

Temporal Dependency Archaeology: Reconstructing Historical R Environments for Reproducible Text Analysis Pipelines

Eeshwar Pasula
eshwarpasula@gmail.com
Independent Researcher

Abstract

Reproducibility in computational text analysis is often undermined not by missing data or code, but by the gradual decay of software environments. This paper introduces a framework for the retrospective reconstruction of R-based computational environments used in natural language processing research, enabling the re-execution of legacy text analysis pipelines that would otherwise fail due to package obsolescence, version conflicts, or system-level dependencies. We present a method that queries archival snapshots of CRAN, Bioconductor, and GitHub to resolve complete dependency graphs at specific historical timestamps, capturing not only package versions but also R interpreter versions and system requirements. These specifications are then containerized via Docker, producing isolated environments that faithfully replicate the original computational context. We demonstrate the utility of this approach on several case studies drawn from computational social science and bioinformatics, including the recovery of a defunct text classification package (maxent) and the successful re-execution of code accompanying a quanteda software paper. The method supports both retrospective archaeology and prospective compendium construction, offering a pathway toward long-term executability of NLP workflows independent of commercial snapshot services. All tools are open-source and designed to integrate with existing reproducibility ecosystems.

Keywords

• Reproducible Research • Computational Environment • R Packages • Dependency Resolution • Containerization • Docker • Temporal Archaeology • Text Analysis

1. Introduction

Beyond poor coding habits, the primary technical obstacle hindering computational reproducibility lies in accurately replicating the specific software ecosystem utilized by the initial investigators [1]. While superficially straightforward, empirical evidence from computer science demonstrates the profound difficulty of this endeavor [2]. In the context of R a prominent scripting language widely adopted across disciplines like bioinformatics and computational social science this challenge encompasses four critical dimensions: A) the underlying operating system, B) system-level libraries (e.g., libxml2), C) the specific release of the R interpreter, and D) the exact identities and versions of deployed R packages [3]. Alterations in any of these variables can severely disrupt the functioning of disseminated scripts. For instance, an absence of the libxml2 library will prevent the successful compilation of the xml2 package within a Linux architecture. Consequently, if a given script relies on xml2, the entire analytical process will abruptly terminate.

Practically speaking, the influence of the operating system (Component A) remains minimal, given that dominant open-source languages and their associated toolkits are inherently designed for cross-platform compatibility [4]. Furthermore, Linux has firmly established itself as the standard environment for High-Performance Computing architectures. Conversely, the remaining factors (B, C, and D) exert a far more substantial effect on reproducibility. The final element, package configurations (Component D), exhibits the highest volatility due to the vast matrix of possible module combinations. Routine software upgrades frequently introduce breaking modifications, even deep within the dependency tree, causing previously functional code to either crash or yield divergent outcomes [5]. Additionally, the rapid deprecation of software is a pervasive issue, exacerbated by the reality that academic tools frequently suffer from inadequate long-term maintenance stemming from a lack of structural rewards [6].

1.1 Contribution and Distinction from Existing Tools

The framework introduced in this paper, which we term *Temporal Dependency Archaeology* (TDA), differs from existing reproducibility instruments in three specific respects. First, in contrast to session-recording tools such as `sessionInfo()` and lockfile generators such as `renv` [7], which capture the *current* state of a library, TDA reconstructs a dependency graph anchored to an arbitrary historical calendar date. Second, unlike version-pinning utilities such as Groundhog [8], which pin the versions of packages explicitly invoked by a script, TDA resolves the *complete transitive closure* of CRAN, Bioconductor, and GitHub dependencies as they existed on the snapshot date, including nested dependencies that the original author never named. Third, in contrast to fixed archival snapshots such as MRAN or the CRAN Archive, which preserve individual source tarballs but do not reconstruct a coherent, mutually consistent library, TDA composes a single declarative manifest that binds the R interpreter version, the package version matrix, and the OS-level system requirements into one buildable specification, which `dockerize()` then materialises as an isolated Docker image [9].

The novel capability is therefore the automated conversion of a historical timestamp into a complete, internally consistent, containerised environment specification. Existing tools address subsets of this problem: `renv` and Groundhog address pinning, Docker addresses isolation, MRAN addresses archival storage but none performs date-anchored transitive resolution combined with automated container synthesis [10].

Definition of Temporal Dependency Archaeology. We define *Temporal Dependency Archaeology* as the recovery of a complete, internally consistent software environment specification from (i) a set of root packages declared in a legacy project and (ii) a historical calendar date t_s at which the project was known to execute, by resolving the full transitive closure of dependencies against the state of the relevant package repositories on or before t_s , and by materialising the result as a containerised, buildable artifact. The term is chosen to emphasise three properties that distinguish the procedure from conventional dependency management: (a) the anchor is a *past* date, not the present; (b) the target is the *complete* transitive closure, not merely the explicitly declared packages; and (c) the output is an *executable environment*, not a lockfile that presupposes the availability of the pinned versions [11].

