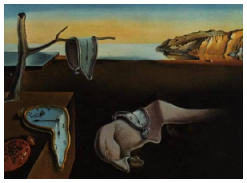

Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

Nix for Polyglot, Reproducible Data Science Workflows

Bruno Rodrigues
Ministry of Research and Higher education, Luxembourg

Philipp Baumann
Alliance SwissPass, Switzerland

Abstract

Reproducible analysis requires more than clean, well-documented code; it demands the precise management of software dependencies, computational environments, and workflow execution. While researchers can combine tools like **Docker**, **renv** for R packages (or **uv** for Python packages) and **Make** to address these needs, the resulting toolchain is often complex and fragile, requiring the coordination of multiple disparate systems. This paper introduces a unified framework built on the **Nix** package manager, made accessible through two R packages. The first, **rix**, generates declarative **Nix** expressions to define version-pinned, reproducible environments that encompass R, Python, Julia, and system-level dependencies. The second, **rixpress** (and its Python port, **ryxpress**), leverages these environments to orchestrate polyglot pipelines where each computational step runs in a hermetically sealed, language-specific environment with automatic caching. By treating the entire computational environment as code, this integrated approach ensures a high degree of reproducibility, supports collaboration across heterogeneous systems, and helps guarantee that analyses remain executable long into the future.

Keywords: reproducibility, R, Python, Julia, **Nix**.

1. Introduction: Reproducibility is also about software

Peng (2011) introduced the concept of reproducibility as a *continuum*. At one end lies the least reproducible state, where only a paper describing the study is available. Reproducibility improves when authors share the original source code, improves further when they include the underlying data, and reaches its highest level when what Roger Peng called *linked and executable code and data* are provided.

By *linked and executable code and data*, Peng referred to compiled source code and runnable scripts. In this paper, we interpret this notion more broadly as the *computational environment*:

the complete set of software required to execute an analysis. Here too, a continuum exists. At the minimal end, authors might only name the main software used—say, the R programming language (R Core Team 2021). More careful authors might also specify the version of R, or list the additional packages and their versions. Rarely, however, do authors specify the operating system on which the analysis was performed, even though differences in operating systems can lead to divergent results when using the same code and software versions, as shown by Bhandari Neupane, Neupane, Luo, Yoshida, Sun, and Williams (2019). It is even less common for authors to provide step-by-step installation instructions for the required software stack, an omission often driven by institutional constraints. Journals, for instance, can inadvertently hinder reproducibility by imposing strict page or word limits that leave no room for the necessary technical documentation, discouraging thoroughness in the name of brevity.

Even when such instructions are given, they often fail across different platforms or versions of the same platform. This lack of portability not only hinders reproducibility but also complicates everyday research workflows. Researchers using multiple machines must recreate environments consistently, and collaborators must share identical computational setups to avoid inconsistencies.

Finally, once the execution environment is correctly configured, additional clarity is needed on how to *run* the project itself. Which packages should be loaded first? Which scripts should be executed, and in what order? Without clear documentation or automated orchestration, these operational details become yet another barrier to reproducibility.

This paper focuses on two critical but often overlooked aspects of reproducibility: *computational environment management* and *workflow orchestration*. We present a comprehensive framework addressing both challenges using the **Nix** package manager (Dolstra, De Jonge, and Visser 2004), making it accessible to researchers through two R packages: **rix** and **rixpress**. Before introducing these packages, we survey existing tools and their limitations to contextualise our contributions.

A range of tools now exist to help researchers approach the gold standard of full reproducibility, or to consistently deploy the same development environment across multiple machines. Let us first consider the most basic step in this process: listing the software used. In R, the `sessionInfo()` function provides a concise summary of the software environment, including the R version, platform details, and all loaded packages. Its output can be saved to a file and included as part of a study's reproducibility record. Below is an example output from `sessionInfo()`:

```
R> sessionInfo()

R version 4.3.2 (2023-10-31)
Platform: aarch64-unknown-linux-gnu (64-bit)
Running under: Ubuntu 22.04.3 LTS

Matrix products: default
BLAS:   /usr/lib/aarch64-linux-gnu/openblas-pthread/libblas.so.3
LAPACK: /usr/lib/aarch64-linux-gnu/openblas-pthread/libopenblasp[...]

locale:
```

```
 LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 ...

attached base packages:
 stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
 nnet_7.3-19  mgcv_1.9-0   nlme_3.1-163

loaded via a namespace (and not attached):
 compiler_4.3.2  Matrix_1.6-1.1  tools_4.3.2     splines_4.3.2
 grid_4.3.2      lattice_0.21-9
```

When an author includes this information, others attempting to reproduce the study (future readers, collaborators, or even the author at a later time) can see which version of R and which packages (with their versions) were used. However, reproducing the environment still requires manually installing the correct package versions — a process that becomes difficult when packages depend on system-level libraries. For example, the **sf** package requires **GDAL**, **GEOS**, and **PROJ** libraries. Installing and configuring these dependencies varies substantially across operating systems and can be prohibitively difficult on some platforms.

A more robust approach than simply listing package versions is to use tools that define per-project environments. This is common for Python (Van Rossum and Drake 2009) users, who have many tools at their disposal, such as the built-in *virtual environments* module, or third-party tools like **uv** (Astral 2024), which manages both project-specific interpreters and package libraries using a lock file mechanism.

In the R ecosystem, virtual environments are not built-in as they are in Python, however tools like **renv** provide equivalent functionality. **renv** captures the project's software state and writes it to a lock file (`renv.lock`), which includes the exact versions of R and all required packages. This lockfile serves as a blueprint for restoring the environment automatically, ensuring others can recreate the same setup with minimal effort.

However, both **renv** and **uv** have limitations. While **uv** manages Python interpreter versions directly, **renv** does not restore the version of R itself—installing the correct version must be done separately using tools like **rig** (R Infrastructure 2023). More critically, neither handles system-level dependencies. If **sf** requires **GDAL** version 3.0 but the system has version 2.4 installed, neither tool can resolve this conflict. Users must manually install system libraries, and the process differs across operating systems.

Other R packages such as **groundhog** (Simonsohn and Gruson 2023) and **rang** (hong Chan and Schoch 2023) enable date-based package installation, while the Posit Package Manager provides dated CRAN snapshots. However, like **renv**, these focus on R packages and do not address system-level dependencies or R version management. Furthermore, using pre-compiled binaries of packages simplifies installation but do not eliminate the need for system libraries required at runtime. These tools represent significant progress in reproducibility within their ecosystems, but they share a common limitation: none handles system-level dependencies or the increasingly polyglot nature of modern data science.

Indeed, there is evidence that the era of a single tool dominating an entire analytical pipeline is giving way to a more pragmatic approach where researchers combine the complementary strengths of multiple languages (Peng, Wang, Ma, Leong, Wakefield, Melott, Chiu, Du, and Weinstein 2018). This trend is particularly evident in the synergistic use of R, with its rich ecosystem for statistical modeling, Python, with its extensive libraries for machine learning, and Julia for complex simulations through the use of interoperability libraries such as **PythonCall** (calling Python from Julia), **JuliaCall** (calling Julia from Python or R), **reticulate** (calling Python from R), **rpy2** and **ryp** (calling R from Python) (Osborne, Pivarski, and Ling 2024; Navarro, Koneva, Sánchez-Macián, and Hernández 2024). Recent survey data confirms this trend, showing that research scientists use, on average, nearly two programming languages in their work, with the R and Python combination being the most prevalent (Chen, Wong, Sharif, and Peruma 2025). This creates a significant challenge for reproducibility: a project using R, Python, and system tools like **Quarto** or LaTeXrequires coordinating multiple, disparate package managers, each with its own configuration and potential for conflict and using interoperability libraries.

The demand for solutions that can manage this complexity is not merely academic; it is reflected in major strategic shifts within the industry. A prominent example is the evolution of Posit (formerly RStudio), which has pivoted from its origins as a premier R-centric service and software supplier to a company that treats Python as a first-class citizen alongside R. This industry trend, coupled with the proliferation of academic research into polyglot tooling, underscores a critical reality: the central challenge for modern reproducible science is no longer *if* polyglot workflows should be used, but *how* they can be managed in a unified, efficient, and reproducible manner.

