Why Risk it, When You Can {rix} it: A Tutorial for Computational Reproducibility Focused

on Simulation Studies

Felipe Fontana Vieira[1], Jason Geller[2], and Bruno Rodrigues[3]

[1]Department of Data Analysis, Ghent University

[2]Department of Psychology and Neuroscienc, Boston College

[3]Statistics and Data Strategy Departments, Ministry of Research and Higher Education,

Luxembourg

Author Note

Felipe Fontana Vieira ⓘ https://orcid.org/0009-0006-0949-6569

Jason Geller ⓘ https://orcid.org/0000-0002-7459-4505

Bruno Rodrigues ⓘ https://orcid.org/0000-0002-3211-3689

Correspondence concerning this article should be addressed to Felipe Fontana Vieira, Email: felipe.fontanavieira@ugent.be

Abstract

Computational reproducibility remains limited in psychological research, despite widespread norms for sharing data and analysis code. One reason is that reproducibility exists on a continuum, ranging from partial transparency—such as providing scripts or software version numbers—to fully executable research compendia that regenerate all results from raw code. In this article, we introduce Nix and the {rix} R package as a practical framework for achieving full computational reproducibility in simulation-based research. We provide a step-by-step tutorial demonstrating how {rix} can be used to define, build, and share isolated, project-specific software environments that precisely capture R versions, package dependencies, system libraries, and integrated development environments. We further illustrate this workflow by reproducing a complete manuscript using Quarto and the {apaquarto} extension, showing how analyses, figures, and text can be regenerated in a single, executable pipeline. Together, these tools lower the technical barrier to robust, end-to-end reproducibility and offer a scalable solution for simulation studies and methodological research in psychology and related fields.

Keywords: reproducibility, Nix, simulation studies, R, computational methods

Why Risk it, When You Can {rix} it: A Tutorial for Computational Reproducibility Focused

on Simulation Studies

Psychological science is in the midst of a credibility revolution, which has prompted substantial progress in how research is conducted and evaluated (Vazire, 2018). Yet, despite notable progress, a key cornerstone of science, reproducibility (i.e., the ability to precisely reproduce the results of a study or studies based on provided data, code, materials, and software/hardware) remains limited (Hardwicke et al., 2020). Hence, ensuring reproducibility remains an open and pressing challenge for psychological science.

Addressing this gap is complicated by the fact that reproducibility is not a binary feature but instead exists along a continuum (Peng, 2011). At the lower end, reproducibility may be interpreted as sharing only a manuscript. Further along the spectrum, it may involve providing partial code, complete analysis scripts, or publicly accessible datasets. At the highest level, reproducibility entails documenting a fully specified computational environment that allows others to recreate identical results—from raw data to final manuscript output—with minimal friction. As a result, researchers may implicitly target different points on this continuum, and efforts to improve reproducibility can diverge substantially in both goals and implementation.

Open science initiatives have made considerable progress in encouraging movement along this spectrum. For example, journals now offer open-science badges (Kidwell et al., 2016), best practices have been developed to make data sharing routine (Levenstein & Lyle, 2018), and platforms such as the Open Science Framework (OSF) provide infrastructure for storing and sharing research materials (Nosek et al., 2015). However, these efforts primarily target the lower to middle portions of the continuum, emphasizing *what* is shared rather than *how* shared materials can be executed in practice.

Data and code are never fully self-sufficient to reproduce a set of findings (Epskamp, 2019; Peikert & Brandmaier, 2021; Wiebels & Moreau, 2021). Assuming the data and code are error-free, reproducibility depends on a hierarchy of software components—collectively referred to as *dependencies*—including the programming language version, the packages used in the

analysis, and the system libraries on which those packages rely. When these dependencies differ from those used in the original analysis, code may fail, behave inconsistently across machines, or yield conflicting numerical results (Baker et al., 2024; Glatard et al., 2015; Hodges et al., 2023; Nosek et al., 2022). These issues are particularly acute for simulation studies, which rely on complex codebases, versioned dependencies, and intricate software configurations (Luijken et al., 2024; Siepe et al., 2024).

To make this concrete, we use *computational environment* to refer to the complete software context required for an analysis to run successfully—the programming language version, package versions, system libraries, and operating system (Rodrigues, 2023; Rodrigues & Baumann, 2026). We define *computational environment reproducibility* as the ability to reconstruct this entire set of software dependencies on any machine and at any future time, such that executing the same code yields the same numerical results. Empirical assessments show that current practice falls short of this ideal. Siepe et al. (2024) report that nearly two-thirds of simulation studies in psychology provide no accompanying code, and among those that do, documentation of the computational environment is rarely included. This gap is consequential: simulation studies inform methodological recommendations, meaning that insufficient reproducibility undermines confidence in those recommendations (Luijken et al., 2024; White et al., 2024).

Arguably, these challenges persist because researchers must navigate a fragmented landscape of solutions, each addressing only part of the problem. Package-level managers such as {renv} (Ushey, 2024) and {groundhog} (Simonsohn, 2020) stabilize R package versions but do not manage the R interpreter itself or the system-level libraries those packages depend on. Workflow orchestration tools such as {targets} (Landau, 2021) and Make (Feldman, 1979) support reproducibility in a different sense: they specify the structure of an analysis by formalizing the order in which steps should run and by tracking dependencies among intermediate results. These tools clarify *how* an analysis proceeds, but they assume that the software stack required to run each step is already stable. Containerization tools such as Docker, including R-focused implementations like Rocker (Boettiger, 2015; Boettiger & Eddelbuettel, 2017) offer a

more comprehensive approach by bundling the full environment (i.e., operating system, system libraries, interpreter versions, and packages) into a single executable image. Yet their use requires familiarity with Linux system administration, and even containerization may suffer from temporal drift when Dockerfiles rely on mutable upstream repositories (Malka et al., 2024). For a detailed comparison of these tools and their limitations, see Rodrigues and Baumann (2026). Researchers thus face a difficult choice between solutions that are accessible but incomplete or approaches that are powerful but demand technical expertise.

In this article, therefore, we focus specifically on computational environment reproducibility as the foundation upon which other reproducibility practices depend. For that, we introduce Nix (Dolstra et al., 2004), a functional software[1] ecosystem designed to make software installation deterministic, and {rix} (Rodrigues & Baumann, 2025), an R interface that allows researchers to use Nix without needing deep knowledge of its underlying language or infrastructure. Our main objective with the tutorial is not to introduce a specific workflow orchestration system or to prescribe a particular analytic structure. Instead, we aim to show what Nix and {rix} are and how they can establish a stable, cross-platform environment within which any simulation study—whether organized in simple, documented script sequences (e.g., .R files that `source()` others), through more formal orchestration tools (e.g., {targets}) or embedded as code chunks in `.Rmd` or `.qmd`—can be executed reliably.

We illustrate these ideas through a reproducible simulation study conducted in R, culminating in this automated APA-formatted manuscript generated with `apaquarto` (Schneider, 2024). Although the example centers on R because of its prominence in psychological methodology, the principles underlying environment reproducibility with {rix} apply equally to other languages, including Python and Julia, and to different development environments such as RStudio, VS Code, Emacs, or Positron. At the end of the article, we also briefly introduce {rixpress} (Rodrigues, 2025), a workflow orchestration tool that builds on Nix to coordinate

---

[1] *Functional* is used in the sense of functional programming (pure functions, immutability, explicit inputs), not in the colloquial sense of *working*.

pipelines across R, Python, and Julia.

