Building Reproducible Analytical Pipelines

Master of Data Science, University of Luxembourg - 2025

Bruno Rodrigues

2025-09-15

Table of contents

Introduction

This is the 2025 edition of the course. If you're looking for the 2024 edition, you can click here

What's new:

- Focus on Nix as the canonical tool for reproducibility and build automation
- Integration of LLMs as an additional tool in the reproducers toolbox.

This course is based on my book titled Building Reproducible Analytical Pipelines with R. This course focuses only on certain aspects that are discussed in greater detail in the book.

Schedule

- 2025/09/04 - 4 hours,
- 2025/09/11 - 4 hours,
- 2025/09/14 - 4 hours,
- 2025/09/02 - 4 hours,
- 2025/09/05 - 2 hours,
- 2025/09/09 - 5 hours,
- 2025/09/16 - 4 hours,
- 2025/09/19 - 3 hours,

Reproducible analytical pipelines?

This course is my take on setting up code that results in some *data product*. This code has to be reproducible, documented and production ready. Not my original idea, but introduced by the UK's Analysis Function.

The basic idea of a reproducible analytical pipeline (RAP) is to have code that always produces the same result when run, whatever this result might be. This is obviously crucial in research and science, but this is also the case in businesses that deal with data science/data-driven decision making etc.

A well documented RAP avoids a lot of headache and is usually re-usable for other projects as well.

Data products?

In this course each of you will develop a *data product*. A data product is anything that requires data as an input. This can be a very simple report in PDF or Word format or a complex web app. This website is actually also a data product, which I made using the R programming language. In this course we will not focus too much on how to create automated reports or web apps (but I'll give an introduction to these, don't worry) but our focus will be on how to set up a pipeline that results in these data products in a reproducible way.

Machine learning?

No, being a master in machine learning is not enough to become a data scientist. Actually, the older I get, the more I think that machine learning is almost optional. What is not optional is knowing how:

- to write, test, and properly document code;
- to acquire (reading in data can be tricky!) and clean data;
- to work inside the Linux terminal/command line interface;
- to use Git, Docker for Dev(Git)Ops;
- the Internet works (what's a firewall? what's a reverse proxy? what's a domain name? etc, etc…);

But what about machine learning? Well, depending what you'll end up doing, you might indeed focus a lot on machine learning and/or statistical modeling. That being said, in practice, it is very often much more efficient to let some automl algorithm figure out the best hyperparameters of a XGBoost model and simply use that, at least as a starting point (but good luck improving upon automl…). What matters, is that the data you're feeding to your model is clean, that your analysis is sensible, and most importantly, that it could be understood by someone taking over (imagine you get sick) and rerun with minimal effort in the future. The model here should simply be a piece that could be replaced by another model without much impact. The model is rarely central… but of course there are exceptions to this, especially in research, but every other point I've made still stands. It's just that not only do you have to care about your model a lot, you also have to care about everything else.

So in this course we're going to learn a bit of all of this. We're going to learn how to write reusable code, learn some basics of the Linux command line, Git and Docker.

What actually is reproducibility?

A reproducible project means that this project can be rerun by anyone at 0 (or very minimal) cost. But there are different levels of reproducibility, and I will discuss this in the next section. Let's first discuss some requirements that a project must have to be considered a RAP.

The requirements of a RAP

For something to be truly reproducible, it has to respect the following bullet points:

- Source code must obviously be available and thoroughly tested and documented (which is why we will be using Git and Github);
- All the dependencies must be easy to find and install (we are going to deal with this using dependency management tools);
- To be written with an open source programming language (nocode tools like Excel are by default non-reproducible because they can't be used non-interactively, and which is why we are going to use the R programming language);
- The project needs to be run on an open source operating system (thankfully, we can deal with this without having to install and learn to use a new operating system, thanks to Docker);
- Data and the paper/report need obviously to be accessible as well, if not publicly as is the case for research, then within your company.