The most comprehensive approach to managing a complete computational environment to date is containerization, for which **Docker** has become the de facto standard. Docker's principal strength lies in its capacity to package an entire data product—including the operating system, all system-level libraries, programming language runtimes, and packages—into a single, self-contained artifact known as an image. Once this image is built, the analysis can be executed within a container, which is a running instance of that image. This methodology effectively solves the problem of missing system-level dependencies; because Docker images are typically minimal Linux systems, they can bundle the specific versions of libraries like **GDAL** or **GEOS** required by packages such as **sf**. These images can then be shared via public registries, enabling a high degree of reproducibility across different host systems. The Rocker project, for instance, provides invaluable pre-built Docker images tailored for the R community, offering a convenient foundation for reproducible research environments (Boettiger and Eddelbuettel 2017).

However, while Docker excels at providing this form of spatial reproducibility, its application in interactive research workflows reveals several significant limitations. First, the container model is often poorly aligned with the iterative, interactive nature of data analysis. Running graphical applications like RStudio Desktop requires complex and platform-specific configuration, while ephemeral container filesystems mean that changes made during a session are lost by default. Consequently, a common workflow involves developing an analysis interactively on a host system and only containerizing it post-hoc, which achieves reproducibility as an afterthought rather than as a continuous guarantee during the research process. Packages like **dockerfiler** (Fay, Guyader, Parry, and Rochette 2024) are specifically made to enable this

type of post-hoc reproducibility.

Second, the effective use of **Docker** introduces a considerable learning curve. Authoring a reliable and truly reproducible `Dockerfile` requires a degree of familiarity with Linux system administration. Best practices, such as referencing base images by their immutable content digests rather than mutable tags, are necessary for long-term stability but are not widely adopted. Furthermore, while **Docker** can house polyglot environments, it offers no native mechanism for orchestrating a pipeline where distinct steps might require different, and potentially conflicting, sets of dependencies. The entire analysis is typically confined to a single, monolithic environment.

The most critical limitation, however, lies in its guarantees of temporal reproducibility. As noted by Dellaiera (2024), reproducibility is best viewed as a spectrum. Docker provides strong run-time reproducibility (consistency across space), but offers weaker build-time reproducibility (consistency over time). The imperative commands within a `Dockerfile`, such as `apt-get update`, are non-deterministic by nature and can introduce temporal drift, where rebuilding the same file at different times yields different binary artifacts. As empirically demonstrated by Malka, Zacchiroli, and Zimmermann (2025), achieving deterministic builds requires a system designed for this purpose, a role for which functional package managers like **Nix** are architecturally better suited.[1]

This is not to say that **Docker** has no place in a reproducible research workflow. Its utility for production deployment, continuous integration, and cross-platform distribution is well-established. Indeed, the two approaches are not mutually exclusive. As shall be explained in Section 2, it is possible to use **Nix** within **Docker** to benefit from the advantages of both tools.

Making sure the development environment is reproducible is but one piece of the puzzle, though. Even with a perfectly reproducible environment, researchers face another challenge: reliably executing the analysis workflow. Which scripts run first? What are the dependencies between steps? Manual execution is error-prone and poorly documented. Build automation tools like **Make** address this by defining analyses as ordered, reproducible steps.

Within R, the **targets** package (Landau 2021) provides a modern, declarative approach to workflow management. It tracks dependencies, caches results, and only recomputes affected steps, improving efficiency for complex analyses. However, **targets** operates within a single R session. For truly polyglot pipelines—where steps may require incompatible dependencies or different language versions—this becomes problematic. Running **targets** inside a **Docker** container ensures reproducibility but forces the entire pipeline to share one environment.

The current state-of-the-art in reproducible research exemplifies the complexity of navigating this fragmented landscape. The work of McDermott (2021) provides an excellent case study in best practices, with an accompanying repository that is fully reproducible.[2] Achieving this gold standard, however, required the masterful integration of multiple complex and disparate tools: **renv** for managing R packages, **Docker** for containerizing the system environment, and **Make** for orchestrating the analysis. This demonstrates that while a high degree of

[1]This contrast between the procedural, imperative commands of a `Dockerfile` and the declarative, functional model of **Nix** highlights a fundamental dichotomy in approaches to reproducibility, a concept we will examine in detail as we introduce the **Nix** package manager in Section Section 2.

[2]See https://github.com/grantmcdermott/skeptic-priors

reproducibility is attainable, it requires a heroic effort, forcing researchers to become experts in a brittle and complex toolchain.

Furthermore, the complexity increases for polyglot projects, which must manage multiple ecosystems simultaneously.

Researchers therefore need a tool that:

1. Manages complete environments including languages, packages, and system dependencies;
2. Supports multiple languages natively;
3. Works interactively without container complexity;
4. Provides step-level isolation for different environments;
5. Ensures bit-for-bit reproducibility through deterministic builds;
6. Remains simple enough for researchers without systems administration expertise.

The **Nix** package manager provides these capabilities. It ensures reproducible installation by deploying *component closures*—packages bundled with all their direct and transitive dependencies (Dolstra *et al.* 2004). Think of installing a package as packing for a trip: traditional package managers assume you'll find essentials at your destination, while **Nix** packs everything you need. This produces self-contained software environments that behave identically across machines and over time.

Nix can replace **Docker** for isolation, **renv** for package management, and even **Make** for orchestration—all within a single, unified framework. However, it has a steep learning curve, with its own functional language for declaratively defining software builds and configurations. While this ensures reproducibility, it also creates a barrier to adoption.

To make **Nix** accessible to researchers, we developed two R packages:

- **rix** generates **Nix** expressions from intuitive R function calls, eliminating the need to learn the Nix language. It handles environment definitions including R, Python, Julia (Bezanson, Edelman, Karpinski, and Shah 2017), system tools, and dependencies.

- **rixpress** orchestrates polyglot analytical pipelines using **Nix** as the build engine. Each step runs in its own hermetic environment with automatic caching and dependency tracking, enabling true polyglot workflows across R, Python, and Julia. Python users can use **ryxpress**, a direct port.

Together, these packages provide a single framework for managing reproducible, polyglot environments and executing complex workflows with step-level isolation. While **Nix** remains complex for advanced use, **rix** and **rixpress** abstract this complexity, making deep reproducibility accessible without systems administration expertise.

The remainder of this paper proceeds as follows. Section 2 introduces **Nix** and explains how its functional model enables reproducibility. Section 3 presents **rix** and demonstrates environment definition. Section 4 discusses the `rstats-on-nix` fork of the package repository. Section 5 introduces **rixpress** and demonstrates polyglot pipeline orchestration. Section 6 concludes with discussion of future directions.

2. The Nix Package Manager

Nix is a cross-platform package manager designed for reproducible software installation and deployment. Unlike traditional package managers, **Nix** emphasizes immutable infrastructure and a functional programming paradigm to ensure computational environment consistency. The primary **Nix** package collection, `nixpkgs`, provides access to over 120,000 packages, including nearly all of CRAN and Bioconductor.[3] According to Repology, a service tracking package repositories across distributions, `nixpkgs` is both the largest and among the most up-to-date repositories available (Figure Figure 1).

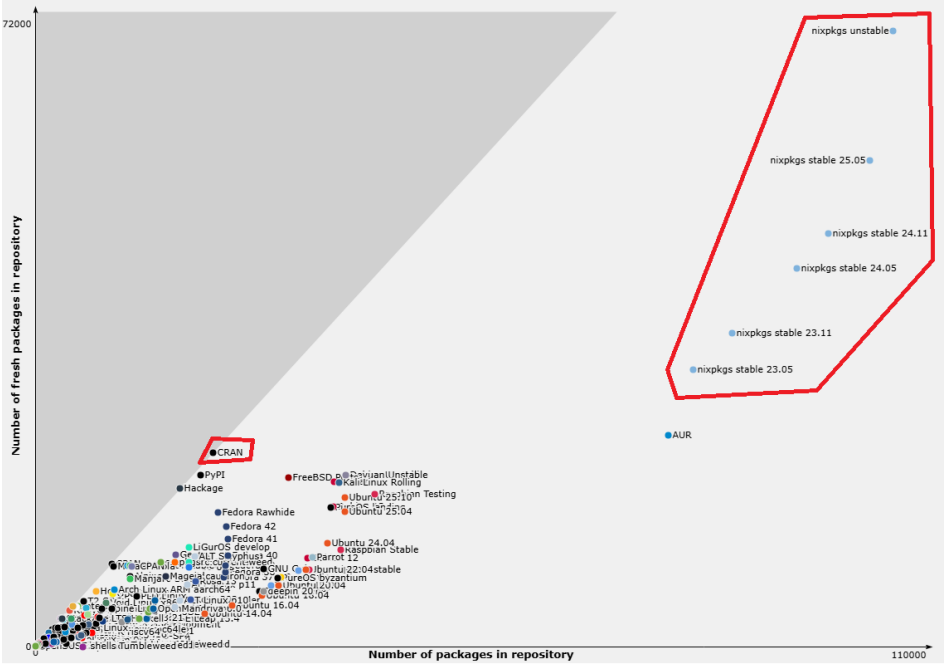

Figure 1: Package repository size and freshness according to Repology. `nixpkgs` leads in both total package count and update recency, making it well-suited for reproducible research environments.