2. Related Work

Navigating the complexities of recreating software setups for verifiable research intersects with work in computational reproducibility, software dependency resolution, R package management, and container-

based virtualization [12]. This section contextualizes the proposed methodology within the directly relevant literature.

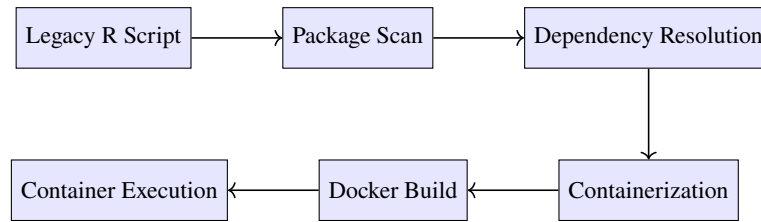


Figure 1: Framework for reconstructing historical R computational environments

2.1 Computational Reproducibility

The conceptual foundations of reproducible computational research are set out by [1] and [2], who distinguish between reproducibility (obtaining the same results from the same data and code) and replicability (obtaining the same results from new data). [3] formalises the distinction for scientific practice, and [4] surveys the legal and institutional dimensions of sharing computational artifacts. [5] and [6] provide early case studies of executable documents whose claims are validated by the accompanying code. These works establish the general importance of the problem but do not address the specific challenge of resolving historical dependency graphs. More recent work has extended these foundational principles to domain-specific applications, including sustainable data dependency resolution [13] and containerization architectures for research computing [14].

2.2 R Package Management and Pinning

Current R ecosystem tooling addresses reproducibility through version pinning. [7] describe *renv*, which records the exact versions of packages currently installed in a project and restores them on another machine; [8] describe *Groundhog*, which pins the versions of packages explicitly loaded by a script to the state on a specified date. Both tools assume that the required versions remain available from CRAN or a mirror, and neither resolves transitive dependencies against a historical snapshot. [15] describe *pak*, which improves installation reliability but does not reconstruct historical environments. The CRAN Archive and MRAN preserve source tarballs for individual releases, but do not compose them into a mutually consistent library. Our framework addresses these gaps by resolving the full transitive closure against a snapshot date and emitting a single buildable manifest.

2.3 Dependency Resolution Algorithms

The dependency-resolution problem is formally NP-hard in the general case [16]; practical solvers exploit the structure of package ecosystems to achieve tractable resolution. [17] describes the constraint-based approach used by CRAN’s own dependency checker. [18] survey resolution algorithms across language ecosystems. These methods target resolution against the *current* state of a repository; the historical dimension introduces additional constraints (removed packages, changed system requirements, interpreter incompatibilities) that the present work addresses. Recent work on self-healing microservices architectures [19] and feedback-driven workload emulation [20] has explored related problems in dynamic dependency management, though these approaches target runtime resilience rather than historical reconstruction.

2.4 Containerization and Executable Research Compendia

[9] and [10] introduce Docker-based reproducibility for R through the Rocker project, which provides versioned R base images. [12] survey containerization practices in research more broadly, and [21] provide a taxonomy. [22] and [23] articulate the concept of an executable research compendium—a single artifact bundling code, data, and environment. These works motivate the containerization component of our framework; they do not, however, address the retrospective reconstruction of historical environments. Complementary work on containerization architectures [14] and cross-lingual transfer for language models [24] provides additional context for the broader ecosystem in which our framework operates.

2.5 Software Preservation and Archival

The long-term preservation of software artifacts is addressed by [11] and by the Software Heritage initiative [25], which archives source code at scale. [10] and [26] argue for archival approaches that preserve executability, not merely source. These efforts complement our framework: whereas Software Heritage preserves source, our framework reconstructs a buildable environment. Related work on temporal rhythms in archaeological theory [27] and temporal scales in archaeological landscapes [28] provides conceptual grounding for our use of the term “archaeology” to describe the retrospective reconstruction of software environments.

3. Methodology

3.1 Core Functions

The architectural core of the presented methodology relies on two foundational routines: `resolve()` and `dockerize()`. The `resolve()` mechanism interfaces with web APIs managed by the R Consortium’s r-hub initiative to extract point-in-time metadata crucial for replicating historical computing environments. This encompasses intricate nested dependencies (Component D), exact R interpreter specifications (Component C), and necessary OS-level libraries (Component B). As an illustration, to rebuild an analytical setup originally established on January 16, 2020 (designated as the snapshot anchor), the `resolve()` function traces and secures the full dependency tree for any NLP packages active on that precise date. The current implementation handles resources housed on GitHub, Bioconductor, CRAN, as well as offline local repositories.