Also, reproducibility is on a continuum, and depending on the constraints you face your project can be "not very reproducible" to “totally reproducible”. Let’s consider the following list of anything that can influence how reproducible your project truly is:

- Version of the programming language used;
- Versions of the packages/libraries of said programming language used;
- Operating System, and its version;
- Versions of the underlying system libraries (which often go hand in hand with OS version, but not necessarily).
- And even the hardware architecture that you run all that software stack on.

So by “reproducibility is on a continuum”, what I mean is that you could set up your project in a way that none, one, two, three, four or all of the preceding items are taken into consideration when making your project reproducible.

This is not a novel, or new idea. Peng (2011) already discussed this concept but named it the *reproducibility spectrum*.

Large Language Models

LLMs have rapidly become an essential powertool in the data scientist’s toolbox. But as with any powertool, beginners risk cutting their fingers if they’re not careful. So it is important to learn how to use them. This course will give you some pointers on how to integrate LLMs into your workflow.

Why R? Why not [insert your favourite programming language]

R is a domain-specific language whose domain is statistics, data analysis/science and machine learning, and as such has many built-in facilities to make handling data very efficient.

If you learn R you have access to almost 25'000 packages (as of June 2025, including both CRAN and Bioconductor packages) to:

- clean data (see: `{dplyr}`, `{tidyr}`, `{data.table}`…);
- work with medium and big data (see: `{arrow}`, `{sparklyr}`…);
- visualize data (see: `{ggplot2}`, `{plotly}`, `{echarts4r}`…);
- do literate programming (using Rmarkdown or Quarto, you can write books, documents even create a website);
- do functional programming (see: `{purrr}`…);
- call other languages from R (see: `{reticulate}` to call Python from R);
- do machine learning and AI (see: `{tidymodels}`, `{tensorflow}`, `{keras}`…)
- create webapps (see: `{shiny}`…)
- domain specific statistics/machine learning (see CRAN Task Views for an exhaustive list);
- and more

It's not just about what the packages provide: installing R and its packages and dependencies is rarely frustrating, which is not the case with Python (Python 2 vs Python 3, `pip` vs `conda`, `pyenv` vs `venv` vs `uv`, …, dependency hell is a real place full of snakes)

That doesn't mean that R does not have any issues. Quite the contrary, R sometimes behaves in seemingly truly bizarre ways (as an example, try running `nchar("1000000000")` and then `nchar(1000000000)` and try to make sense of it). To know more about such bizarre behaviour, I recommend you read *The R Inferno* (linked at the end of this chapter). So, yes, R is far from perfect, but it sucks less than the alternatives (again, in my absolutely objective opinion).

```
nchar("1000000000")
```

That being said, the reality of data science is that the future is becoming more and more polyglot. Data products are evermore complex, and necessity are built using many languages; so ideally we would like to find a way to use whatever tool is best fit for the job at hand. Sometimes it can be R, sometimes Python, sometimes shell scripts, or any other language. This is where Nix will help us.

Nix

Nix is a package manager for Linux distributions, macOS and it even works on Windows if you enable WSL2. What's a package manager? If you're not a Linux user, you may not be aware. Let me explain it this way: in R, if you want to install a package to provide some functionality not included with a vanilla installation of R, you'd run this:

```
install.packages("dplyr")
```

It turns out that Linux distributions, like Ubuntu for example, work in a similar way, but for software that you'd usually install using an installer (at least on Windows). For example you could install Firefox on Ubuntu using:

```
sudo apt-get install firefox
```

(there's also graphical interfaces that make this process "more user-friendly"). In Linux jargon, `packages` are simply what we call software (or I guess it's all "apps" these days). These packages get downloaded from so-called repositories (think of CRAN, the repository of R packages, or Pypi, in the case of Python) but for any type of software that you might need to make your computer work: web browsers, office suites, multimedia software and so on.

So Nix is just another package manager that you can use to install software.

But what interests us is not using Nix to install Firefox, but instead to install R, Python and the R and Python packages that we require for our analysis. But why use Nix instead of the usual ways to install software on our operating systems?