This extensive coverage allows users to install not only R but also all required packages and system-level dependencies for any given project.

A key advantage of **Nix** over traditional installation methods lies in its comprehensive dependency management. Consider the **sf** package for spatial data analysis in R. It requires several complex system libraries: **GDAL**, **GEOS**, and **PROJ**. Manually installing and configuring these libraries is challenging and platform-specific. With **Nix**, users simply declare **sf** as a project requirement, and **Nix** automatically installs and configures all necessary system libraries as transitive dependencies.

[3]Only about 80 packages currently fail to build on the NixOS build farm, plus roughly 30 packages marked as broken by the maintainers of the R ecosystem. These could, in principle, be fixed, but doing so would require significant effort. Marking them as broken signals to users that their build failures are known and intentional. It should be further noted that some packages may build successfully, but then fail at runtime as they try to further install runtime dependencies on first run. One such example is **torch**.

This seamless process stems from **Nix**'s concept of *component closures*. As Dolstra *et al.* (2004) explains:

> The idea is to always deploy component closures: if we deploy a component, then we must also deploy its dependencies, their dependencies, and so on. [...] Since closures are self-contained, they are the units of complete software deployment.

In other words, every package deployment includes everything it needs to run, with no hidden dependencies on the host system.

Comprehensive dependency tracking, however, is not unique to **Nix**. The **r2u** project by Eddelbuettel (2025) achieves something similar for Ubuntu: once the r2u repository is configured, every **R** package is installed as a binary with all its dependencies automatically resolved. In principle, any package manager could implement such tracking.

What distinguishes **Nix** (and its sibling **Guix**, the only other functional package manager) is its *functional package* management model. This model makes full dependency specification intrinsic to the system, mandatory by design, and replaces the imperative, step-by-step nature of traditional package managers (*install this, then that*) with a declarative one: users describe the desired end state (*an environment with these packages*), and **Nix** determines how to realize it.

A further advantage is that packages built by **Nix** act as universal binaries across any Linux system (on the same architecture) where **Nix** is installed. This "build once, run anywhere" property has long been solved by **Nix**, and is now being revisited by Posit through its initiative to provide portable CRAN binaries (Posit, PBC 2025).

This universality stems from **Nix**'s use of derivations, declarative blueprints that specify all inputs (source code, build commands, dependencies) required to build a package. When software is installed, **Nix** evaluates expressions written in the Nix language to produce these derivations. Because every input is explicitly declared, builds are fully reproducible and independent of the surrounding environment.

For example, the **rJava** package definition includes the correct `JAVA_HOME` configuration pointing to its **jdk** dependency. Outside **Nix**, users must manually set this variable if automatic detection fails. Within **Nix**, maintainers encode such details once, ensuring every user benefits from a consistent, reliable installation.

At its core, **Nix** treats software builds as pure functions, a concept borrowed from functional programming. A pure function always produces the same output for a given input—for example, if $f(x) = x^2$, then $f(2) = 4$ invariably holds. Likewise, **Nix** guarantees that identical inputs (source code, dependencies, build instructions) yield identical outputs (built packages), regardless of when or where the build occurs.

This determinism is achieved through hermetic (isolated) builds with no hidden dependencies on global system state nor undeclared side effects. The effectiveness of this model was validated at unprecedented scale by Malka *et al.* (2025), who rebuilt 709,816 packages from historical `nixpkgs` snapshots (2017–2023), finding rebuildability rates exceeding 99% and bitwise reproducibility rates between 69% and 91%, with an upward trend. Importantly, about 15% of unreproducible builds failed due to embedded timestamps—a solvable implementation issue rather than a fundamental model limitation.

All **Nix** package definitions are hosted in the `nixpkgs` GitHub repository. By *pinning* to a specific commit (or equivalently, a specific date), users guarantee that all installed packages derive from identical build instructions. An environment built today using a pinned revision will produce the same software versions and configurations if rebuilt years later, provided the `nixpkgs` repository remains accessible (it is archived and distributed across multiple mirrors).

Nix installs all packages into a content-addressed directory called the Nix store (typically `/nix/store`). Each package occupies a unique subdirectory whose name includes a cryptographic hash of all its inputs: source code, dependencies, and build instructions. This means `/nix/store/abc123-dplyr-1.0.0` and `/nix/store/xyz789-dplyr-1.1.0` can coexist without conflicts, allowing multiple R versions on the same system and eliminating "dependency hell."

While this functional approach enhances reproducibility, it can introduce complexity for package maintainers, particularly for software downloading external assets during installation (e.g., certain Bioconductor packages). For end-users, however, this complexity is largely abstracted away by **rix** and **rixpress**, which we introduce in the following sections.

For a more in-depth technical discussion of **Nix**'s design principles, see Dolstra *et al.* (2004).

By leveraging these principles, **Nix** provides a unified framework that can subsume the functionalities of disparate tools. For R projects, it can replace the combination of **renv** for package management and **Docker** for system-level isolation. Similarly, for Python projects, it offers a more comprehensive alternative to `requirements.txt` files and virtual environments. Furthermore, **Nix** excels at composing polyglot environments, seamlessly integrating R, Python, Julia, a LaTeX distribution, and any of the numerous other tools available in `nixpkgs`. This enables the creation of a truly complete, project-specific, and deeply reproducible environment suitable for both interactive development and non-interactive, automated analysis. The long-term viability of this reproducibility is contingent only on the continued accessibility of the `nixpkgs` repository or a suitable fork, as will be discussed in Section Section 4.

Nix offers robust cross-platform support, running natively on Linux (x86_64) with optimal functionality. On macOS (Intel and Apple Silicon), however, its reproducibility guarantees are somewhat compromised by dependencies on proprietary system frameworks and Xcode toolchains that reside outside of the **Nix** store; consequently, macOS users may need to update project pins following system upgrades. For Windows users, **Nix** operates effectively through the Windows Subsystem for Linux (WSL2), where environments can be integrated into development workflows using editors like VS Code or Positron.

While **Nix** is a powerful system, its adoption is not without notable challenges. The most significant is its steep learning curve; it uses its own programming language, also called Nix, which is purely functional, a paradigm that may be unfamiliar to many researchers. This complexity is most apparent when authoring custom build instructions or troubleshooting intricate dependency issues. Additionally, build times can be substantial for large environments when pre-built binary artifacts are not available from a cache. The immutable, content-addressed nature of the **Nix** store also leads to higher disk usage compared to traditional package managers. Finally, while the `nixpkgs` repository is extensive, certain niche packages may be unavailable or fail to build on all platforms.

Beyond these technical aspects, a practical hurdle to adoption is the ubiquity of **Docker**. Given

the extensive infrastructure and community knowledge built around containerization, **Nix** can be perceived as a competing rather than a complementary technology. This perception, however, is a false dichotomy. **Docker** and **Nix** allow researchers to solve similar issues but these tools are not alternatives of one another. Thus, a powerful hybrid model involves using **Nix** to deterministically build the environment *during the construction of a* ***Docker*** *image*, which is then deployed using standard container infrastructure. This approach leverages the strengths of both tools: **Nix** provides robust build-time reproducibility, while **Docker** serves as a universal distribution and deployment mechanism. For many research workflows, however, where interactive development is paramount and the overhead of containerization is a barrier, the unified model provided by **Nix** alone can provide a more direct and streamlined path to reproducibility.

To make these capabilities more approachable for R users seeking reproducible, project-specific environments without learning the full Nix language, we created the **rix** and **rixpress** packages.

3. Reproducible development environments with Nix

As mentioned, **Nix** expressions are written in the Nix programming language, which is purely functional. Here is a simple example that creates a shell environment containing version 4.3.1 of R:

```
let
  pkgs = import (fetchTarball
    "https://github.com/NixOS/nixpkgs/archive/976fa336.tar.gz"
  ) {};
  system_packages = builtins.attrValues {
    inherit (pkgs) R;
  };
in
  pkgs.mkShell {
    buildInputs = [ system_packages ];
    shellHook = "R --vanilla";
  }
```

In this expression, the `let` keyword is used to define variables. The variable `pkgs` imports the set of packages from the `nixpkgs` repository at the specified commit `976fa336`. The variable `system_packages` lists the packages to include in the environment; in this case, it is just the R programming language, along with all its dependencies and their transitive dependencies. The `mkShell` function then creates a development shell with the specified packages. The `shellHook` is set to `"R --vanilla"`, meaning that entering the shell automatically starts R in vanilla mode, ignoring any startup options.