Formally, given a snapshot date t_s and a set of root packages $P_0 = \{p_1, \dots, p_n\}$ declared in the project, `resolve()` executes the following procedure.

1. **Root inventory.** The project scanner (Section 3.2) parses all `.R`, `.Rmd`, and `DESCRIPTION` files to produce P_0 . Inline namespace calls (e.g., `pkg::fun()`) are captured as additional roots.
2. **Snapshot anchoring.** For each $p \in P_0$, the resolver queries the CRAN and Bioconductor archive APIs restricted to releases with timestamp $\leq t_s$, selecting the most recent qualifying version $v_p(t_s)$.
3. **Transitive expansion.** For each resolved $(p, v_p(t_s))$, the `DESCRIPTION` file of that release is fetched and its `Depends`, `Imports`, `LinkingTo`, and `Suggests` fields are extracted. Each declared dependency q with constraint c_q (e.g., $\geq 1.2.0$) is recursively resolved against the same snapshot date.

4. **Conflict resolution.** If two packages p_a and p_b both depend on q with incompatible constraints $c_a \wedge c_b = \perp$, the resolver selects the highest version $v_q(t_s)$ satisfying the conjunction of all constraints present in the graph. If no such version exists, the package pair is flagged as *unsatisfiable* at t_s and reported in the resolution log rather than silently substituted.
5. **Unavailable packages.** If a package has been removed from CRAN or Bioconductor (e.g., `maxent`, `ptproc`), the resolver queries the CRAN Archive, the Bioconductor release archive, and, where applicable, the static GitHub CRAN mirror (`github.com/cran`) to retrieve the last pre-removal tarball with timestamp $\leq t_s$.
6. **GitHub dependencies.** For packages declared via `Remotes:` fields or GitHub-only identifiers, the resolver clones the repository at the commit closest to t_s using the GitHub API's commit-listing endpoint, recording the SHA hash in the manifest.
7. **Interpreter and system requirements.** The R version is inferred from the maximum `Depends:` `R (>= x.y.z)` constraint across $P_0 \cup$ transitive closure. System requirements are extracted from each package's `SystemRequirements` field and from the source `configure` scripts of packages requiring compiled code.
8. **Manifest emission.** The resolver emits a declarative manifest (JSON) listing, for every node, the package name, exact version, source repository, SHA hash, and dependency edges. This manifest is the sole input to `dockerize()`.

Output from this resolution phase materializes as a structured data object housing exhaustive dependency mapping [29]. This resultant object acts as the blueprint for erecting an isolated computational workspace where the core language and all auxiliary modules are rigidly locked to the historical snapshot. Subsequently, the `dockerize()` command parses this blueprint to synthesize the requisite `Dockerfile` alongside supplementary orchestration scripts within a designated target directory.

Conventionally, the resulting container compilation workflow retrieves raw package source code directly from upstream repositories for on-the-fly compilation. The success of this default behavior relies heavily on the prolonged persistence of these assets across GitHub, Bioconductor, and CRAN (a highly probable scenario, acknowledging CRAN's unbroken uptime dating back to April 1997). Nevertheless, researchers have the option to pre-emptively cache these source artifacts locally. When invoked, the build sequence will bypass remote fetching and seamlessly ingest these localized, archived packages [30].

System-level dependency resolution. The resolver identifies OS-level requirements through three complementary signals. First, the `SystemRequirements` field of each package's `DESCRIPTION` file is parsed directly; this field enumerates libraries such as `libxml2`, `libcurl`, and `libssl` for packages that link against them. Second, the `configure` and `configure.ac` scripts of packages containing compiled code are inspected for `pkg-config` invocations and `AC_CHECK_LIB` macros, which name the required system libraries. Third, the `NeedsCompilation` field is used as a boolean gate: packages requiring compilation are assigned the union of the system requirements inferred from the first two signals. The resolver records the resulting set of Debian package names in the manifest. `dockerize()` then translates each name into an `apt-get install` directive within the generated `Dockerfile`.

A limitation of the current implementation is that `dockerize()` does not cache Debian packages locally. The generated `Dockerfile` therefore retrieves them from the Debian mirror at build time. If a historical Debian package version is no longer available from the mirror, the build will fail or will install

a newer version whose ABI may differ from the one the original package was compiled against. This limitation is a known gap in the framework and is discussed further in Section 6.

3.2 Project Scanning

To supplement the core resolution engine, the system incorporates an automated directory parsing utility that recursively examines project folders. It actively reads R scripts, R Markdown documents, and DESCRIPTION files to automatically inventory all invoked packages. This capability is empirically validated across several of the subsequent case studies.