A Practical Example: Setting up a Reproducible Simulation Study with {rix}

Imagine you have just been awarded a grant to conduct a large-scale simulation study. The study is designed to evaluate the performance of a statistical estimator under varying data-generating conditions (see Appendix A for full technical details). This tutorial is organized around this scenario. We use this example to ground our discussion into a typical methods section, but readers can follow the tutorial without engaging deeply with the simulation itself.

In practice, simulation studies are typically organized into multiple component files, each corresponding to a distinct analytical stage, a modular structure that facilitates development and debugging. In our case, the simulation is organized into five sequential scripts: data generation (`01_data_generation.R`), model specification (`02_models.R`), simulation execution (`03_run_simulation.R`), performance metric calculation (`04_performance_metrics.R`), and results visualization (`05_plots.R`). However, because our focus is on the reproducibility of the *entire manuscript*, we embed all code directly within this document as executable chunks in a single `.qmd` file. When rendered, the simulation runs from start to finish, producing results and figures automatically. This approach would be impractical for many real-world simulation studies, which are often too computationally intensive. We return to this trade-off later in the tutorial.

Now suppose a researcher attempts to reproduce the simulation results reported in the article. What might prevent them from obtaining identical outcomes? The natural first concern is the version of packages. Installing R packages at a later time may lead to errors if functions have been renamed or deprecated (e.g., `lavaan:::lav_utils_get_ancestors` renamed to `lavaan:::lav_graph_get_ancestors`), or to subtly different results due to changes in default settings (e.g., `stringsAsFactors` defaulting to FALSE as of R 4.0) or numerical implementations. Beyond package versioning, many packages rely on system-level libraries that must be installed separately from R. Our simulation illustrates this dependency structure directly: the {rvinecopulib} package interfaces with a C++ backend and links against external libraries such as Boost, Eigen, and RcppThread (Nagler & Vatter, 2025).

The R language version introduces another layer of dependency. Code written for R 4.0 may rely on syntax or functionality that is unavailable in earlier versions (e.g., the native pipe `|>` introduced in R 4.1). More subtly, changes to R's random number generation across major versions mean that identical code executed with the same seed can nevertheless produce different random sequences (Ottoboni & Stark, 2018). For simulation studies, where specific random draws often underpin reported results, this version sensitivity is consequential.

Finally, when analyses are embedded in a literate programming workflow (i.e., documents that combine narrative text and executable code; dynamic document generation) additional layers of software dependencies arise. For example, rendering R Markdown (`.rmd`) or Quarto documents (`.qmd`) requires both a document conversion tool (e.g., Pandoc, which converts `.rmd` or `.qmd` files into formats such as PDF or HTML) and a typesetting system such as a LaTeX distribution or Typst. Each of these components introduces its own versioning constraints and platform-specific installation requirements. Taken together, these layers highlight that reproducibility depends not only on code and data, but also on the broader computational environment in which analyses are executed.

Nix and {rix}: A Comprehensive Solution

A potential solution to this problem is Nix (Dolstra et al., 2004), a software ecosystem whose primary concern is reproducible, declarative builds. To achieve this Nix is a programming language, a build tool and a package manager. This article focuses mostly on Nix the package manager. Most package managers (think of Apple's or Android's app stores) are imperative: they modify a system's state as they install or update software. Nix, in contrast, treats build instructions and dependencies as immutable, enforcing reproducible, declarative, and isolated environments across platforms. This allows researchers to specify exactly which versions of programming languages, packages, and system libraries an analysis requires, and to recreate that environment reliably on any machine.

Unlike familiar tools such as `install.packages()` in R, `apt-get` on Linux, or `uv` in Python—which typically manage only a single layer of the software stack—Nix handles language

versions, package versions, and system-level dependencies within a single framework (Rodrigues & Baumann, 2025). Rather than installing software into shared system directories, Nix builds each environment as an explicit, self-contained specification. As a result, multiple environments can coexist without conflict, and analyses can be rerun months or years later under identical computational conditions.

This unified approach directly addresses the fragmented landscape described above. Where researchers would otherwise need to coordinate separate tools for package management, interpreter versions, and system dependencies, Nix brings all three together within a single declarative model, lowering the barrier to fully reproducible computational workflows.

Core Principles

Rather than installing software into global directories (e.g., `/usr/lib`), Nix places every package in its own directory under `/nix/store`. Each package path contains a cryptographic hash representing its precise inputs—source code, dependencies, and build instructions. Because these paths are content-addressed, multiple versions of the same software can coexist without conflict. A researcher can, for example, maintain projects requiring R 4.1.0 and R 4.3.3 side by side, or use different package versions across analyses, switching between them seamlessly (Rodrigues & Baumann, 2025).

The Nix ecosystem is built around nixpkgs, a version-controlled repository comprising more than 120,000 packages, including nearly all of CRAN and Bioconductor. By pinning a specific commit or date, researchers freeze the entire software stack (i.e., R itself, R packages, and all system libraries) at the time of this writing. This eliminates the system-dependency problems that tools like {renv} cannot address (Rodrigues & Baumann, 2025). This architecture also ensures stability over time. Empirical work has shown strong rebuildability and reproducibility rates for historical nixpkgs snapshots (Rodrigues & Baumann, 2026). Combined with binary caches, which often allow environments to materialize in seconds, Nix becomes practical for interactive research workflows (Rodrigues & Baumann, 2025).

The {rix} Package: R Interface to Nix

As previously stated, Nix is also a functional programming language. Because the Nix package manager is declarative, it requires expressions written in this language to install software. However, since Nix is a functional programming language unfamiliar to most researchers, we recommend using {rix} to lower this barrier. The {rix} package provides an R-native interface: a single call to `rix()` generates complete Nix configurations from standard R syntax, specifying R versions, CRAN packages, system libraries, and even Python or Julia components when required. Users never need to read or write Nix code directly, as {rix} translates automatically (Rodrigues & Baumann, 2025).

A key feature of {rix} is its integration with rstats-on-nix, a community-maintained fork offering daily CRAN snapshots and weekly tested environments on Linux and macOS. Researchers can request, for example, `rix(date = "2024-12-14")` to obtain a validated and reproducible environment without manually assessing compatibility. After the configuration is generated, `nix_build()` instantiates the environment, and binary caches typically allow this to complete within seconds (Rodrigues & Baumann, 2025).

Although Nix is capable of replacing tools like Docker for isolation or {renv} for package management, it does not require an all-or-nothing transition. Researchers can adopt it gradually and use it alongside familiar tooling. For instance, by building Docker images with Nix, converting existing {renv} lockfiles, or running {targets} pipelines within a Nix-defined environment (Rodrigues & Baumann, 2025). This allows Nix to strengthen reproducibility while preserving established workflows. We will come back to this after the tutorial.

Step 1: Installing Nix and {rix}

Before proceeding, both Nix and the {rix} R package need to be installed. Installation procedures differ across operating systems (Windows via WSL2, Linux, and macOS), and detailed, up-to-date instructions are maintained in the official {rix} documentation:

- **Linux and Windows (WSL2)**: https: //docs.ropensci.org/rix/articles/b1-setting-up-and-using-rix-on-linux-and-windows.html

- **macOS**: https://docs.ropensci.org/rix/articles/b2-setting-up-and-using-rix-on-macos.html

Once Nix is installed[2], there are two ways to access {rix}, depending on whether R is already installed on your system. In this tutorial, we proceed as if R was already installed[3] (Listing 1):

Listing 1 Installing {rix} from CRAN or developmental version

```r
# CRAN version
install.packages("rix")
# Development version
install.packages(
  "rix",
  repos = c(
    "https://ropensci.r-universe.dev"
  )
)
```

Step 2: Specifying the Computational Environment

After that, we need to establish a reproducible environment by creating a script that will generate the environment specification. We recommend creating a file named `generate-env.R` (or similar) in the project directory. This script will use the `rix()` function from the {rix} package to produce a `default.nix` file—a declarative specification that precisely defines all software dependencies required for the project.