This expression can be saved in a file called `default.nix`. The environment can then be built on a system with **Nix** installed using the `nix-build` command.[4]. Once the build completes, the user can enter the interactive shell with `nix-shell`, which executes the following steps:

[4]For installing **Nix**, we recommend the Determinate Systems installer: https://determinate.systems/posts/determinate-nix-installer

- **Nix** reads the expression and identifies what needs to be installed;
- It checks if these packages already exist in `/nix/store`;
- If not, it downloads or builds them (with all dependencies);
- It creates a new shell environment and updates the `$PATH` to include the right binaries from `/nix/store`.
- The user is now in an interactive **Nix** shell session.

This shell contains all the packages specified in `default.nix` and can be used for development, similar to activating a virtual environment in the Python ecosystem.

Writing **Nix** expressions can be challenging for users unfamiliar with the Nix language. However, the ability to define a fully reproducible development environment in a single text file and then rebuild it anywhere is highly appealing. **rix** aims to lower the barrier to adoption of **Nix** for reproducibility.

rix, an R package, provides the `rix()` function, which simplifies generating **Nix** expressions. It is available on CRAN and can be installed like any other R package. Additionally, it can bootstrap an R development environment on a system where R is not yet installed but **Nix** is available. This can be done by running (inside of a terminal):

```
$> nix-shell -I \
+    nixpkgs=https://github.com/rstats-on-nix/nixpkgs/tarball/2025-10-20 -p \
+    R rPackages.rix
```

(the `-I` flag allows one to pass a specific revision of `nixpkgs`, ensuring temporary shells are also reproducible).

This command opens a temporary R session with **rix** available.[5] From there, users can generate new **Nix** expressions for building environments. For example, the following generates a `default.nix` file that installs R 4.3.1 along with the **dplyr** and **chronicler** packages:

```
R> library('rix')

R> rix(r_ver = "4.3.1",
+    r_pkgs = c("dplyr", "chronicler"),
+    project_path = ".",
+    overwrite = TRUE)
```

rix can also handle more complex setups, and users can provide a date instead of a specific R version:

```
R> rix(date = "2025-10-20",
+    r_pkgs = c("rix", "dplyr", "chronicler", "AER@1.2-8"),
+    system_pkgs = c("quarto", "git"),
+    tex_pkgs = c(
+          "amsmath",
+          "framed",
+          "fvextra",
+          "environ",
+          "fontawesome5",
```

[5] `nix-shell -p` starts an interactive shell with the specified packages.

```
+          "orcidlink",
+          "pdfcol",
+          "tcolorbox",
+          "tikzfill"
+    ),
+    git_pkgs = list(
+        package_name = "fusen",
+        repo_url = "https://github.com/ThinkR-open/fusen",
+        commit = "60346860111be79fc2beb33c53e195f97504a667"
+    ),
+    ide = "positron",
+    project_path = ".",
+    overwrite = TRUE)
```

This call to `rix()` generates a `default.nix` file for a development shell that encapsulates a complete and reproducible research environment. It provides R and its packages (including **AER** at version 1.2-8), several TeXLive packages for LaTeXdocument authoring, development versions of **rix** and **fusen** pulled directly from GitHub at a specific commit, and the Positron editor. The reproducibility of this environment is guaranteed by pinning all components to a single point in time: the R packages are resolved from the CRAN snapshot of October 20th 2025, while all other system tools are fixed to the version of `nixpkgs` from that same date. Typing `nix-shell` in a terminal within the folder that contains this `default.nix` will drop the user into a new shell. It is also possible to configure IDEs to dynamically load this new shell to provide the proper development tooling for better interactive use.

It is also possible to include Python and Julia packages using the `py_conf` and `jl_conf` arguments respectively. Julia packages work the same way as R packages: they are pinned to their versions as of the specified date.

Python packages, however, require special handling. Unlike CRAN or the Julia package registry, PyPI (the Python Package Index) contains over 500,000 packages with highly variable quality—many lack proper build specifications, have broken dependencies, or are abandoned. Because of this heterogeneity, **Nix** maintainers cannot automatically package all of PyPI as they do for CRAN or the Julia registry. Instead, they manually curate and package individual Python packages that meet quality standards.[6] This means some Python packages, or specific versions, may not be available through **Nix**.

For projects requiring Python packages not available in **Nix**, **rix** supports integration with **uv**, a modern Python package manager. This allows users to combine Nix's system-level reproducibility (Python interpreter, system libraries) with **uv**'s comprehensive package availability. While this hybrid approach sacrifices some of Nix's deterministic build guarantees for Python packages, it provides a pragmatic solution when the needed packages are unavailable in `nixpkgs`.

Users can include **uv** as a system package:

```
R> rix(date = "2025-10-20",
+    r_pkgs = c("rix", "dplyr", "chronicler"),
```

[6] On October 2025, this amounts to roughly 10000 packages, according to the search engine for **Nix** packages.

```
+    system_pkgs = c("uv"),
+    project_path = ".",
+    overwrite = TRUE)
```

This hybrid approach offers complementary strengths: **Nix** manages the Python interpreter, system libraries (e.g., **GDAL**, **HDF5**), and R packages with full determinism, while **uv** handles Python packages through its own lockfile mechanism (`uv.lock`). Within the Nix environment, users run standard **uv** commands (e.g., `uv pip install pandas==2.0.0`) to install Python packages. The **uv** lockfile records exact package versions and their hashes, providing reproducibility for the Python package layer.

While this approach doesn't achieve Nix's level of build-time determinism for Python packages (**uv** still downloads wheels or builds packages at install time rather than using pre-built, content-addressed artefacts) it provides a pragmatic balance: comprehensive Python package availability with reasonable reproducibility guarantees. For polyglot projects, this means R and system-level dependencies remain fully reproducible through **Nix**, while Python packages gain the reproducibility that **uv** can provide. We recommend of course to use **Python** through **Nix** whenever possible and only resort to the hybrid approach with **uv** as a last resort.[7]

Anywhere **Nix** can be installed, users can benefit from its features. For instance, the repository containing the source code for this article[8] uses GitHub Actions to compile the paper. Each time a push is made to the master branch, a runner installs **Nix**, generates the environment from the hosted `default.nix` file, and compiles the paper using **Quarto** within the reproducible environment. This ensures that *exactly* the same environment is used on the authors' machines and on the CI/CD platform without any additional, platform-specific, configuration.

Instead of first entering a **Nix** shell, it is also possible to run a program directly from the environment:

```
cd /path/to/project/ && nix-shell default.nix --run "Rscript analysis.R"
```

This command runs `Rscript` and executes the `analysis.R` script, which in this example should be located in the same directory as `default.nix`.

4. The rstats-on-nix fork of nixpkgs

As explained earlier, **Nix** uses expressions from the `nixpkgs` GitHub repository to build software. However, when generating expressions with **rix**, the fork `rstats-on-nix/nixpkgs` is used instead.

Using a fork offers several advantages. First, it provides flexibility that the official `nixpkgs` repository cannot always accommodate. Because **Nix** serves as the package manager for the NixOS Linux distribution, upstream priorities center on system-level stability rather than rapid updates to language ecosystems like R or Julia.

[7]It is also possible to package the missing **Python** for **Nix** and submit a pull request to `nixpkgs`, but we recognise that this option may not be beginner-friendly.

[8]https://github.com/b-rodrigues/rix_paper

For instance, while **Nix** can theoretically support multiple versions (or *variants*) of the same package, in practice maintainers cannot provide several variants for all R packages, given the size of the ecosystem (over 20,000 CRAN and Bioconductor packages). This makes it difficult to install a specific version of an R package not included in a particular `nixpkgs` commit. With **rix**, users can install a specific package version from source, e.g.:

```
R> rix(..., r_pkgs = "dplyr@1.0.7", ...)
```

However, installing from source might fail, especially if the package needs to be compiled.

Additionally, updating the full R package set on **Nix** daily is impractical. While CRAN and Bioconductor update daily, the R packages in `nixpkgs` are only updated with new R releases. This limitation is due to **Nix**'s governance as a Linux distribution package manager.

