especially, may change over time and the stability of their interfaces or functionality is typically not guaranteed [10]. Projects which do not precisely describe their computational environment may therefore cease to function in the near future, complicating attempts at replicating their findings.

Building on Schöch's conception of replicability, in this contribution, I aim to investigate some of the preconditions to replicability, namely source code availability and the reproducibility of the computational environment's software dependencies. For the purposes of this paper, practices that aim to control for these challenges by providing information on the original software environment are understood as aiding the *reproducibility* of the computational environment. For one, this allows for a distinction between the *replicability* of a study and the *reproducibility* of its computational environment.

In addition, this term aligns more closely with the underlying technical mechanisms, as commonly used dependency management systems evaluate, fetch, and collate remote artefacts to reproduce the desired configuration from its constituent components. Finally, it relates to the concept of *reproducible builds* in open-source software development, though this is specifically concerned with deterministic, bit-perfect reproductions [5]. For the purposes of this paper, reproduction of computational environments is understood more loosely, in terms of its functional equivalence, as bit-for-bit identical reproduction of environments is either unviable or unattainable across operating systems.

3 Method

All published papers were retrieved from the CHR 2024 proceedings page (`https://ceur-ws.org/Vol-3834/`). Titles and author names were extracted from the HTML of the proceedings page using Python, and the corresponding PDFs were downloaded programmatically for a total of 78 cases.

As this study is concerned with the replicability of computational workflows, only papers addressing empirical or methodological questions through code-based analysis were included. Contributions that primarily introduced tools or software without their computational evaluation itself being the object of the study were excluded. This excludes, for instance, papers which discuss the development of new computational tools to aid in research, because, even though they may make their source code available, there may be no research question to answer or findings to replicate.

However, if the study's focus is, for example, the evaluation or benchmarking of a newly developed tool or method, resulting in both source code and empirical findings that can be assessed for replicability, then such a contribution qualifies for inclusion.

An initial scan for mentions of source code availability was conducted by searching each paper for the keywords "code" and "scripts". Subsequently, each paper was manually inspected and annotated for the following variables: *Source code availability*, *Git URL*, *repository URL*, *programming language(s)*, *dependency management schemes*, and *additional dependency management features*. This annotation scheme was developed inductively based on the features encountered as the annotation work progressed.

Source code availability, was categorized into six classes: *Yes*, *No*, *Unfulfilled / No location*, *Unfulfilled / Empty*, *Data only*, and *Not applicable*. The *Yes* and *No* categories indicate, respectively, whether a functioning code repository was clearly linked and accessible, or absent altogether. *Data only* refers to cases where only datasets were made available, despite the apparent presence or necessity of source code based on the contributions' descriptions. Finally, the two *Unfulfilled* categories were used when authors indicated an intention to share code, but none could be identified. *Unfulfilled / No location* was used when no specific identifier or access point for the resource was provided (e.g. no URL, DOI, or repository name), while *Unfulfilled / Empty* was applied when the referenced location (e.g. a GitHub repository) existed but did not contain the relevant source code. Finally, where the availability of source code was stated, but not properly linked, an attempt was made to identify the correct location using a web search and included in the analysis where successful.

The project's URLs in open-access repositories (e.g. Zenodo) and/or Git-based code forges (e.g. GitHub) were recorded. Where applicable, the projects' repositories were reviewed using their platforms' web interfaces. In cases where this was impractical, files were downloaded and inspected locally. For GitHub-hosted repositories, the platform's language analysis was used to inform annotation of the main programming language [3].

One of the main variables of interest was the presence and form of dependency management features. Given the centrality of dependency versioning for replicability, seeing as library APIs may change considerably between versions, each project's repository was inspected for mechanisms used to specify dependencies. As with the other categories, rather than starting from a fixed set, they were developed inductively over the course of the annotation.

These included a number of different approaches: from informal methods like dependencies listed in code comments or referenced through basic installation commands, such as *pip install* in a README file or a Jupyter notebook cell, to more structured approaches such as Python's *requirements.txt* files, and, for more robust version control, the use of lockfiles including hashes, such as *poetry.lock* in Python or *renv.lock* in R. The use of external tools for declarative environment configuration, such as Nix, was also recorded.

In addition to identifying specific dependency management schemes, each project was evaluated in terms of the reproducibility of its dependency specifications. This judgment does not necessarily follow from the dependency scheme used, as some schemes may provide insufficient details to infer the original environment. A project's dependencies were marked as reproducible only if the versions of all declared dependencies were explicitly and unambiguously specified. This makes this category rather fragile: Even a single undeclared version, such as a line in a requirements file lacking a version constraint, was sufficient for the project to be marked as non-reproducible.