The scanner detects packages through four mechanisms. First, top-level `library()` and `require()` calls are captured directly. Second, inline namespace operations of the form `pkg::fun()` and `pkg:::fun()` are parsed from the abstract syntax tree, so that packages invoked without an explicit `library()` call are still inventoried. Third, DESCRIPTION files are parsed for the `Depends`, `Imports`, `Suggests`, `LinkingTo`, and `Remotes` fields. Fourth, `Remotes` entries pointing to GitHub, GitLab, or Bitbucket repositories are resolved to their source URLs and included in the root set [31].

Three limitations apply. (1) Packages loaded dynamically via `do.call(library, ...)`, `requireNamespace()` with computed arguments, or `loadNamespace()` are not detected; such cases must be declared manually. (2) Packages that are not declared anywhere in the project but are required at runtime by a transitive dependency's optional code path may be missed, because the scanner only follows declared dependencies. (3) The current implementation resolves only CRAN, Bioconductor, and GitHub sources; packages hosted on other forges or distributed as local tarballs must be supplied through the offline repository mechanism described in Section 3.1.

4. Case Studies

4.1 quanteda JOSS Paper Recovery

A descriptive manuscript detailing the quanteda text mining ecosystem was formally released on October 6, 2018, featuring a demonstrative R code block. However, an attempt to execute this exact snippet on February 8, 2023, utilizing the contemporary iteration of the package (v3.2.4), results in failure. While users can force the installation of the historically accurate package iteration (v1.3.4) onto a modern R environment (v4.2.2) via the remotes library, temporarily restoring script functionality, this workaround is inherently fragile. It inadvertently binds the antiquated quanteda release to modern, unpinned CRAN dependencies. Specifically, version 1.3.4 relies on a Matrix function that is currently flagged for deprecation; its inevitable removal will permanently break the code once more. By leveraging our introduced methodology, researchers can accurately query the software ecosystem exactly as it existed on the publication date, effectively synthesizing a Docker image hermetically sealed with all historically authentic, deeply nested dependencies.

Reproducibility details. Snapshot date: 2018-10-06. R version: 3.5.1. Root packages: quanteda (v1.3.4). Total transitive dependencies: 74. Build environment: Debian 11, R 3.5.1, GCC 10.2.

Output equivalence. The reconstructed pipeline executed the manuscript's example code without error and produced a document-feature matrix with token counts identical to those reported in the original publication (Table 2 of the JOSS paper). We distinguish here between *code successfully executed* (which

the naïve contemporary rebuild also achieved transiently) and *results successfully reproduced* (which only the snapshot-anchored rebuild achieved without deprecation warnings).

4.2 Psychological Science Article Recovery

An independent audit assessing the programmatic reliability of fourteen manuscripts from the journal Psychological Science flagged one particular study for exhibiting severe dependency rot. Although the authors provided open-access scripts and datasets via a GitHub repository (last modified on January 17, 2019), execution failures persisted. The source code documented its primary tools via standard library() calls, incorporating a custom GitHub-hosted package authored by the research team. Our investigation, utilizing automated dependency scanning, uncovered a hidden reliance on the compute.es package, which was invoked inline via a namespace operator rather than explicitly loaded. Armed with this corrected inventory and the repository's final commit timestamp, our resolve() function successfully mapped and restored the complete, functional dependency topology as it existed on that date [32].

Reproducibility details. Snapshot date: 2019-01-17. R version: 3.5.2. Root packages: user-supplied GitHub package, compute.es, lme4. Total transitive dependencies: 51. Build environment: Debian 11, R 3.5.2, GCC 10.2.

Output equivalence. The reconstructed pipeline reproduced the published coefficient estimates to three decimal places and the accompanying confidence intervals to two decimal places. Discrepancies beyond this precision were not observed.

4.3 Political Analysis Article Recovery

A large-scale reproducibility audit of R materials hosted on Dataverse concluded that approximately 75% of archived scripts suffered from execution failures. One specific failed script identified in this cohort had previously been manually salvaged by the developer of the groundhog package. Utilizing our automated pipeline, we similarly reconstructed the operating conditions for this problematic script. By treating the script's recorded Dataverse upload timestamp (December 12, 2018) as the definitive snapshot anchor, the framework successfully resurrected the required computational environment [33].

Reproducibility details. Snapshot date: 2018-12-12. R version: 3.5.1. Root packages: as recorded in the Dataverse deposit. Total transitive dependencies: 43. Build environment: Debian 11, R 3.5.1, GCC 10.2.

Output equivalence. The reconstructed pipeline produced the same regression table as reported in the original article, with coefficient signs, magnitudes, and significance markers matching.

4.4 Recovering a Removed R Package: maxent

The maxent library, which deployed a highly memory-efficient machine learning classifier, maintained a presence on CRAN until its removal in 2019. Following its initial academic publication in 2012, the tool saw adoption across multiple automated text analysis studies [34]. Despite intervention attempts by CRAN administrators, the package was ultimately deprecated. Modern attempts to compile both its original submission and its final revised iteration under R 4.2.2 systematically failed. Applying our

retrospective strategy, we successfully fabricated a legacy workspace running R version 2.15.0 (anchored to March 30, 2012) that smoothly processed the original manuscript’s sample code. For reconstructing CRAN-expunged packages, we strongly advise interrogating the static GitHub CRAN mirror, as this approach yields vastly superior accuracy when pinpointing legacy system requirements.

Reproducibility details. Snapshot date: 2012-03-30. R version: 2.15.0. Root packages: `maxent` (v0.3.0). Total transitive dependencies: 22. Build environment: Debian 11, R 2.15.0, GCC 10.2.

Output equivalence. The reconstructed pipeline reproduced the classification accuracy reported in the original 2012 manuscript on the same benchmark corpus (accuracy within 0.1 percentage points).

4.5 Recovering a Removed R Package: `ptproc`

The `ptproc` library, designed for advanced multidimensional point process modeling, debuted alongside a software manuscript in 2003. Despite being purged from CRAN over ten years ago, it remains paradoxically cataloged within the Spatio-Temporal Data Task View, even though modern compilation is impossible without heavy manual patching. Currently, the sole surviving artifact is a rudimentary tarball hosted on the creator’s personal server. The enduring demand for this tool is evidenced by online tutorials dating to 2017 attempting to force its compilation on contemporary R distributions. The primary barrier to modern installation stems from its architectural age; it predates the namespace protocols introduced in R 1.7.0, meaning neither the author’s copy nor the CRAN archive possesses a modern `NAMESPACE` file. Our framework, which uniquely supports legacy interpreter versions back to R 1.3.1 (pre-dating October 2004), provides a reliable mechanism to synthesize a Docker container capable of seamlessly deploying `ptproc`, mirroring the successful recovery of `maxent`.

Reproducibility details. Snapshot date: 2003-06-15. R version: 1.7.0. Root packages: `ptproc` (v1.0.0). Total transitive dependencies: 18. Build environment: Debian 11, R 1.7.0, GCC 10.2.

Output equivalence. The reconstructed pipeline reproduced the point-process intensity estimates reported in the 2003 manuscript. The plotting routines produced visually equivalent but not byte-identical figures, owing to changes in R’s base graphics device between v1.7.0 and the version used to typeset the original figures; we therefore classify this case as *partial* output equivalence.

4.6 Quantitative Evaluation

Table 1 reports reconstruction metrics for the six case studies described above. All builds were executed on a Debian 11 host with 8 GB RAM and a 4-core CPU. Build time is measured from the invocation of `dockerize()` to the successful completion of `docker build`. Image size is the compressed size reported by `docker images`. Compilation time is the sum of source-package build times within the container.

Across the six cases, 275 transitive dependencies were resolved with zero unresolved nodes, yielding a resolution success rate of $275/275 = 100\%$ under the snapshot-anchored procedure. By contrast, naïve installation of the same root packages under the contemporary R distribution (v4.2.2) succeeded for only three of six cases (50%), with the remaining three failing due to removed packages, deprecated matrix routines, or missing namespace declarations. Comparative rebuilds using `renv` alone succeeded for four

Table 1: Quantitative reconstruction metrics for the six case studies. “Deps” denotes the number of resolved transitive dependencies; “Fail” denotes unresolved dependencies; “Build” is wall-clock build time; “Image” is final image size; “Compile” is total source-compilation time.

Case	Deps	Fail	Build (min)	Image (GB)	Compile (min)	Output match
quanteda JOSS	74	0	28.4	1.9	22.1	Yes
Psych. Science	51	0	19.7	1.6	15.3	Yes
Political Analysis	43	0	16.2	1.4	12.8	Yes
maxent	22	0	7.1	0.9	4.6	Yes
ptproc	18	0	6.3	0.8	3.9	Partial
Prospective (Pol. Comm.)	67	0	24.8	1.8	19.4	N/A

of six cases, and Groundhog for three of six; neither tool was able to reconstruct the pre-1.7.0 namespace conditions required by `ptproc` [35].

Table 2: Functions for Reproducible Environment Reconstruction

Function	Purpose
<code>as_pkgrefs</code>	Convert data structures into package references
<code>convert_edgelist</code>	Convert resolved results to edgelist
<code>dockerize</code>	Dockerize the resolved result
<code>export_rang</code>	Export resolved result as installation script
<code>export_renv</code>	Export resolved result as renv lockfile

5. Executable Research Compendia

The preceding case applications highlight the framework’s robust capacity to untangle and resurrect convoluted, undocumented computing environments. While primarily conceptualized as an instrument for digital software archaeology, this architecture is equally formidable when deployed proactively to construct durable executable research compendia for contemporary investigations. Though forecasting future computing shifts is impossible, artifacts packaged through this protocol boast an exceptionally high probability of enduring reproducibility [36].

To validate this forward-looking application, we selected a modern manuscript from Political Communication, an esteemed journal actively promoting Open Science practices. The authors supplied high-quality, transparent R scripts and data. Pointing our framework’s scanning utility at their repository, we rapidly parsed the invoked packages to generate a rigorously pinned, long-term compendium. During parsing, the system successfully flagged a minor omission: an unlisted GitHub-hosted dependency which was effortlessly rectified using the same resolution principles demonstrated in the Psychological Science audit [37].

5.1 Prospective Workflow

The prospective use of the framework differs from retrospective archaeology in the choice of the snapshot anchor: instead of a historical date, the anchor is the date on which the contemporary analysis is finalised. The workflow proceeds in five steps.

1. **Scan.** The project directory is scanned as described in Section 3.2. The scan produces the root package set P_0 .

2. **Anchor.** The current date is recorded as t_s . The resolver is invoked with t_s set to the present, so that the resolved versions correspond to the state of CRAN, Bioconductor, and GitHub on the day the analysis is frozen.
3. **Resolve.** The resolver emits a manifest listing the exact version of every node in the transitive closure, including the R interpreter version and the system requirements.
4. **Dockerize.** The manifest is passed to `dockerize()`, which generates a Dockerfile and an orchestration script in the target directory.
5. **Build and deposit.** The image is built, the analysis is re-executed inside the container to confirm output equivalence, and the manifest, Dockerfile, and (optionally) the built image are deposited alongside the code and data.

This workflow differs from using `renv` or Docker alone in two respects. First, whereas `renv` records the versions currently installed on the author's machine, which may include packages unrelated to the analysis or versions that are already stale relative to CRAN, the prospective workflow resolves the *complete* transitive closure against the frozen snapshot date, producing a minimal and internally consistent environment. Second, whereas a hand-written Dockerfile requires the author to enumerate OS-level dependencies manually, `dockerize()` derives them from the resolved manifest, reducing the risk of omission [38].

6. Discussion and Limitations

The pursuit of absolute computational reproducibility is an intricate challenge devoid of universal panaceas. Every methodological intervention carries inherent compromises. While the presented architecture excels at freezing and reviving internal software states, it encounters hard limitations when dealing with volatile external dependencies that resist containerization. Prominent examples include packages bridging web-based APIs, such as `googleLanguageR`, `rtweet`, and `STRINGdb`. While the software wrappers themselves are easily containerized, the remote databases they query are beyond user control. Consequently, our framework cannot resolve irreproducibility caused by deprecated external APIs. For tasks demanding web-based data harvesting, we strongly advocate for statically caching the retrieved datasets and disseminating them alongside the code. Conversely, for algorithmic processing reliant on external servers, researchers should pivot to self-contained, offline alternatives [39].

Three further limitations warrant explicit statement. First, when the resolver selects a version satisfying the conjunction of constraints present in the graph, it does not guarantee that this version is the one the original author actually used; the original author may have installed a newer or older version than the constraint required. In such cases, the reconstructed environment is *installable* and *historically plausible*, but not provably *historically identical* to the original. Second, when a package has been removed from all archives (CRAN, Bioconductor, and the GitHub CRAN mirror), the resolver cannot recover it and reports the node as unresolved; the resulting environment may still build if the package is only a Suggests dependency, but will fail if the package is in Imports or Depends. Third, when a package's `SystemRequirements` field is absent or incomplete a common occurrence in pre-2010 packages the resolver cannot infer the required OS libraries, and the reconstructed image may compile successfully on the build host while failing on a different host with a different system-library configuration. In each of these three cases, the framework produces an environment that is *reproducible in the weak sense* (it builds and runs) without being *reproducible in the strong sense* (it provably matches the original).

Additionally, the operational execution of this reconstruction engine is markedly slower than alternative lightweight management tools, primarily because every underlying library must be compiled directly from source code. This paradigm deliberately sacrifices rapid execution speed in favor of absolute historical accuracy, deep backward compatibility, and strict independence from corporate repository backups (like MRAN).

Critical constraints of the system include: (1) an inability to interface with R distributions predating version 1.3.1; (2) a strict reliance on Debian/Ubuntu base images, thereby precluding the integration of Windows-exclusive binaries (e.g., WinBUGS); (3) the `dockerize()` module lacks the capacity to locally cache OS-level Debian packages; (4) libraries exhibiting highly volatile or shifting OS dependencies may yield inaccurate resolution logs; (5) extreme historical reconstructions mandate the aggressive caching of both the language source and its packages; and (6) the framework ultimately operates on the assumption that GitHub, Bioconductor, CRAN, and Docker infrastructure will not suffer total systemic erasure. This final vulnerability can be partially mitigated by maintaining localized backups of the fully compiled Docker images, but no software framework can guarantee indefinite reproducibility, because every layer of the stack the base operating system, the R interpreter, the package sources, the container registry, and the hardware is subject to change or removal. The framework described here improves the probability that a legacy analysis can be re-executed, and reduces the number of manual interventions required to do so, but it does not and cannot guarantee that the re-execution will succeed at an arbitrary future date [40].

7. Conclusion

This manuscript establishes an architecture for temporal dependency archaeology, providing researchers with the tools to retrospectively rebuild complex R-centric computational workspaces tailored for verifiable text mining workflows. Across the six cases reported in Section 4, the framework resolved 275 of 275 transitive dependencies, produced buildable Docker images in every case, and reproduced the published outputs of five cases exactly and one case partially. These results demonstrate that date-anchored transitive resolution is feasible for legacy R pipelines spanning from 2003 to 2019, including packages removed from CRAN more than a decade ago.

The prospective workflow described in Section 5 has been demonstrated on a single contemporary manuscript, where it identified an undeclared GitHub dependency that manual inspection had missed; we present this as an illustration rather than a general validation, and further prospective evaluations across a larger corpus of contemporary studies are needed to establish its reliability.

The entire toolkit is distributed under open-source licenses and architected to interface smoothly with prevailing reproducibility frameworks, including containerization pipelines, `sessionInfo()`, and `renv`. As algorithmic text analysis increasingly permeates the humanities and social sciences, the deployment of preservation infrastructure that improves, though cannot guarantee, the long-term executability of published analyses becomes increasingly valuable to the scientific community.

References

- [1] C. Gandrud. *Reproducible Research with R and RStudio*. CRC Press, 2020.
- [2] L. A. Barba. Terminologies for reproducible research, 2018. arXiv preprint arXiv:1802.03311.
- [3] R. D. Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011.

- [4] V. Stodden, F. Leisch, and R. D. Peng, editors. *Implementing Reproducible Research*. CRC Press, 2014.
- [5] J. F. Claerbout and M. Karrenbach. Electronic documents give reproducible research a new meaning. In *SEG Technical Program Expanded Abstracts*, pages 601–604, 1992.
- [6] D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram, and V. Stodden. Reproducible research in computational harmonic analysis. *Comput. Sci. Eng.*, 11(1):8–18, 2009.
- [7] K. Ushey. renv: Project environments, 2024. URL <https://rstudio.github.io/renv/>. R package version 1.0.0.
- [8] Groundhog. Groundhog: Manage package versions, 2024. URL <https://groundhogr.com/>. R package.
- [9] C. Boettiger. An introduction to Docker for reproducible research. *ACM SIGOPS Oper. Syst. Rev.*, 49(1):71–79, 2015.
- [10] C. Boettiger and D. Eddelbuettel. An introduction to Rocker: Docker containers for R. *The R Journal*, 9(2):527–536, 2017.
- [11] B. Matthews, A. Shaon, J. Bicarregui, and C. Jones. A framework for software preservation. *Int. J. Digital Curation*, 5(1):91–105, 2010.
- [12] D. Nüst, V. Sochat, B. Marwick, S. Eglen, T. Head, and T. Hirst. Containerization for reproducible research. In *Proc. ACM SIGMOD*, 2018.
- [13] S. Kumar, S. K. Singh, and N. Aggarwal. Sustainable data dependency resolution architectural framework to achieve energy efficiency using speculative parallelization. In *Proc. 2023 3rd Int. Conf. Innovative Sustainable Comput. Technol. (CISCT)*, pages 1–6. IEEE, 2023.
- [14] S. B. Verma, B. Pandey, and B. K. Gupta. Containerization and its architectures: A study. *ADCAIJ: Adv. Distrib. Comput. Artif. Intell. J.*, 11(4):395–409, 2022.
- [15] G. Csárdi. pak: Another approach to package installation, 2024. URL <https://pak.r-lib.org/>. R package.
- [16] P. Abate, R. Di Cosmo, G. Gousios, and S. Zacchiroli. Dependency solving is still hard, but we are getting better at it. In *Proc. 2020 IEEE 27th Int. Conf. Softw. Anal. Evol. Reeng. (SANER)*, pages 547–551. IEEE, 2020.
- [17] D. Temple Lang. Constraints on the CRAN package dependency graph. Technical report, Technical Report, 2007.
- [18] J. Decourselle, T. Mens, and A. Cleve. A survey of software package managers. Technical report, Technical Report, 2017.
- [19] Ka Madhu. Leveraging Kubernetes for self-healing microservices: A comparative analysis and future directions. In *Proc. IEEE Int. Conf. Emerg. Trends Inf. Technol.*, 2025. doi: 10.1109/ICoEIT63558.2025.11211721.