The `rstats-on-nix` fork allows us to circumvent these limitations. For example, it provides daily snapshots of CRAN and the Julia. Each day, the R package set is updated and committed to a dated branch using GitHub Actions. Users can select a specific date with:

```
R> rix(date = "2024-12-14", ...)
```

We strive to provide an available date per week: each Monday, a GitHub Action tests popular R packages on Linux and macOS, and only if all tests succeed is the date added to the list of available dates in **rix**. Users can see all available dates with `rix::available_dates()`. This ensures users can reliably install packages, and allows us to backport fixes if needed. For example, when RStudio was temporarily broken due to a dependency issue (**boost**), a pull request was submitted to the official `nixpkgs` repository. I backported the fix to the `rstats-on-nix` fork, making RStudio available to users of **rix** earlier than upstream, as merging PRs in the official repository can take some time.

We have backported fixes to the `rstats-on-nix nixpkgs` fork as far back as March 2019. The process involves checking out a `nixpkgs` commit on the selected date, updating the R package set using Posit CRAN and Bioconductor snapshots, backporting fixes, and ensuring popular packages work on both x86-linux (including WSL2) and aarch64-darwin (Apple Silicon). These changes are committed to a dated branch in `rstats-on-nix/nixpkgs`.

A drawback of forking `nixpkgs` is that backported packages are not included upstream and thus are not prebuilt by Nix's CI platform, Hydra. Users may need to build many packages from source, which can be time-consuming. To mitigate this, we provide a binary cache sponsored by Cachix[9], complementing the public Nix cache. Instructions for using Cachix are in **rix**'s documentation. Using the cache significantly speeds up installations, as prebuilt packages are downloaded rather than compiled.

The fork also allows us to catch issues (such as packages' builds breaking) early on, and prepare fixes that can then be contributed upstream.

While the reliance on the `rstats-on-nix` fork might initially raise sustainability concerns, this risk should be contextualized. The fork serves a focused purpose: it updates CRAN and Julia package registries daily and backports fixes when builds break. Importantly, users are never locked into this fork—they retain the ability to point directly to any commit in the upstream `nixpkgs` repository, maintaining full access to the official **Nix** ecosystem. The

[9] https://www.cachix.org/

fork introduces no fundamental architectural changes or proprietary extensions; it functions as a convenience layer providing tested package sets and dated snapshots. Should maintenance cease, users could seamlessly transition to upstream `nixpkgs` commits, though with trade-offs: less frequent R and Julia package updates. The fork's value lies not in creating dependency, but in reducing friction through testing infrastructure and curated "known-good" dates. Thus, the fork represents an enhancement to user experience rather than a critical single point of failure—the underlying **Nix** infrastructure and official `nixpkgs` repository remain the foundation, ensuring long-term viability of environments created with **rix**.

5. Orchestrating the workflow with rixpress

Defining a reproducible environment with **rix** addresses the first major challenge of reproducibility: environment capture. The second challenge, reliably and efficiently executing the analysis workflow within that environment, is addressed by **rixpress** (or **ryxpress** for Python users).

As mentioned in the introduction, a build automation tool like **targets** is invaluable for managing complex analyses. It tracks dependencies between code and data, caches results, and only recomputes steps that have changed. One can run a **targets** pipeline inside a Nix environment to make it reproducible. However, this approach has limitations: the entire pipeline must run in a single environment, and orchestrating steps across different languages (e.g., R and Python) requires manual handling via packages like **reticulate**.

rixpress overcomes these limitations by using **Nix** not just as a package manager, but as the build automation engine itself. In a **rixpress** pipeline, each step is defined as a **Nix** derivation, providing two key benefits:

1. True Polyglot Pipelines: Each step can have its own **Nix** environment. A Python step can run in a pure Python environment, an R step in an R environment, and a Quarto rendering step in yet another, all within the same pipeline.
2. Deep Reproducibility: Each step is a hermetically sealed **Nix** derivation whose output is cached in the **Nix** store based on the hash of all its inputs. All artefacts depend directly on their computational environment, ensuring that any change in dependencies triggers a rebuild.

As an example, we demonstrate a polyglot pipeline that simulates a Real Business Cycle (RBC) model in Julia, trains an **XGBoost** forecasting model in Python on the simulated data, visualises results in R, and compiles a **Quarto** report. The code for this example can be found in the following repository[10], with additional details provided in the Appendix of this manuscript. Although this example is purely synthetic and not of scientific significance, it effectively demonstrates the capabilities of **rixpress**. It should also be noted that the code Julia, Python and R was generated by a large language model.

For new projects, **rixpress** provides `rxp_init()`, which generates the necessary boilerplate structure including a pipeline definition file and template helper scripts. Users then define their pipeline in the script `gen-pipeline.R` using functions inspired by **targets**, and the computational environment the pipeline will be executed in the `gen-env.R` script. If functions

[10]https://github.com/b-rodrigues/rixpress_demos/tree/master/case-study

are needed for pipeline steps, these are to be specified, in language-specific helper scripts: `functions.jl` for Julia, `functions.py` for Python, and `functions.R` for R. This separation keeps the pipeline definition clean and readable while encapsulating the scientific logic in appropriate language-native files. Importantly, these helper scripts are standard language files—they contain no **rixpress**-specific code and can be reused in other projects or executed independently outside of **Nix**. This design allows users to preserve standard development practices while still benefiting from declarative orchestration.

The following example demonstrates how **rixpress** orchestrates a granular, multi-step data science workflow that crosses language boundaries. (Complete setup instructions and a detailed walkthrough are provided in Appendix)

```
R> library('rixpress')

R> pipeline_steps <- list(
+  # STEP 0: Define RBC Model Parameters in Julia
+  rxp_jl(alpha, 0.3),
+  rxp_jl(beta, 1 / 1.01),
+  # ... (other parameters omitted for brevity) ...
+
+  # STEP 1: Julia - Simulate the RBC model
+  rxp_jl(
+    name = simulated_rbc_data,
+    expr = "simulate_rbc_model(alpha, beta, delta, rho, sigma, sigma_z)",
+    user_functions = "functions/functions.jl",
+    encoder = "arrow_write"
+  ),
+
+  # STEP 2.1: Python - Prepare features
+  rxp_py(
+    name = processed_data,
+    expr = "prepare_features(simulated_rbc_data)",
+    user_functions = "functions/functions.py",
+    decoder = "pyarrow.feather.read_feather"
+  ),
+
+  # STEP 2.2: Python - Split data (X_train, y_train, etc.)
+  rxp_py(name = X_train, expr = "get_X_train(processed_data)", ...),
+  # ... (other data splits omitted for brevity) ...
+
+  # STEP 2.3: Python - Train the XGBoost model
+  rxp_py(
+    name = trained_model,
+    expr = "train_model(X_train, y_train)",
+    user_functions = "functions/functions.py"
+  ),
+
+  # STEP 2.4: Python - Make predictions and format results
```

```
+  # ... (prediction and formatting steps omitted for brevity) ...
+  rxp_py(
+    name = final_predictions_df,
+    expr = "format_results(y_test, model_predictions)",
+    user_functions = "functions/functions.py",
+    encoder = "save_arrow"
+  ),
+
+  # STEP 3: R - Visualise the predictions
+  rxp_r(
+    name = output_plot,
+    expr = plot_predictions(final_predictions_df),
+    user_functions = "functions/functions.R",
+    decoder = arrow::read_feather
+  ),
+
+  # STEP 4: Quarto - Compile the final report
+  rxp_qmd(
+    name = final_report,
+    qmd_file = "readme.qmd"
+  )
+)

# Generate the 'pipeline.nix' file from the R list
R> rxp_populate(pipeline_steps, build = TRUE)
```

Computation steps, referred to as *derivations*, are defined using the functions `rxp_r()`, `rxp_jl()`, and `rxp_py()`, corresponding to R, Julia, and Python environments, respectively. Variants of these functions suffixed with `_file()` designate *import derivations*, which introduce external data into the pipeline. It is important to note that any data imported in this way is stored within the **Nix** store, ensuring reproducibility but also persisting the data in the build environment. The function `rxp_qmd()` is used to compile a **Quarto** document (there is also `rxp_rmd()` which serves the same purpose for **R Markdown** documents).

The `rxp_populate()` function translates this R list into a `pipeline.nix` file, which declaratively defines the entire workflow. Data flows from derivation to derivation by being serialised into the efficient and language-agnostic **Arrow** format using the pair of `encoder/decoder` functions. Other universal formats, such as `csv` or `json` could have been used. It should also be noted that if interoperability libraries such as **reticulate** or **rds2py** are available, it becomes possible to seamlessly transfer supported objects to and from R and Python (for now, such a feature is not yet integrated for conversion of arbitrary Julia objects).