4 Findings

The most immediate variable concerns the availability of source code, as this forms the basis for further analysis. For this, an almost equal split could be observed among projects (Table 1). Out of the 78 annotated projects, 33 (42%) made their source code available, while 40 (roughly half)

did not, although some expressed an (“unfulfilled”) intention to do so: in seven cases, no location was provided, and in 3, the linked repository was empty. The most common reasons for this were issues with the published PDFs, where text was marked as a hyperlink, but no link was actually encoded, as well as leftover anonymisation placeholders from the peer review process as required by CHR [2], such as “github.com/xxxx”. In four cases, only data was shared. For five projects, code availability was deemed not applicable due to the nature of the work.

Source code availability	Count	% of projects
Yes	33	42
No	40	51
of which provided data only	4	5
of which provided no location	7	9
of which were empty	3	4
Not applicable	5	6

Table 1: Source code availability among the 78 annotated projects

The most commonly used programming language across the annotated projects was Python (Table 2). It appeared in 18 projects’ source files (55%), and in 18 projects in the form of Jupyter notebooks. There is substantial overlap between these two categories, as many projects combined Python utility scripts with notebook-based analysis workflows.

Programming language	Count	% of projects
Python	18	55
Jupyter/Python	18	55
R	5	15
MATLAB	1	3
C++	1	3
Shell	1	3

Table 2: Programming languages used in the 33 projects with available code

Other languages appeared far less frequently: R was used in 5 projects (15%), while MATLAB, C++, and shell scripts each appeared once. Projects could and often did use multiple languages in parallel. Given the prominence of Python, this section will also discuss language-specific practices where applicable. That said, the analysis is not concerned with language or tool choice per se, but with how precisely dependencies are declared and managed in general.

Across the corpus, dependency management practices varied widely in both structure and reproducibility (Table 3). Of the 33 projects with available source code, only seven (about 20%) specified all dependencies with clearly defined version constraints and were thus classified as having reproducible dependency specifications. The remaining 26 projects lacked sufficient versioning information to allow for a faithful reconstruction of the original software environment.

The most common pattern among projects with non-reproducible dependencies was the absence of any structured dependency specification beyond the code itself. In 19 projects,

dependencies could only be inferred from import statements or equivalent constructs. While these indicate which packages are used, they provide no versioning information and offer no guarantees for replicability. Some projects attempted more structured approaches, but still fell short of full reproducibility. Eight projects included a requirements file (e.g. requirements.txt for Python), but only half of these fully pinned all versions. In the other half, some packages were listed with-

Dependency management scheme	Count	% of projects
None (import only)	19	58
Requirements file	8	24
Lockfile	3	9
In README	1	3
In notebook cell	1	3
In comment	1	3

Table 3: Highest-fidelity dependency management schemes used in the 33 projects with available code

out version constraints, presumably in part as a result of inconsistent editing after an initial pip freeze. This practice was a relatively frequent obstacle to dependency reproducibility, suggesting that even the use of standard tooling does not reliably translate into reproducible results.

Three projects included proper lockfiles with both version and hash information, allowing for high-fidelity reproduction (or at least verifiable reproduction, depending on availability) of their dependencies. Among these projects, one used PDM, a more recent Python-specific package manager [8]. Another employed the Nix package manager, which enables the declarative configuration of software and its dependencies [6].

A handful of projects relied on more informal or unconventional methods: One project included a "pip install" command in their README file, another included a "!pip install" shell command as the top-most cell in their Jupyter notebook. Unfortunately, while technically possible, neither specified the version numbers of their dependencies, meaning that the package manager will resolve the latest releases as it sees fit, leading to potential breakage. One R-based project included package names and version numbers in a comment at the top of their script.

Additionally, only some of these dependency management schemes are machine-actionable, that is to say that they are properly structured and interpretable by package management software in order to reproduce the correct environment. As noted with dependency definition files such as requirements.txt, manual editing can compromise their version pinning guarantees. Specifically, only seven projects' dependencies could be considered reproducible as well as machine-actionable, four of which used a requirements file as their highest-fidelity dependency pinning scheme. In fact, while requirements files were the most common dependency management scheme encountered, they only achieved full version pinning in four out of eight cases.

A single project offered an Open Container Initiative (OCI)/Docker image of their computational environment in addition to reproducible version definitions. These images are a special case, as they already include all the programs and dependencies required for replicating findings, and are termed self-contained and self-executable research artefacts [1]. It is, therefore, less appropriate to speak of *reproducing* the computational environment, as there is nothing to be evaluated, fetched, or built: Instead, the original environment itself has already been built and compressed, ready to be loaded on a new machine. OCI images offer high fidelity with respect to the original environment and are expected to be quite resilient, as they are self-contained and do not rely on external sources when properly crafted [1]. In fact, encouraging the use of executable self-contained research artefacts is one of the main recommendations made by Arvan et al. following their extensive survey of replicability practices in computational linguistics [1].