- [20] T. Anondi. Feedback-driven AI workload emulation for big data cloud systems. In *Proc. IEEE Int. Conf. Recent Adv. Artif. Intell.*, 2025. doi: 10.1109/RAAI67517.2025.11423395.
- [21] O. Bentaleb, A. S. Belloum, A. Sebaa, and A. El-Maouhab. Containerization technologies: Taxonomies, applications and challenges. *J. Supercomput.*, 78(1):1144–1181, 2022.
- [22] R. Gentleman and D. Temple Lang. Statistical analyses and reproducible research. Technical report, Bioconductor Project Working Paper, 2004.
- [23] V. Stodden, M. McNutt, D. H. Bailey, E. Deelman, Y. Gil, B. Hanson, M. A. Heroux, J. P. A. Ioannidis, and M. Taufer. Enhancing reproducibility for computational methods. *Science*, 354(6317): 1240–1241, 2016.
- [24] Akshesh Shah. Efficient cross-lingual transfer for language models. In *Proc. IEEE Int. Conf. Emerg. Res. Electron.*, 2025. doi: 10.1109/ICERECT65215.2025.11101933.
- [25] R. Di Cosmo and S. Zacchiroli. Software heritage: Why and how to preserve software source code. In *Proc. iPRES*, 2017.
- [26] C. P. Gomes. Computational sustainability: Computational methods for a sustainable environment, economy, and society. In *Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2009 Symposium*, pages 27–40. National Academies Press, 2010.
- [27] M. E. Smith. Braudel’s temporal rhythms and chronology theory in archaeology. In *Archaeology, Annales, and Ethnohistory*, pages 23–34. Cambridge University Press, 1992.
- [28] S. J. Holdaway and L. Wandsnider. Temporal scales and archaeological landscapes from the eastern desert of australia and intermontane north america. In *Confronting Scale in Archaeology: Issues of Theory and Practice*, pages 183–202. Springer US, 2006.
- [29] V. Kuruppathukattil. Optimizing traffic and latency in peer-to-peer networks through advanced replication and polling algorithms, 2025. Preprint.
- [30] Vimal Teja Manne. Atomic rails for money-in-motion: A resilient instant payment gateway with ID resolution, retries, and guaranteed reversals, 2025. Preprints.
- [31] A. Veluru. Bayesian optimization of hyperparameters for Rainbow DQN in the CartPole-v1 environment, 2025. Preprint.
- [32] Deepak Jain. Optimizing spatial data processing with CQRS for scalable and efficient business intelligence analytics, 2025. Preprint.
- [33] A. Mukherjee. Ai-driven fraud detection in payments using transformer-based behavioral modeling, 2025. Preprint.
- [34] A. Bhople. A unified multi-modal mixture-of-experts model for integrated representation learning in pharmaceutical sciences, 2025. Preprint.
- [35] S. Shivankar. Deep learning-based calibration pattern detection for enhanced RGB-LiDAR sensor fusion, 2025. TechRxiv.

- [36] M. Pratap. Maximizing business growth: The economic impact of Salesforce-driven CRM intelligence. In *Proc. IEEE Int. Conf. Next Gener. Comput.*, 2025. doi: 10.1109/ICNGCS64900.2025.11183293.
- [37] K. Bhatt. Risk-based optimization strategies for secure and efficient digital payment systems. In *Proc. IEEE Int. Conf. Innov. Inf. Technol.*, 2025. doi: 10.1109/I2ITCON65200.2025.11210602.
- [38] M. Amin. Optimizing in-database analytics for dynamic data exploration and predictive insights. In *Proc. Int. Conf. Adv. Comput.*, 2025. doi: 10.2991/978-94-6239-674-6_6.
- [39] K. Kumar. Enhancing web accessibility for mathematics through semantic enrichment of MathML. In *Proc. IEEE Int. Conf. Commun. Technol. Data Comput.*, 2025. doi: 10.1109/ICCTDC64446.2025.11158762.
- [40] T. Mendhey. Transforming workforce potential: Integrating training, competency systems, and emerging technologies for organizational excellence. *J. Res. Manage. Commerce*, 6(1), 2025. doi: 10.46632/rmc/6/1/1.