As a sidenote: Python users who wish to use **ryxpress** define pipelines using the same R-based DSL shown above. This design choice keeps the pipeline definition language consistent across both R and Python ecosystems. **ryxpress** calls R and **rixpress** under the hood to generate and build the `pipeline.nix` file. To use **ryxpress**, one should simply add this package to the `py_conf` argument of `rix()`.

The package can then generate a visual representation of the pipeline's directed acyclic graph

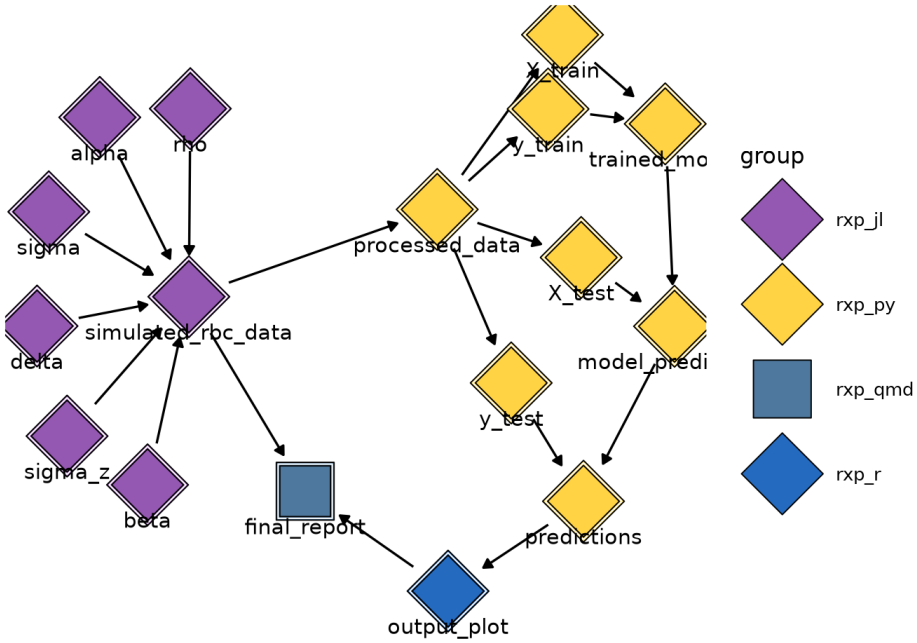

Figure 2: Graphical representation of the polyglot pipeline. Purple nodes are Julia, yellow nodes are **Python**, light blue nodes are R, and dark blue nodes are Quarto derivations.

by running `rxp_dag()`, as can be seen in Figure 2.

To execute the pipeline, one can either set `build = TRUE` in `rxp_populate()` or call `rxp_make()` separately. **Nix** executes each step in order, building dependencies as needed. Outputs are cached, so subsequent runs only recompute steps with changed inputs or code. This provides the efficiency of **targets** with polyglot support and bit-for-bit reproducibility. Artefacts can be inspected interactively in R using `rxp_read("artefact_name")` or `rxp_load("artefact_name")`.

rixpress also includes several additional features not covered here for brevity. As mentioned previously, it is also possible to configure popular IDEs to work interactively and seamlessly with both **rix** and **rixpress**, enabling a smooth, reproducible workflow from within the development environment. Detailed setup instructions are provided in the vignettes of both packages.

In summary, **rixpress** extends the guarantees of **rix** from static environments to dynamic workflows, enabling end-to-end, polyglot reproducibility.

6. A Comparison of Pipeline Paradigms: Imperative vs. Declarative

To illustrate the practical benefits of our framework, we implemented the RBC pipeline example from Section Section 5 using two distinct paradigms. The first is a traditional, imperative, stack combining Docker for environment isolation, Make for workflow orchestration,

and language-specific package managers.[11] The second is our proposed declarative framework using `rix` and `rixpress`. While both approaches successfully execute the same polyglot analysis to produce identical outputs, their comparison illuminates fundamental differences in conceptual models, system complexity, and the nature of the reproducibility guarantees they provide.

6.1. The Imperative Composition of Specialized Tools

The traditional approach represents a composition of specialized tools, each solving part of the reproducibility puzzle. This paradigm is purely imperative; the researcher must explicitly script the procedural steps for both environment construction and workflow execution.

The codification of the environment is a procedural build process specified in a `Dockerfile`. Successfully authoring such a file necessitates that the researcher act as a system administrator, manually scripting a sequence of shell commands to install each component. This includes managing `apt` repositories for R, invoking external installers like `uv` and `rig` for Python and R runtimes, and using `curl` and `tar` for Julia. Subsequently, separate commands must be run to install packages within each language's ecosystem. The procedural verbosity and intertwined concerns of this approach are reflected in a `Dockerfile` spanning 108 lines to define the complete environment.

Here is a sketch of this `Dockerfile`:

```
# Add R repository and install specific version
RUN apt-get update && apt-get install -y software-properties-common
RUN add-apt-repository ppa:...
RUN curl -L https://rig.r-pkg.org/... | sh
RUN rig add 4.5.1

# Install Python with uv
RUN curl -LsSf https://astral.sh/uv/install.sh | sh
RUN uv python install 3.13

# Download and extract Julia
RUN curl -fsSL https://julialang-s3.julialang.org/... -o julia.tar.gz
RUN tar -xzf julia.tar.gz -C /opt/

# Install packages for each language separately
RUN echo 'options(repos = c(CRAN = ...))' > /root/.Rprofile
RUN Rscript -e 'install.packages(...)'

# Install Python packages using uv with specific versions for reproducibility.
RUN echo "pandas==2.3.3" > /tmp/requirements.txt && \
    echo "scikit-learn==1.7.2" >> /tmp/requirements.txt && \
    # ... more packages ...
```

[11] This choice of tools is illustrative: we could have used, for instance, **Snakemake** (Mölder, Jablonski, Letcher *et al.* 2021) for workflow orchestration or **conda** (ana 2016) instead of **uv**, but the broader point remains—these are all imperative approaches

```
RUN uv pip install --no-cache -r /tmp/requirements.txt && \
    rm /tmp/requirements.txt

# Install specific versions of Julia packages for reproducibility
RUN julia -e 'using Pkg; \
    Pkg.add(name="Arrow", version="2.8.0"); \
```

This procedural philosophy extends from environment definition to workflow orchestration. A `Makefile` defines the pipeline as a directed acyclic graph of file dependencies. While robust, this model requires the researcher to manually encode the dependency graph, tightly coupling the analytical logic to specific file paths. Furthermore, since `make` is agnostic to the content of these files, it provides no mechanism to enforce type or schema consistency between steps.

A direct consequence of this file-based orchestration model is the proliferation of wrapper scripts. Each step in the `Makefile` invokes a script containing a significant amount of procedural boilerplate unrelated to the scientific logic. This includes parsing command-line arguments, handling data serialization and deserialization, and managing file paths. The core analytical logic is thus obscured by necessary but extraneous scaffolding.

The cumulative effect is a system of high structural complexity. The traditional implementation requires nine separate files to fully specify the environment, workflow, and analysis. This includes the `Dockerfile`, `Makefile`, three wrapper scripts for orchestration, and three `functions` files containing the core logic, in addition to the final report. The cognitive overhead is substantial, as a user must comprehend the conventions and interactions of this entire toolchain.

6.2. A Unified Declarative Framework

The Nix-based framework inverts this imperative model. Instead of scripting the procedural construction of the environment and workflow, the researcher declares the desired end-state.

Environment specification is reduced to a single declarative statement in the `gen-env.R` script. The researcher specifies *what* the environment should contain, leaving the procedural implementation of *how* to build it entirely to the Nix toolchain. The `gen-env.R` script replaces the 108-line `Dockerfile`, abstracting away all system administration details. `rix` translates this high-level specification into a formal Nix expression that deterministically resolves all transitive dependencies for the complete polyglot environment.

```
R> library(rix)
R> rix(
+   date = "2025-10-14",
+   r_pkgs = c("ggplot2", "dplyr", "arrow"),
+   jl_conf = list(jl_version = "lts", ...),
+   py_conf = list(py_version = "3.13", ...),
+   ...
+ )
```

Workflow orchestration undergoes a similar conceptual shift from procedural to declarative.