5 Conclusion

Overall, while some form of dependency declaration was present in most projects, careful and consistent versioning remained the exception. The analysis shows that even when tooling is available

and ostensibly used, reproducibility often breaks down in the details. This aligns with Arvan, Pina, and Parde's observations, based on both their own and earlier studies, that many released artefacts still fall short of reproducibility standards, despite recent progress [1].

While this contribution has focused on programming library dependencies, these are, arguably, the low-hanging fruit of computational reproducibility: standardised tooling exists, and most programming languages offer support for dependency specification. Yet even here, versioning practices remain inconsistent, and critical details are often missing. Importantly, this study did not attempt to execute the published research artefacts. Actual replicability rates may therefore be lower than the data suggests, as dependency specifications may be incomplete, code may be erroneous, or factors like hardware differences and hard-coded paths may further impede replicability.

Beyond package dependencies, research workflows often involve further components: datasets, language models, command-line utilities, and orchestration across mixed-language toolchains. These fall outside the purview of programming language's regular dependency managers and are rarely versioned. As such, a more holistic approach is needed to treat data and auxiliary tools as first-class dependencies. The FAIR Principles already lay the groundwork for this in their recommendation of stably addressable and machine-retrievable datasets [4]. Future research could investigate barriers that prevent researchers from adopting such practices. For instance, interviews with authors might reveal time constraints, lack of infrastructure, or uncertainty around best practices as common obstacles. Further, authors may lack the incentive to invest in replicability practices when these are often neither required nor reviewed: While many conferences encourage the publication of source code, including CHR, they rarely require it, in part to reduce reviewer burden [1]. For CHR specifically, to ensure double-blindness during the double-blind peer review phase, repository URLs must be either anonymised using appropriate tools, or redacted, which effectively excludes code from the review entirely [2]. Together, these factors may represent considerable challenges for advancing replicability practices within the community.

More is needed to make findings truly replicable, and while reproducible computational environments are a considerable prerequisite in achieving this, they are not the full picture.

Coding practices need to account for execution contexts that differ from the original in terms of time, people and systems. In other words, researchers should be enabled to write code that still works in the future, for users other than themselves, and on machines unlike their own.

Code and Data Availability

Source code and data for this contribution are available on GitHub at `https://github.com/v-ji/chr2025-works-on-my-machine`.

Funding Statement

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy in the context of the Cluster of Excellence *Temporal Communities: Doing Literature in a Global Perspective* – EXC 2020 – Project ID 390608380.

References

[1] Arvan, Mohammad, Pina, Luís, and Parde, Natalie. "Reproducibility in Computational Linguistics: Is source code enough?" In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. EMNLP 2022, ed. by Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang. Abu Dhabi, United Arab Emirates: Association for Computa-

tional Linguistics, Dec. 2022, pp. 2350–2361. DOI: 10.18653/v1/2022.emnlp-main.150.

[2] Computational Humanities Research. “Call for papers”. Computational Humanities Research 2024. 2024. URL: https://2024.computational-humanities-research.org/cfp/.

[3] GitHub. “About repository languages”. GitHub Docs. 2025. URL: https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-repository-languages.

[4] GO FAIR Initiative. “FAIR Principles”. 2025. URL: https://www.go-fair.org/fair-principles/.

[5] Lamb, Chris and Zacchiroli, Stefano. “Reproducible builds: Increasing the integrity of software supply chains”. In: *IEEE Software 39*, no. 2 (Mar. 2022), pp. 62–70. ISSN: 1937-4194. DOI: 10.1109/MS.2021.3073045.

[6] NixOS contributors. “Nix & NixOS: Declarative builds and deployments”. 2025. URL: https://nixos.org/.

[7] Patil, Prasad, Peng, Roger D., and Leek, Jeffrey T. “A statistical definition for reproducibility and replicability”. July 29, 2016. DOI: 10.1101/066803.

[8] PDM contributors. “PDM”. 2025. URL: https://github.com/pdm-project/pdm.

[9] Ries, Thorsten, Van Dalen-Oskam, Karina, and Offert, Fabian. “Reproducibility and explainability in digital humanities”. In: *International Journal of Digital Humanities 6*, no. 1 (Jan. 3, 2024), pp. 1–7. ISSN: 2524-7840. DOI: 10.1007/s42803-023-00083-w.

[10] Schöch, Christof. “Repetitive research: a conceptual space and terminology of replication, reproduction, revision, reanalysis, reinvestigation and reuse in digital humanities”. In: *International Journal of Digital Humanities 5*, no. 2 (Nov. 1, 2023), pp. 373–403. ISSN: 2524-7840. DOI: 10.1007/s42803-023-00073-y.