In our case, where we use literate programming for generating the manuscript, we implement the following environment specification, which can be found on the GitHub repository as a file named `gen-env.R` (Listing 2):

---

[2] It is worth noting that {rix} can generate Nix expressions even without Nix installed on your system—you can write a `default.nix` file without Nix, but you cannot build or enter the resulting environment unless Nix is installed (Rodrigues & Baumann, 2025).

[3] We, however, recommend uninstalling your local R and letting Nix manage R, R packages, and other tools entirely. This approach avoids potential conflicts between system-installed and Nix-managed software, an issue we will illustrate later in this tutorial. See https://docs.ropensci.org/rix/articles/setting-up-linux-windows.html#case-1-you-dont-have-r-installed-and-wish-to-install-it-using-nix-as-well for more details.

Listing 2 Environment specification for the manuscript using rix()

```r
library(rix)

rix(
  date = "2026-01-14",
  r_pkgs = c(
    "rix", "quarto", "knitr", "marginaleffects",
    "simhelpers", "ggplot2", "doParallel", "doRNG", "cowplot",
    "dplyr", "svglite", "rvinecopulib"
  ),
  system_pkgs = c("quarto"),
  tex_pkgs = c("amsmath", "ninecolors", "apa7", "scalerel",
    "threeparttable", "threeparttablex", "endfloat", "environ",
    "multirow", "tcolorbox", "pdfcol", "tikzfill", "fontawesome5",
    "framed", "newtx", "fontaxes", "xstring", "wrapfig", "tabularray",
    "siunitx", "fvextra", "geometry", "setspace", "fancyvrb",
    "anyfontsize"
  ),
  ide = "rstudio",
  project_path = ".",
  overwrite = TRUE
)
```

Thus, note that we have more than just the R packages specified for the simulation scripts. This happens because we also included what is needed for the manuscript generation, not solely for the simulation code. In Appendix B, we mention more specifically the reasons for adding each package in `r_pkgs()` and `tex_pkgs()`. For now, we focus more on clarifying the different arguments for the `rix()` function.

The Environment Generation Script

The `rix()` function[4] constructs this specification through a series of parameters that collectively describe the computational environment. Each parameter serves a distinct purpose in defining the environment's characteristics.

---

[4] For an overarching information on the function `rix()`, we suggest the following {rix} documentation: https://docs.ropensci.org/rix/articles/project-environments.html

Specifying the R version. Researchers must first determine which version of R to use. This can be accomplished in two ways: The `r_ver` argument accepts an exact version string (e.g., "4.3.3") or special designations such as "latest-upstream" for the most recent stable release. Alternatively, the `date` argument specifies a particular date (e.g., "2024-11-15"), which ensures that R and all packages correspond to the versions available on that date. The date-based approach is generally preferable for reproducibility, as it captures a complete snapshot of the R ecosystem at a single point in time. For this tutorial, as shown on top, we use the `date` parameter to ensure temporal consistency across all software components (Rodrigues & Baumann, 2025) (see {rix} documentation for more: https://docs.ropensci.org/rix/articles/project-environments.html).

Declaring R package dependencies. The `r_pkgs` argument accepts a character vector listing all required R packages by their CRAN names. These packages will be installed from the version repository corresponding to the specified date or R version. It is important to list all packages that the analysis will load directly; dependencies of these packages are automatically resolved by Nix. For packages requiring specific versions not corresponding to the chosen date, researchers can specify exact versions using the syntax `"packagename@version"` (e.g., `"ggplot2@2.2.1"`). For packages available only on GitHub or other Git repositories, the `git_pkgs` argument accepts a list structure containing repository URLs and specific commit hashes. For example:

Listing 3 Example for the git_pkgs argument

```
git_pkgs = list(
  package_name = "marginaleffects",
  repo_url = "https://github.com/vincentarelbundock/marginaleffects",
  commit = "304bff91dc31ae28b227a8485bfa4f7bdc86d625"
)
```

This ensures that exact development versions are obtained (Rodrigues & Baumann, 2025). For our simulation study, all packages were used with their CRAN versions (see {rix} documentation for more details: https://docs.ropensci.org/rix/articles/installing-r-packages.html).

Including system-level dependencies. Many R-based workflows require tools beyond R packages. The `system_pkgs` parameter specifies system-level software such as Quarto for document generation, Git for version control, or Pandoc for document conversion. Critically, we include Quarto as a system package because this tutorial demonstrates full computational reproducibility—not merely of the simulation code, but of the complete manuscript itself. Our manuscript uses the `apaquarto` extension for APA formatting, stored in the project's `_extensions/` directory (Rodrigues & Baumann, 2025) (see {rix} documentation for more: https://docs.ropensci.org/rix/articles/installing-system-tools.html).

Specifying LaTeX packages. The `tex_pkgs` parameter specifies LaTeX packages needed for PDF compilation. When any packages are listed, Nix automatically includes a minimal TexLive distribution (`scheme-small`) as a base, to which the specified packages are added. Determining the required LaTeX packages may involve some trial and error—Quarto's error messages during, for example, the PDF rendering indicate which packages are missing, and these can then be added to `tex_pkgs` (see {rix} documentation for more: https://docs.ropensci.org/rix/articles/installing-system-tools.html). The tex packages included in the above code offer a nice starting point for researchers wanting to create manuscripts using LaTeX.

Configuring the development environment. The `ide` parameter controls whether an integrated development environment (IDE) is included in the Nix environment, allowing users to interactively develop and run code within their editor of choice. When `ide` is specified, the project can be opened directly in the corresponding IDE, with all dependencies provided by the Nix environment. For example, setting `ide = "rstudio"` installs a project-specific version of RStudio inside the Nix environment. This is required for RStudio because, unlike most other editors, it cannot attach to an external Nix shell unless it is itself installed via Nix. On macOS, RStudio is only available through Nix for R versions 4.4.3 or later (or environments dated 2025-02-28 or later); for earlier R versions, alternative editors must be used. Other supported IDEs include Positron (`ide = "positron"`), Visual Studio Code (`ide = "code"`), and

command-line interfaces such as Radian (`ide = "radian"`). These tools may either be installed

directly within the Nix environment using the `ide` parameter, or users may rely on an existing

system installation by setting `ide = "none"` (or `ide = "other"`) and configuring `direnv` to

automatically load the Nix environment when the project directory is opened[5]. All IDEs installed

via Nix are project-specific and do not interfere with system-wide installations. Detailed

configuration instructions are provided in the {rix} documentation:

https://docs.ropensci.org/rix/articles/configuring-ide.html

 Setting file output parameters. The `project_path` parameter indicates where the

`default.nix` file should be written ("." denotes the current directory), while `overwrite`

controls whether an existing file should be replaced. Adding to this, setting `print = TRUE`,

which is another argument, displays the generated specification in the console for immediate

verification (Rodrigues & Baumann, 2025).