The `gen-pipeline.R` script defines the pipeline not as a set of file rules, but as a graph of data objects. Dependencies are not manually specified but are automatically inferred by `rixpress` from the functional relationships in the code. This object-centric model renders the procedural wrapper scripts unnecessary. The scientific logic is expressed as a set of pure functions that accept data structures as inputs and return them as outputs, free of any boilerplate for file I/O or command-line parsing.

```
# functions/functions.py
>>> def train_and_predict_output(simulated_df: pd.DataFrame) -> pd.DataFrame:
+     # ... scientific logic ...
+     return format_results(predictions)
```

This declarative approach significantly reduces system complexity. The entire implementation requires only six files: two for the high-level environment and pipeline declarations, three containing the pure scientific functions, and the final report. The reduction is not merely quantitative; the remaining files are conceptually simpler, containing only analytical logic and specifications, with all procedural scaffolding abstracted away by the framework.

6.3. Reproducibility Guarantees: Space and Time

The critical distinction between the two paradigms lies in the nature of their reproducibility guarantees. The traditional stack, anchored by Docker, provides strong **spatial reproducibility**—the ability to execute the analysis identically across different machines. However, it is vulnerable to **temporal drift**, a concept that distinguishes reproducibility across space from consistency over time (Malka, Zacchiroli, and Zimmermann 2024). This temporal vulnerability arises from several sources, including the resolution of mutable base image tags (e.g., `ubuntu:24.04`) and the potential for language-level package managers to resolve dependencies differently over time.

The Nix-based approach is architected to solve this problem of temporal drift. By pinning the entire software ecosystem to a single, immutable revision of the `nixpkgs` repository, the framework provides strong guarantees of **temporal reproducibility**. This claim is supported by large-scale empirical evidence; a study by Malka *et al.* (2025), which rebuilt over 700,000 historical packages, confirmed that Nix's functional model achieves a bit-for-bit reproducibility rate exceeding 90% and a rebuildability rate over 99% across a multi-year time-frame.

6.4. Summary of Differences

Dimension	Traditional Stack	Nix-Based Framework with rix and rixpress
Paradigm	Imperative composition of specialized tools	Declarative specification in unified framework
Environment setup	Step-by-step shell commands	High-level specification
Workflow definition	File-based rules in Make syntax	Object-based graph in R syntax

Dimension	Traditional Stack	Nix-Based Framework with rix and rixpress
Scientific code	Wrapped in CLI parsing and file I/O boilerplate	Pure functions with no plumbing
File count	9 files (4 for orchestration and plumbing)	6 files (2 for orchestration)
Expertise required	Linux system administration, Make syntax	R programming
Spatial reproducibility	Strong	Strong
Temporal reproducibility	Moderate (vulnerable to drift)	Strong

This declarative rigor, however, is not without pragmatic costs. Our validation on GitHub Actions shows that the imperative Docker-based runner completes the entire pipeline in approximately three minutes, whereas the declarative Nix-based runner takes around five minutes.[12] Nix's deterministic dependency resolution guarantees bit-for-bit reproducibility at the expense of initial computation time, whereas the imperative system prioritizes immediacy over long-term stability.

7. Conclusion

Many tools exist to improve reproducibility, but **Nix** stands out because it deploys complete software environments closed under the "depends on" relation: it installs not only a package but all of its transitive dependencies. This makes **Nix** uniquely suited for reproducible research.

Yet solving such a complex problem makes **Nix** itself complex, as it relies on a functional paradigm still likely unfamiliar to most researchers. **rix** lowers this barrier for R users by providing a familiar interface and workflow. By declaratively specifying reproducible development shells with **Nix**, researchers can accommodate diverse use cases—from running analytical pipelines to developing interactive **shiny** applications or serving **plumber** APIs.

Furthermore, **rixpress** extends this declarative model to entire analysis pipelines. By leveraging **Nix** as a unified build automation system, **rixpress** enables polyglot workflows where each step runs in a hermetically sealed environment. This ensures both spatial and temporal reproducibility, efficient caching, and seamless orchestration of multi-language analyses. Crucially, it provides native polyglot support without the manual orchestration required by container-based approaches, making complex workflows accessible without systems administration expertise.

Adopting **Nix**-based workflows represents a conceptual shift from procedural scripting to declarative specification. A practical adoption pathway begins with using **rix** to generate reproducible environments for new projects while maintaining existing workflows. This allows

[12] To run the benchmark yourself, fork the paper's repository, enable GitHub Actions, and push a single change to trigger the runs.

gradual learning of **Nix** concepts without disrupting active research. As confidence grows, researchers can extend to **rixpress** for smaller analyses before tackling complex pipelines. Institutions can support adoption by offering workshops, maintaining binary caches, and designating local experts. While early projects require investment, later projects benefit from reusable environment definitions and cumulative expertise. Ultimately, adoption depends on project scope, reproducibility needs, and institutional context—but the framework now exists for those who require robust, long-term reproducibility.

While **rix** and **rixpress** (and **ryxpress**) represent significant progress, current limitations remain. Debugging complex **rixpress** pipelines is more challenging than troubleshooting **targets** workflows, especially when failures occur deep in the **Nix** build process. Visualization capabilities, though useful, currently lack features such as automatic detection of outdated derivations. Cross-platform reproducibility remains generally strong but is constrained on macOS by proprietary framework dependencies. These challenges are active areas for development and community contribution. Researchers adopting these tools should do so with realistic expectations: they provide strong guarantees of spatial and temporal reproducibility and scalable polyglot workflows, but they require patience during the learning phase and may occasionally present challenges that simpler tools avoid.

Acknowledgments

We thank the rOpenSci reviewers and contributors who provided valuable feedback on the development of **rix** and **rixpress**. In particular, we are grateful to David Watkins and Jacob Wujiciak-Jens for their reviews of **rix**, and to William Landau and Anthony Martinez for their reviews of **rixpress**. We also acknowledge the contributions of Richard J. Acton, Jordi Rosell, Elio Campitelli, László Kupcsik, and Michael Heming for **rix**. Their expertise and feedback greatly improved the quality and usability of these packages. We would also like to thank Pol Dellaiera and Edvin Syk for providing feedback on the manuscript.

References

(2016). *Anaconda Software Distribution*. URL https://docs.anaconda.com/.

Astral (2024). "uv: An Extremely Fast Python Package and Project Manager." https://github.com/astral-sh/uv. Accessed: 2025-10-28; written in Rust and designed as a drop-in replacement for pip/pip-tools/virtualenv.

Bezanson J, Edelman A, Karpinski S, Shah VB (2017). "Julia: A fresh approach to numerical computing." *SIAM review*, **59**(1), 65–98. URL https://doi.org/10.1137/141000671.

Bhandari Neupane J, Neupane RP, Luo Y, Yoshida WY, Sun R, Williams PG (2019). "Characterization of Leptazolines A–D, Polar Oxazolines from the Cyanobacterium Leptolyngbya sp., Reveals a Glitch with the "Willoughby–Hoye" Scripts for Calculating NMR Chemical Shifts." *Organic Letters*, **21**(20), 8449–8453. doi:10.1021/acs.orglett.9b03332.

Boettiger C, Eddelbuettel D (2017). "An Introduction to Rocker: Docker Containers for R." *The R Journal*, **9**(2), 527–536. doi:10.32614/RJ-2017-065.

Chen A, Wong C, Sharif B, Peruma A (2025). "Exploring Code Comprehension in Scientific Programming: Preliminary Insights from Research Scientists." 2501.10037, URL https://arxiv.org/abs/2501.10037.

De Paoli B (2009). "Slides 1: The RBC Model, Analytical and Numerical solutions." https://personal.lse.ac.uk/depaoli/RBC_slides1.pdf. Lecture Slides.

Dellaiera P (2024). "Reproducibility in Software Engineering." doi:10.5281/zenodo.12666898.

Dolstra E, De Jonge M, Visser E (2004). "**Nix**: A Safe and Policy-Free System for Software Deployment." In *18th Large Installation System Administration Conference*, pp. 79–92.

Eddelbuettel D (2025). "r2u: CRAN as Ubuntu Binaries." https://eddelbuettel.github.io/r2u/. Accessed: 2025-10-28, URL https://eddelbuettel.github.io/r2u/.

Fay C, Guyader V, Parry J, Rochette S (2024). ***dockerfiler**: Easy Dockerfile Creation from R*. R package version 0.2.5, URL https://thinkr-open.github.io/dockerfiler/.

hong Chan C, Schoch D (2023). "**rang**: Reconstructing Reproducible R Computational Environments." *PLOS ONE*, **18**(6), e0286761. doi:10.1371/journal.pone.0286761.

Landau WM (2021). "The **targets** R Package: A Dynamic Make-like Function-Oriented Pipeline Toolkit for Reproducibility and High-Performance Computing." *Journal of Open Source Software*, **6**(57), 2959. doi:10.21105/joss.02959.