The first thing that you should know is that Nix's repository, `nixpkgs`, is huge. Humongously huge. As I'm writing these lines, there's more than 120'000 pieces of software available, and the *entirety of CRAN and Bioconductor* is also available through `nixpkgs`. So instead of installing R as you usually do and then use `install.packages()` to install packages, you could use Nix to handle everything. But still, why use Nix at all?

Nix has an interesting feature: using Nix, it is possible to install software in (relatively) isolated environments. So using Nix, you can install as many versions of R and R packages that you need. Suppose that you start working on a new project. As you start the project, with Nix, you would install a project-specific version of R and R packages that you would only use for that particular project. If you switch projects, you'd switch versions of R and R packages.

Pre-requisites

I will assume basic programming knowledge, and not much more. Ideally you'll be following this course from a Linux machine, but if you're macOS, that's fine as well. On Windows, you will have to set up WSL2 to follow along.

Grading

The way grading works in this course is as follows: during lecture hours you will follow along. At home, you'll be working on setting up your own pipeline. For this, choose a dataset that ideally would need some cleaning and/or tweaking to be usable. We are going first to learn how to package this dataset alongside some functions to make it clean. If time allows, I'll leave some time during lecture hours for you to work on it and ask me and your colleagues for help. At the end of the semester, I will need to download your code and get it running. The less effort this takes me, the better your score. Here is a tentative breakdown:

- Code is on github.com and the repository is documented with a Readme.md file: 5 points;
- Data and functions to run pipeline are documented and tested: 5 points;
- Every software dependency is easily installed: 5 points;
- Pipeline can be executed in one command: 5 points;
- Bonus points: pipeline is dockerized, or uses Nix, and/or uses Github Actions to run? 5 points

The way to fail this class is to write an undocumented script that only runs on your machine and expect me to debug it to get it to run.

Jargon

There's some jargon that is helpful to know when working with R. Here's a non-exhaustive list to get you started:

- CRAN: the Comprehensive R Archive Network. This is a curated online repository of packages and R installers. When you type `install.packages("package_name")` in an R console, the package gets downloaded from there;
- Library: the collection of R packages installed on your machine;
- R console: the program where the R interpreter runs;
- Posit/RStudio: Posit (named RStudio in the past) are the makers of the RStudio IDE and of the *tidyverse* collection of packages;
- tidyverse: a collection of packages created by Posit that offer a common language and syntax to perform any task required for data science — from reading in data, to cleaning data, up to machine learning and visualisation;
- base R: refers to a vanilla installation (and vanilla capabilities) of R. Often used to contrast a *tidyverse* specific approach to a problem (for example, using base R's `lapply()` in constrast to the *tidyverse* `purrr::map()`).
- `package::function()`: Functions can be accessed in several ways in R, either by loading an entire package at the start of a script with `library(dplyr)` or by using `dplyr::select()`.
- Function factory (sometimes adverb): a function that returns a function.
- Variable: the variable of a function (as in `x` in `f(x)`) or the variable from statistical modeling (synonym of feature)
- `<-` vs `=`: in practice, you can use `<-` and `=` interchangeably. I prefer `<-`, but feel free to use `=` if you wish.

Further reading

- An Introduction to R (from the R team themselves)

- What is CRAN?
- The R Inferno
- Building Reproducible Analytical Pipelines with R
- Reproducible Analytical Pipelines (RAP)

License

1 Reproducibility with Nix

1.1 Learning Outcomes

By the end of this chapter, you will:

- Understand the need for environment reproducibility in modern workflows
- Install Nix
- Use `{rix}` to generate `default.nix` files
- Build cross-language environments for data work or software development

1.2 Why Reproducibility? Why Nix? *(2h)*

1.2.1 Motivation: Reproducibility in Scientific and Data Workflows

To ensure that a project is reproducible you need to deal with at least four things:

- Make sure that the required/correct version of R (or any other language) is installed;
- Make sure that the required versions of packages are installed;
- Make sure that system dependencies are installed (for example, you'd need a working Java installation to install the rJava R package on Linux);
- Make sure that you can install all of this for the hardware you have on hand.

But in practice, one or most of these bullet points are missing from projects. The goal of this course is to learn how to fullfill all the requirements to build reproducible projects.

1.2.2 Problems with Ad-Hoc Tools

Tools like Python's `venv` or R's `renv` only deal with some pieces of the reproducibility puzzle. Often, they assume an underlying OS, do not capture native system dependencies (like `libxml2`, `pandoc`, or `curl`), and require users to "rebuild" their environments from partial metadata. Docker helps but introduces overhead, security challenges, and complexity.

Traditional approaches fail to capture the entire dependency graph of a project in a deterministic way. This leads to "it works on my machine" syndromes, onboarding delays, and subtle bugs.

1.2.3 Nix, a declarative package manager

Nix is a tool for reproducible builds and development environments, often introduced as a package manager. It captures complete dependency trees, from your programming language interpreter to every system-level library you rely on. With Nix, environments are not recreated from documentation, but rebuilt precisely from code.

Nix can be installed on Linux distributions, macOS and it even works on Windows if you enable WSL2. In this course, we will use Nix mostly as a package manager (but towards the end also as a build automatino tool).

What's a package manager? If you're not a Linux user, you may not know. Let me explain it this way: in R, if you want to install a package to provide some functionality not included with a vanilla installation of R, you'd run this:

```
install.packages("dplyr")
```

It turns out that Linux distributions, like Ubuntu for example, work in a similar way, but for software that you'd usually install using an installer (at least on Windows). For example you could install Firefox on Ubuntu using:

```
sudo apt-get install firefox
```

(there's also graphical interfaces that make this process "more user-friendly"). In Linux jargon, `packages` are simply what we call software (or I guess it's all "apps" these days). These packages get downloaded from so-called repositories (think of CRAN, the repository of R packages) but for any type of software that you might need to make your computer work: web browsers, office suites, multimedia software and so on.

So Nix is just another package manager that you can use to install software.

But what interests us is not using Nix to install Firefox, but instead to install R and the R packages that we require for our analysis (or any other programming language that we need). But why use Nix instead of the usual ways to install software on our operating systems?

The first thing that you should know is that Nix's repository, `nixpkgs`, is huge. Humongously huge. As I'm writing these lines, there's more than 120'000 pieces of software available, and the *entirety of CRAN and Bioconductor* is also available through `nixpkgs`. So instead of installing R as you usually do and then use `install.packages()` to install packages, you could use Nix to handle everything. But still, why use Nix at all?

Nix has an interesting feature: using Nix, it is possible to install software in (relatively) isolated environments. So using Nix, you can install as many versions of R and R packages that you need. Suppose that you start working on a new project. As you start the project, with Nix, you would install a project-specific version of R and R packages that you would only use for that particular project. If you switch projects, you'd switch versions of R and R packages.

However Nix has quite a steep learning curve, so this is why for the purposes of this course we are going to use an R package called `{rix}` to set up reproducible environments.

1.2.4 rix workflow

The idea of `{rix}` is for you to declare the environment you need using the provided `rix()` function. `rix()` is the package's main function and generates a file called `default.nix` which is then used by the Nix package manager to build that environment. Ideally, you would set up such an environment for each of your projects. You can then use this environment to either work interactively, or run R or Python scripts. It is possible to have as many environments as projects, and software that is common to environments will simply be re-used and not get re-installed to save space. Environments are isolated for each other, but can still interact with your system's files, unlike with Docker where a volume must be mounted. Environments can also interact with the software installed on your computer through the usual means, which can sometimes lead to issues. For example, if you already have R installed, and a user library of R packages, more caution is required to properly use environments managed by Nix.

You don't need to have R installed or be an R user to use `{rix}`. If you have Nix installed on your system, it is possible to "drop" into a temporary environment with R and `{rix}` available and generate the required Nix expression from there.

But first, let's install Nix and try to use temporary shells.

1.2.5 Installing Nix

If you are on Windows, you need the Windows Subsystem for Linux 2 (WSL2) to run Nix. If you are on a recent version of Windows 10 or 11, you can simply run this as an administrator in PowerShell:

```
wsl --install
```

You can find further installation notes at this official MS documentation.

We recommend to activate `systemd` in Ubuntu WSL2, mainly because this supports other users than `root` running Nix. To set this up, please do as outlined this official Ubuntu blog entry:

```
# in WSL2 Ubuntu shell

sudo -i
nano /etc/wsl.conf
```

This will open the `/etc/wsl.conf` in a nano, a command line text editor. Add the following line:

```
[boot]
systemd=true
```

Save the file with CTRL-O and then quit nano with CTRL-X. Then, type the following line in powershell:

```
wsl --shutdown
```

and then relaunch WSL (Ubuntu) from the start menu.

Installing (and uninstalling) Nix is quite simple, thanks to the installer from Determinate Systems, a company that provides services and tools built on Nix, and works the same way on Linux (native or WSL2) and macOS.

Do not use your operating system's package manager to install Nix. Instead, simply open a terminal and run the following line (on Windows, if you cannot or have decided not to activate systemd, then you have to append `--init none` to the command. You can find more details about this on The Determinate Nix Installer page):

```
curl --proto '=https' --tlsv1.2 -sSf \
    -L https://install.determinate.systems/nix | \
     sh -s -- install
```

Then, install the `cachix` client and configure the `rstats-on-nix` cache: this will install binary versions of many R packages which will speed up the building process of environments:

```
nix-env -iA cachix -f
 ↪  https://cachix.org/api/v1/install
```

then use the cache:

```
cachix use rstats-on-nix
```

You only need to do this once per machine you want to use `{rix}` on. Many thanks to Cachix for sponsoring the `rstats-on-nix` cache!

1.2.6 Temporary shells

You now have Nix installed; before continuing, it let's see if everything works (close all your terminals and reopen them) by droping into a temporary shell with a tool you likely have not installed on your machine.

Open a terminal and run:

```
which sl
```

you will likely see something like this:

```
which: no sl in ....
```

now run this:

`nix-shell -p sl`

and then again:

```
which sl
```

this time you should see something like:

```
/nix/store/cndqpx74312xkrrgp842ifinkd4cg89g-sl-5.05/bin/sl
```

This is the path to the `sl` binary installed through Nix. The path starts with `/nix/store`: the *Nix store* is where all the software installed through Nix is stored. Now type `sl` and see what happens!

You can find the list of available packages here.

1.3 Session 1.2 – Dev Environments with Nix *(2h)*

1.3.1 Some Nix concepts

While temporary shells are useful for quick testing, this is not how Nix is typically used in practice. Nix is a declarative package manager: users specify what they want to build, and Nix takes care of the rest.

To do so, users write files called `default.nix` that contain the a so-called Nix expression. This expression will contain the definition of a (or several) *derivations*.

In `Nix` terminology, a derivation is *a specification for running an executable on precisely defined input files to repeatably produce output files at uniquely determined file system paths.* (source)

In simpler terms, a derivation is a recipe with precisely defined inputs, steps, and a fixed output. This means that given identical inputs and build steps, the exact same output will always be produced. To achieve this level of reproducibility, several important measures must be taken:

- All inputs to a derivation must be explicitly declared.
- Inputs include not just data files, but also software dependencies, configuration flags, and environment variables, essentially anything necessary for the build process.
- The build process takes place in a *hermetic* sandbox to ensure the exact same output is always produced.

The next sections of this document explain these three points in more detail.

1.3.2 Derivations

Here is an example of a *simple* `Nix` expression:

```
let
 pkgs = import (fetchTarball
 "https://github.com/rstats-on-nix/nixpkgs/archive/2025-04-11.t
 {};

in

pkgs.stdenv.mkDerivation {
  name = "filtered_mtcars";
  buildInputs = [ pkgs.gawk ];
  dontUnpack = true;
  src = ./mtcars.csv;
  installPhase = ''
    mkdir -p $out
    awk -F',' 'NR==1 || $9=="1" { print }' $src >
    $out/filtered.csv
  '';
}
```

I won't go into details here, but what's important is that this code uses `awk`, a common Unix data processing tool, to filter the `mtcars.csv` file to keep only rows where the 9th column (the `am` column) equals 1. As you can see, a significant amount of boilerplate code is required to perform this simple operation. However, this approach is completely reproducible: the dependencies are declared and pinned to a specific dated branch of our `rstats-on-nix/nixpkgs` fork (more on this later), and the only thing that could make this pipeline fail (though it's a bit of a stretch to call this a *pipeline*) is if the `mtcars.csv` file is not provided to it. This expression can be *instantiated* into a derivation, and the derivation is then built into the actual output that interests us, namely the filtered `mtcars` data.

The derivation above uses the `Nix` builtin function `mkDerivation`: as its name implies, this function *makes a derivation*. But there is also `mkShell`, which is the function that builds a shell instead. Nix expressions that built a shell is the kind of expressions `{rix}` generates for you.

1.3.3 Using `{rix}` to generate development environments

If you have successfully installed Nix, but don't have yet R installed on your system, you could install R as you would usually do on your operating system, and then install the `{rix}` package, and from there, generate project-specific expressions and build them. But you could also install R using Nix. Running the following line in a terminal will drop you in an interactive R session that you can use to start generating expressions:

```
nix-shell -p R rPackages.rix
```

This will drop you in a temporary shell with R and `{rix}` available. Navigate to an empty directory to help a project, call it `rix-session-1`:

```
mkdir rix-session-1
```

and start R and load `{rix}`:

```
R
```

```
library(rix)
```

you can now generate an expression by running the following code:

```
rix(
  date = "2025-06-02",
  r_pkgs = c("dplyr", "ggplot2"),
  py_conf = list(
    py_version = "3.13",
    py_pkgs = c("polars", "great-tables")
  ),
  ide = "positron",
  project_path = ".",
  overwrite = TRUE
)
```

This will write a file called `default.nix` in your project's directory. This `default.nix` contains a Nix expression which will build a shell that comes with R, `{dplyr}` and `{ggplot2}` as they were on the the 2nd of June 2025 on CRAN. This will also add Python 3.13 and the `ploars` and `great-tables` Python packages as they were at the time in `nixpkgs` (more on this later). Finally, this also add the Positron IDE, which is a fork of VS Code for data science. This is just an example, and you can use another IDE if you wish. See this vignette for learning how to setup your IDE with Nix.

1.3.4 Using `nix-shell` to Launch Environments

Once your file is in place, simply run:

```
nix-shell
```

This gives you an isolated shell session with all declared packages available. You can test code, explore APIs, or install further tools within this session.

1.3.5 Pinning with `nixpkgs`

To ensure long-term reproducibility, pin the version of Nixpkgs used. Replace `<nixpkgs>` with a fixed import:

```
let
  pkgs = import (fetchTarball
   ↪  "https://github.com/rstats-on-nix/nixpkgs/archive/2025-
   ↪  {};
```

```
in
pkgs.mkShell {
  buildInputs = [ pkgs.r pkgs.rPackages.dplyr ];
}
```

This avoids unexpected updates and lets others reproduce your environment exactly.

1.4 Hands-On Exercises

1. Extend your previous `shell.nix` to add `r` and `rPackages.dplyr`
2. Add Python with `numpy` and `pytest`
3. Write a one-liner R and Python script and run them inside `nix-shell`
4. Share your folder with a partner; verify their shell behaves identically
5. Pin your nixpkgs version and commit the change

Peng, Roger D. 2011. “Reproducible Research in Computational Science.” *Science* 334 (6060): 1226–27.