 Multi-language environment support. While this tutorial focuses on R, researchers

working across multiple programming languages can include Python or Julia in their

environments. The `py_conf` parameter accepts a list specifying a Python version and required

packages (Listing 4). Similarly, `jl_conf` enables Julia package installation. This capability is

particularly useful, for example, for projects requiring statistical computing in R alongside

machine learning pipelines in Python or numerical optimization in Julia (Rodrigues & Baumann,

2025) (see {rix} documentation for more:

https://docs.ropensci.org/rix/articles/installing-r-packages.html).

Listing 4 Including Python packages

```
py_conf = list(py_version = "3.12", py_pkgs = c("polars", "pandas"))
```

———

[5] `direnv` is a lightweight utility that integrates with the user's shell and automatically loads project-specific
environment settings when navigating into a directory (via a `.envrc` file), and unloads them when leaving. This
makes environment activation implicit and reduces the risk of running analyses in the wrong software context.

Generating the Environment Specification

After defining the computational environment, the `rix()` function must be executed to generate the `default.nix` file. This can be done interactively by running `rix()` in an R console (Listing 2). The resulting `default.nix` file serves as the complete environment specification and contains all information required to recreate the project in a fully reproducible manner.

Step 3: Building and Using the Reproducible Environment

Once Step 2 is complete, build the reproducible environment by navigating to the study directory in a terminal. You may use either the integrated Terminal in RStudio (Tools → Terminal → New Terminal) or an external system terminal from which you are running the Nix project. From the study directory, run the following command (Listing 5):

Listing 5 Building the Nix environment

```
user@computer Why-risk-it-when-you-can-rix-it % nix-build
```

The expected output should look similar to (Listing 6):

Listing 6 Expected output from `nix-build`

```
unpacking 'https://github.com/rstats-on-nix/nixpkgs/archive/2025-08-25.tar.gz'
  into the Git cache...
warning: ignoring untrusted substituter...
warning: ignoring the client-specified setting...
/nix/store/qa7fq20m2f94szsnqzciwv8h4n81w43v-nix-shell
```

This command builds the environment according to the specification. The first execution will download and install all required packages, which may take a few minutes depending on network speed and system resources. Subsequent builds use cached packages and complete in seconds. Upon successful completion, a path to the constructed environment in the Nix store is printed (here, `/nix/store/qa7fq20m2f94szsnqzciwv8h4n81w43v-nix-shell`), and a symbolic link named `result` appears in the project directory pointing to this location.

Note that the warnings indicate that you are not configured as a trusted user, so Nix cannot use the rstats-on-nix binary cache and will instead compile packages from source, which is slower. To enable binary caching, install the cachix client and configure the rstats-on-nix cache. See https://docs.ropensci.org/rix/articles/binary-cache.html for instructions.

To activate the environment, run (Listing 7):

Listing 7 Activating the Nix environment

```
user@computer Why-risk-it-when-you-can-rix-it % nix-shell
```

The expected output (if you have configured yourself as a trusted user, otherwise the same warnings will appear) should look similar to (Listing 8):

Listing 8 Expected output from `nix-shell`

```
unpacking 'https://flakehub.com/f/DeterminateSystems/nixpkgs-weekly/...'
  into the Git cache...
[nix-shell:~/AMPPS/Why-risk-it-when-you-can-rix-it]$
```

This command drops the user into a shell where all specified packages and tools are available. The shell prompt changes to indicate that a Nix environment is active (here, `[nix-shell:~/Desktop/AMPPS/Why-risk-it-when-you-can-rix-it]$`). To verify that R is being provided by Nix rather than a system installation, run `which R`. This should return a path within `/nix/store/`. Moreover, from within the Nix shell, users can launch their IDE by typing its name (e.g., `rstudio` or `positron`), which opens the IDE with the Nix environment active (Listing 9)[6]

---

[6] Please note that activating an RStudio instance via Nix does **not** automatically open the specific project directory you are working in. We therefore recommend creating an **RStudio project file** (`.Rproj`) and opening that file when using Nix to ensure that RStudio is correctly associated with the intended project and environment.

Listing 9 Activating RStudio

```
[nix-shell:~/AMPPS/Why-risk-it-when-you-can-rix-it]$ rstudio
```

Reproducing the Complete Manuscript[7]

Within the nix shell, one is able to render the manuscript as follows in the terminal

(Listing 10):

Listing 10 Rendering the manuscript with Quarto

```
[nix-shell:~/AMPPS/Why-risk-it-when-you-can-rix-it]$
  quarto render Manuscript/article.qmd
```

This command executes all code chunks in the manuscript, incorporates results (e.g.,

Table 1) and figures (e.g., Figure 1), and generates a formatted PDF following APA style

guidelines via the apaquarto extension (Schneider, 2024). This extension is saved in the project

repo already. To download this extension for your own work you can install the extension by using

the terminal (Listing 11):

Listing 11 Installing the apaquarto extension

```
user@computer Why-risk-it-when-you-can-rix-it % quarto use template
  wjschne/apaquarto
```

or in the console (Listing 12):

Listing 12 Installing the apaquarto extension from R

```
quarto::quarto_use_template("wjschne/apaquarto")
```

The final document (.docx, .pdf, or .html) is saved directly in the project folder[8]. Because

---

[7] See the {rix} documentation for more: https://docs.ropensci.org/rix/articles/literate-programming.html

[8] Although we use *apaquarto* in this example, many alternative manuscript templates are available, and Nix is agnostic to the specific template employed, provided the necessary extensions are installed.

Quarto is installed as a system-level package in our Nix specification, the rendering occurs entirely within a fully reproducible environment, ensuring consistent output across machines regardless of local software configurations. If desired, the manuscript can also be reproduced interactively by opening the project folder in the user's preferred IDE.

At this point, it is also worth noting that Nix shells do not fully isolate you from your existing system by default (as mentioned in footnote 2). For R users, this has a practical implication: packages installed in your regular R library (outside of Nix) could potentially be loaded when running R from within the Nix environment. The {rix} package addresses this automatically—when you call `rix()`, it also executes `rix_init()`, which creates a project-specific `.Rprofile`. This file configures R to ignore external package libraries and also disables `install.packages()` within the environment. The rationale is straightforward: any new packages should be added to `default.nix` and the environment rebuilt, preserving full reproducibility (Rodrigues & Baumann, 2025). However, for stricter isolation[9] that also prevents access to other system programs not specified in `default.nix`, use the `--pure` flag (Listing 13):

Listing 13 Activating the Nix environment with strict isolation

---

```
nix-shell --pure
```

Reproducing the Simulation and Results

As previously mentioned, researchers may prefer not to use literate programming, or embedding the simulation within a dynamic document may be impractical. In this case, one could still follow the same steps shown thus far focusing only on the .R files while still benefiting from a reproducible computational environment. For example, `03_run_simulation.R` begins by loading the required packages and sourcing other scripts that are needed:

As an illustration, if we were to focus solely on the computational reproducibility of the

---

[9] For example, when preparing this manuscript without the `--pure` flag, `quarto render` worked successfully. However, when using the `--pure` flag, the build failed. Running `quarto check` from within the Nix shell (i.e., `nix-shell --run "quarto check"`) revealed that Quarto was still accessing the system's LaTeX installation (/Library/TeX/texbin) rather than being restricted to only what was specified in `default.nix`.

Listing 14 Code for running simulation

```r
library(marginaleffects)
...

# Source helper functions
source("Simulation_Scripts/01_data_generation.R")
...
```

---

code underlying the simulation and results, the environment specification would be considerably

simpler (Listing 15):

Listing 15 Environment specification for simulation-only computational reproducibility

```r
rix(
  date = "2026-01-14",
  r_pkgs = c(
    "simhelpers", "ggplot2", "doParallel", "doRNG", "cowplot", "dplyr",
    "rvinecopulib", "marginaleffects"
  ),
  ide = "rstudio",
  project_path = ".",
  overwrite = TRUE
)
```

After following Steps 1-3 with this simpler specification, the simulation study may be

reproduced within the Nix shell (Listing 16) as follows:

Listing 16 Running the complete simulation workflow

```
[nix-shell:~/AMPPS/Why-risk-it-when-you-can-rix-it]$
  Rscript Simulation_Scripts/03_run_simulation.R
```

In the same way, we could proceed with `04_performance_metrics.R` (Listing 17),

which loads the simulation results (in `sim_results.rds`) and calculates the performance metrics:

Similarly, `05_plots.R` (Listing 18) uses those saved metrics (in

`performance_summary.rds`) to create the plots:

Listing 17 Running the performance metrics calculation

```
[nix-shell:~/AMPPS/Why-risk-it-when-you-can-rix-it]$
  Rscript Simulation_Scripts/04_performance_metrics.R
```

Listing 18 Generating the visualization plots

```
[nix-shell:~/AMPPS/Why-risk-it-when-you-can-rix-it]$
  Rscript Simulation_Scripts/05_plots.R
```

Therefore, the key advantage of executing within `nix-shell` is that all dependencies (i.e., R version, packages, and system tools) match exactly those specified in `default.nix`. Note, however, that this approach relies on manually running scripts in sequence. It ensures a reproducible environment but does not formalize the workflow itself; dependencies between scripts remain implicit in the code rather than explicitly declared.

Additional Considerations for Advanced Workflows

Thus far, we have presented Nix and {rix} as standalone solutions for computational reproducibility, contrasting initially them with tools like {renv}, Docker, and {targets}. However, these tools are not mutually exclusive, as in many cases, they can complement each other (Rodrigues & Baumann, 2025). Additionally, as highlighted in previous literature (Peikert & Brandmaier, 2021; Piccolo & Frampton, 2016; Siepe et al., 2024), fully reproducible research benefits not only from a stable computational environment but also from explicit workflow orchestration. We therefore briefly introduce {rixpress}, which extends Nix-based reproducibility to formalized, multi-language pipelines. These topics are not covered in depth; our goal is simply to clarify how these tools relate to one another and orient readers toward resources for more advanced use cases.

Workflow Orchestration: {targets} and {rixpress}

Complex simulation studies often benefit from workflow management systems that track dependencies between computational steps, cache intermediate results, and enable selective

re-execution when inputs change. Two complementary approaches exist within the Nix ecosystem.

Using {targets} Within Nix. The {targets} package (Landau, 2021) provides workflow orchestration for R-based projects. To integrate {targets} with Nix, include `targets` in the `r_pkgs` parameter of `rix()` and execute the pipeline within `nix-shell` using `Rscript -e 'targets::tar_make()'`. A shell hook can also be added via the `shell_hook` argument to run the pipeline automatically when entering the Nix shell. This approach is ideal for projects that remain within R and do not require different environments for different pipeline steps (see {rix} documentation: https://docs.ropensci.org/rix/articles/reproducible-pipelines.html).

Using {rixpress} for Multi-Language Pipelines. The {rixpress} package (Rodrigues, 2025), a sister package to {rix}, uses Nix itself as the build automation tool rather than operating within a Nix environment. Each pipeline step becomes a Nix derivation, built in isolation with automatic caching based on content. The key advantage emerges in multi-language workflows: different steps can execute in different Nix-defined environments (e.g., one step using a specific version of R, another using Python, another using Julia). The interface, inspired by {targets}, uses functions like `rxp_r()`, `rxp_py()`, and `rxp_jl()` to define pipeline steps (see {rixpress} documentation: https://docs.ropensci.org/rixpress/articles/intro-concepts.html). The GitHub repository for this article directs interested readers to a demonstration of {rixpress} applied to this entire project.

Converting Existing {renv} Projects

Researchers with existing {renv} projects can migrate using the `renv2nix()` function, which reads an `renv.lock` file and generates an equivalent Nix expression. This is particularly valuable for projects where {renv} encountered system dependency issues or where stricter reproducibility guarantees are desired. Unlike {renv}, which captures R package versions but not the R interpreter or system libraries, Nix manages all layers of the software stack (see {rix} documentation: https://docs.ropensci.org/rix/articles/renv2nix.html).

Containerization with Docker

Nix and Docker are not necessarily mutually exclusive (Rodrigues & Baumann, 2026). Researchers already using Docker do not need to abandon it to benefit from Nix—the two can be combined by using Nix inside Docker containers to handle environment setup (Rodrigues & Baumann, 2025). This is particularly useful for deployment to cloud platforms or high-performance computing clusters where Docker is standard but Nix may not be available (see {rix} documentation: https://docs.ropensci.org/rix/articles/nix-inside-docker.html).

Discussion

Reproducibility in computational research is often treated as a matter of transparency—making data and code available. This tutorial has argued that transparency alone is insufficient without the ability to reliably reconstruct the computational environments in which analyses are executed. For simulation studies in particular, where results depend critically on software versions, system libraries, and random number generation, environment-level reproducibility is not optional but essential.

By introducing Nix and the {rix} package, we demonstrated a practical and accessible approach to fully specifying and rebuilding computational environments for simulation-based research. This approach enables analyses and manuscripts to be rerun identically across machines and over time, transforming reproducibility from an aspirational goal into a verifiable property of the research workflow.

Importantly, adopting environment reproducibility does not require abandoning existing analytic practices. Nix is agnostic to programming language, editor, workflow structure, and manuscript template, allowing researchers to retain familiar tools while strengthening the reliability of their work. In this sense, reproducible environments serve as enabling infrastructure—supporting, rather than replacing, other best practices such as version control, workflow orchestration, and transparent reporting.

If reproducibility is to function as a cornerstone of cumulative science, then the ability to reconstruct computational environments must become a routine part of methodological practice.

Tools such as Nix and {rix} lower the barrier to achieving this goal, making fully reproducible simulation research feasible without requiring deep systems expertise. We hope this tutorial helps normalize environment-level reproducibility as a standard component of rigorous computational research in psychology and beyond.

References

Baker, D. H., Berg, M., Hansford, K., Quinn, B., Segala, F., & Warden-English, E. (2024). ReproduceMe: Lessons from a pilot project on computational reproducibility. *Meta-Psychology*, *8*, MP.2023.4021. https://doi.org/10.15626/MP.2023.4021

Boettiger, C. (2015). An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, *49*(1), 71–79. https://doi.org/10.1145/2723872.2723882

Boettiger, C., & Eddelbuettel, D. (2017). An introduction to Rocker: Docker containers for R. *The R Journal*, *9*(2), 527–536. https://doi.org/10.32614/RJ-2017-065

Dolstra, E., De Jonge, M., & Visser, E. (2004). Nix: A safe and policy-free system for software deployment. *18th Large Installation System Administration Conference*, 79–92.

Epskamp, S. (2019). Reproducibility and replicability in a fast-paced methodological world. *Advances in Methods and Practices in Psychological Science*, *2*(2), 145–155. https://doi.org/10.1177/2515245919847421

Feldman, S. I. (1979). Make — a program for maintaining computer programs. *Software: Practice and Experience*, *9*(4), 255–265. https://doi.org/10.1002/spe.4380090402

Glatard, T., Lewis, L. B., Ferreira da Silva, R., Adalat, R., Beck, N., Lepage, C., Rioux, P., Rousseau, M.-É., Sherif, T., Deelman, E., Khalili-Mahani, N., & Evans, A. C. (2015). Reproducibility of neuroimaging analyses across operating systems. *Frontiers in Neuroinformatics*, *9*, 12. https://doi.org/10.3389/fninf.2015.00012

Hardwicke, T. E., Wallach, J. D., Kidwell, M. C., Bendixen, T., Crüwell, S., & Ioannidis, J. P. A. (2020). An empirical assessment of transparency and reproducibility-related research practices in the social sciences (2014–2017). *Royal Society Open Science*, *7*(2), 190806. https://doi.org/10.1098/rsos.190806

Hodges, C. B., Stone, B. M., Johnson, P. K., Carter, J. H., III, Sawyers, C. K., Roby, P. R., & Lindsey, H. M. (2023). Researcher degrees of freedom in statistical software contribute to unreliable results: A comparison of nonparametric analyses conducted in SPSS, SAS, Stata, and R. *Behavior Research Methods*, *55*(6), 2813–2837.

https://doi.org/10.3758/s13428-022-01932-2

Kidwell, M. C., Lazarević, L. B., Baranski, E., Hardwicke, T. E., Piechowski, S., Falkenberg, L.-S., Kennett, C., Slowik, A., Sonnleitner, C., Hess-Holden, C., Errington, T. M., Fiedler, S., & Nosek, B. A. (2016). Badges to acknowledge open practices: A simple, low-cost, effective method for increasing transparency. *PLOS Biology*, *14*(5), e1002456. https://doi.org/10.1371/journal.pbio.1002456

Landau, W. M. (2021). *Targets: Dynamic function-oriented make-like declarative workflows*.

Levenstein, M. C., & Lyle, J. A. (2018). Data: Sharing is caring. *Advances in Methods and Practices in Psychological Science*, *1*(1), 95–103.

Luijken, K., Lohmann, A., Alter, U., Claramunt Gonzalez, J., Clouth, F. J., Fossum, J. L., Hesen, L., Huizing, A. H. J., Ketelaar, J., Montoya, A. K., Nab, L., Nijman, R. C. C., Penning de Vries, B. B. L., Tibbe, T. D., Wang, Y. A., & Groenwold, R. H. H. (2024). Replicability of simulation studies for the investigation of statistical methods: The RepliSims project. *Royal Society Open Science*, *11*(1), 231003. https://doi.org/10.1098/rsos.231003

Malka, J., Zacchiroli, S., & Zimmermann, T. (2024). Reproducibility of build environments through space and time. *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 97–101. https://doi.org/10.1145/3639476.3639767

Morris, T. P., White, I. R., & Crowther, M. J. (2019). Using simulation studies to evaluate statistical methods. *Statistics in Medicine*, *38*(11), 2074–2102. https://doi.org/10.1002/sim.8086

Nagler, T., & Vatter, T. (2025). *Rvinecopulib: High performance algorithms for vine copula modeling*. https://CRAN.R-project.org/package=rvinecopulib

Nosek, B. A., Alter, G., Banks, G. C., Borsboom, D., Bowman, S. D., Breckler, S. J., Buck, S., Chambers, C. D., Chin, G., Christensen, G., Contestabile, M., Dafoe, A., Eich, E., Freese, J., Glennerster, R., Goroff, D., Green, D. P., Hesse, B., Humphreys, M., … Yarkoni, T. (2015). Promoting an open research culture. *Science*, *348*(6242), 1422–1425.

https://doi.org/10.1126/science.aab2374

Nosek, B. A., Hardwicke, T. E., Moshontz, H., Allard, A., Corker, K. S., Dreber, A., Fidler, F., Hilgard, J., Kline Struhl, M., Nuijten, M. B., Rohrer, J. M., Romero, F., Scheel, A. M., Scherer, L. D., Schönbrodt, F. D., & Vazire, S. (2022). Replicability, robustness, and reproducibility in psychological science. *Annual Review of Psychology*, *73*, 719–748. https://doi.org/10.1146/annurev-psych-020821-114157

Ottoboni, K., & Stark, P. B. (2018). *Random problems with r*. https://arxiv.org/abs/1809.06520

Pawel, S., Bartoš, F., Siepe, B. S., & Lohmann, A. (2025). Handling missingness, failures, and non-convergence in simulation studies: A review of current practices and recommendations. *The American Statistician*, 1–18. https://doi.org/10.1080/00031305.2025.2540002

Peikert, A., & Brandmaier, A. M. (2021). A reproducible data analysis workflow. *Quantitative and Computational Methods in Behavioral Sciences*, *1*, e3763. https://doi.org/10.5964/qcmb.3763

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

Piccolo, S. R., & Frampton, M. B. (2016). Tools and techniques for computational reproducibility. *GigaScience*, *5*, 30. https://doi.org/10.1186/s13742-016-0135-4

Rodrigues, B. (2023). *Building reproducible analytical pipelines with R*. https://raps-with-r.dev

Rodrigues, B. (2025). *Rixpress: Build reproducible analytical pipelines with 'nix'*. https://CRAN.R-project.org/package=rixpress

Rodrigues, B., & Baumann, P. (2025). *Rix: Reproducible data science environments with 'nix'*. https://CRAN.R-project.org/package=rix

Rodrigues, B., & Baumann, P. (2026). *Nix for polyglot, reproducible data science workflows* (Version v0.0.1). Zenodo. https://doi.org/10.5281/zenodo.18138618

Schneider, W. J. (2024). *Apaquarto* [Computer software]. https://github.com/wjschne/apaquarto

Siepe, B. S., Bartoš, F., Morris, T. P., Boulesteix, A.-L., Heck, D. W., & Pawel, S. (2024). Simulation studies for methodological research in psychology: A standardized template for planning, preregistration, and reporting. *Psychological Methods*.

https://doi.org/10.1037/met0000695

Simonsohn, U. (2020). *Groundhog: Version-control for CRAN, github, and gitlab packages*.

Ushey, K. (2024). *Renv: Project environments*.

Vazire, S. (2018). Implications of the credibility revolution for productivity, creativity, and progress. *Perspectives on Psychological Science*, *13*(4), 411–417. https://doi.org/10.1177/1745691617751884

White, I. R., Pham, T. M., Quartagno, M., & Morris, T. P. (2024). How to check a simulation study. *International Journal of Epidemiology*, *53*(1), dyad134. https://doi.org/10.1093/ije/dyad134

Wiebels, K., & Moreau, D. (2021). Leveraging containers for reproducible psychological research. *Advances in Methods and Practices in Psychological Science*, *4*(2). https://doi.org/10.1177/25152459211017853

Table 1

Performance metrics for ACE estimator across simulation conditions. Values in parentheses are Monte Carlo standard errors (MCSE).

Sample Size	Confounding	Relative Bias	Relative RMSE	Coverage	CI Width
50	none	0.957 (0.042)	0.422 (0.035)	0.910 (0.029)	0.224 (0.004)
100	none	1.013 (0.026)	0.256 (0.020)	0.930 (0.026)	0.159 (0.002)
2000	none	0.997 (0.006)	0.059 (0.005)	0.960 (0.020)	0.036 (0.000)
50	mild	1.173 (0.036)	0.395 (0.034)	0.890 (0.031)	0.222 (0.004)
100	mild	1.157 (0.028)	0.318 (0.026)	0.870 (0.034)	0.154 (0.002)
2000	mild	1.114 (0.006)	0.130 (0.006)	0.510 (0.050)	0.035 (0.000)
50	severe	1.200 (0.044)	0.482 (0.054)	0.870 (0.034)	0.223 (0.005)
100	severe	1.235 (0.030)	0.379 (0.027)	0.830 (0.038)	0.157 (0.002)
2000	severe	1.257 (0.007)	0.266 (0.007)	0.020 (0.014)	0.035 (0.000)

Figure 1

Performance of ACE estimator across sample sizes and confounding severity. Panel A shows relative bias, Panel B shows relative RMSE, Panel C shows coverage probability of 95% confidence intervals (dashed line at nominal 0.95 level), and Panel D shows average confidence interval width. Results demonstrate that model misspecification induces systematic bias that persists across sample sizes, while increasing sample size improves precision but not accuracy under misspecification.
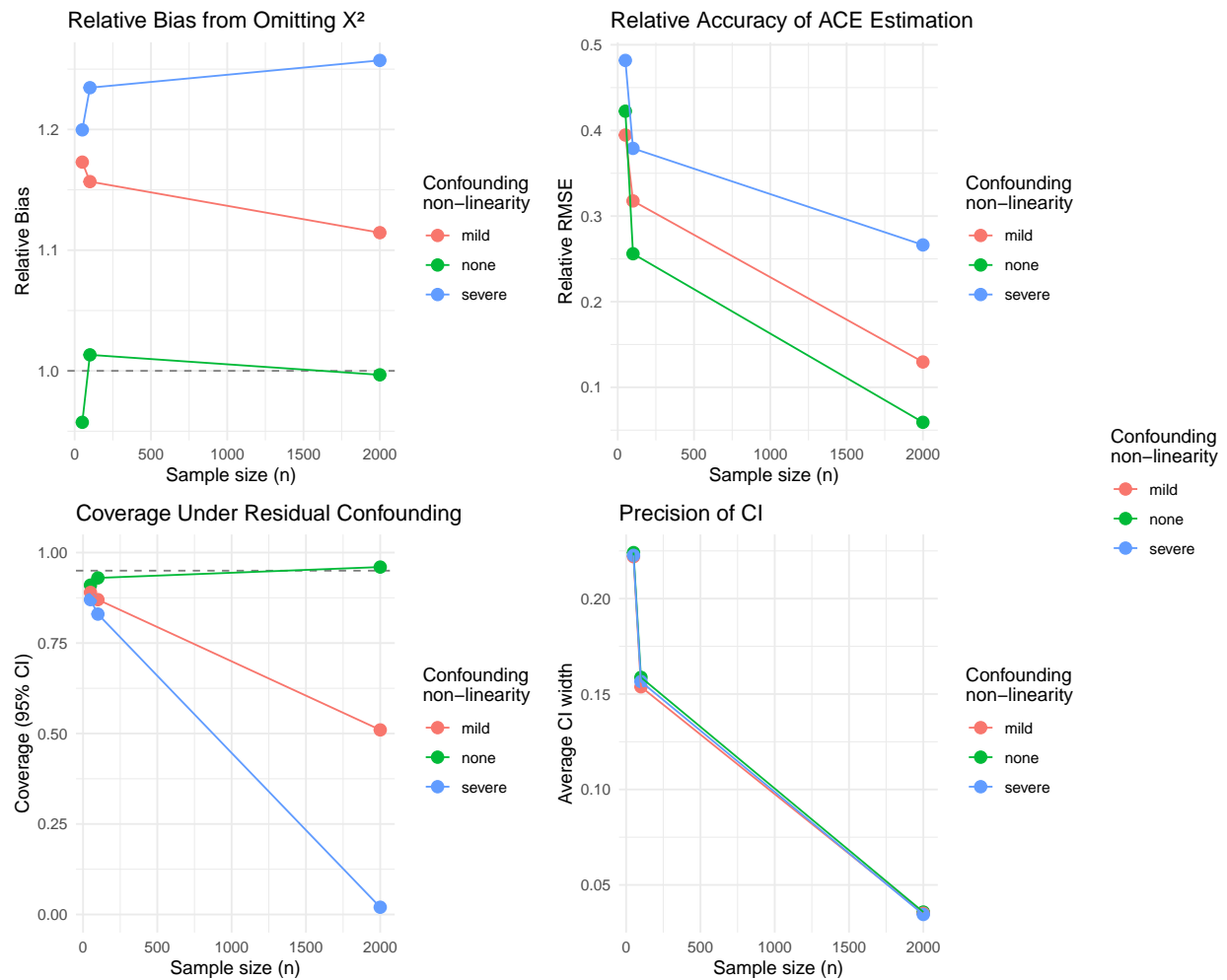
Appendix A

Simulation Study Design

Here we present a rather short description following recommendations from previous research, but ideally even more may be reported (Morris et al., 2019; Pawel et al., 2025; Siepe et al., 2024; White et al., 2024). This mimics a methods or similar section in articles.

Factorial Design. The simulation employs a full factorial design with two factors: sample size ($n \in \{50, 100, 2000\}$) and degree of confounding non-linearity ($\gamma_2 \in \{0, 0.3, 0.8\}$, labeled as none, mild, and severe). The parameter γ_2 controls the strength of the quadratic confounder effect on the outcome (see Data Generation). This yields nine conditions, each replicated $K = 100$ times.

Data Generation. For each replication, data are generated following a causal structure where a confounder X_2 affects both treatment assignment and the outcome. The confounder and treatment error term are generated using the {rvinecopulib} package: pairs (U_1, U_2) are drawn from an independence copula via `rbicop()`, then transformed to standard normals via $X_2 = \Phi^{-1}(U_1)$ and $\epsilon = \Phi^{-1}(U_2)$. The independence copula is simply $C(u, v) = uv$, meaning the resulting uniforms are independent—mathematically equivalent to calling `rnorm()` directly. We use {rvinecopulib} intentionally because it depends on C++ libraries.

Treatment assignment follows $X_1 = \alpha_0 + \alpha_1 X_2 + \alpha_2 X_2^2 + \epsilon$ where $\alpha_0 = 0$, $\alpha_1 = 0.5$, and $\alpha_2 = 0.2$. This creates confounding because X_2 influences treatment assignment through both linear and quadratic terms. The binary outcome is generated from the true logistic regression model:

$$\text{logit}(P(Y = 1 \mid X_1, X_2)) = \beta_0 + \beta_1 X_1 + \gamma_1 X_2 + \gamma_2 X_2^2$$

with parameters $\beta_0 = -0.5$, $\beta_1 = 0.7$ (the causal effect of interest), $\gamma_1 = -0.4$, and γ_2 varying by condition. The analyst misspecifies the outcome model by omitting the quadratic confounder term, fitting instead:

$$\text{logit}(P(Y = 1 \mid X_1, X_2)) = \beta_0 + \beta_1 X_1 + \gamma_1 X_2$$

This misspecification creates residual confounding because the omitted term $\gamma_2 X_2^2$ is

correlated with X_1 (since X_1 depends on both X_2 and X_2^2), violating the conditional

exchangeability assumption given linear adjustment alone.

Estimand. The target estimand is the average causal effect (ACE) of X_1 on Y, properly

adjusted for confounding:

$$\text{ACE}(X_1) = \mathbb{E}\left[\frac{\partial P(Y = 1 \mid X_1, X_2)}{\partial X_1}\right] = \mathbb{E}\left[\beta_1 \cdot \frac{\exp(\eta)}{(1 + \exp(\eta))^2}\right]$$

where $\eta = \beta_0 + \beta_1 X_1 + \gamma_1 X_2 + \gamma_2 X_2^2$ is the correctly specified linear predictor, and the expectation

is taken over the joint distribution of (X_1, X_2). For each γ_2 condition, the "true" ACE (denoted θ)

is approximated once using a very large sample ($N = 200{,}000$) with the correctly specified model

including X_2^2.

Estimator. The causal effect is estimated from the misspecified model (omitting X_2^2) as:

$$\widehat{\text{ACE}}(X_1) = \frac{1}{n} \sum_{i=1}^{n} \tilde{\beta}_1 \cdot \frac{\exp(\tilde{\eta}_i)}{(1 + \exp(\tilde{\eta}_i))^2}$$

where $\tilde{\eta}_i = \tilde{\beta}_0 + \tilde{\beta}_1 X_{i1} + \tilde{\gamma}_1 X_{i2}$ and $\tilde{\boldsymbol{\beta}} = (\tilde{\beta}_0, \tilde{\beta}_1, \tilde{\gamma}_1)$ are maximum likelihood estimates from the

misspecified logistic regression.

Performance Criteria. Table B1 presents the performance criteria used to evaluate the

ACE estimator across simulation conditions.

Computational Details. The simulation was conducted on a MacBook Pro (Apple M4

Pro Chip), running macOS Sequoia 15.7.3. All analyses were performed in R (version 4.5.2).

Parallel processing was implemented through the {doParallel} package (version 1.0.17), with

{doRNG} (version 1.8.6.2) to ensure independent and reproducible random number streams. For

data generation we used the {rvinecopulib} package (version 0.7.3.1.0). The estimator was

implemented using the {marginaleffects} package (version 0.31.0). Data wrangling was

performed with {dplyr} (version 1.1.4). Method performance was assessed through multiple

metrics following the formulas from the {simhelpers} package (version 0.3.1). Figures were

produced with {ggplot2} (version 4.0.1) and {cowplot} (version 1.2.0).

Appendix B

Clarifying the packages used

R Packages

Reproducibility Infrastructure

rix: Generates the Nix expression (`default.nix`) for reproducible environments.

Simulation Study

These packages are used in the `Simulation_Scripts/` folder:

rvinecopulib (used in `01_data_generation.R`): Generates correlated data via copulas using the `rbicop()` function.

marginaleffects (used in `02_models.R`): Computes average causal effects via the `avg_slopes()` function.

doParallel (used in `03_run_simulation.R`): Enables parallel `foreach` loops across CPU cores.

doRNG (used in `03_run_simulation.R`): Makes parallel RNG reproducible.

simhelpers (used in `04_performance_metrics.R`): Calculates bias, RMSE, and coverage metrics.

ggplot2 (used in `05_plots.R`): Creates simulation result visualizations.

cowplot (used in `05_plots.R`): Combines plots with `plot_grid()` and extracts legends.

dplyr (used in `article.qmd`): Data wrangling for results reported in Table 1.

Dynamic Document Generation

quarto: R interface to invoke Quarto rendering.

knitr: Executes R code chunks in `.qmd` files.

svglite: SVG graphics device; apaquarto sets `dev: svglite` for HTML output.

LaTeX Packages

Required by apaquarto Extension

These are loaded in apaquarto's `header.tex` or `apatemplate.tex`:

amsmath: Math environments (`align`, `equation`, etc.).

threeparttablex: Tables with notes below (APA table format).

tcolorbox: Callout boxes (note, warning, tip blocks).

fontawesome5: Icons in callouts.

multirow: Table cells spanning multiple rows.

newtx: Times-like fonts (default when no custom `mainfont`).

geometry: Page margins.

Dependencies of apaquarto Packages

environ: Dependency of tcolorbox.

pdfcol: Dependency of tcolorbox.

tikzfill: Dependency of tcolorbox.

fontaxes: Dependency of newtx.

xstring: Template conditionals.

scalerel: Dependency of apa7 class.

Required by apa7 Document Class

apa7: The document class itself (\documentclass{apa7}).

endfloat: Moves floats to end of document in manuscript mode.

threeparttable: Dependency for threeparttablex.

Required by Quarto PDF Rendering

framed: Shaded/framed regions for callouts.

fvextra: Enhanced verbatim for syntax-highlighted code blocks.

fancyvrb: Verbatim environments for code display.

setspace: Line spacing; also used in `article.qmd` for single-spaced code blocks.

anyfontsize: Arbitrary font sizes in code blocks.

Additional Packages

ninecolors: Extended color palettes.

wrapfig: Text wrapping around figures.

tabularray: Modern table typesetting.

siunitx: SI units and number formatting.

Table B1

Performance criteria for evaluating the ACE estimator. $\hat{\theta}_k$ denotes the ACE estimate from replication k (for $k = 1, \ldots, K$), where $K = 1000$ is the number of replications, and θ denotes the true ACE for a given condition. For coverage and width criteria, A_k and B_k denote the lower and upper endpoints of the 95% confidence interval from replication k, $W_k = B_k - A_k$ is the interval width, c_β is the estimated coverage probability, and $I(\cdot)$ is an indicator function equaling 1 if the condition is true and 0 otherwise. The Monte Carlo standard error (MCSE) quantifies the simulation uncertainty in each performance measure estimate

Criterion	Estimate	MCSE
Bias	$\frac{1}{K} \sum_{k=1}^{K} \hat{\theta}_k - \theta$	$\sqrt{\frac{1}{K(K-1)} \sum_{k=1}^{K} (\hat{\theta}_k - \bar{\hat{\theta}})^2}$
Variance	$\frac{1}{K-1} \sum_{k=1}^{K} (\hat{\theta}_k - \bar{\hat{\theta}})^2$	$\sqrt{\frac{K-1}{K} \cdot \frac{1}{K-1} \sum_{k=1}^{K} (\hat{\theta}_k - \bar{\hat{\theta}})^2}$
RMSE	$\sqrt{\frac{1}{K} \sum_{k=1}^{K} (\hat{\theta}_k - \theta)^2}$	$\sqrt{\frac{K-1}{K} \sum_{j=1}^{K} \left(\sqrt{(\hat{\theta}_j - \theta)^2} - \overline{RMSE} \right)^2}$
Relative Bias	$\frac{1}{\theta K} \sum_{k=1}^{K} \hat{\theta}_k$	$\frac{1}{\theta} \sqrt{\frac{1}{K(K-1)} \sum_{k=1}^{K} (\hat{\theta}_k - \bar{\hat{\theta}})^2}$
Relative RMSE	$\frac{1}{\theta} \sqrt{\frac{1}{K} \sum_{k=1}^{K} (\hat{\theta}_k - \theta)^2}$	$\frac{1}{\theta} \sqrt{\frac{K-1}{K} \sum_{j=1}^{K} \left(\sqrt{(\hat{\theta}_j - \theta)^2} - \overline{RMSE} \right)^2}$
Coverage	$\frac{1}{K} \sum_{k=1}^{K} I(A_k \leq \theta \leq B_k)$	$\sqrt{\frac{c_\beta(1-c_\beta)}{K}}$
Width	$\frac{1}{K} \sum_{k=1}^{K} (B_k - A_k)$	$\sqrt{\frac{1}{K(K-1)} \sum_{k=1}^{K} (W_k - \bar{W})^2}$