Malka J, Zacchiroli S, Zimmermann T (2024). "Reproducibility of Build Environments through Space and Time." In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, p. 97–101. ACM. doi:10.1145/3639476.3639767. URL http://dx.doi.org/10.1145/3639476.3639767.

Malka J, Zacchiroli S, Zimmermann T (2025). "Does Functional Package Management Enable Reproducible Builds at Scale? Yes." In *22nd International Conference on Mining Software Repositories*. Ottawa, Canada. URL https://hal.science/hal-04913007.

McDermott GR (2021). "Skeptic Priors and Climate Consensus." *Climatic Change*, **166**(1-2), 7. doi:10.1007/s10584-021-03114-6.

Mölder F, Jablonski KP, Letcher B, *et al.* (2021). "Sustainable data analysis with Snakemake." *F1000Research*, **10**, 33. doi:10.12688/f1000research.29032.1. URL https://doi.org/10.12688/f1000research.29032.1.

Navarro ALG, Koneva N, Sánchez-Macián A, Hernández JA (2024). "A Comprehensive Guide to Combining R and Python code for Data Science, Machine Learning and Reinforcement Learning." 2407.14695, URL https://arxiv.org/abs/2407.14695.

Osborne I, Pivarski J, Ling J (2024). "Bridging Worlds: Achieving Language Interoperability between Julia and Python in Scientific Computing." 2404.18170, URL https://arxiv.org/abs/2404.18170.

Peng B, Wang G, Ma J, Leong MC, Wakefield C, Melott J, Chiu Y, Du D, Weinstein JN (2018). "SoS Notebook: an interactive multi-language data analysis environment." *Bioinformatics*, **34**(21), 3768–3770. ISSN 1367-4803. doi:10.1093/bioinformatics/bty405. https://academic.oup.com/bioinformatics/article-pdf/34/21/3768/48921347/bioinformatics_34_21_3768.pdf, URL https://doi.org/10.1093/bioinformatics/bty405.

Peng RD (2011). "Reproducible Research in Computational Science." *Science*, **334**(6060), 1226–1227. doi:10.1126/science.1213847.

Posit, PBC (2025). "Introducing Portable Linux R Binary Packages." https://posit.co/blog/introducing-portable-linux-r-binary-packages/. Accessed: 2025-10-28, URL https://posit.co/blog/introducing-portable-linux-r-binary-packages/.

R Core Team (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

R Infrastructure (2023). ***rig**: The R Installation Manager*. URL https://github.com/r-lib/rig.

Simonsohn U, Gruson H (2023). *groundhog: Version-Control for CRAN, GitHub, and GitLab Packages*. R package version 3.1.2, https://github.com/CredibilityLab/groundhog, URL https://groundhogr.com/.

Van Rossum G, Drake FL (2009). *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA. ISBN 1441412697.

8. Appendix

8.1. Reproducing this paper

The source code of this paper is hosted on GitHub and can be found at following link[13]. The paper can be easily compiled by running the following command:

```
$> nix-shell --run "quarto render paper.qmd --to jss-pdf"
```

The `default.nix` file defines the exact computational environment required to compile the manuscript, ensuring full reproducibility of the build process. To guarantee the reproducibility of the manuscript, it is automatically recompiled via GitHub Actions upon each commit, using the same **Nix** environment. An HTML version is additionally deployed to GitHub Pages, providing an accessible format for viewing on smaller devices at this link[14]. The PDF version of the manuscript can be found at link[15].

8.2. A Complete Polyglot Example with rixpress

[13] https://github.com/b-rodrigues/rix_paper
[14] https://b-rodrigues.github.io/rix_paper/
[15] https://b-rodrigues.github.io/rix_paper/paper.pdf

This appendix provides a conceptual walk-through of the polyglot pipeline example discussed in Section 5. The pipeline simulates a RBC model in Julia, trains an **XGBoost** model in Python, visualises the results in R, and compiles a final report with **Quarto**.

The complete, runnable source code for this example is available in the paper's GitHub repository. A shell script, `repro_script.sh`, is provided to automate the entire process from environment creation to final output, as described at the end of this section[16].

Project Structure and Components

The project is organized into a set of scripts, each with a distinct responsibility. The core logic is separated from the environment definition and pipeline orchestration.

- `functions/`: This directory contains the helper scripts with the core analytical code for each language.
  - `functions.jl`: The Julia script for the economic simulation.
  - `functions.py`: The Python script for the machine learning workflow.
  - `functions.R`: The R script for data visualisation.
- `gen-env.R`: An R script that defines the reproducible computational environment.
- `gen-pipeline.R`: The main R script that defines and orchestrates the entire polyglot pipeline.

Step 1: The Environment Definition

The foundation of the project is the reproducible environment, defined in `gen-env.R`. This script uses the `rix()` function to programmatically generate a `default.nix` file. This file serves as a complete blueprint for the computational environment, specifying:

- The exact versions of R, Python, and Julia.
- A list of required packages for each language (e.g., **ggplot2** for R, **xgboost** for Python, **DataFrames** for Julia).
- System-level dependencies like **Quarto**.

Crucially, the entire environment is pinned to a specific date (`2025-10-14`), ensuring that anyone who builds the environment, now or in the future, will get the exact same software versions, guaranteeing reproducibility.

By dropping into a temporary shell using the following command:

```
$> nix-shell -I \
+  nixpkgs=https://github.com/rstats-on-nix/nixpkgs/tarball/2025-10-20 -p \
+  R rPackages.rix
```

it is possible to generate the `default.nix` by sourcing `gen-env.R`. Then, one leaves this temporary shell, and builds the environment using the command `nix-shell`, which also drop the user into the development shell.

Step 2: The Core Analytical Logic

[16] https://github.com/b-rodrigues/rix_paper/blob/master/repro-script/repro_script.sh

The scientific logic for each stage of the pipeline is encapsulated in the separate helper scripts within the `functions/` directory.

- The RBC Model Simulation (`functions.jl`): This Julia script contains a single pure function that implements the state-space solution to the RBC model, based on De Paoli (2009). It takes the model's economic parameters as inputs and returns the simulated time-series data as a **DataFrame**.
- The **XGBoost** Forecasting Model (`functions.py`): This Python script handles the complete machine learning workflow through a series of modular functions. Its responsibilities include feature engineering (creating lagged variables), splitting the data into training and testing sets, training the **XGBoost** model, and generating predictions.
- The Visualisation (`functions.R`): This R script contains a function that uses **ggplot2** to create the final visualisation. It is designed to take the data frame of actual and predicted values produced by the Python script and generate a plot comparing the two series.

Step 3: Orchestrating the Pipeline

The entire workflow is defined and orchestrated by `gen-pipeline.R`. This script acts as the master plan, using functions from the **rixpress** package to define each computational step as a *derivation*. It declaratively outlines the dependencies between steps:

1. It begins by defining the RBC model parameters in Julia.
2. It specifies that the RBC simulation in `functions.jl` depends on these parameters.
3. It then defines the series of Python steps (feature preparation, training, prediction), making each one dependent on the output of the previous one. The first Python step is explicitly made dependent on the output from the Julia simulation. **rixpress** handles the passing of data between languages, in this case using the **Arrow** file format for efficiency.
4. Finally, it defines the **R visualisation** step, which depends on the final predictions from the Python model, and a **Quarto rendering** step that depends on the generated plot.

When this script is run in the development shell (by executing `source("gen-pipeline.R")`), **rixpress** translates the declared pipeline into a master Nix expression that **Nix** can execute, automatically handling the caching of results and re-running only the necessary steps if a piece of code or data changes.

To build the pipeline from an interactive Python session, one would execute the following lines:

```
>>> from ryxpress import rxp_make
>>> rxp_make()
```

Running the Project

This entire example can be executed by running the `repro_script.sh` script available in the root of the paper's repository. This script automates the full process: 1. It first executes `gen-env.R` inside a temporary **Nix** shell to build the `default.nix` file. 2. It then executes

`gen-pipeline.R` inside the newly defined environment. This triggers **Nix** to run the entire polyglot pipeline in the correct order.

Upon completion, the script will have generated all intermediate artefacts and the final `readme.html` report found in the `pipeline-output` folder containing the visualisation.

Affiliation:

Bruno Rodrigues
Department of Statistics
18, Montée de la Pétrusse
Luxembourg Luxembourg
E-mail: bruno@brodrigues.co
URL: https://www.brodrigues.co

Philipp Baumann
Data and Analytics
7, Länggassstrasse
Bern Switzerland
E-mail: baumann-philipp@protonmail.com

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd