Reproducible Polyglot Data Science

Bruno Rodrigues

2026-02-01

Table of contents

Welcome!

A modern, unified, and language-agnostic workflow for data science using Nix.

This is still WIP! Expected first finalized release at the end of Q1 2026. This book is a complete reimagining of my previous work, "Building Reproducible Analytical Pipelines with R." If you're looking for that book, you can find it here. But if you're ready for the next step, you're in the right place.

Data scientists, statisticians, analysts, researchers, and many other professionals write a lot of code.

Not only do they write a lot of code, but they must also read and review a lot of code as well. They either work in teams and need to review each other's code, or need to be able to reproduce results from past projects, be it for peer review or auditing purposes. And yet, they never, or very rarely, get taught the tools and techniques that would make the process of writing, collaborating, reviewing and reproducing projects possible.

Which is truly unfortunate because software engineers face the same challenges and solved them decades ago.

The aim of this book is to teach you how to use some of the best practices from software engineering and DevOps to make your projects robust, reliable and reproducible. It doesn't matter if

you work alone, in a small or in a big team. It doesn't matter if your work gets (peer-)reviewed or audited: the techniques presented in this book will make your projects more reliable and save you a lot of frustration!

As someone whose primary job is analysing data, you might think that you are not a developer. It seems as if developers are these genius types that write extremely high-quality code and create these super useful packages. The truth is that you are a developer as well. It's just that your focus is on writing code for your purposes to get your analyses going instead of writing code for others. Or at least, that's what you think. Because in others, your team-mates are included. Reviewers and auditors are included. Any people that will read your code are included, and there will be people that will read your code. At the very least future you will read your code. By learning how to set up projects and write code in a way that future you will understand and not want to murder you, you will actually work towards improving the quality of your work, naturally.

The book can be read for free on https://b-rodrigues.github.io/reproducible-data-science/ and you'll be able buy a DRM-free Epub or PDF on Leanpub[1] once there's more content.

This book is the culmination of my previous works. I started by writing a book focused on R, and then began working on a Python edition. During that process, I had a realization: tackling reproducibility one language at a time was solving the symptoms, not the root cause. The real solution needed to be universal, powerful, and capable of handling any language or tool we might need.

That universal solution is Nix.

[1] https://leanpub.com/

This book moves beyond language-specific tooling. It presents a holistic workflow where R, Python, and Julia are not competitors, but collaborators in a single, cohesive, and perfectly reproducible environment. We will cover:

- The Nix Philosophy: Why Nix is the ultimate tool for solving the "it works on my machine" problem, once and for all.
- Declarative Environments with `{rix}`: How to use a simple R interface to define exact, bit-for-bit reproducible software environments that include specific versions of R, Python, Julia, their packages, and any system-level dependencies.
- Polyglot Pipelines with `{rixpress}` (R) or `ryxpress` (Python): How to orchestrate complex analytical pipelines that seamlessly pass data between different languages, all managed by the Nix build system.
- Unit Testing and Functional Programming: Core principles for writing robust, testable, and maintainable code, no matter the language.
- Distribution and Automation: How to package your entire reproducible pipeline into a Docker container for easy sharing and automate your workflow with GitHub Actions.

While this is not a book for beginners (you should be familiar with at least one data-centric programming language before reading this), I will not assume that you have any knowledge of the tools discussed. But be warned, this book will require you to take the time to read it, and then type on your computer. Type *a lot*.

I hope that you will enjoy reading this book and applying the ideas in your day-to-day, ideas which hopefully should improve the reliability, traceability and reproducibility of your code.

If you find this book useful, don't hesitate to let me know! You can submit issues, suggest improvements, and ask questions on

the book's Github repository.

If you want to get to know me better, read my bio[2].

[2]https://www.brodrigues.co/about/me/

Preface

Three years ago, I wrote a book with a straightforward premise: by borrowing a few key ideas from software engineering, people who analyse data could save themselves a great deal of frustration. The response to that book was more positive than I could have ever hoped for, and it confirmed a suspicion I had: we, as a community, are hungry for better ways to work.

That book, however, focused exclusively on the R ecosystem. A recurring question I received was, "This is great, but what about Python?" It was a fair question. The world of data science is not a monologue; it's a conversation between languages. So, I began what felt like the logical next step: writing a Python edition.

I mapped out the chapters and identified the equivalent tools, `pipenv` for dependency management, `ploomber` for pipelines, and started writing. But as I went deeper, a nagging feeling grew. I was solving the same problems all over again, just with a different set of tools. This feeling was compounded by the rapid churn within the Python ecosystem itself. How many package managers have been created to solve virtual environment management? As of writing, `uv` is all the rage, and while it may be here to stay, history suggests a new contender is always just around the corner.

This pointed to a larger issue. I am convinced that the future of data science is polyglot. An R user and a Python user, both following my original advice, would end up with reproducible

projects, but their workflows would be fundamentally incompatible. They couldn't easily share an environment or build a single pipeline that leverages the strengths of both languages. While companies like Posit have made excellent progress in making it easier to call Python from R, setting up a truly integrated development environment remains a challenge. And what if you wish to bring Julia, the other language of data analysis, into the fold? It is not as popular as Python or R, but it has its own distinct appeal and advantages.

I realised I was treating the symptoms, not the disease. The root problem wasn't "How do I make R reproducible?" or "How do I make Python reproducible?". The real challenge was the lack of a universal foundation that could handle *any* language, *any* tool, and *any* system dependency with absolute, bit-for-bit precision.

That's when I stopped writing the Python book.

The solution wasn't to create another language-specific guide but to find a tool that operated at a more fundamental level. That tool, I am now convinced, is **Nix**.

Nix is not just another package manager; it is a powerful, declarative system for building and managing software environments. It allows us to define the *entire* computational environment—from the operating system libraries up to the specific versions of our R and Python packages—in a single, simple text file. When you use Nix, the phrase "it works on my machine" becomes obsolete. It is replaced by the guarantee: "it builds identically, everywhere, every time"—with a few caveats that we will explore, of course.

This book is the result of that realisation. It is a complete reimagining of the original. We are moving away from language-specific patchworks and toward a unified, polyglot workflow. We will use Nix as our bedrock, with the `{rix}` and `{rixpress}` R packages

(or `ryxpress` for Python) serving as our friendly interface to its power. You will learn to build pipelines where R, Python, and Julia aren't just neighbours; they are collaborators, working together in a single, perfectly reproducible environment.

A note for Python-first users: do not be deterred by the fact that `{rix}` and `{rixpress}` are R packages: there is a Python version called `ryxpress` that will allow you to run your pipelines from an interactive Python session. You will be able to use them to define your environments (even integrating tools like `uv`) and orchestrate your pipelines, while doing all of your analytical work exclusively in Python. In this workflow, R simply becomes a convenient configuration language.

The core message from three years ago remains unchanged. You, as someone who writes code to analyse data, are a developer. Your work is important, and it deserves to be reliable. This book aims to give you the tools and the mindset to achieve that. The journey is more ambitious this time, but the payoff is far greater.

I hope you'll join me.

You can read this book for free online at https://b-rodrigues.github.io/reproducible-data-science/.

You can submit issues, suggest improvements, and ask questions on the book's Github repository.

1 Introduction

Just like the previous, R-focused edition of this book, this one will not teach you about machine learning, statistics, or visualisation.

The goal is to teach you a set of tools, practices, and project management techniques that should make your projects easier to reproduce, replicate, and retrace, with a focus on polyglot (multilingual) projects. I believe that with LLMs, polyglot projects will become increasingly common.

Before LLMs, if you were a Python user, you would avoid R like the plague, and vice versa. This is understandable: even though both are high-level scripting languages, and an R user could likely read and understand Python code (and vice versa), it is still a pain in the loins to set up another language just to run a few lines of code. Now, with LLMs, you can have a model generate the code, and depending on what you're doing, it's likely to be correct. However, setting up another language is still quite annoying. This is where Nix comes in. Nix makes adding another language to a project extremely simple.

Even though tools like `{rix}`, `{rixpress}`, and Docker might be new to you, you can become productive with them very quickly by leveraging LLMs. The key is to provide the LLM with the right context. I have built a tool called pkgctx[1] that extracts structured, compact API specifications from R or Python packages

[1] https://github.com/b-rodrigues/pkgctx

for use in LLMs, minimising tokens while maximising context. Each of the repositories for these tools—`{rix}`, `{rixpress}` (R), and `rixpress` (Python)—contains a `.pkgctx.yaml` file that you can feed to an LLM to help it understand the package's API. With this context, the LLM can assist you in writing correct code using these tools, even if you've never used them before.

You can run `pkgctx` directly without installing it, thanks to Nix:

```
# Extract context for an R package from CRAN
nix run github:b-rodrigues/pkgctx -- r dplyr >
 ↪  dplyr.pkgctx.yaml

# Extract context for an R package from GitHub
nix run github:b-rodrigues/pkgctx -- r
 ↪  github:ropensci/rix > rix.pkgctx.yaml

# Extract context for a Python package from PyPI
nix run github:b-rodrigues/pkgctx -- python requests
 ↪  > requests.pkgctx.yaml
```

I encourage you to do the same for the packages you use in your analyses. By generating context files using `pkgctx`, you enable LLMs to help you write code more efficiently and correctly. This is especially useful for packages with complex APIs or for packages you don't use frequently. Simply generate the context file once and include it in your project repository—or feed it to your LLM whenever you need assistance with that package.

1.1 Who is this book for?

This book is for anyone who uses raw data to build any type of output. This could be a simple quarterly report, in which data is used for tables and graphs, a scientific article for a peer-reviewed journal, or even an interactive web application. The specific output doesn't matter, because the process is, at its core, always very similar:

- Get the data;
- Clean the data;
- Write code to analyse the data;
- Put the results into the final product.

This book assumes some familiarity with programming, particularly with the R and Python languages. I will not discuss Julia in great detail, as I am not familiar enough with it to do it justice. That being said, I will show you how to add Julia to a project and use it effectively if you need to.

1.2 What is the aim of this book?

The aim of this book is to make the process of analysing data as reliable, retraceable, and reproducible as possible, and to do this by design. This means that once you're done with the analysis, you're done. You don't want to spend time, which you often don't have anyway, to rewrite or refactor an analysis to make it reproducible after the fact. We both know this is not going to happen. Once an analysis is finished, it's on to the next one. If you need to rerun an older analysis (for example, because the data has been updated), you'll simply figure it out at that point, right? That's a problem for Future You. Hopefully, Future You

will remember every quirk of your code, know which script to run at which point, which comments are outdated, and what features of the data need to be checked… You had better hope Future You is a more diligent worker than you are!

Going forward, I'm going to refer to a project that is reproducible as a "Reproducible Analytical Pipeline", or RAP for short. There are only two ways to create a RAP: either you are lucky enough to have someone on your team whose job is to turn your messy code into a RAP, or you do it yourself. The second option is by far the most common. The issue, as stated above, is that most of us simply don't do it. We are always in a rush to get to the results and don't think about making the process reproducible, because we assume it takes extra time that is better spent on the analysis itself. This is a misconception, for two reasons.

The first is that employing the techniques we will discuss in this book won't actually take much time. As you will see, they are not things you "add on top of" the analysis, but are part of the analysis itself, and they will also help with managing the project. Some of these techniques, especially testing, will even save you time and headaches.

The second reason is that an analysis is never a one-shot. Only the simplest tasks, like pulling a single number from a database, might be. Even then, chances are that once you provide that number, you'll be asked for a variation of it (for example, disaggregated by one or several variables). Or perhaps you'll be asked for an update in six months. You will quickly learn to keep that SQL query in a script somewhere to ensure consistency. But what about more complex analyses? Is keeping the script enough? It is a good start, of course, but very often, there is no single script, or a script for each step of the analysis is missing.

I've seen this play out many times in many different organisations.

It's that time of the year again, and a report needs to be written. Ten people are involved, and just gathering the data is already complicated. Some get their data from Word documents attached to emails, some from a website, some from a PDF report from another department. I remember a story a senior manager at my previous job used to tell: once, a client put out a call for a project that involved setting up a PDF scraper. They periodically needed data from another department that only came in PDFs. The manager asked what was, at least from our perspective, an obvious question: "Why can't they send you the underlying data in a machine-readable format?" They had never thought to ask. So, my manager went to that department and talked to the people putting the PDF together. Their answer? "Well, we could send them the data in any format they want, but they've asked for the tables in a PDF."

So the first, and probably most important, lesson here is: when starting to build a RAP, make sure you talk with all the people involved.

1.3 Prerequisites

You should be comfortable with the command line. This book will not assume any particular Integrated Development Environment (IDE), so most of what I'll show you will be done via the command line. That said, I will spend some time helping you set up a data science-focused IDE, Positron, to work seamlessly with this workflow. The command line may be over 50 years old, but it is not going anywhere. In fact, thanks to the rise of LLMs, it seems to be enjoying a resurgence. Since these models generate text, it is far simpler to ask one for a shell command to solve a problem than to have it produce detailed instructions on where to click

in a graphical user interface (GUI). Knowing your way around the command line is also essential for working with modern data science infrastructure: continuous integration platforms, Docker, remote servers… they all live in the terminal. So, if you're not at all familiar with the command line, you might need to brush up on the basics. Don't worry, though; this isn't a book about the intricacies of the Unix command line. The commands I'll show you will be straightforward and directly applicable to the task at hand.

This means however that if you are using Windows, first of all, why? and second of all, you will have to set up Windows Subsystem for Linux. This is because there is no native Nix implementation for Windows, and so we need to run the Linux version through WSL. Don't worry though, it's not that hard, and you can then use an IDE from Windows to work with the environments managed by Nix, in a very seamless way (but seriously, consider, if you can, switching to Linux. How can you tolerate ads (ADS!) in the start menu).

Ideally, you should be comfortable with either R or Python. This book will assume that you have been using at least one of these languages for some projects and want to improve how you manage complex projects. You should know about packages and how to install them, have written some functions, understand loops, and have a basic knowledge of data structures like lists. While this is not a book on visualisation, we will be making some graphs as well.

Our aim is to write code that can be executed non-interactively. This is because a necessary condition for a workflow to be reproducible and to qualify as a RAP is for it to be executed by a machine, automatically, without any human intervention. This is the second lesson of building RAPs: there should be no human intervention needed to get the outputs once the RAP has started.

If you achieve this, then your workflow is likely reproducible, or can at least be made so much more easily than if it requires special manipulation by a human somewhere in the loop.

1.4 What actually is reproducibility?

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

1.4.1 Using open-source tools is a hard requirement

Open source is a hard requirement for reproducibility. No ifs, ands, or buts. I'm not just talking about the code you wrote for your research paper or report; I'm talking about the entire ecosystem you used to write your code and build the workflow.

Is your code open? Good. Or is it at least available to others in your organisation, in a way that they could re-execute it if needed? Also good.

But is it code written in a proprietary program, like STATA, SAS, or MATLAB? Then your project is not reproducible. It doesn't matter if the code is well-documented, well-written, and available on a version control system. The project is simply not reproducible. Why?

Because on a long enough time horizon, there is no way to re-execute your code with the exact same version of the proprietary

language and operating system that were used when the project was developed. As I'm writing these lines, MATLAB, for example, is at version R2025a. Buying an older version may not be simple. I'm sure if you contact their sales department, they might be able to sell you an older version. Maybe you can even re-download older versions you've already purchased from their website. But maybe it's not that straightforward. Or maybe they won't offer this option in the future. In any case, if you search for "purchase old version of Matlab," you will see that many researchers and engineers have this need. ::: {.content-hidden when-format="pdf"}

Wanting to run older versions of analytics software is a recurrent need.

:::

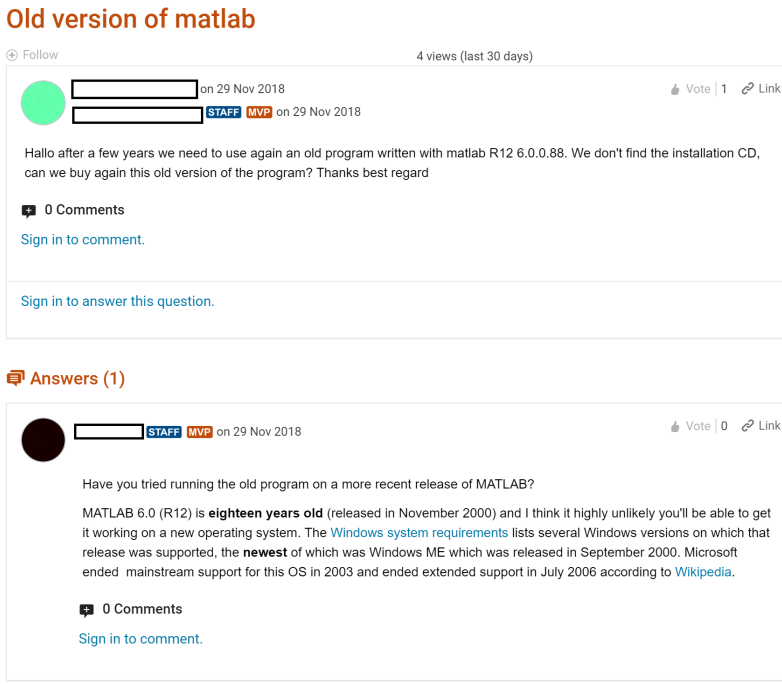


Figure 1.1: Wanting to run older versions of analytics software is a recurrent need.

And if you're running old code written for version, say, R2008a, there's no guarantee that it will produce the exact same results on version R2025a. That's without even mentioning the toolboxes (if you're not familiar with them, they're MATLAB's equivalent of packages or libraries). These evolve as well, and there's no guarantee that you can purchase older versions. It's also likely that newer toolboxes cannot even run on older versions of MATLAB.

Let me be clear: what I'm describing here for MATLAB could also be said for any other proprietary programs still commonly (and unfortunately) used in research and statistics, like STATA,

SAS, or SPSS. Even if some of these vendors provide ways to run older versions of their software, the fact that you have to rely on them for this is a barrier to reproducibility. There is no guarantee they will provide this option forever. Who can guarantee that these companies will even be around forever? More likely, they might shift from a program you install on your machine to a subscription-based model.

For just 199€ a month, you can execute your SAS (or whatever) scripts on the cloud! Worried about data confidentiality? No problem, data is encrypted and stored safely on our secure servers! Run your analysis from anywhere and don't worry about your cat knocking coffee over your laptop! And if you purchase the pro licence, for an additional 100€ a month, you can even execute your code in parallel!

Think this is science fiction? There is a growing and concerning trend for vendors to move to a Software-as-a-Service model with monthly subscriptions. It happened to Adobe's design software, and the primary reason it hasn't yet happened for data analytics tools is data privacy concerns. Once those are deemed "solved," I would not be surprised to see a similar shift.

1.4.2 Hidden dependencies can hinder reproducibility

Then there's another problem. Let's suppose you've written a thoroughly tested and documented workflow and made it available on GitHub (and let's even assume the data is freely available and the paper is open access). Or, if you're in the private sector, you've done all of the above, but the workflow is only available internally.

Let's further assume that you've used R, Python, or another open-source language. Can this analysis be said to be reproducible? Well, if it ran on a proprietary operating system, then the conclusion is: your project is not fully reproducible.

This is because the operating system the code runs on can also influence the outputs. There are particularities in operating systems that may cause certain things to work differently. Admittedly, this is rarely a problem in practice, but it does happen[2], especially if you're working with high-precision floating-point arithmetic, as you might in the financial sector, for instance.

Thankfully, as you will see in this book, there is no need to change your operating system to deal with this issue.

1.4.3 The requirements of a RAP

So where does that leave us? For a project to be truly reproducible, it has to respect the following points:

- Source code must be available and thoroughly tested and documented (which is why we will be using Git and GitHub).
- All dependencies must be easy to find and install (we will deal with this using dependency management tools).
- It must be written in an open-source programming language (no-code tools like Excel are non-reproducible by default because they can't be used non-interactively, which is why we will be using languages like R, Python, and Julia).
- The project needs to run on an open-source operating system (we can deal with this without having to install and learn a new OS, thanks to tools like Docker).

[2] https://github.com/numpy/numpy/issues/9187

- The data and the final report must be accessible—if not publicly, then at least within your company. This means the concept of "scripts and/or data available upon request" belongs in the bin.

Figure 1.2: A real sentence from a real paper published in *THE LANCET Regional Health*. How about *make the data available and I won't scratch your car*, how's that for a reasonable request?

1.4.4 Are there different types of reproducibility?

Let's take one step back. We live in the real world, where constraints outside of our control can make it impossible to build a true RAP. Sometimes we need to settle for something that might not be perfect, but is the next best thing.

In what follows, let's assume the code is tested and documented, so we will only discuss the pipeline's execution.

The *least* reproducible pipeline would be something that works, but only on your machine. This could be due to hardcoded paths that only exist on your laptop. Anyone wanting to rerun the pipeline would need to change them. This should be documented

in a README, which we've assumed is the case. But perhaps the pipeline only runs on your laptop because the computational environment is hard to reproduce. Maybe you use software, even open-source software, that is not easy to install (anyone who has tried to install R packages on Linux that depend on `{rJava}` knows what I'm talking about).

A better, though still imperfect, pipeline would be one that could be run more easily on any similar machine. This could be achieved by avoiding hardcoded absolute paths and by providing instructions to set up the environment. For example, in Python, this could be as simple as providing a `requirements.txt` file that lists the project's dependencies, which could be installed using `pip`:

```
pip install -r requirements.txt
```

Doing this helps others (or Future You) install the required packages. However, this is not enough, as other software on your system, outside of what `pip` manages, can still impact the results.

You should also ensure that people run the same analysis on the same versions of R or Python that were used to create it. Just installing the right packages is not enough. The same code can produce different results on different versions of a language, or not run at all. If you've been using Python for some time, you certainly remember the switch from Python 2 to Python 3. Who knows, the switch to Python 4 might be just as painful!

The take-away message is that relying on the language itself being stable over time is not a sufficient condition for reproducibility. We have to set up our code in a way that is *explicitly* reproducible by dealing with the versions of the language itself.

So what does this all mean? It means that reproducibility exists on a continuum. Depending on the constraints you face, your project can be "not very reproducible" or "totally reproducible". Let's consider the following list of factors that can influence how reproducible your project truly is:

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

By "reproducibility is on a continuum," I mean that you can set up your project to take none, one, two, three, four, or all of the preceding items into consideration.

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

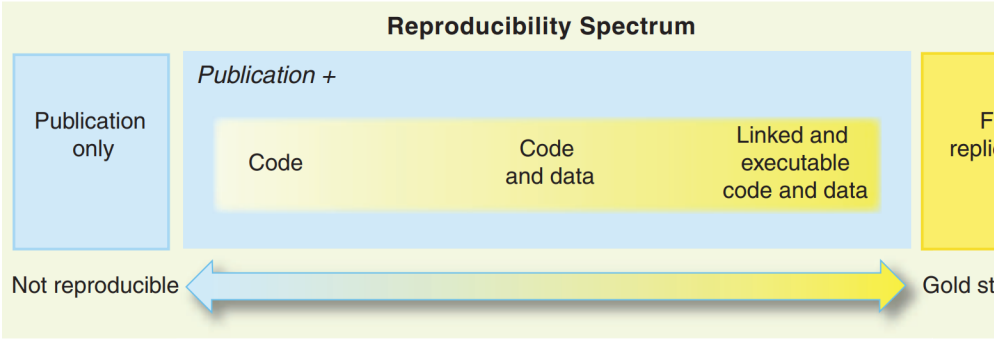


Figure 1.3: The reproducibility spectrum from Peng's 2011 paper.

Let me finish this introduction by discussing the last item on the list: hardware architecture. In 2020, Apple changed the hardware architecture of their computers. Their new machines no longer use Intel CPUs, but instead Apple's own proprietary architecture (Apple Silicon) based on the ARM specification. Concretely, this means that binary packages built for Intel-based Apple computers cannot run on their new machines, at least not without a compatibility layer. If you have a recent Apple Silicon Mac and need to install old packages to rerun a project (and we will learn how to do this), they need to be compiled to work on Apple Silicon first. While a compatibility layer called Rosetta 2 exists, my point is that you never know what might come in the future. The ability to compile from source is important because it requires the fewest dependencies outside of your control. Relying on pre-compiled binaries is not future-proof, which is another reason why open-source tools are a hard requirement for reproducibility.

For you Windows users, don't think that the preceding paragraph does not concern you. It is very likely that Microsoft will push for OEM manufacturers to build more ARM-based

computers in the future. There is already an ARM version of Windows, and I believe Microsoft will continue to support it. This is because ARM is much more energy-efficient than other architectures, and any manufacturer can build its own ARM CPUs by purchasing a license—a very interesting proposition from a business perspective.

It is also possible that we will move towards more cloud-based computing, though I think this is less likely than the hardware shift. In that case, it is quite likely that the actual code will be running on Linux servers that are ARM-based, due to energy and licensing costs. Here again, if you want to run your historical code, you'll have to compile old packages and programming language versions from source.

Ok, this might all seem incredibly complicated. How on earth are we supposed to manage all these risks and balance the immediate need for results with the future need to rerun an old project? And what if rerunning it is never needed?

As you shall see, this is not as difficult as it sounds, thanks to Nix and the tireless efforts of the `nixpkgs` maintainers who work to build truly reproducible packages.

Let's dive in!

2 The Nix Package Manager

2.1 Introduction

Nix is a package manager that can be installed on your computer, regardless of the operating system (but requires WSL2 on Windows). If you are familiar with the Ubuntu Linux distribution, you have likely used `apt-get` to install software. On macOS, you may have used `homebrew` for similar purposes. Nix functions in a comparable way but has many advantages over classic package managers, as it focuses on reproducible builds and downloads packages from `nixpkgs`, currently the largest software repository[1].

In this chapter, we will explore the critical need for environment reproducibility in modern workflows. We will see why ad-hoc tools often fail, and how Nix's declarative approach and "component closures" provide a robust solution. We will also cover the core concepts of Nix—derivations, the store, and hermetic builds—that make this possible.

[1] https://repology.org/repositories/graphs

2.2 Why Reproducibility? Why Nix?

2.2.1 Motivation: Reproducibility in Scientific and Data Workflows

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

- Ensure the required version of your programming language (R, Python, etc.) is installed.
- Ensure the required versions of all packages are installed.
- Ensure all necessary system dependencies are installed (for example, a working Java installation for the `{rJava}` R package on Linux).
- Ensure you can install all of this on the hardware you have on hand.

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

The current consensus for tackling the first three points is often a mixture of tools: Docker for system dependencies, `{renv}` or `uv` for package management, and tools like the R installation manager (`rig`) for language versions. As for the last point, hardware architecture, the only way out is to be able to compile the software for the target platform. This involves a lot of moving parts and requires significant knowledge to get right.

2.2.2 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 system-level 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, and just adding it to your project doesn't make it reproducible if you don't explicitly take some precautionary steps.

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.

2.2.3 Nix: A Declarative Solution

With Nix, we can handle all of these challenges with a single tool.

The first advantage of Nix is that its repository, `nixpkgs`, is humongous. As of this writing, it contains over 120,000 pieces of software, including the *entirety of CRAN and Bioconductor*. This means you can use Nix to handle everything: R, Python, Julia, their respective packages, and any other software available through `nixpkgs`, making it particularly useful for polyglot pipelines.

The second and most crucial advantage is that Nix allows you to install software in (relatively) isolated environments. When you start a new project, you can use Nix to install a project-specific version of R and all its packages. These dependencies are used only for that project. If you switch to another project, you switch to a different, independent environment. But this also means that all the dependencies of R and R packages, plus all of their dependencies and so on get installed as well. Your project's development environment will not depend on anything outside

of it (well, there are some caveats which we will explore as we move on).

This is similar to `{renv}`, but the difference is profound: you get not only a project-specific library of R packages but also a project-specific R version and all the necessary system dependencies. For example, if you need `{xlsx}`, Nix automatically figures out that Java is required and installs and configures it for you, without any intervention.

What's more, you can *pin* your project to a specific revision of the `nixpkgs` repository. This ensures that every package Nix installs will always be at the exact same version, regardless of when or where the project is built. The environment is defined in a simple plain-text file, and anyone using that file will get a byte-for-byte identical environment, even on a different operating system.

2.3 Important Concepts

Before we start using Nix, it is important to spend some time learning about some Nix-centric concepts, starting with the *derivation*.

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 (and "inputs" here is meant in a very broad sense; for example, configuration flags are also inputs!).
- 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 explain these three points in more detail.

2.3.1 Derivations

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

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

Without going into too much detail, this code uses `awk`, a common Unix data processing tool, to filter the `mtcars.csv` file. As you can see, a significant amount of boilerplate is required for this simple operation. However, this approach is completely reproducible: the dependencies are declared and pinned to a specific version of the `nixpkgs` repository. The only thing that could make this small pipeline fail is if the `mtcars.csv` file is not provided to it.

`Nix` builds the `filtered.csv` output file in two steps: it first generates a *derivation* from this expression, and only then does it build the output. For clarity, I will refer to code like the example above as a *derivation* rather than an expression, to avoid confusion with the concept of an *expression* in R.

The goal of the tools we will use in this book, `{rix}` and `{rixpress}` (or `ryxpress` if you prefer using Python), is to help you create pipelines from such derivations without needing to learn the Nix language itself, while still benefiting from its powerful reproducibility features.

2.3.2 Dependencies of derivations

Nix requires that the dependencies of any derivation be explicitly listed and managed by Nix itself. If you are building an output that requires Quarto, then Quarto must be explicitly listed as an input, even if you already have it installed on your system. The same applies to Quarto's dependencies, and their dependencies, all the way down. To run a linear regression with R, you essentially need Nix to build the entire universe of software that R depends on first.

In Nix terms, this complete set of packages is what its author, Eelco Dolstra, refers to as a *component closure*:

> 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. That is, we must always deploy a set of components that is closed under the 'depends on' relation.

(*Nix: A Safe and Policy-Free System for Software Deployment*, Dolstra et al., 2004).

Figure 4 of Dolstra et al. (2004)

In the figure, `subversion` depends on `openssl`, which itself depends on `glibc`. Similarly, if you write a derivation to filter `mtcars`, it requires an input file, R, `{dplyr}`, and all of their respective dependencies. All of these must be managed by Nix. If any dependency exists "outside" this closure, the pipeline will only *work on your machine*, defeating the purpose of reproducibility.

2.3.3 The Nix store and hermetic builds

When building derivations, their outputs are saved into the **Nix store**. Typically located at `/nix/store/`, this folder contains all the software and build artefacts produced by Nix.

For example, the output of a derivation might be stored at a path like `/nix/store/81k4s9q652jlka0c36khpscnmr8wk7jb-filtered`. The long cryptographic hash uniquely identifies the build output and is computed based on the content of the derivation and all its inputs. This ensures that the build is fully reproducible.

As a result, building the same derivation on two different machines will yield the same cryptographic hash. You can substitute the built artefact with the derivation that generates it one-to-one,

just as in mathematics, where writing $f(2)$ is the same as writing 4 for the function $f(x) := x^2$.

To guarantee that derivations always produce identical outputs, builds must occur in an isolated environment known as a **hermetic sandbox**. This process ensures that the build is unaffected by external factors, such as the state of the host system. This isolation extends to environment variables and even network access. If you need to download data from an API, for example, it will not work from within the build sandbox. This may seem restrictive, but it makes perfect sense for reproducibility: an API's output can change over time.

For a truly reproducible result, you should obtain the data once, version it, and use that archived data as an input to your analysis.

2.3.4 Other key Nix concepts

We've covered derivations, dependencies, closures, the Nix store, and hermetic builds. That's the core of what makes Nix tick. But there are a few more concepts worth knowing about before we move on:

- **Purity**: Nix tries very hard to keep builds "pure": the output only depends on what you explicitly list as inputs. If a build script tries to reach out to the internet or read some random file on your machine, Nix will block it. That can feel restrictive at first, but it's what guarantees reproducibility.
- **Binary caches**: You don't always need to build everything yourself. Think back to the math analogy: if you already know that $f(2) = 4$, there's no need to compute

it again; just reuse the result! Nix does the same with binary caches: because every build is identified by its unique cryptographic hash made from its inputs, this means a prebuilt package fetched from `cache.nixos.org` (or your own cache you may want to set up) is bit-for-bit identical to what you would have built locally. This is why Nix is both reproducible and fast.

- **Garbage collection**: Since Nix never overwrites anything in the store, old packages can pile up. Running `nix-store --gc` will run the garbage collector to free up space.
- **Overlays**: If you want to tweak a package or add your own without forking all of `nixpkgs`, you can use overlays. They let you extend or override existing definitions in a clean, composable way.
- **Flakes**: The newer way to define and pin Nix projects. Flakes make it easier to share and reuse Nix setups across machines and repositories. There is a lot of discussion around flakes, as they're officially still considered not stable, even though they've been widely adopted by the community. But don't worry, this is not something you'll need to think about for this book.

Together, these features explain why Nix isn't *just another package manager*. It's more like a framework for reproducible environments that can scale from a single project to an entire operating system (called NixOS[2]).

As a little sidenote: I want also to highlight that Nix can even be used to declaratively and reproducibly configure your operating system (be it NixOS, macOS or other Linux distributions) using a tool that integrates with it called `homemanager`.[3] This is outside

[2]https://nixos.org/
[3]https://github.com/nix-community/home-manager

the scope of this book, but I wanted to highlight it, as it's extremely powerful. What this means in practice is that you could write a whole Nix expression that not only downloads and configures software, but even sets up users with their specific software, and preferences like wallpapers, colour schemes and so on.

2.4 Caveats

While Nix is powerful, there are some limitations and practical hurdles to be aware of if you plan to use it for actual work:

- **Hardware acceleration**: On non-NixOS systems, it can be difficult to set up GPU acceleration (CUDA, ROCm, OpenCL). Drivers are tightly coupled to the host kernel and libraries, while Nix builds aim for strict isolation. On NixOS this integration is smoother, but on macOS or other Linux distros you may encounter limitations or extra manual steps.
- **macOS-specific issues**: Reproducibility is harder to achieve on macOS (both Intel and Apple Silicon) than on Linux. Nix packages often rely on Apple system frameworks (e.g., CoreFoundation, Security) that live outside the Nix store and compromise hermetic builds[4]. The macOS sandbox is also weaker than Linux's and sometimes leaks system tools like Xcode or Rosetta into builds[5]. Hydra cache coverage is thinner for Darwin platforms, especially Apple Silicon, so cache misses and local builds are much

[4] https://github.com/NixOS/nixpkgs/issues/67166

[5] https://discourse.nixos.org/t/nix-macos-sandbox-issues-in-nix-2-4-and-later/17475

more common[6]. Finally, pinned environments can break after macOS or Xcode updates because of changes in system libraries or compiler flags[7].

That said, in practice most packages build fine on macOS, and the ecosystem continues to improve. When reproducibility problems do appear, the simplest fix is often just to use another nearby `nixpkgs` revision for pinning that is known to work. This makes the situation less fragile than it might first appear. Another solution would be to use Docker to deploy the Nix environment and use that as a dev container. All of this will be discussed in detail in this book.

- **Steep learning curve**: The Nix language and ecosystem (flakes, overlays, derivations) can be conceptually difficult if you come from traditional package managers. Even basic customizations require some ramp-up time. This was my main motivation to write `{rix}`, `{rixpress}` (for R) and `ryxpress` (for Python).
- **Disk space and builds**: Because Nix never mutates software in place, the store can accumulate large amounts of data. Cache misses sometimes force local builds, which can be slow and resource-intensive. However, it is of course possible to empty the Nix store to recover disk space, and it is also possible to set up your own project cache if you wish so. Setting up your own cache will also be something that we will explore in this book.

These caveats don't diminish Nix's strengths but highlight that its guarantees are strongest on Linux (and thus WSL), and especially

[6] https://www.reddit.com/r/NixOS/comments/17uxj6q/how_does_nix_package_manager_work_on_apple_silicon/

[7] https://github.com/NixOS/nix/issues/11679

on NixOS. On macOS, reproducibility is possible but sometimes requires extra work, a bit of flexibility, and occasionally picking a different `nixpkgs` snapshot.

2.5 In summary

Nix makes it possible to *actually* build software reproducibly. To achieve this, it introduces several core concepts that are quite specific, and definitely worth taking the time to understand.

In the next chapter, we will learn how to install Nix, configure `cachix`, and set up Positron for a seamless development experience.

3 Setting Up Your Environment

In this chapter, we will install Nix, configure the `rstats-on-nix` binary cache to speed up installations, and set up our development environment (Positron, VS Code, or Emacs) to work seamlessly with reproducible Nix environments.

3.1 Installing Nix

3.1.1 For Windows Users: WSL2 Prerequisites

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.

I recommend activating `systemd` in Ubuntu WSL2, mainly because this supports users other than `root` running Nix. To set this up, please follow this official Ubuntu blog entry:

```
# in WSL2 Ubuntu shell

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

This will open `/etc/wsl.conf` in 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. For those of you running Windows, we will be working exclusively from WSL2 now. If that is not an option, then I highly recommend you set up a virtual machine with Ubuntu using VirtualBox for example, or dual-boot Ubuntu.

3.1.2 The Determinate Systems installer

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, run this inside WSL, and after the prerequisites listed above):

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

Just follow the instructions on screen, and in no time Nix will be available on your machine!

3.2 Configuring the Cache

Next, 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!

If you get this warning when trying to install software with Nix:

```
warning: ignoring the client-specified setting
'trusted-public-keys', because it is a restricted
setting and you are not a trusted user
warning: ignoring untrusted substituter
'https://rstats-on-nix.cachix.org', you are not a
trusted user.
Run `man nix.conf` for more information on the
`substituters` configuration option.
warning: ignoring the client-specified setting
'trusted-public-keys', because it is a restricted
setting and you are not a trusted user
```

Then this means that configuration was not successful. You need to add your user to `/etc/nix/nix.custom.conf`:

```
sudo nano /etc/nix/nix.custom.conf
```

then simply add this line in the file:

```
trusted-users = root YOURUSERNAME
```

where `YOURUSERNAME` is your current login user name.

3.3 Verifying installation with Temporary Shells

You now have Nix installed; before continuing, let's see if everything works (close all your terminals and reopen them) by dropping 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!

Temporary shells are quite useful, especially if you want to simply run a command using some tool that you only need infrequently. But this is not how we are going to be using Nix.

3.4 Configuring your IDE

3.4.1 Pre-requisites

We now need to configure an IDE to use our Nix shells as development environments. You are free to use whatever IDE you want but the instructions below are going to focus on Positron, which is a fork of VS Code geared towards data science. It works well with both Python and R and makes it quite easy to choose the right R or Python interpreter (which you'll have to do to make sure you're using the one provided by Nix, see here).

If you want to use VS Code proper, you can follow all the instructions here, but you need to install the REditorSupport and the Python extension. You will also need to add the `{languageserver}` R package to your Nix shells.

On Windows, you need to install Positron on Windows, not inside WSL.

If you want to use RStudio, you can, but you will need to install it through Nix: an RStudio installed through the usual means for your system is not going to be able to interact with Nix! This is a limitation of RStudio and there is currently no workaround. If you want to use RStudio, just skip the rest of the chapter, as it's irrelevant to you. Later, when we use `{rix}` to set up our first Nix development environment, I'll tell you what to do to use RStudio.

Other editors work well with Nix too. Emacs users can use the envrc or emacs-direnv packages to automatically load Nix environments. Neovim users can use direnv.nvim. The key is that any editor with `direnv` support will work with the setup described below.

3.4.2 direnv

Once Positron is installed, you need to install a piece of software called `direnv`: `direnv` will automatically load Nix shells when you open a project that contains a `default.nix` file in an editor.[1] It works on any operating system and many editors support it, including Positron. If you're using Windows, install `direnv` in WSL (even though you've just installed Positron for Windows). To install `direnv` run this command:

```
nix-env -f '<nixpkgs>' -iA direnv
```

This will install `direnv` and make it available even outside of Nix shells!

Then, I highly recommend to install the `nix-direnv` extension:

```
nix-env -f '<nixpkgs>' -iA nix-direnv
```

It is not mandatory to use `nix-direnv` if you already have `direnv`, but it'll make loading environments much faster and seamless.

Finally, if you haven't used `direnv` before, don't forget this last step to make your terminal detect and load `direnv` automatically.

Then, in Positron, install the direnv extension. Finally, add a file called `.envrc` and simply write the following two lines in it (this `.envrc` file should be in the same folder as your project's `default.nix`):

[1] A `default.nix` file is a file that contains the specification of our development environments, and is the file that will be automatically generated by `{rix}`

```
use nix
mkdir $TMP
```

in it. On Windows, *remotely connect to WSL* first, but on other operating systems, simply open the project's folder using `File > Open Folder...` and you will see a pop-up stating `direnv: /PATH/TO/PROJECT/.envrc is blocked` and a button to allow it. Click `Allow` and then open an R script. You might get another pop-up asking you to restart the extension, so click `Restart`. Be aware that at this point, `direnv` will run `nix-shell` and so will start building the environment. If that particular environment hasn't been built and cached yet, it might take some time before Code will be able to interact with it. You might get yet another popup, this time from the R Code extension complaining that R can't be found. In this case, simply restart Positron and open the project folder again: now it should work every time.

3.4.3 In summary and next steps

For a new project, simply repeat this process:

- Generate the project's `default.nix` file (we will see how in the next chapter);
- Build it using `nix-build`;
- Create an `.envrc` and write the two lines from above in it;
- Open the project's folder in Positron and click allow when prompted;
- Restart the extension and Positron if necessary.

Another option is to create the `.envrc` file and write the two required lines, then open a terminal, navigate to the project's folder,

and run `direnv allow`. Doing this before opening Positron should not prompt you anymore.

If you're on Windows, using Positron like this is particularly interesting, because it allows you to install Positron on Windows as usual, and then you can configure it to interact with a Nix shell, even if it's running from WSL. This is a very seamless experience.

4 Reproducible Development Environments with rix

4.1 Introduction

Now that we have Nix installed, and our IDE configured we can actually tackle the reproducibility puzzle.

4.1.1 The Reproducibility Challenge

Reproducibility in research and data science exists on a continuum. At one end, authors might only describe their methods in prose. Moving along the spectrum, they might share code, then data, and finally what we call a *computational environment*: the complete set of software required to execute an analysis.

Even when researchers share code and data, they rarely specify the full software stack: the exact version of R, all package versions, and crucially, the system-level dependencies. Yet differences in any of these can lead to divergent results from the same code.

Tools like `{renv}` address part of the puzzle: they capture R package versions in a lockfile. But `{renv}` does not manage the R version itself (you need `rig` for that), and neither handles system libraries. If `{sf}` requires GDAL 3.0 but your system has 2.4, `{renv}` can't help. And if your project uses both R and

Python? Now you're coordinating multiple package managers, each with its own configuration.

4.1.2 Component Closures: The Nix Approach

This is where Nix shines. Nix deploys *component closures*: when you install a package, Nix also installs all its dependencies, their dependencies, and so on. Think of it like packing for a trip—traditional package managers assume you'll find essentials at your destination, while Nix packs everything you need.

As the original Nix paper 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.

This means when you install `{sf}` through Nix, you automatically get the correct versions of GDAL, GEOS, and PROJ—no manual system configuration needed.

4.1.3 The Polyglot Challenge

Modern data science is increasingly polyglot. Research shows that data scientists use, on average, nearly two programming languages in their work, with R and Python being the most common combination. Python dominates machine learning, R excels at statistical modelling, and Julia offers high-performance numerics. Projects increasingly combine these strengths.

This creates a reproducibility challenge: a project using R, Python, and Quarto requires coordinating multiple package managers. Nix solves this by providing a unified framework for all languages and system tools.

4.1.4 Enter rix

However, Nix has a steep learning curve. Its functional programming language can be daunting for researchers focused on their analysis, not system administration.

`{rix}` bridges this gap. It's an R package that generates Nix expressions from intuitive R function calls. You describe *what* you want, and `{rix}` figures out *how* to express it in Nix. The workflow is simple:

1. **Declare** your environment using `rix()`
2. **Build** the environment using `nix-build`
3. **Use** the environment using `nix-shell`

This chapter covers everything you need to know to create project-specific, reproducible development environments for your R projects.

4.2 Transitioning to Nix-Managed R

Now that Nix is installed, I strongly recommend uninstalling any system-wide R installation and removing the packages in your user library (typically found in `~/R` on Linux or `~/Library/R` on macOS). From this point forward, let Nix handle everything. If you are using Windows, you can keep your Windows-specific R installation, since Nix will not interfere with it (remember, Nix

is installed inside WSL and Positron will automatically load Nix environments installed in WSL as well).

If you are not ready to take this step, you can continue using your local R library. The `{rix}` package makes efforts not to interfere with your existing setup, and the `.Rprofile` it generates helps keep Nix environments isolated. However, for the cleanest experience and to avoid subtle conflicts, I recommend fully committing to Nix.

4.2.1 Bootstrapping `{rix}` without a local R installation

But if R is uninstalled, how do you use `{rix}` to generate environments? The answer lies in the temporary shells we saw in the previous chapter. Use Nix itself to bootstrap a temporary R session with `{rix}` available.

Running the following line in a terminal will drop you into an interactive R session:

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

This gives you a temporary shell with R and `{rix}` ready to use. From here, you can generate project-specific Nix expressions.

For example, navigate to an empty directory for a new project:

```
mkdir my-project
cd my-project
```

Then start R:

```
R
```

Load {rix} and generate an expression:

```
library(rix)

rix(
  date = "2025-04-11",
  r_pkgs = c("dplyr", "ggplot2"),
  ide = "none",
  project_path = ".",
  overwrite = TRUE
)
```

This writes a `default.nix` file to your project directory. The expression defines a shell with R, {dplyr}, and {ggplot2} as they were on the 11th of April 2025 on CRAN. The `ide` argument is set to `"none"` because we configured Positron in the previous chapter rather than having Nix manage an IDE.

I recommend saving this code in a file called `gen-env.R` in your project directory. If you later need to add packages or change the R version, simply edit `gen-env.R`, run it to regenerate `default.nix`, and rebuild with `nix-build`. This keeps your environment definition readable and versioned alongside your project.

> **Getting LLM assistance with {rix}**
>
> If the {rix} syntax is new to you, remember that you can use `pkgctx` to generate LLM-ready context (as mentioned in the introduction). The {rix} repository includes

a `.pkgctx.yaml` file you can feed to your LLM to help it understand the package's API. You can also generate your own context file:

```
nix run github:b-rodrigues/pkgctx -- r
 ↪  github:ropensci/rix > rix.pkgctx.yaml
```

With this context, your LLM can help you write correct `rix()` calls, even if you've never used the package before. You can do so for any package hosted on CRAN, GitHub, or local `.tar.gz` files.

You can now exit this temporary session (type `q()` in R, then `exit` in the shell) and build your new environment:

```
nix-build
```

Once built, use `nix-shell` to enter your project's environment. This workflow of bootstrapping `{rix}` via a temporary shell, generating a `default.nix`, and then building it is the pattern you will follow for every new project.

This is also the reason why there is no Python version of `{rix}`: you can bootstrap a Python environment using `{rix}` in the same way, by only temporarily having the R interpreter available through the Nix shell.

4.3 The rix() Function

The `rix()` function is the heart of the package. It generates a `default.nix` file—a Nix expression that defines your development environment. Here are its main arguments:

- `r_ver`: the version of R you need (or use `date` instead)
- `date`: a date corresponding to a CRAN snapshot
- `r_pkgs`: R packages to install from CRAN/Bioconductor
- `system_pkgs`: system tools like `quarto` or `git`
- `git_pkgs`: R packages to install from GitHub
- `local_r_pkgs`: local `.tar.gz` packages to install
- `tex_pkgs`: TexLive packages for literate programming
- `ide`: which IDE to configure (`"rstudio"`, `"code"`, `"positron"`, `"none"`). Since we've configured Positron in the previous chapter, you'll want to set this to `"none"`. If you want to use RStudio, then set it to `"rstudio"`. This will install RStudio using Nix and make it available from the development shell. If you set it to `"code"` or `"positron"`, this will then install VS Code or Positron using Nix.
- `project_path`: where to save the `default.nix` file
- `overwrite`: whether to overwrite an existing `default.nix`

Let's create our first environment. Suppose you need R with `{dplyr}` and `{ggplot2}`:

```
library(rix)

rix(
  r_ver = "4.4.2",
  r_pkgs = c("dplyr", "ggplot2"),
  ide = "rstudio",
  project_path = ".",
  overwrite = TRUE
)
```

This generates two files:

1. `default.nix`: The Nix expression defining your environment
2. `.Rprofile`: Created by `rix_init()` (called automatically), this file prevents conflicts with any system-installed R packages

The `.Rprofile` is important: it ensures that packages from your user library don't get loaded into the Nix environment, and it redefines `install.packages()` to throw an error, because you should never install packages that way in a Nix environment.

4.4 Choosing an R Version or Date

The `r_ver` argument (or alternatively `date`) controls which version of R and which package versions you'll get. Here's a summary of the options:

`r_ver` / `date`	Intended Use	R Version	Package Versions
`"latest-upstream"`	New project, versions don't matter	Current/previous	Up to 6 months old
`"4.4.2"` (or similar)	Reproduce old project or start new	Specified version	Up to 2 months old
`date = "2024-12-14"`	Precise CRAN snapshot	Current at that date	Exact versions from date
`"bleeding-edge"`	Develop against latest CRAN	Always current	Always current

r_ver / date	Intended Use	R Version	Package Versions
"frozen-edge"	Latest CRAN, manual updates	Current at generation	Current at generation
"r-devel"	Test against R development version	R-devel	Always current

To see which R versions are available:

```
available_r()
```

To see which dates are available for snapshotting:

```
available_dates()
```

4.4.1 Using dates vs versions

Using a specific date is often the best choice for reproducibility. When you specify a date, you get the exact state of CRAN on that day:

```
rix(
  date = "2024-12-14",
  r_pkgs = c("dplyr", "ggplot2"),
  ide = "none",
  project_path = ".",
```

```
  overwrite = TRUE
)
```

I find that this is often easier and clearer than using an R version. Be careful though, as using a date doesn't reflect the state of PyPi on that date. So if you also need Python packages, the versions of packages for Python that will be provided are the ones available from `nixpkgs` on that day (but more on this later).

4.4.2 The rstats-on-nix fork

When you use a specific R version or date (rather than `"latest-upstream"`), `{rix}` uses our `rstats-on-nix` fork of `nixpkgs` rather than the upstream repository. This fork:

- Snapshots CRAN more frequently
- Offers newer R releases faster than the official channels
- Includes many fixes, especially for Apple Silicon (even though as time goes by, this is becoming much rarer, because Nix is getting better and more support for Apple Silicon)

4.5 Installing R Packages

4.5.1 From CRAN/Bioconductor

The simplest case—just list package names in `r_pkgs`:

```
rix(
  r_ver = "4.4.2",
  r_pkgs = c("dplyr", "ggplot2", "tidyr", "readr"),
  ide = "none",
  project_path = "."
)
```

Both CRAN and Bioconductor packages can be specified this way.

4.5.2 Installing archived versions

Need a specific old version of a package? Use the `@` syntax:

```
rix(
  r_ver = "4.2.1",
  r_pkgs = c("dplyr@0.8.0", "janitor@1.0.0"),
  ide = "none",
  project_path = "."
)
```

This will install `{dplyr}` version 0.8.0 from the CRAN archives. Note that archived packages are built from source, which may fail for packages requiring compilation. Thus I recommend you use a date on which that specific version of `{dplyr}` was current.

4.5.3 Installing from GitHub

For packages on GitHub, use the `git_pkgs` argument with a list containing the package name, repository URL, and commit hash:

```
rix(
  r_ver = "4.4.2",
  r_pkgs = c("dplyr", "ggplot2"),
  git_pkgs = list(
    list(
      package_name = "housing",
      repo_url =
       ↪  "https://github.com/rap4all/housing/",
      commit =
       ↪  "1c860959310b80e67c41f7bbdc3e84cef00df18e"
    )
  ),
  ide = "none",
  project_path = "."
)
```

Always specify a commit hash, not a branch name. This ensures reproducibility: branch names can change, but commits are immutable.

If the R package lives in a subfolder of the repository, append the subfolder to the URL:

```
git_pkgs = list(
  package_name = "BPCells",
  repo_url = "https://github.com/bnprks/BPCells/r",
   ↪  # Note the /r suffix
  commit =
   ↪  "16faeade0a26b392637217b0caf5d7017c5bdf9b"
)
```

Note that this will install the package from source, so if it's a package that requires specific system dependencies, you will

need to specify them manually (Nix takes care of this for you for packages that are available through `nixpkgs`, but not for ad-hoc packages from other sources).

4.5.4 Installing local packages

For local `.tar.gz` archives, place them in the same directory as your `default.nix` and use `local_r_pkgs`:

```
rix(
  r_ver = "4.3.1",
  local_r_pkgs = c("mypackage_1.0.0.tar.gz"),
  ide = "none",
  project_path = "."
)
```

Just like with packages from Git, note that this will install the package from source, so if it's a package that requires specific system dependencies, you will need to specify them manually (Nix takes care of this for you for packages that are available through `nixpkgs`, but not for ad-hoc packages from other sources).

4.5.5 Why NOT to use install.packages()

It's crucial to understand: **never call `install.packages()` from within a Nix environment**. Here's why:

1. **Declarative environments**: If you install packages imperatively, your `default.nix` no longer matches your actual environment.

2. **Leaking packages**: Packages installed via `install.packages()` go to your user library, not the Nix environment. They'll be visible to *all* Nix shells, breaking isolation.
3. **Reproducibility**: The whole point of Nix is that your environment is fully defined by the `default.nix`. Ad-hoc installations defeat this.

Instead, add packages to your `rix()` call and rebuild the environment.

4.6 System Tools and TexLive

4.6.1 Adding system packages

Need command-line tools like Quarto or Git? Add them via `system_pkgs`:

```
rix(
  r_ver = "latest-upstream",
  r_pkgs = c("quarto"),
  system_pkgs = c("quarto", "git"),
  ide = "none",
  project_path = "."
)
```

Note that here we install both the R `{quarto}` package and the `quarto` command-line tool—they're different things!

To find available packages, search at search.nixos.org.

4.6.2 TexLive for literate programming

For PDF output from Quarto, R Markdown, or Sweave, you need TexLive packages:

```
rix(
  r_ver = "latest-upstream",
  r_pkgs = c("quarto"),
  system_pkgs = "quarto",
  tex_pkgs = c("amsmath", "framed", "fvextra"),
  ide = "none",
  project_path = "."
)
```

This installs the `scheme-small` TexLive distribution plus the specified packages.

4.6.3 Python and Julia integration

`{rix}` can also add Python or Julia to your environment:

```
rix(
  date = "2025-02-17",
  r_pkgs = "ggplot2",
  py_conf = list(
    py_version = "3.12",
    py_pkgs = c("polars", "great-tables")
  ),
  ide = "none",
  project_path = "."
)
```

This creates a polyglot environment with R, Python, and the specified packages for each.

4.6.4 Installing Python packages with uv (impure)

Not all Python packages (or versions of packages) are available through Nix. Unlike CRAN, PyPI doesn't get automatically mirrored—individual packages must be packaged by volunteers. If a Python package you need isn't in nixpkgs, you can use `uv` as an escape hatch.

This approach is also useful when collaborating with colleagues who use `uv` but haven't adopted Nix yet. `uv` is 10–100x faster than `pip` and generates a lock file for improved reproducibility.

The idea is to install `uv` in your shell (but not Python or Python packages through Nix):

```
rix(
  date = "2025-02-17",
  r_pkgs = "ggplot2",
  system_pkgs = c("uv"),
  ide = "none",
  project_path = "."
)
```

Then use `uv` from within your shell. We recommend:

1. Specify Python packages in a `requirements.txt` file with explicit versions (e.g., `scanpy==1.11.4`)
2. Set up a shell hook to automatically configure the virtual environment

Here's a complete example with a shell hook:

```
rix(
  date = "2025-02-17",
  r_pkgs = "ggplot2",
  system_pkgs = c("uv"),
  shell_hook = "
    if [ ! -f pyproject.toml ]; then
      uv init --python 3.13.5
    fi
    uv add --requirements requirements.txt
    alias python='uv run python'
  ",
  ide = "none",
  project_path = "."
)
```

After running `nix-shell`, `uv` initialises a Python project with the specified version and installs packages from `requirements.txt`. This happens each time you enter the shell, but `uv` caches everything so it's nearly instant after the first run.

4.6.4.1 Troubleshooting wheel issues

When using wheels (pre-compiled Python packages), you may encounter errors like:

```
ImportError: libstdc++.so.6: cannot open shared
object file
```

This happens because wheels expect certain libraries in certain locations. Add this to your shell hook to fix it:

```
shellHook = ''
  export LD_LIBRARY_PATH="${pkgs.lib.makeLibraryPath
 ↪  (with pkgs; [
    zlib gcc.cc glibc stdenv.cc.cc
  ])}":$LD_LIBRARY_PATH
  # ... rest of your hook
'';
```

If this seems complicated: yes, it is. This is exactly the kind of problem Nix aims to solve. When possible, prefer Python packages included in nixpkgs.

4.7 Building and Using Environments

4.7.1 Building with nix-build

Once you have a `default.nix`, build the environment:

```
nix-build
```

This downloads/builds all required packages and creates a `result` symlink in your project directory. The `result` file prevents the environment from being garbage-collected.

4.7.2 Entering with nix-shell

To use the environment interactively:

```
nix-shell
```

You'll drop into a shell where R and all your packages are available. Type `R` to start an R session. If you're using RStudio (and specified `ide = "rstudio"` in the call to `rix()`) then type `rstudio` to launch RStudio. If instead you configured Positron, (or any other IDE), open Positron, and open the project folder that contains the `default.nix` (make sure you also have an `.envrc` there for `direnv` to load the environment directly).

You can even run scripts directly without entering the shell:

```
nix-shell default.nix --run "Rscript analysis.R"
```

This is quite useful on CI/CD platforms.

4.7.3 Pure shells for complete isolation

By default, `nix-shell` can still see programs installed on your system. This is quite important to understand: building the environment happens in an isolated, hermetic sandbox, but when inside a shell, isolation is more porous and it is possible to use other systems tools. For example, you don't need to install `git` inside the Nix development environment: you can just keep using the `git` executable already available on your system.

But it is possible to run the environment with increased isolation:

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

This hides everything not explicitly included in your environment.

4.7.4 Garbage collection

Nix never deletes old packages automatically. To clean up:

1. Delete the `result` symlink that will appear in your project's folder after calling `nix-build`
2. Run `nix-store --gc`

This removes all packages that are no longer referenced by any environment.

4.8 Converting renv Projects

If you have existing projects using `{renv}`, the `renv2nix()` function can help you migrate:

```
renv2nix(
  renv_lock_path = "path/to/project/renv.lock",
  project_path = "path/to/new_nix_project"
)
```

4.8.1 Recommended workflow

1. Copy the `renv.lock` file to a new, empty folder
2. Run `renv2nix()` pointing to that folder
3. Build the environment with `nix-build`

Do **not** convert in the same folder as the original `{renv}` project—the generated `.Rprofile` will conflict with `{renv}`'s `.Rprofile`.

4.8.2 Caveats

- Package versions may not match exactly due to how Nix handles snapshotting
- If the `renv.lock` lists an old R version but recent packages, use the `override_r_ver` argument to specify a more appropriate R version

4.9 Summary

Let's step back and recap why we have gone through all this trouble.

Traditional tools like `{renv}` or Python's `venv` only capture part of the reproducibility puzzle. They track package versions but not the language version itself, nor system-level dependencies like GDAL or Java. This means your project can still break on a different machine, or even on your own machine after a system update.

Nix solves this by managing *everything*: R, Python, all packages, and all system dependencies. When you define an environment with `{rix}`, you get a complete, self-contained specification that anyone can use to recreate the exact same environment, on any machine, at any point in the future.

4.9.1 Quick Reference: Starting a New Project

Here is the workflow you will follow for every new project:

1. **Create an empty folder** for your project:

```
mkdir my-project
cd my-project
```

2. **Write a `gen-env.R` file** with your environment definition:

```
library(rix)

rix(
  date = "2025-04-11",
  r_pkgs = c("dplyr", "ggplot2"),
  ide = "none",  # or "rstudio" if you prefer
  project_path = ".",
  overwrite = TRUE
)
```

3. **Add a `.envrc` file** (only needed for Positron or VS Code, skip if using RStudio):

```
use nix
mkdir $TMP
```

4. **Bootstrap `{rix}` and generate `default.nix`**:

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

Then in R:

```
source("gen-env.R")
```

Exit R with `q()` and the shell with `exit`.

5. **Build the environment**:

```
nix-build
# you could also run `direnv allow` to allow
 ↪  direnv to load the environment
```

```
# automatically and build the environment on
 ↪  first use
```

6. **Start working**: Open the folder in Positron (which will load the environment via `direnv`), or run `nix-shell` followed by `rstudio` if you set `ide = "rstudio"`.

That's it. Your project is now reproducible. Anyone with Nix installed can clone your repository, run `nix-build`, and get the exact same environment you have.

In the next chapter, we'll explore **Functional Programming** principles—a foundation you'll need before we can build pipelines with `{rixpress}`. After that, we'll learn how to **test** our code (Chapter 6), and then put everything together in Chapters 7 and 8 where we build complete analytical pipelines.

5 Functional Programming

In this chapter, we will see why functional programming is crucial for reproducible, testable, and collaborative data science. We will compare how to write self-contained, "pure" functions in both R and Python, and how to use functional concepts like `map`, `filter`, and `reduce` to replace error-prone loops. Finally, we will discuss how writing functions makes your code easier to review, debug, and even generate with LLMs.

5.1 Introduction: From Scripts to Functions

In the previous chapter, we learned how to create reproducible development environments with `{rix}`. We can now ensure everyone has the *exact same tools* (R, Python, system libraries) to run our code.

We could have stopped there: after all, we have now reproducible environments, as many as we need for each of our projects. But having the right tools is only half the battle. Now we turn to **writing reproducible code itself**. A common way to start a data analysis is by writing a script: a sequence of commands executed from top to bottom.

```r
# R script example
library(dplyr)
data(mtcars)
heavy_cars <- filter(mtcars, wt > 4)
mean_mpg_heavy <- mean(heavy_cars$mpg)
print(mean_mpg_heavy)
```

```python
# Python script example
import pandas as pd
mtcars = pd.read_csv("mtcars.csv")
heavy_cars = mtcars[mtcars['wt'] > 4]
mean_mpg_heavy = heavy_cars['mpg'].mean()
print(mean_mpg_heavy)
```

This works, but it has a hidden, dangerous property: **state**. The script relies on an implicit execution order and on variables like `heavy_cars` existing in the global environment at the right moment. This makes the code surprisingly fragile: rename a variable, reorder a few lines, or run a subset of the script, and things silently break, or worse, silently produce wrong answers.

Of course, this is a four-line toy example. Now imagine the reality of production data science: dozens of scripts, each hundreds of lines long, maintained by multiple people over months or years. Variables are reused, overwritten, and passed around in an unspoken contract of "run these files in *this* order." Debugging becomes archaeology: you must reconstruct not just what the code says, but what the global environment *was* when it ran. Testing is nearly impossible, because there is no clear boundary between inputs and outputs. And onboarding a new team member? They inherit a minefield.

The root problem is **implicit dependencies**. When code depends on global state, the dependencies are invisible. A function

call like `mean(heavy_cars$mpg)` *looks* self-contained, but it secretly relies on a hidden precondition: that `heavy_cars` exists, has the expected columns, and was computed correctly upstream. If any step in that chain fails silently, the error propagates, and may only surface as a mysterious wrong number in a final report, weeks later.

5.1.1 The Notebook Problem

If scripting with state is a crack in the foundation of reproducibility, then using computational notebooks is a gaping hole. I will not make many friends in the Python community with the following paragraphs, but that's because only truth hurts.

Notebooks like Jupyter introduce an even more insidious form of state: the cell execution order. You can execute cells out of order, meaning the visual layout of your code has no relation to how it actually ran. This is a recipe for non-reproducible results and a primary cause of the "it worked yesterday, why is it broken today?" problem.

In a famous talk from JupyterCon 2018, Joel Grus (a research engineer at the Allen Institute for AI) played the contrarian at a conference dedicated to the very tool he was criticising.[1] His central thesis is that data science code should follow software engineering best practices, and Jupyter notebooks actively discourage those practices.

His biggest complaint is **hidden state and out-of-order execution**. The state of the variables depends on the *execution history*, not the order of the code on the screen. You can delete a cell that defined a variable, but that variable still exists in

[1] I don't like notebooks

memory. This leads to unreproducible results where the code works right now (because of hidden state in memory) but will fail if you restart the kernel and run top-to-bottom. Worse, it confuses beginners who do not understand why their code works one minute and breaks the next.

Notebooks also **discourage modular code**. Because it is difficult to import code from one notebook into another, users tend to write massive, monolithic scripts, copy and paste the same code blocks into multiple notebooks, and avoid creating functions or modules that can be tested and reused.

Then there is the **"works on my machine" problem**. Notebooks often lack clear dependency specifications, users hardcode file paths that only exist on their specific computer, and reusing someone else's work usually involves manually copying and pasting cells, which is error-prone.

Grus also argues that notebooks have **poor tooling compared to IDEs**. Actual text editors provide linting (identifying stylistic errors or unused variables), type checking, superior autocomplete, and the ability to run unit tests. All of these are difficult or impossible in notebooks.

Finally, **version control is a nightmare**. Notebooks are JSON files. If two people edit a notebook and try to merge their changes in Git, the diffs are unreadable blocks of JSON metadata, and merge conflicts are incredibly difficult to resolve. This encourages workflows where people email files back and forth rather than using proper version control.

What does Grus suggest instead? Write code in modules (`.py` or `.R` files) using a proper editor, write unit tests to ensure the code works, and use notebooks *only* for the final step: importing those modules to visualise the data or present the results. Ideally,

the notebook should contain very little logic and mostly just function calls.

This is exactly the approach we will take in this book.

5.1.2 The Functional Solution

The solution is to embrace a paradigm that minimises state: **Functional Programming (FP)**. Instead of a linear script, we structure our code as a collection of self-contained, predictable functions.

The power of FP comes from the concept of **purity**, borrowed from mathematics. A mathematical function has a beautiful property: for a given input, it always returns the same output. `sqrt(4)` is always `2`. Its result doesn't depend on what you calculated before or on a random internet connection.

Our Nix environments handle the "right library" problem; purity handles the "right logic" problem. Our goal is to write our analysis code with this same level of rock-solid predictability.

5.1.3 FP vs OOP: Transformations vs Actors

To appreciate what FP brings, it helps to contrast it with **Object-Oriented Programming (OOP)**, arguably the dominant paradigm in many software systems.

OOP organises computation around *who does what*: a network of objects communicating with each other and managing their own internal state. You send a message to an object, asking it to perform an action, without needing to know how it works internally.

Functional programming, by contrast, organises computation around *how data changes*. It replaces a network of interacting objects with a flow of transformations: data goes in, data comes out, and nothing else changes in the process.

This shift is especially powerful in data science:

- Analyses are naturally expressed as **pipelines of transformations** (cleaning, filtering, aggregating, modelling)
- Pure functions make results **reproducible**: same inputs always yield same outputs
- **Immutability** prevents accidental side effects on shared data
- Because transformations can be composed, tested, and reused independently, FP encourages **modular, maintainable** analysis code

There is also a structural reason why FP fits data science better than OOP. OOP excels when you have *many different types of objects*, each with a *small set of methods*. A graphical user interface, for example, has buttons, menus, windows, and dialogs, each responding to a few actions like `click()` or `resize()`. But data science is the opposite: we typically work with *a small number of data structures* (data frames, arrays, models) and apply *a large number of operations* to them (filtering, grouping, joining, summarising, plotting). FP handles this case naturally: adding a new function is trivial, while OOP would require modifying every class.

5.1.4 Why Does This Matter for Data Science?

Adopting a functional style brings massive benefits:

1. **Unit Testing is Now Possible:** You can't easily test a 200-line script. But you *can* easily test a small function that does one thing.
2. **Code Review is Easier:** A Pull Request that just adds or modifies a single function is simple for your collaborators to understand and approve.
3. **Working with LLMs is More Effective:** It's incredibly effective to ask, "Write a Python function that takes a pandas DataFrame and a column name, and returns the mean of that column, handling missing values. Also, write three `pytest` unit tests for it."
4. **Readability:** Well-named functions are self-documenting:

   ```
   starwars %>%
     group_by(species) %>%
     summarize(mean_height = mean(height))
   ```

   is instantly understandable. The equivalent `for` loop is a puzzle.

5.2 Purity and Side Effects

A **pure function** has two rules:

1. It only depends on its inputs. It doesn't use any "global" variables defined outside the function.
2. It doesn't change anything outside of its own scope. It doesn't modify a global variable or write a file to disk. This is called having "no side effects."

Consider this "impure" function in Python:

```
# IMPURE: Relies on a global variable
discount_rate = 0.10

def calculate_discounted_price(price):
    return price * (1 - discount_rate)  # What if
     ↪  discount_rate changes?

print(calculate_discounted_price(100))
```

```
90.0
```

```
discount_rate = 0.20  # Someone changes the state
print(calculate_discounted_price(100))
```

```
80.0
```

The pure version passes *all* its dependencies as arguments:

```
# PURE: All inputs are explicit arguments
def calculate_discounted_price_pure(price, rate):
    return price * (1 - rate)

print(calculate_discounted_price_pure(100, 0.10))
```

```
90.0
```

```
print(calculate_discounted_price_pure(100, 0.20))
```

```
80.0
```

Now the function is predictable and self-contained.

5.2.1 Handling "Impure" Operations like Randomness

Some operations, like generating random numbers, are inherently impure. Each time you run `rnorm(10)` or `numpy.random.rand(10)`, you get a different result.

The functional approach is not to avoid this, but to *control* it by making the source of impurity (the random seed) an explicit input.

In R, the `{withr}` package helps create a temporary, controlled context:

```
library(withr)

# This function is now pure! For a given seed, the
 ↪  output is always the same.
pure_rnorm <- function(n, seed) {
  with_seed(seed, {
    rnorm(n)
  })
}

pure_rnorm(n = 5, seed = 123)
```

```
[1] -0.56047565 -0.23017749  1.55870831  0.07050839
0.12928774
```

```
pure_rnorm(n = 5, seed = 123)  # Same result!
```

```
[1] -0.56047565 -0.23017749  1.55870831  0.07050839
0.12928774
```

In Python, `numpy` provides an object-oriented way to handle this:

```
import numpy as np

# Create a random number generator instance with a
 ↪  seed
rng = np.random.default_rng(seed=123)
print(rng.standard_normal(5))
```

```
[-0.98912135 -0.36778665  1.28792526  0.19397442
0.9202309 ]
```

```
# If we re-create the same generator, we get the
 ↪  same numbers
rng2 = np.random.default_rng(seed=123)
print(rng2.standard_normal(5))
```

```
[-0.98912135 -0.36778665  1.28792526  0.19397442
0.9202309 ]
```

The key is the same: the "state" (the seed) is explicitly managed, not hidden globally.

5.2.2 The OOP Caveat in Python

This introduces a concept from OOP: the `rng` variable is an *object* that bundles together data (its internal seed state) and methods (`.standard_normal()`). This is **encapsulation**.

This is a double-edged sword for reproducibility. The `rng` object is now a stateful entity. If we called `rng.standard_normal(5)`

a second time, it would produce different numbers because its internal state was mutated.

Core Python libraries like `pandas`, `scikit-learn`, and `matplotlib` are fundamentally object-oriented. Our guiding principle must be:

> **Use functions for the flow and logic of your analysis, and treat objects from libraries as values that are passed between these functions.**

Avoid building your own complex classes with hidden state for your data pipeline. A pipeline composed of functions (`df2 = clean_data(df1); df3 = analyze_data(df2)`) is almost always more transparent than an OOP one (`pipeline.load(); pipeline.clean(); pipeline.analyze()`, where `pipeline` is an object that keeps mutating after each call of one of its methods).

5.3 Functions: A Refresher and Beyond

You likely already know how to write functions in R and Python. This section serves as a quick refresher, but also introduces some concepts you may not have encountered: **higher-order functions**, **closures**, and **decorators**.

5.3.1 The Basics

In R, functions are first-class citizens. You assign them to variables and pass them around like any other value:

```r
calculate_ci <- function(x, level = 0.95) {
  se <- sd(x, na.rm = TRUE) / sqrt(length(x))
  mean_val <- mean(x, na.rm = TRUE)
  alpha <- 1 - level
  lower <- mean_val - qnorm(1 - alpha/2) * se
  upper <- mean_val + qnorm(1 - alpha/2) * se
  c(mean = mean_val, lower = lower, upper = upper)
}
```

In Python, the `def` keyword defines functions. Type hints are recommended:

```python
import statistics
import scipy.stats as stats

def calculate_ci(x: list[float], level: float =
 ↪  0.95) -> dict:
    """Calculate confidence interval for a list of
     ↪  numbers."""
    n = len(x)
    mean_val = statistics.mean(x)
    se = statistics.stdev(x) / (n ** 0.5)
    alpha = 1 - level
    z = stats.norm.ppf(1 - alpha / 2)
    return {"mean": mean_val, "lower": mean_val - z
     ↪  * se, "upper": mean_val + z * se}
```

Imports and Language Design

Notice that the Python version requires importing `statistics` and `scipy.stats` for basic statistical operations. This is a consequence of Python being a general-

purpose language: statistical functions are not built in, so we must import libraries that provide them.
R, by contrast, is a language designed for statistics. Functions like `mean()`, `sd()`, and `qnorm()` are available out of the box. When you do need a function from an external package, R offers the `::` notation (e.g., `dplyr::filter(df, x > 0)`) to call a single function without loading the entire package into your namespace.
But this raises a question: if you have many functions spread across multiple files, how do you manage which imports or `library()` calls each file needs? This is exactly what Joel Grus advocates: write your code in proper modules (`.py` or `.R` files), not notebooks, and structure them as **packages**. When you do, each module declares its own dependencies, and the package manager ensures everything is available. We will explore packaging in detail later in this book.

5.3.2 Higher-Order Functions

A **higher-order function** is a function that takes another function as an argument, returns a function, or both. This is the foundation of functional programming.

You have already seen examples: `map()`, `filter()`, and `reduce()` are all higher-order functions because they take a function as their first argument.

Here is a simple example in R:

```
apply_twice <- function(f, x) {
 f(f(x))
}
```

```r
apply_twice(sqrt, 16)  # sqrt(sqrt(16)) = sqrt(4) =
 ↪  2
```

```
[1] 2
```

And in Python:

```python
def apply_twice(f, x):
    return f(f(x))

apply_twice(lambda x: x ** 2, 2)  # (2^2)^2 = 16
```

```
16
```

5.3.3 Closures: Functions That Remember

A **closure** is a function that "remembers" variables from its enclosing scope, even after that scope has finished executing. This is useful for creating specialised functions. These are sometimes called *function factories*.

In R:

```r
make_power <- function(n) {
  function(x) x^n
}

square <- make_power(2)
cube <- make_power(3)

square(4)  # 16
```

```
[1] 16
```

```
cube(4)    # 64
```

```
[1] 64
```

In Python:

```
def make_power(n):
    def power(x):
        return x ** n
    return power

square = make_power(2)
cube = make_power(3)

square(4)  # 16
```

```
16
```

```
cube(4)    # 64
```

```
64
```

The inner function "closes over" the variable `n`, preserving its value.

5.3.4 Decorators (Python)

Python has a special syntax for a common use of higher-order functions: **decorators**. A decorator wraps a function to extend its behaviour without modifying its code.

```
import time

def timer(func):
    """A decorator that prints how long a function
     ↪ takes to run."""
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end -
         ↪ start:.4f} seconds")
        return result
    return wrapper

@timer
def slow_sum(n):
    return sum(range(n))

slow_sum(1_000_000)
```

```
slow_sum took 0.0169 seconds
499999500000
```

The `@timer` syntax is equivalent to `slow_sum = timer(slow_sum)`. Decorators are widely used in Python frameworks (Flask, FastAPI, pytest) for logging, authentication, and caching.

If you find decorators elegant, you will appreciate the chapter on **monads** later in this book. Monads take the idea of wrapping and chaining functions further, providing a principled way to handle errors, missing values, and side effects in a purely functional style.

R does not have built-in decorator syntax, but you can achieve the same effect with higher-order functions:

```
timer <- function(f) {
  function(...) {
    start <- Sys.time()
    result <- f(...)
    end <- Sys.time()
    message(sprintf("Elapsed: %.4f seconds", end -
     ↪  start))
    result
  }
}

slow_sum <- timer(function(n) sum(seq_len(n)))
slow_sum(1e6)
```

```
Elapsed: 0.0000 seconds
```

```
[1] 500000500000
```

5.3.5 Tidy Evaluation in R

For data analysis, you will often want functions that work with column names. The `{dplyr}` package uses "tidy evaluation" with the `{ }` (curly-curly) syntax:

```
library(dplyr)
```

```
Attaching package: 'dplyr'

The following objects are masked from
'package:stats':

    filter, lag

The following objects are masked from
'package:base':

    intersect, setdiff, setequal, union
```

```
summarise_variable <- function(data, var) {
  data %>%
    summarise(
      n = n(),
      mean = mean({{ var }}, na.rm = TRUE),
      sd = sd({{ var }}, na.rm = TRUE)
    )
}

starwars %>%
  group_by(species) %>%
  summarise_variable(height)
```

```
# A tibble: 38 x 4
   species       n  mean    sd
   <chr>     <int> <dbl> <dbl>
 1 Aleena        1    79    NA
 2 Besalisk      1   198    NA
```

```
 3 Cerean         1  198   NA
 4 Chagrian       1  196   NA
 5 Clawdite       1  168   NA
 6 Droid          6  131.  49.1
 7 Dug            1  112   NA
 8 Ewok           1   88   NA
 9 Geonosian      1  183   NA
10 Gungan         3  209.  14.2
# i 28 more rows
```

The `{ var }` tells `dplyr` to treat `var` as a column name rather than a literal variable.

Python has no equivalent to tidy evaluation. In pandas, column names must always be passed as strings:

```python
def summarise_variable(df, var):
    return df[[var]].agg(['count', 'mean', 'std'])

summarise_variable(starwars_py, 'height')
```

This works, but there are trade-offs. With tidy evaluation, you write unquoted column names, which means your editor can provide autocomplete suggestions if it knows the data frame's structure. With strings, you are on your own. More importantly, writing programmatic functions that accept column names as arguments is more cumbersome: you end up passing strings around and using methods like `.loc[]` or `.agg()` with string keys, rather than the natural "column as variable" style that `{dplyr}` enables.

5.3.6 Anonymous Functions and Lambdas

Sometimes you need a quick, throwaway function that does not deserve a name. Both R and Python support **anonymous functions**.

In Python, the `lambda` keyword creates a small function in a single expression:

```
squares = list(map(lambda x: x ** 2, [1, 2, 3, 4]))
squares
```

```
[1, 4, 9, 16]
```

In R, the base syntax is verbose, but `{purrr}` introduced the **formula shorthand** using `~` and `.x`:

```
library(purrr)

# Full anonymous function
map_dbl(1:4, function(x) x^2)
```

```
[1]  1  4  9 16
```

```
# Formula shorthand (purrr style)
map_dbl(1:4, ~ .x^2)
```

```
[1]  1  4  9 16
```

R 4.1 also introduced a shorter base syntax with `\(x)`:

```
# Base R shorthand (R >= 4.1)
sapply(1:4, \(x) x^2)
```

```
[1]  1  4  9 16
```

Use anonymous functions when the logic is simple and a full function definition would be overkill.

5.3.7 Partial Application

Partial application means fixing some arguments of a function to create a more specialised version. This is closely related to function factories but works with existing functions rather than defining new ones.

In R, use `purrr::partial()`:

```
library(purrr)

# Create a function that always rounds to 2 decimal
 ↪  places
round2 <- partial(round, digits = 2)

round2(3.14159)  # 3.14
```

```
[1] 3.14
```

```
round2(2.71828)  # 2.72
```

```
[1] 2.72
```

In Python, use `functools.partial()`:

```
from functools import partial

# Create a function that always rounds to 2 decimal
 ↪  places
round2 = partial(round, ndigits=2)

round2(3.14159)  # 3.14
```

```
3.14
```

```
round2(2.71828)  # 2.72
```

```
2.72
```

Partial application is useful for creating callbacks, simplifying repetitive code, and making functions fit the signature expected by `map()` or similar.

5.3.8 Immutability: Data That Does Not Change

A core principle of functional programming is **immutability**: once data is created, it is never modified in place. Instead, transformations produce *new* copies of the data.

R has **copy-on-modify** semantics. When you "modify" a data frame, R actually creates a new copy:

```
df <- data.frame(x = 1:3)
df2 <- df
df2$x[1] <- 99
df$x[1]  # Still 1, df was not mutated
```

```
[1] 1
```

Python, by contrast, has **mutable defaults**, which can surprise newcomers:

```
import pandas as pd

df = pd.DataFrame({"x": [1, 2, 3]})
df2 = df  # This is a reference, not a copy!
df2.loc[0, "x"] = 99
df.loc[0, "x"]  # Now 99, df was mutated!
```

```
np.int64(99)
```

```
# To avoid this, explicitly copy:
df2 = df.copy()
```

Immutability prevents entire classes of bugs where one part of your code unexpectedly modifies data used elsewhere. When using Python, be explicit about copying when you need independent data.

5.4 The Functional Toolkit: Map, Filter, and Reduce

Most `for` loops can be replaced by one of three core functional concepts: **mapping**, **filtering**, or **reducing**. These are "higher-order functions": functions that take other functions as arguments.

5.4.1 1. Mapping: Applying a Function to Each Element

The pattern: You have a list of things, and you want to perform the same action on each element, producing a new list of the same length.

5.4.1.1 In R with `purrr::map()`

The {purrr} package is the gold standard for functional programming in R:

- `map()`: Always returns a list
- `map_dbl()`: Returns a vector of doubles (numeric)
- `map_chr()`: Returns a vector of characters (strings)
- `map_lgl()`: Returns a vector of logicals (booleans)

```
library(purrr)

# The classic for-loop way (verbose)
means_loop <- vector("double", ncol(mtcars))
for (i in seq_along(mtcars)) {
  means_loop[[i]] <- mean(mtcars[[i]], na.rm = TRUE)
}

# The functional way with map_dbl()
means_functional <- map_dbl(mtcars, mean, na.rm =
 ↪  TRUE)
```

The `map()` version is not just shorter; it's safer. You can't make an off-by-one error.

5.4.1.2 In Python with List Comprehensions

Python's most idiomatic tool for mapping is the **list comprehension**:

```
numbers = [1, 2, 3, 4, 5]
squares = [n**2 for n in numbers]
# > [1, 4, 9, 16, 25]
```

Python also has a built-in `map()` function:

```
def to_upper_case(s: str) -> str:
    return s.upper()

words = ["hello", "world"]
upper_words = list(map(to_upper_case, words))
# > ['HELLO', 'WORLD']
```

5.4.2 2. Filtering: Keeping Elements That Match a Condition

The pattern: You have a list of things, and you want to keep only the elements that satisfy a certain condition (don't confuse this with filtering rows of a data frame).

5.4.2.1 In R with `purrr::keep()`

```r
df1 <- data.frame(x = 1:50)
df2 <- data.frame(x = 1:200)
df3 <- data.frame(x = 1:75)
list_of_dfs <- list(a = df1, b = df2, c = df3)

# Keep only data frames with more than 100 rows
large_dfs <- keep(list_of_dfs, ~ nrow(.x) > 100)
```

5.4.2.2 In Python with List Comprehensions

List comprehensions have a built-in `if` clause:

```python
numbers = [1, 10, 5, 20, 15, 30]
large_numbers = [n for n in numbers if n > 10]
# > [20, 15, 30]
```

5.4.3 3. Reducing: Combining All Elements into a Single Value

The pattern: You have a list of things, and you want to iteratively combine them into a single summary value.

5.4.3.1 In R with `purrr::reduce()`

```r
# Sum all elements
total_sum <- reduce(c(1, 2, 3, 4, 5), `+`)

# Find common columns across multiple data frames
```

```r
list_of_colnames <- map(list_of_dfs, names)
common_cols <- reduce(list_of_colnames, intersect)
```

5.4.3.2 In Python with `functools.reduce`

```python
from functools import reduce
import operator

numbers = [1, 2, 3, 4, 5]
total_sum = reduce(operator.add, numbers)
# > 15
```

5.5 The Power of Composition

The final, beautiful consequence of a functional style is **composition**. You can chain functions together to build complex workflows from simple, reusable parts.

This R code is a sequence of function compositions:

```r
starwars %>%
  filter(!is.na(mass)) %>%
  select(species, sex, mass) %>%
  group_by(sex, species) %>%
  summarise(mean_mass = mean(mass), .groups =
   ↪  "drop")
```

Method chaining provides something similar to function composition:

```
(starwars_py
 .dropna(subset=['mass'])
 .filter(items=['species', 'sex', 'mass'])
 .groupby(['sex', 'species'])
 ['mass'].mean()
 .reset_index()
)
```

Each step is a function that takes a data frame and returns a new, transformed data frame. By combining `map`, `filter`, and `reduce` with this compositional style, you can express complex data manipulation pipelines without writing a single `for` loop.

5.5.1 Composition in Python

Method chaining in pandas is elegant but limited to the methods defined for `DataFrame` objects. R's pipe operators (`|>` and `%>%`) are more flexible because functions are not strictly owned by objects, and they can be more easily combined.

This reflects the languages' different philosophies:

- **R**'s lineage traces back to Scheme (a Lisp dialect), making functional composition natural
- **Python** was designed as an imperative, object-oriented language. In fact, Guido van Rossum, Python's creator, once proposed removing `map()`, `filter()`, and `reduce()` from the language entirely.[2] The community pushed back, but functional programming remains a second-class citizen in Python's design.

[2]The fate of reduce() in Python 3000

R is fundamentally a functional language that acquired OOP features, while Python is an OOP language with functional capabilities that got almost entirely removed from the language. I think that this the reason "Python feels weird" to R programmers and vice-versa.

5.6 Handling Errors Functionally

What happens when a function in your pipeline fails? In imperative code, you might wrap everything in try/catch blocks. Functional programming offers a cleaner approach: functions that *capture* errors rather than throw them.

But first, it is worth noting that **throwing an error is inherently impure**. When a function raises an exception, it does not return a value; instead, it transfers control flow to some unknown handler elsewhere in the program. This is a side effect. A pure function should always return a value, even when something goes wrong. The functional solution is to return a value that *represents* failure, rather than throwing an exception that breaks the normal flow.

5.6.1 In R with `purrr::safely()` and `purrr::possibly()`

The `{purrr}` package provides wrappers that turn error-prone functions into safe ones:

```
library(purrr)

# A function that might fail
```

```
risky_log <- function(x) {
  if (x <= 0) stop("x must be positive")
  log(x)
}

# safely() returns a list with $result and $error
safe_log <- safely(risky_log)
safe_log(10)   # list(result = 2.302585, error =
 ↪  NULL)
```

```
$result
[1] 2.302585

$error
NULL
```

```
safe_log(-1)   # list(result = NULL, error =
 ↪  <error>)
```

```
$result
NULL

$error
<simpleError in .f(...): x must be positive>
```

```
# possibly() returns a default value on error
maybe_log <- possibly(risky_log, otherwise = NA)
map_dbl(c(10, -1, 5), maybe_log)  # c(2.30, NA,
 ↪  1.61)
```

```
[1] 2.302585       NA 1.609438
```

This lets your pipeline continue even when some elements fail, and you can inspect failures afterwards.

5.6.2 In Python with Decorators

We saw decorators earlier as a way to wrap functions with extra behaviour. We can use the same pattern to capture errors:

```
import math
from functools import wraps

def maybe(default=None):
    """A decorator that catches exceptions and
     ↪  returns a default value."""
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            try:
                return func(*args, **kwargs)
            except Exception:
                return default
        return wrapper
    return decorator

@maybe(default=None)
def safe_log(x):
    if x <= 0:
        raise ValueError("x must be positive")
    return math.log(x)

results = [safe_log(x) for x in [10, -1, 5]]
```

This is cleaner than scattering try/except blocks throughout your code, and the `@maybe` decorator makes the error-handling strategy explicit.

5.6.3 The Limits of This Approach

While `safely()` and decorators help, they are still working around the fundamental impurity of exceptions. The returned `None` or `NA` values can propagate silently, causing confusing errors downstream. For a more principled approach, you need **monads**: data structures that explicitly represent success or failure and force you to handle both cases. We will explore this in **Chapter 9: Robust Pipelines with Monads**, where we introduce `{chronicler}` for R and `talvez` for Python.

5.7 When NOT to Use Functional Programming

Functional programming is powerful, but it is not always the best choice. Here are situations where imperative or object-oriented code may be clearer:

1. **Complex stateful algorithms**: Some algorithms (like graph traversals or simulations) naturally require mutable state. Forcing them into a functional style can make the code harder to read.
2. **Performance-critical inner loops**: Functional abstractions like `map()` introduce overhead. In tight loops where microseconds matter, a simple `for` loop may be faster.

The goal is not functional purity for its own sake, but **clarity and correctness**. Use functional techniques where they help, and step back to simpler approaches when they do not. Thankfully, LLMs here are quite useful again, as you can use them if you need to write complex imperative code with loops.

5.8 Summary

This chapter has laid the groundwork for writing reproducible code by embracing **Functional Programming**.

Key takeaways:

- **Pure functions** guarantee the same output for the same input, with no hidden dependencies on global state
- Make impure operations (like randomness) explicit by controlling the seed
- Replace error-prone `for` loops with **map, filter, and reduce**
- Use **composition** to build complex pipelines from simple, reusable functions
- In Python, treat stateful library objects as values passed between pure functions

Understanding the distinction between R's functional heritage and Python's OOP nature is key to becoming an effective data scientist in either language. By mastering the functional paradigm, you're building a foundation for code that is robust, easy to review, simple to debug, and truly reproducible.

But first, in the next chapter, we'll learn how to **test** these pure functions—a skill that will make your code far more reliable. Then, in Chapters 7 and 8, we'll put everything into practice with `{rixpress}`, a package that leverages functional composition and Nix to build fully reproducible, polyglot analytical pipelines.

6 Professional Workflows: Testing and Git

6.1 Introduction

In the previous chapter, we learned that pure functions—those with no side effects and deterministic outputs—are the building blocks of reliable code. Small, testable functions are the foundation of reproducible pipelines.

But how do we *prove* that our functions actually do what we claim they do? And how do we manage the complexity of collaborating with others (and with AI) without breaking everything?

This brings us to two final, crucial questions: how do we prove that our functions actually do what we claim they do? And how do we manage the complexity of collaborating with others (and with AI) without breaking everything?

This chapter addresses both questions by introducing unit testing and advanced Git workflows. You will learn what unit tests are and why they are essential for reliable data analysis, how to write and run them in both R (with `{testthat}`) and Python (with `pytest`), and how to use LLMs to accelerate test writing while embracing your role as a code reviewer. We will also cover professional Git techniques like Trunk-Based Development and

interactive staging, which are especially valuable when managing code generated by AI assistants.

6.2 Part 1: Unit Testing

The answer to "how do we prove it works?" is unit testing. A unit test is a piece of code whose sole job is to check that another piece of code, a "unit", works correctly. In our functional world, the "unit" is almost always a single function. This is why we spent so much time on FP in the previous chapter. Small, pure functions are not just easy to reason about; they are incredibly easy to test.

Writing tests is your contract with your collaborators and your future self. It's a formal promise that your function, `calculate_mean_mpg()`, given a specific input, will always produce a specific, correct output. It's the safety net that catches bugs before they make it into your final analysis and the tool that gives you the confidence to refactor and improve your code without breaking it.

6.2.1 The Philosophy of a Good Unit Test

So, what should we test? Writing good tests is a skill, but it revolves around answering a few key questions about your function. For any function you write, you should have tests that cover:

- The "Happy Path": does the function return the expected, correct value for a typical, valid input?

- Bad Inputs: does the function fail gracefully or throw an informative error when given garbage input (e.g., a string instead of a number, a data frame with the wrong columns)?
- Edge Cases: how does the function handle tricky but valid inputs? For example, what happens if it receives an empty data frame, a vector with `NA` values, or a vector where all the numbers are the same?

Writing tests forces you to think through these scenarios, and in doing so, almost always leads you to write more robust and well-designed functions.

6.2.2 Unit Testing in Practice

Let's imagine we've written a simple helper function to normalise a numeric vector (i.e., scale it to have a mean of 0 and a standard deviation of 1). We'll save this in a file named `utils.R` or `utils.py`.

R version (`utils.R`):

```r
normalize_vector <- function(x) {
  (x - mean(x, na.rm = TRUE)) / sd(x, na.rm = TRUE)
}
```

Python version (`utils.py`):

```python
import numpy as np

def normalize_vector(x):
  return (x - np.nanmean(x)) / np.nanstd(x)
```

Now, let's write tests for it.

6.2.2.1 Testing in R with `{testthat}`

In R, the standard for unit testing is the `{testthat}` package. The convention is to create a `tests/testthat/` directory in your project, and for a script `utils.R`, you would create a test file named `test-utils.R`.

Inside `test-utils.R`, we use the `test_that()` function to group related expectations.

```
# In file: tests/testthat/test-utils.R

# First, we need to load the function we want to
 ↪  test
source("../../utils.R")

library(testthat)

test_that("Normalization works on a simple vector
 ↪  (the happy path)", {
  # 1. Setup: Create input and expected output
  input_vector <- c(10, 20, 30)
  expected_output <- c(-1, 0, 1)

  # 2. Action: Run the function
  actual_output <- normalize_vector(input_vector)

  # 3. Expectation: Check if the actual output
   ↪  matches the expected output
  expect_equal(actual_output, expected_output)
})
```

```
test_that("Normalization handles NA values
 ↪  correctly", {
  input_with_na <- c(10, 20, 30, NA)
  expected_output <- c(-1, 0, 1, NA)

  actual_output <- normalize_vector(input_with_na)

  # We need to use expect_equal because it knows how
   ↪  to compare NAs
  expect_equal(actual_output, expected_output)
})
```

The `expect_equal()` function checks for near-exact equality. `{testthat}` has many other `expect_*()` functions, like `expect_error()` to check that a function fails correctly, or `expect_warning()` to check for warnings.

6.2.2.2 Testing in Python with `pytest`

In Python, the de facto standard is `pytest`. It's incredibly simple and powerful. The convention is to create a `tests/` directory, and your test files should be named `test_*.py`. Inside, you just write functions whose names start with `test_` and use Python's standard `assert` keyword.

```
# In file: tests/test_utils.py

import numpy as np
from utils import normalize_vector # Import our
 ↪  function
```

```
def test_normalize_vector_happy_path():
    # 1. Setup
    input_vector = np.array([10, 20, 30])
    expected_output = np.array([-1.0, 0.0, 1.0])

    # 2. Action
    actual_output = normalize_vector(input_vector)

    # 3. Expectation
    # For floating point numbers, it's better to
     ↪  check for "close enough"
    assert np.allclose(actual_output,
     ↪  expected_output)

def test_normalize_vector_with_nas():
    input_with_na = np.array([10, 20, 30, np.nan])
    expected_output = np.array([-1.0, 0.0, 1.0,
 ↪  np.nan])

    actual_output = normalize_vector(input_with_na)

    # `np.allclose` doesn't handle NaNs, but
     ↪  `np.testing.assert_allclose` does!
    np.testing.assert_allclose(actual_output,
 ↪  expected_output)
```

Because this isn't a package though (yet), but a simple project with scripts, you also need to create another file called `pytest.ini`, which will tell `pytest` where to find the tests:

```
[pytest]
# Discover tests in the tests/ directory
```

```
testpaths = tests/

# Include default patterns and your hyphenated
 ↪  pattern
python_files = test_*.py *_test.py test-*.py

# Adds the root directory to the pythonpath, without
 ↪  this
# it'll be impossible to import normalize_vector()
 ↪  from
# utils.py
pythonpath = .
```

To run your tests, you simply navigate to your project's root directory in the terminal and run the command `pytest`. It will automatically discover and run all your tests for you. That being said, you will get an error:

```
>       assert np.allclose(actual_output,
 ↪  expected_output)
E       assert False
E        +  where False = <function allclose at
 ↪  0x7f0e81c959f0>(array([-1.22474487,  0.        ,
 ↪  1.22474487]), array([-1.,  0.,  1.])    def
 ↪  test_normalize_vector_with_nas():
        input_with_na = np.array([10, 20, 30,
 ↪  np.nan])
        expected_output = np.array([-1.0, 0.0, 1.0,
 ↪  np.nan])

        actual_output =
 ↪  normalize_vector(input_with_na)
```

```
        # `np.allclose` doesn't handle NaNs, but
         ↪  `np.testing.assert_allclose` does!
>       np.testing.assert_allclose(actual_output,
 ↪  expected_output)
E       AssertionError:
E       Not equal to tolerance rtol=1e-07, atol=0
E
E       Mismatched elements: 2 / 4 (50%)
E       Max absolute difference among violations:
 ↪  0.22474487
E       Max relative difference among violations:
 ↪  0.22474487
E        ACTUAL: array([-1.224745,  0.      ,
 ↪  1.224745,       nan])
E        DESIRED: array([-1.,  0.,  1., nan])

tests/test_utils.py:25: AssertionError
=============================================================
 ↪  short test summary info
 ↪  =============================================================
FAILED
 ↪  tests/test_utils.py::test_normalize_vector_happy_path
 ↪  - assert False
FAILED
 ↪  tests/test_utils.py::test_normalize_vector_with_nas
 ↪  - AssertionError:
=============================================================
 ↪  2 failed in 0.15s
 ↪  =============================================================
```

You don't encounter this issue in R and understanding why reveals how valuable unit tests can be. (Hint: how does the

implementation of the `sd` function, which computes the standard deviation, differ between R and NumPy?)

6.2.3 Testing as a Design Tool

Testing can also help you with programming, by thinking about edge cases. For example, what happens if we try to normalise a vector where all the elements are the same?

Let's write a test for this edge case first.

`pytest` version:

```
# tests/test_utils.py
def test_normalize_vector_with_zero_std():
    input_vector = np.array([5, 5, 5, 5])
    actual_output = normalize_vector(input_vector)
    # The current function will return `[nan, nan,
     ↪  nan, nan]`
    # Let's assert that we expect a vector of zeros
     ↪  instead.
    assert np.allclose(actual_output, np.array([0,
     ↪  0, 0, 0]))
```

If we run `pytest` now, this test will fail. Our test has just revealed a flaw in our function's design. This process is a core part of Test-Driven Development (TDD): write a failing, but correct, test, then write the code to make it pass.

Let's improve our function:

```
import numpy as np

def normalize_vector(x):
  std_dev = np.nanstd(x)
  if std_dev == 0:
    # If std is 0, all elements are the mean. Return
     ↪  a vector of zeros.
    return np.zeros_like(x, dtype=float)
  return (x - np.nanmean(x)) / std_dev
```

Now, if we run `pytest` again, our new test will pass. We used testing not just to verify our code, but to actively make it more robust and thoughtful.

6.2.4 The Modern Data Scientist's Role: Reviewer and AI Collaborator

In the past, writing tests was often seen as a chore. Today, LLMs make this process very easy.

6.2.4.1 Using LLMs to Write Tests

LLMs are fantastic at writing unit tests. They are good at handling boilerplate code and *thinking* of edge cases. You can provide your function to an LLM and give it a prompt like this:

> "Here is my Python function `normalize_vector`. Please write three `pytest` unit tests for it. Include a test for the happy path with a simple array, a test for

> an array containing `np.nan`, and a test for the edge case where all elements in the array are identical."

The LLM will likely generate high-quality test code that is very similar to what we wrote above.

> **All you need is context**
>
> When using an LLM to generate tests, context is everything. If you are writing tests for functions that heavily use specific packages (like `{dplyr}` or `pandas`), providing the `pkgctx` output for those packages ensures the LLM writes idiomatic and correct tests.
> For example, if you are testing a function that uses `{rix}`, feed the `rix.pkgctx.yaml` to the LLM so it knows exactly how to mock or assert the outputs of `rix()` functions. This is very useful particularly for packages that are not very well-known (or internal packages to your company that aren't even public) or packages that have been updated very recently, as the LLMs training data cutoff might not include the latest versions of the package.

This is a massive productivity boost. However, this introduces a new, critical role for the data scientist: you are the reviewer.

An LLM does not write your tests; it generates a draft. It is your professional responsibility to:

1. Read and understand every line of the test code.
2. Verify that the `expected_output` is actually correct.
3. Confirm that the tests cover the cases you care about.
4. Commit that code under your name, taking full ownership of it.

"A COMPUTER CAN NEVER BE HELD ACCOUNTABLE THEREFORE A COMPUTER MUST NEVER MAKE A MANAGEMENT DECISION" - IBM Training Manual, 1979.

This principle is, in my opinion, even more important for tests than for production code. Even before LLMs, we relied on code we didn't write ourselves. At my first job, my boss insisted that I avoid external packages entirely and rewrite everything from scratch. I ignored her and continued using the packages I trusted. At some point, you have to trust strangers. That was true then, and it is true now, except that "strangers" now includes LLMs and developers who themselves rely heavily on LLMs.

But here is the key difference: tests are small. Unlike a sprawling codebase, a unit test is short enough to read and understand completely. When you review an LLM-generated test, you can verify that the expected output is correct and that the test covers the right cases. This makes tests uniquely suited to the "trust but verify" workflow that AI-assisted development demands.

In the next chapter, we will start building reproducible pipelines with `{rixpress}`, putting our tested functions to work.

6.3 Part 2: Advanced Version Control

I assume you already know the basics of Git: `clone`, `add`, `commit`, and `push`. If you don't, there are countless tutorials available, and I highly recommend you check one out and then come back to this.

In this section, we will focus on the techniques that separate the "I submit code via Dropbox" novice from the professional data scientist: Trunk-Based Development and managing AI-generated code.

6.3.1 A Better Way to Collaborate: Trunk-Based Development

A common mistake for new teams is to use branches to add new features, and then let these feature branches live for a very long time. A data scientist might create a branch called `feature-big-analysis`, work on it for three weeks, and then try to merge it back into `main`. The result is often what's called "merge hell": `main` has changed so much in three weeks that merging the branch back in creates dozens of conflicts and is a painful, stressful process.

To avoid this, many professional teams use a workflow called Trunk-Based Development (TBD). The philosophy is simple but powerful:

> All developers integrate their work back into the main branch (the "trunk") as frequently as possible, at least once a day.

This means that feature branches are incredibly short-lived. Instead of a single, massive feature branch that takes weeks, you create many tiny branches that each take a few hours or a day at most.

6.3.1.1 How to Work with Short-Lived Branches

But how can you merge something back into `main` if the feature isn't finished? The `main` branch must always be stable and runnable. You can't merge broken code.

The first way to solve this issue is to use feature flags.

A feature flag is just a simple variable (like a `TRUE`/`FALSE` switch) that lets you turn a new, unfinished part of the code on or off.

This allows you to merge the code into `main` while keeping it "off" until it's ready.

Imagine you are adding a new, complex plot to `analysis.R`, but it will take a few days to get right.

```
# At the top of your analysis.R script
# --- Configuration ---
use_new_scatterplot <- FALSE # Set to FALSE while in
 ↪  development

# ... lots of existing, working code ...

# --- New Feature Code ---
if (use_new_scatterplot) {
  # All your new, unfinished, possibly-buggy
   ↪  plotting code goes here.
  # It won't run as long as the flag is FALSE.
  library(scatterplot3d)
  scatterplot3d(mtcars$mpg, mtcars$hp, mtcars$wt)
}
```

With this `if` block, you can safely merge your changes into `main`. The new code is there, but it won't execute and won't break the existing analysis. Other developers can pull your changes and won't even notice. Once you've finished the feature in subsequent small commits, the final change is just to flip the switch: `use_new_scatterplot <- TRUE`.

6.3.2 Collaborative Hygiene: Rebase vs. Merge

When you and a colleague both work on `main`, your histories can diverge. When you `git pull`, Git behaves in two ways:

1. Merge (Default): creates a "merge commit" that ties the histories together. This results in a messy, non-linear history graph if done frequently.
2. Rebase (Recommended): temporarily lifts your commits, pulls your colleague's changes, and then replays your commits on top.

In data science projects, where identifying when a plot or number changed is critical, a linear history is valuable. I strongly recommend configuring Git to use rebase by default:

```
git config --global pull.rebase true
```

Now, when you `git pull`, your local changes will stay at the "tip" of the history, keeping the log clean.

6.3.3 Working with LLMs and Git: Managing AI-Generated Changes

When working with LLMs like GitHub Copilot, it's crucial to review changes carefully before committing them. Git provides a powerful tool for this: interactive staging.

6.3.3.1 Interactive Staging: Accepting changes chunk by chunk

Git's interactive staging feature (`git add -p`) is perfect for reviewing LLM changes. Instead of blindly adding all changes with `git add .`, `git add -p` lets you review each "hunk" (chunk of changes) individually.

Suppose an LLM refactored your Python script. You run:

```
git add -p analysis.py
```

Git will show you a chunk of changes and prompt you:

```
@@ -1,2 +1,4 @@
 # Load required libraries
 import pandas as pd
+import matplotlib.pyplot as plt
+import seaborn as sns
Stage this hunk [y,n,q,a,d,s,e,?]?
```

You can reply:

- `y`: Yes, stage this (I approve).
- `n`: No, do not stage this (I reject or want to edit later).
- `s`: Split this hunk into smaller pieces (if the LLM did two unrelated things in one block).
- `e`: Edit the hunk manually before staging.

This forces you to be the reviewer. It prevents "accidental" AI hallucinations (like deleting a critical import) from slipping into your repository.

6.3.3.2 Example LLM Workflow

1. Save state: `git commit -m "Working state before LLM"`
2. Prompt LLM: "Please refactor this function to be more efficient."
3. Review: Run `git diff` to see what it did.
4. Select: Run `git add -p` and say `y` only to the parts that look correct.

5. Clean: Run `git checkout .` to discard the parts you rejected.
6. Verify: Run your unit tests!

This workflow ensures you maintain full control over your codebase while benefiting from LLM assistance.

6.4 Summary

In this chapter, we covered the safety protocols of professional software engineering:

1. Unit tests prove your code works and protect you from regressions.
2. Git workflows (Trunk-Based Development, Rebasing) keep your collaboration clean and history linear.
3. Code review (via Interactive Staging) is your primary defence against AI-generated bugs.

7 Building Reproducible Pipelines with rixpress

7.1 Introduction: From Scripts and Notebooks to Pipelines

With reproducible environments (Nix, `{rix}`), functional code (Chapter 5), and testing skills (Chapter 6) in hand, we're now ready to orchestrate complete analytical pipelines. But there's one more piece to the puzzle: **orchestration**.

How do we take our collection of functions and data files and run them in the correct order to produce our final data product? This problem of managing computational workflows is not new, and a whole category of **build automation tools** has been created to solve it.

7.1.1 The Evolution of Build Automation

The original solution, dating back to the 1970s, is **`make`**. Created by Stuart Feldman at Bell Labs in 1976, `make` reads a `Makefile` that describes the dependency graph of a project. If you change the code that generates `plot.png`, `make` is smart enough to only re-run the steps needed to rebuild the plot and the final report.

The strength of these tools is their language-agnosticism, but their weaknesses are twofold:

1. **File-centric**: You must manually handle all I/O. Your first script saves `data.csv`, your second loads it. This adds boilerplate and creates surfaces for error.
2. **Environment-agnostic**: They track files but know nothing about the *software environment* needed to create those files.

This is where R's **`{targets}`** package shines. It tracks dependencies between **R objects directly**, automatically handling serialisation. But `{targets}` operates within a single R session; for polyglot pipelines, you must manually coordinate via `{reticulate}`.

7.1.2 The Separation Problem

All these tools (from `make` to `{targets}` to Airflow) separate workflow management from environment management. You use one tool to run the pipeline and another (Docker, `{renv}`) to set up the software.

This separation creates friction. Running `{targets}` inside Docker ensures reproducibility, but forces the entire pipeline into one monolithic environment. What if your Python step requires TensorFlow 2.15 but your R step needs reticulate with Python 3.9? You're stuck.

7.1.3 The Imperative Approach: Make + Docker

To illustrate this, consider the traditional setup for a polyglot pipeline. You'd need:

1. A `Dockerfile` to set up the environment
2. A `Makefile` to orchestrate the workflow
3. Wrapper scripts for each step

Here's what a `Makefile` might look like:

```
# Makefile for a Python → R pipeline

DATA_DIR = data
OUTPUT_DIR = output

$(OUTPUT_DIR)/predictions.csv: $(DATA_DIR)/raw.csv
 ↪  scripts/train_model.py
python scripts/train_model.py $(DATA_DIR)/raw.csv $@

$(OUTPUT_DIR)/plot.png:
 ↪  $(OUTPUT_DIR)/predictions.csv
 ↪  scripts/visualise.R
Rscript scripts/visualise.R $< $@

$(OUTPUT_DIR)/report.html: $(OUTPUT_DIR)/plot.png
 ↪  report.qmd
quarto render report.qmd -o $@

all: $(OUTPUT_DIR)/report.html

clean:
rm -rf $(OUTPUT_DIR)/*
```

This looks clean, but notice the procedural boilerplate required in each script. Your `train_model.py` must parse command-line arguments and handle file I/O:

```
# scripts/train_model.py
import sys
import pandas as pd
from sklearn.ensemble import RandomForestClassifier

def main():
    input_path = sys.argv[1]
    output_path = sys.argv[2]

    # Load data
    df = pd.read_csv(input_path)

    # ... actual ML logic ...

    # Save results
    predictions.to_csv(output_path, index=False)

if __name__ == "__main__":
    main()
```

And your `visualise.R` script needs the same boilerplate:

```
# scripts/visualise.R
args <- commandArgs(trailingOnly = TRUE)
input_path <- args[1]
output_path <- args[2]

# Load data
predictions <- read.csv(input_path)

# ... actual visualisation logic ...

# Save plot
```

```
ggsave(output_path, plot)
```

The scientific logic is buried under file I/O scaffolding. And the environment? That's a separate 100+ line `Dockerfile` you must maintain.

7.1.4 rixpress: Unified Orchestration

This brings us to a key insight: a reproducible pipeline should be nothing more than a **composition of pure functions**, each with explicit inputs and outputs, no hidden state, and no reliance on execution order beyond what the data dependencies require.

`{rixpress}` solves this by using Nix not just as a package manager, but as the build automation engine itself. Each pipeline step is a **Nix derivation**: a hermetically sealed build unit.

Compare the same pipeline in `{rixpress}`:

```
library(rixpress)

list(
  rxp_py_file(
    name = raw_data,
    path = "data/raw.csv",
    read_function = "lambda x: pandas.read_csv(x)"
  ),

  rxp_py(
    name = predictions,
    expr = "train_model(raw_data)",
    user_functions = "functions.py"
```

```
  ),

  rxp_py2r(
    # rxp_py2r uses reticulate for conversion
    name = predictions_r,
    expr = predictions
  ),

  rxp_r(
    name = plot,
    expr = visualise(predictions_r),
    user_functions = "functions.R"
  ),

  rxp_qmd(
    name = report,
    qmd_file = "report.qmd"
  )
) |>
  rxp_populate()
```

And your `functions.py` contains *only* the scientific logic:

```
# functions.py
def train_model(df):
    # ... pure ML logic, no file I/O ...
    return predictions
```

The difference is stark:

Aspect	Make + Docker	rixpress
Files needed	Dockerfile, Makefile, wrapper scripts	gen-env.R, gen-pipeline.R, function files
I/O handling	Manual in every script	Automatic via encoders/decoders
Dependencies	Explicit file rules	Inferred from object references
Environment	Separate Docker setup	Unified via `{rix}`
Expertise needed	Linux admin, Make syntax	R programming

This provides two key benefits:

1. **True Polyglot Pipelines**: Each step can have its own Nix environment. A Python step runs in a pure Python environment, an R step in an R environment, a Quarto step in yet another, all within the same pipeline. Julia is also supported, and `{reticulate}` can be used for data transfer, but arbitrary encoder and decoder as well (more on this later).

2. **Deep Reproducibility**: Each step is cached based on the cryptographic hash of *all* its inputs: the code, the data, *and the environment*. Any change in dependencies triggers a rebuild. This is reproducibility at the build level, not just the environment level.

The interface is heavily inspired by `{targets}`, so you get the ergonomic, object-passing feel you're used to, combined with the bit-for-bit reproducibility of the Nix build system.

Getting LLM assistance with `{rixpress}` and `ryxpress`

If the `{rixpress}` syntax is new to you, remember that you can use `pkgctx` to generate LLM-ready context (as mentioned in the introduction). Both the `{rixpress}` (R) and `ryxpress` (Python) repositories include `.pkgctx.yaml` files you can feed to your LLM to help it understand the package's API. You can also generate your own context files:

```
# For the R package
nix run github:b-rodrigues/pkgctx -- r
 ↪  github:b-rodrigues/rixpress >
 ↪  rixpress.pkgctx.yaml

# For the Python package
nix run github:b-rodrigues/pkgctx -- python
 ↪  ryxpress > ryxpress.pkgctx.yaml
```

With this context, your LLM can help you write correct pipeline definitions, even if the syntax is completely new to you. You can do so for any package hosted on CRAN, GitHub, or local `.tar.gz` files.

7.2 What is rixpress?

`{rixpress}` streamlines creation of *micropipelines* (small-to-medium, single-machine analytic pipelines) by expressing a pipeline in idiomatic R while delegating build orchestration to the Nix build system.

Key features:

- Define pipeline *derivations* with concise `rxp_*()` helper functions
- Seamlessly mix R, Python, Julia, and Quarto steps
- Reuse hermetic environments defined via `{rix}` and a `default.nix`
- Visualise and inspect the DAG; selectively read, load, or copy outputs
- Automatic caching: only rebuild what changed

`{rixpress}` provides several functions to define pipeline steps:

Function	Purpose
`rxp_r()`	Run R code
`rxp_r_file()`	Read a file using R
`rxp_py()`	Run Python code
`rxp_py_file()`	Read a file using Python
`rxp_qmd()`	Render a Quarto document
`rxp_py2r()`	Convert Python object to R using reticulate
`rxp_r2py()`	Convert R object to Python using reticulate

Here is what a basic pipeline looks like:

```
library(rixpress)

list(
  rxp_r_file(
    mtcars,
    'mtcars.csv',
    \(x) read.csv(file = x, sep = "|")
  ),
```

```
  rxp_r(
    mtcars_am,
    filter(mtcars, am == 1)
  ),

  rxp_r(
    mtcars_head,
    head(mtcars_am)
  ),

  rxp_qmd(
    page,
    "page.qmd"
  )
) |>
  rxp_populate()
```

7.3 Getting Started

7.3.1 Initialising a project

If you're starting fresh, you can bootstrap a project using a temporary shell:

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

Once inside, start R and run:

```
rixpress::rxp_init()
```

This creates two essential files:

- **gen-env.R**: Where you define your environment with {rix}
- **gen-pipeline.R**: Where you define your pipeline with {rixpress}

7.3.2 Defining the environment

Open gen-env.R and define the tools your pipeline needs:

```
library(rix)

rix(
  date = "2025-10-14",
  r_pkgs = c("dplyr", "ggplot2", "quarto",
   ↪  "rixpress"),
  ide = "none",
  project_path = ".",
  overwrite = TRUE
)
```

Run this script to generate default.nix, then build and enter your environment:

```
nix-build
nix-shell
```

7.3.3 Defining the pipeline

Open gen-pipeline.R and define your pipeline:

```
library(rixpress)

list(
  rxp_r_file(
    name = mtcars,
    path = "data/mtcars.csv",
    read_function = \(x) read.csv(x, sep = "|")
  ),

  rxp_r(
    name = mtcars_am,
    expr = dplyr::filter(mtcars, am == 1)
  ),

  rxp_r(
    name = mtcars_head,
    expr = head(mtcars_am)
  )
) |>
  rxp_populate()
```

Running `rxp_populate()` generates a `pipeline.nix` file and builds the entire pipeline. It also creates a `_rixpress` folder with required files for the project.

You don't need to write `gen-pipeline.R` completely in one go. Instead, start with a single derivation, usually the one loading the data, and build the pipeline using `source("gen-pipeline.R")` and then `rxp_make()`. Alternatively, if you set `build = TRUE` in `rxp_populate()`, sourcing the script alone would be enough, as the pipeline would be built automatically.

In an interactive session, you can use `rxp_read()`, `rxp_load()` to read or load artifacts into the R session respectively and

`rxp_trace()` to check the lineage of an artifact. This way, you can also work interactively with `{rixpress}`: add derivations one by one and incrementally build the pipeline. If you change previous derivations or functions that affect them, you don't need to worry about re-running past steps—these will be executed automatically since they've changed!

7.4 Real-World Examples

The rixpress_demos repository contains many complete examples. Here are a few practical patterns to get you started.

7.4.1 Example 1: Reading Many Input Files

When you have multiple CSV files in a directory:

```
library(rixpress)

list(
  # R approach: read all files at once
  rxp_r_file(
    name = mtcars_r,
    path = "data",
    read_function = \(x) {
      readr::read_delim(list.files(x, full.names =
↪   TRUE), delim = "|")
    }
  ),

  # Python approach: custom function
  rxp_py_file(
```

```
    name = mtcars_py,
    path = "data",
    read_function = "read_many_csvs",
    user_functions = "functions.py"
  ),

  rxp_py(
    name = head_mtcars,
    expr = "mtcars_py.head()"
  )
) |>
  rxp_populate()
```

The key insight: `rxp_r_file()` and `rxp_py_file()` can point to a directory, and your `read_function` handles the logic.

7.4.2 Example 2: Machine Learning with XGBoost

This pipeline trains an XGBoost classifier in Python, then passes predictions to R for evaluation with `{yardstick}`:

```
library(rixpress)

list(
  # Load data as NumPy array
  rxp_py_file(
    name = dataset_np,
    path = "data/pima-indians-diabetes.csv",
    read_function = "lambda x: loadtxt(x,
     ↪  delimiter=',')"
```

```
),

# Split features and target
rxp_py(name = X, expr = "dataset_np[:,0:8]"),
rxp_py(name = Y, expr = "dataset_np[:,8]"),

# Train/test split
rxp_py(
  name = splits,
  expr = "train_test_split(X, Y, test_size=0.33,
   ↪  random_state=7)"
),

# Extract splits

rxp_py(name = X_train, expr = "splits[0]"),
rxp_py(name = X_test, expr = "splits[1]"),
rxp_py(name = y_train, expr = "splits[2]"),
rxp_py(name = y_test, expr = "splits[3]"),

# Train XGBoost model
rxp_py(
  name = model,
  expr = "XGBClassifier(use_label_encoder=False,
   ↪  eval_metric='logloss').fit(X_train,
   ↪  y_train)"
),

# Make predictions
rxp_py(name = y_pred, expr =
 ↪  "model.predict(X_test)"),

# Export predictions to CSV for R
```

```
  rxp_py(
    name = combined_df,
    expr = "DataFrame({'truth': y_test, 'estimate':
     ↪  y_pred})"
  ),

  rxp_py(
    name = combined_csv,
    expr = "combined_df",
    user_functions = "functions.py",
    encoder = "write_to_csv"
  ),

  # Compute confusion matrix in R
  rxp_r(
    combined_factor,
    expr = mutate(combined_csv, across(everything(),
     ↪  factor)),
    decoder = "read.csv"
  ),

  rxp_r(
    name = confusion_matrix,
    expr = yardstick::conf_mat(combined_factor,
     ↪  truth, estimate)
  )
) |>
  rxp_populate(build = FALSE)

# Adjust Python imports
adjust_import("import numpy", "from numpy import
 ↪  array, loadtxt")
adjust_import("import xgboost", "from xgboost import
 ↪  XGBClassifier")
```

```
adjust_import("import sklearn", "from
 ↪  sklearn.model_selection import
 ↪  train_test_split")
add_import("from pandas import DataFrame",
 ↪  "default.nix")

rxp_make()
```

This demonstrates:

- Python-heavy computation with XGBoost
- Custom serialization via `encoder`/`decoder`
- Adjusting Python imports with `adjust_import()` and `add_import()`
- Passing results to R for evaluation

7.4.3 Example 3: Simple Python→R→Quarto Workflow

A complete pipeline that bounces data between languages and renders a report:

```
library(rixpress)

list(
  # Read with Python polars
  rxp_py_file(
    name = mtcars_pl,
    path = "data/mtcars.csv",
    read_function = "lambda x: polars.read_csv(x,
     ↪  separator='|')"
```

```
),

# Filter in Python, convert to pandas for
 ↪  reticulate
rxp_py(
  name = mtcars_pl_am,
  expr = "mtcars_pl.filter(polars.col('am') ==
   ↪  1).to_pandas()"
),

# Convert to R
rxp_py2r(name = mtcars_am, expr = mtcars_pl_am),

# Process in R
rxp_r(
  name = mtcars_head,
  expr = my_head(mtcars_am),
  user_functions = "functions.R"
),

# Back to Python
rxp_r2py(name = mtcars_head_py, expr =
 ↪  mtcars_head),

# More Python processing
rxp_py(name = mtcars_tail_py, expr =
 ↪  "mtcars_head_py.tail()"),

# Back to R
rxp_py2r(name = mtcars_tail, expr =
 ↪  mtcars_tail_py),

# Final R step
```

```
  rxp_r(name = mtcars_mpg, expr =
   ↪  dplyr::select(mtcars_tail, mpg)),

  # Render Quarto document
  rxp_qmd(
    name = page,
    qmd_file = "my_doc/page.qmd",
    additional_files = c("my_doc/content.qmd",
     ↪  "my_doc/images")
  )
) |>
  rxp_populate()
```

Note the `additional_files` argument for `rxp_qmd()`: this includes child documents and images that the main Quarto file needs.

7.5 Working with Pipelines

7.5.1 Building the pipeline

You can build the pipeline in two steps:

```
# Generate pipeline.nix only (don't build)
rxp_populate(build = FALSE)

# Build the pipeline
rxp_make()
```

Or in a single step with `rxp_populate(build = TRUE)`. Once you've built the pipeline, you can inspect the results.

7.5.2 Inspecting outputs

Because outputs live in `/nix/store/`, `{rixpress}` provides helpers:

```
# List all built artifacts
rxp_inspect()

# Read an artifact into R
rxp_read("mtcars_head")

# Load an artifact into the global environment
rxp_load("mtcars_head")
```

It is important to understand that all artifacts are stored in the `/nix/store/`. So if you want to save the outputs somewhere else, you need to use `rxp_copy()`.

```
rxp_copy("clean_data")
```

This will copy the `clean_data` artifacts into the current working directory. If you call `rxp_copy()` without arguments, every artifact will be copied over.

Over time, the Nix store can accumulate many artifacts from previous pipeline runs. Use `rxp_gc()` to clean up old build outputs and reclaim disk space:

```
# Preview what would be deleted (dry run)
rxp_gc(keep_since = "2026-01-01", dry_run = TRUE)

# Delete artifacts older than a specific date
```

```
rxp_gc(keep_since = "2026-01-01")

# Full garbage collection (removes all unreferenced
 ↪  store paths)
rxp_gc()
```

The `keep_since` argument lets you preserve recent builds while cleaning up older ones. This is particularly useful on shared machines or CI runners where disk space is limited.

7.5.3 Visualising the pipeline

There are three ways to visualise pipelines. The first is to look at the *trace* of a derivation or of the full pipeline. This is similar to a genealogy tree. You can start by running `rxp_inspect()` to double check how the derivation names are written:

```
rxp_inspect()
```

```
derivation build_success
1              X_test          TRUE
2             X_train          TRUE
3               alpha          TRUE
4                beta          TRUE
5    all-derivations          TRUE
6               delta          TRUE
7        final_report          TRUE
8   model_predictions          TRUE
9         output_plot          TRUE
10        predictions          TRUE
11     processed_data          TRUE
```

```
12                rho          TRUE
13              sigma          TRUE
14            sigma_z          TRUE
15 simulated_rbc_data          TRUE
16      trained_model          TRUE
17             y_test          TRUE
18            y_train          TRUE
```

Let's check the trace of `output_plot`:

```
rxp_trace("output_plot")
```

```
==== Lineage for: output_plot ====
Dependencies (ancestors):
  - predictions
    - y_test*
      - processed_data*
        - simulated_rbc_data*
          - alpha*
          - beta*
          - delta*
          - rho*
          - sigma*
          - sigma_z*
    - model_predictions*
      - X_test*
        - processed_data*
      - trained_model*
        - X_train*
          - processed_data*
        - y_train*
          - processed_data*
```

```
Reverse dependencies (children):
  - final_report

Note: '*' marks transitive dependencies (depth >=
 ↪  2).
```

We see that `output_plot` depends directly on `predictions`, and `predictions` on `y_test`, which itself depends on all the other listed dependencies. The children of the derivation are also listed, in this case `final_report`.

Calling `rxp_trace()` without arguments shows the entire pipeline:

```
rxp_trace()
```

```
- final_report
  - simulated_rbc_data
    - alpha*
    - beta*
    - delta*
    - rho*
    - sigma*
    - sigma_z*
  - output_plot
    - predictions*
      - y_test*
        - processed_data*
          - simulated_rbc_data*
      - model_predictions*
        - X_test*
```

```
          - processed_data*
        - trained_model*
          - X_train*
            - processed_data*
          - y_train*
            - processed_data*

Note: '*' marks transitive dependencies (depth >=
 ↪  2).
```

Another way to visualise the pipeline is by using `rxp_ggdag()`. This requires the `{ggdag}` package to be installed in the environment. Calling `rxp_ggdag()` shows the following plot:

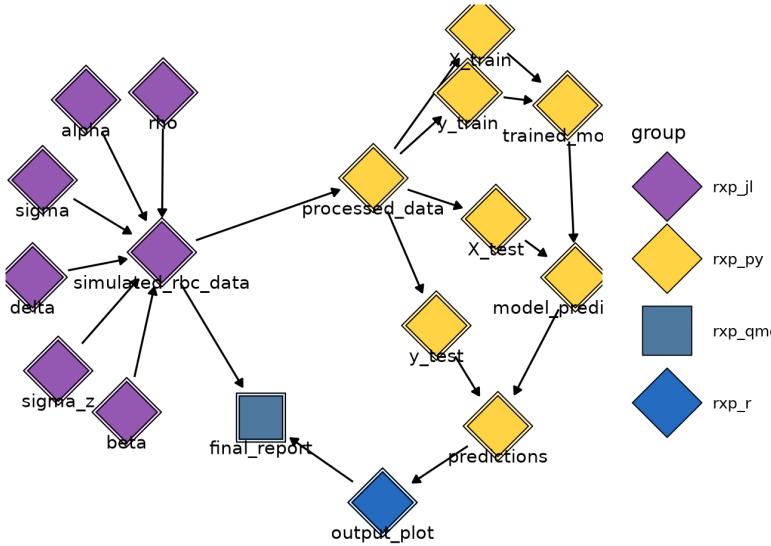


Figure 7.1: Derivations are coloured by programming language.

Finally, you can also use `rxp_visnetwork()` (which requires the

`{visNetwork}` package). This opens an interactive JavaScript visualization in your browser, allowing you to zoom into and explore different parts of your pipeline.

7.5.4 Caching, Incremental Builds and Build Logs

One of the most powerful features of using Nix for pipelines is automatic caching. Because Nix tracks all inputs to each derivation, it knows exactly what needs to be rebuilt when something changes.

Try this:

1. Build your pipeline with `rxp_make()`
2. Change one step in your pipeline
3. Run `rxp_make()` again

Nix will detect that unchanged steps are already cached and instantly reuse them. It only rebuilds the steps affected by your change. Also, your pipeline is actually also a *child* of the environment. So if the environment changes (for example, you update the date argument in the `rix()` call), your entire pipeline will be rebuilt. While this might seem like overkill, this is actually the only way you can guarantee that the pipeline is reproducible and works. If the pipeline weren't rebuilt when updating the environment, and cached results were re-used, you'd never know if a non-backwards-compatible change in a function in some package would break your pipeline!

Every time you run `rxp_populate()`, a timestamped log is saved in the `_rixpress/` directory. This is like having a Git history for your pipeline's *outputs*. Suppose I update a derivation in my pipeline, and build the pipeline again. Calling `rxp_list_logs()` shows me this:

```
rxp_list_logs()
```

```
> rxp_list_logs()
filename
1
 ↪  build_log_20260131_104726_bpjnnq2nax40w45rzbl83ggp5y766dsv
2
 ↪  build_log_20260131_101941_y4gz22nl1aym5yfzp3wgj5glmzv3wx5v
modification_time size_kb
1 2026-01-31 10:47:26    3.06
2 2026-01-31 10:19:41    3.06
```

I also recommend committing the change, so the time of the rixpress log matches the time of the change:

```
git log
```

```
commit 043dd99cb56966af449796f636470462b5624d85
Author: Bruno Rodrigues <bruno@brodrigues.co>
Date:   Sat Jan 31 10:46:02 2026 +0100

  updated alpha from 0.3 to 0.4
```

I can now update the derivation again, and rebuild the pipeline. `rxp_list_logs()` shows the following:

```

 ↪  filename
1
 ↪  build_log_20260131_110120_v470crwiqx2k0g5bfvd4mqm73ssjdaa5
```

```
2
 ↪  build_log_20260131_104726_bpjnnq2nax40w45rzbl83ggp5y766ds
3
 ↪  build_log_20260131_101941_y4gz22nl1aym5yfzp3wgj5glmzv3wx5
    modification_time size_kb
1 2026-01-31 11:01:20    3.06
2 2026-01-31 10:47:26    3.06
3 2026-01-31 10:19:41    3.06
```

I can now compare results:

```
#| eval: false
rxp_list_logs()

old_result <- rxp_read("output_plot", which_log =
 ↪  "20260131_101941")

new_result <- rxp_read("output_plot", which_log =
 ↪  "20260131_110120")
```

The `which_log` argument is used to select the log you want. This is incredibly powerful for debugging and validation. You can go back in time to inspect any output from any previous pipeline run.

8 Advanced Pipeline Patterns

In the previous chapter, we learned the fundamentals of `{rixpress}`: defining derivations, building pipelines, and inspecting outputs. Now we'll explore advanced patterns for larger, more complex projects—including troubleshooting, sub-pipelines for organisation, polyglot workflows, and CI integration.

8.1 Troubleshooting and Common Issues

When working with `{rixpress}`, you will inevitably encounter build failures. Understanding how to read Nix error messages and debug your pipelines is essential for productive development. This section covers the most common issues and how to resolve them.

8.1.1 Understanding Build Output

The `verbose` argument in `rxp_populate()` and `rxp_make()` controls how much information you see during builds:

Level	Description	Use Case
0	Progress indicators only (default)	Normal development
1	Informational messages	**Recommended for debuggin**
2	Talkative	Detailed troubleshooting
3+	Debug/Vomit	Deep Nix internals

For most debugging, `verbose = 1` provides the right balance:

```
rxp_make(verbose = 1)
# or
rxp_populate(build = TRUE, verbose = 1)
```

8.1.2 Common Errors and Solutions

8.1.2.1 R Syntax Errors

Symptom: Build fails immediately with an R parse error before any Nix activity.

Example: Missing a comma between derivations in `gen-pipeline.R`:

```
list(
  rxp_jl(alpha, 0.5)  # Missing comma here!
  rxp_jl(beta, 1 / 1.01),
  ...
)
```

Error output:

```
Error: unexpected symbol in:
"  rxp_jl(alpha, 0.5)
  rxp_jl"
Execution halted
```

Solution: These are standard R syntax errors. Check the line number and the context shown in the error message. Common culprits:

- Missing commas between list elements
- Unmatched parentheses or brackets
- Missing closing quotes

> **IDE Syntax Checking**
>
> Use an IDE like RStudio or VS Code with R language support. These will highlight syntax errors before you even try to run the pipeline.

8.1.2.2 Missing Function Errors

Symptom: The build starts but a derivation fails with "could not find function".

Example: Typo in a function name:

```
rxp_r(
  name = output_plot,
  expr = plot_predictions_typo(predictions),  #
   ↪  Typo!
  user_functions = "functions/functions.R"
)
```

Error output (with `verbose = 1`):

```
building '/nix/store/...-output_plot.drv'...
Running phase: buildPhase
Error in plot_predictions_typo(predictions) :
  could not find function "plot_predictions_typo"
Execution halted

error: Cannot build
'/nix/store/...-output_plot.drv'.
       Reason: builder failed with exit code 1.
```

Solution: Check for:

1. **Typos** in function names
2. **Missing `user_functions`** argument—did you forget to specify the file containing your function?
3. **Wrong file path** in `user_functions`—is the path relative to your project root?
4. **Function not exported**—is the function defined in the file you specified?

8.1.2.3 Python Import Errors

Symptom: Python derivation fails with `ModuleNotFoundError`.

Example: Using a package not listed in your `default.nix`:

```
# In functions.py
def prepare_features(df):
    import nonexistent_module  # Not in environment!
    ...
```

Error output:

```
Running phase: buildPhase
Traceback (most recent call last):
  File "<string>", line 5, in <module>
    exec('processed_data =
    prepare_features(simulated_rbc_data)')
  File "<string>", line 1, in <module>
  File "<string>", line 22, in prepare_features
ModuleNotFoundError: No module named
'nonexistent_module'
```

Solution:

1. **Add the package** to `py_pkgs` in your `gen-env.R`:

```
rix(
  ...
  py_conf = list(
    py_version = "3.13",
    py_pkgs = c("pandas", "numpy",
     ↪  "your_missing_package")
  ),
  ...
)
```

2. **Rebuild the environment**: Run `source("gen-env.R")`, then exit and re-enter the shell with `nix-shell`.
3. **Check import syntax** using `adjust_import()` or `add_import()` if your Python code uses non-standard import patterns.

8.1.2.4 Decoder/Encoder Mismatches

Symptom: A derivation fails when trying to read data from an upstream step.

Example: Using `readRDS` to read an Arrow file:

```
rxp_r(
  name = output_plot,
  expr = plot_predictions(predictions),
  user_functions = "functions/functions.R",
  decoder = readRDS  # Wrong! predictions is an
   ↪  Arrow file
)
```

Error output:

```
Error in readRDS("/nix/store/.../predictions") :
  read error
Execution halted
```

Solution: Match the decoder to the upstream encoder:

Upstream Encoder	Correct Decoder
Default (pickle) for Python	`rxp_py2r()` or keep in Python
`save_arrow` / `feather.write_feather`	`arrow::read_feather`
`write_to_csv`	`read.csv`
Default (`saveRDS`) for R	`readRDS` (default)

```
rxp_r(
  name = output_plot,
  expr = plot_predictions(predictions),
  user_functions = "functions/functions.R",
```

```
  decoder = arrow::read_feather  # Correct!
)
```

8.1.3 Debugging Strategies

8.1.3.1 1. Enable Verbose Output

Always start debugging with `verbose = 1`:

```
rxp_make(verbose = 1)
```

This shows you exactly which derivation failed and the actual error message from R, Python, or Julia.

8.1.3.2 2. Inspect Build Results

Use `rxp_inspect()` to see the status of all derivations:

```
rxp_inspect()
```

This returns a data frame showing which derivations succeeded, failed, and their paths. You can filter for failures:

```
results <- rxp_inspect()
subset(results, !build_success)
```

8.1.3.3 3. Test Functions Interactively First

Before adding a function to your pipeline, test it in an interactive R or Python session within the Nix shell:

```
# In R
source("functions/functions.R")
test_data <-
 ↪  arrow::read_feather("path/to/test/data.arrow")
plot_predictions(test_data)
```

```
# In Python
from functions.functions import prepare_features
import pyarrow.feather as feather
df = feather.read_feather("path/to/test/data.arrow")
prepare_features(df)
```

8.1.3.4 4. Check Intermediate Outputs

Use `rxp_read()` to inspect the output of any successful derivation:

```
# What does the upstream data look like?
upstream_data <- rxp_read("predictions")
str(upstream_data)
```

This helps you understand what format the data is in and whether your decoder/encoder is appropriate.

8.1.3.5 5. View Full Nix Logs

For complex failures, Nix provides the full build log:

```
nix log /nix/store/...-derivation_name.drv
```

The path is shown in the error output. This gives you the complete stdout/stderr from the failed build.

8.1.4 Quick Reference: Error → Solution

Error Message	Likely Cause	Solution
`unexpected symbol`	R syntax error	Check for missing commas, brackets
`could not find function`	Missing/misspelled function	Verify `user_functions` path and function name
`ModuleNotFoundError`	Python package not installed	Add to `py_pkgs` in `gen-env.R`
`read error` in decoder	Wrong decoder for data format	Match decoder to upstream encoder

Error Message	Likely Cause	Solution
`object 'x' not found`	Upstream derivation failed	Fix the upstream error first
`1 dependency failed`	Cascading failure	Look for the root cause higher in the log
`No such file or directory`	Missing `additional_files`	Add required files to derivation

i The Cascade Effect

When one derivation fails, all downstream derivations that depend on it will also fail with "dependency failed". Always look for the **first** failure in the build output—that's your root cause. The error message `Reason: N dependencies failed` means you need to scroll up to find the actual error.

8.2 Organizing Large Projects with Sub-Pipelines

As pipelines grow, a single `gen-pipeline.R` file can become difficult to manage. Consider the typical data science project:

- Data extraction and cleaning (ETL)
- Feature engineering
- Model training
- Model evaluation

- Report generation

Putting all derivations in one file makes it hard to:

- Navigate the code
- Understand which derivations belong to which phase
- Collaborate across team members
- Reuse pipeline components in other projects

To solve this issue, you can define your project using sub-pipelines and join them into a master pipeline using `rxp_pipeline()`.

This allows you to organise derivations into named groups and even colour-code them for making it easy to visualise.

A project with sub-pipelines would look something like this:

```
my-project/
   default.nix           # Nix environment
(generated by rix)
   gen-env.R             # Script to generate
default.nix
   gen-pipeline.R        # MASTER SCRIPT: combines
all sub-pipelines
   pipelines/
       01_data_prep.R    # Data preparation
     sub-pipeline
       02_analysis.R     # Analysis sub-pipeline
       03_reporting.R    # Reporting sub-pipeline
```

Each sub-pipeline file returns a list of derivations:

```
# Data Preparation Sub-Pipeline
# pipelines/01_data_prep.R
library(rixpress)
```

```
list(
  rxp_r(
    name = raw_mtcars,
    expr = mtcars
  ),
  rxp_r(
    name = clean_mtcars,
    expr = dplyr::filter(raw_mtcars, am == 1)
  ),
  rxp_r(
    name = selected_mtcars,
    expr = dplyr::select(clean_mtcars, mpg, cyl, hp,
     ↪  wt)
  )
)
```

The `rxp_pipeline()` function takes:

- **name**: A descriptive name for this group of derivations
- **path**: Either a **file path** to an R script returning a list of derivations (recommended), or a list of derivation objects.
- **color**: Optional CSS color name or hex code for DAG visualisation

The second sub-pipeline:

```
# Analysis Sub-Pipeline
# pipelines/02_analysis.R
library(rixpress)

list(
  rxp_r(
    name = summary_stats,
```

```
    expr = summary(selected_mtcars)
  ),
  rxp_r(
    name = mpg_model,
    expr = lm(mpg ~ hp + wt, data = selected_mtcars)
  ),
  rxp_r(
    name = model_coefs,
    expr = coef(mpg_model),
    )
)
```

The master script becomes very clean, as `rxp_pipeline()` handles sourcing the files:

```
# gen-pipeline.R
library(rixpress)

# Create named pipelines with colours by pointing to
 ↪  the files
pipe_data_prep <- rxp_pipeline(
  name = "Data Preparation",
  path = "pipelines/01_data_prep.R",
  color = "#E69F00"
)

pipe_analysis <- rxp_pipeline(
  name = "Statistical Analysis",
  path = "pipelines/02_analysis.R",
  color = "#56B4E9"
)

# Build combined pipeline
```

```
rxp_populate(
  list(pipe_data_prep, pipe_analysis),
  project_path = ".",
  build = TRUE)
```

8.2.1 Visualising Sub-Pipelines

When sub-pipelines are defined, visualisation tools use pipeline colours:

1. **Interactive Network** (`rxp_visnetwork()`) and **Static DAG** (`rxp_ggdag()`) both use a dual-encoding approach:
    - **Node fill (interior)**: Derivation type colour (R = blue, Python = yellow, etc.)
    - **Node border (thick stroke)**: Pipeline group colour This allows you to see both what *type* of computation each node is and which *pipeline* it belongs to.

Subpipelines are coloured.

2. **Trace**: `rxp_trace()` output in the console is coloured by pipeline (using the `cli` package).

If your terminal supports it, derivation names are coloured according to the chosen sub-pipeline colour.

It is also possible to not highlight sub-pipelines by using the `colour_by` argument (allows you to switch between colouring the pipeline, or by derivation type):

```
rxp_ggdag(colour_by = "pipeline")

rxp_ggdag(colour_by = "type")
```

8.3 Polyglot Pipelines

One of `{rixpress}`'s strengths is seamlessly mixing languages. Here's a pipeline that reads data with Python's `polars`, processes it with R's `dplyr`, and renders a Quarto report.

> **Polyglot Development Is Now Cheap**
>
> Historically, using multiple languages in one project meant significant setup overhead: installing interpreters, managing conflicting dependencies, writing glue code. With Nix, that cost drops to near zero. You declare your R and Python dependencies in one file, and Nix handles the rest.
> LLMs lower the barrier further. Even if you are primarily an R programmer, you can ask an LLM to generate the Python code for a specific step, or vice versa. You don't need to master both languages; you just need to know enough to recognise when each shines. Use R for statistics, Bayesian modelling, and visualisation with `{ggplot2}`. Use Python for deep learning, web scraping, or leveraging a library that only exists in the Python ecosystem. With Nix handling environments and LLMs helping with syntax, the "cost" of crossing language boundaries becomes negligible.

Open `gen-env.R` and define the tools your pipeline needs:

```
library(rix)

rix(
  date = "2026-01-26",
  r_pkgs = c(
    "chronicler",
    "dplyr",
```

```
    "igraph",
    "quarto",
    "reticulate",
    "rix",
    "rixpress"
  ),
  py_conf = list(
    py_version = "3.14",
    py_pkgs = c(
      "biocframe",
      "numpy",
      "phart",
      "polars",
      "pyarrow",
      "rds2py",
      "ryxpress"
    )
  ),
  ide = "none",
  project_path = ".",
  overwrite = TRUE
)
```

The Python packages `biocframe`, `phart`, and `rds2py` are optional but highly recommended when working with `ryxpress`. `biocframe` enables direct transfer to `polars` or Bioconductor DataFrames, bypassing the need for `pandas`. `rds2py` (also from the BiocPy project) allows Python to read `.rds` files, R's native binary format. Together, these packages ensure seamless interoperability between your R and Python steps. `phart` is a nice little package that allows you to visualise pipelines from a Python interpreter prompt: not as sexy as R's `{ggdag}` but it gets the job done!

8.3.1 Transferring data between Python and R

The `rxp_py2r()` and `rxp_r2py()` functions use `{reticulate}` to convert objects between languages:

```
rxp_py2r(
  name = mtcars_r,
  expr = mtcars_py
)
```

However, `{reticulate}` conversions can sometimes be fragile or behave unexpectedly with complex objects. For robust production pipelines, I recommend using the `encoder` and `decoder` arguments to explicitly serialize data to intermediary formats like CSV, JSON, or Apache Arrow (Parquet/Feather).

You can define a custom encoder in Python and a decoder in R:

```
# Python step: serialize to JSON
rxp_py(
  name = mtcars_json,
  expr = "mtcars_pl.filter(polars.col('am') == 1)",
  user_functions = "functions.py",
  encoder = "serialize_to_json"
),

# R step: deserialize from JSON
rxp_r(
  name = mtcars_head,
  expr = my_head(mtcars_json),
  user_functions = "functions.R",
  decoder = "jsonlite::fromJSON"
)
```

The Python `serialize_to_json` function would be defined in `functions.py`:

```
#| eval: false
def serialize_to_json(pl_df, path):
    with open(path, 'w') as f:
        f.write(pl_df.write_json())
```

For larger datasets, Apache Arrow is ideal because it allows zero-copy reads and is supported natively by `polars`, `pandas`, and R's `{arrow}` package.

Here is how you would transfer data from Python to R:

```
rxp_py(
  name = processed_data,
  expr = "prepare_features(raw_data)",
  user_functions = "functions.py",
  encoder = "save_arrow"
),

rxp_r(
  name = analysis_results,
  expr = "run_analysis(processed_data)",
  user_functions = "functions.R",
  decoder = "arrow::read_feather"
)
```

The Python `save_arrow` encoder function:

```
#| eval: false
def save_arrow(df: pd.DataFrame, path: str):
```

```
    """Encoder function to save a pandas DataFrame
     ↪  to an Arrow file."""
    feather.write_feather(df, path)
```

Both `encoder` and `decoder` functions must accept the object as the first argument and the file path as the second argument. This pattern works for any serialization format supported by your languages of choice.

8.4 Advanced Topics

8.4.1 Complete Polyglot Pipeline Walkthrough

You can find the complete code source to this example over here[1].

For this polyglot pipeline, I use three programming languages: Julia to simulate synthetic data from a macroeconomic model, Python to train a machine learning model and R to visualize the results. Let me make it clear, though, that scientifically speaking, this is nonsensical: this is merely an example to showcase how to setup a complete end-to-end project with `{rixpress}`.

The model I use is the foundational Real Business Cycle (RBC) model. The theory and log-linearized equations used in the Julia script are taken directly from the excellent lecture slides by Bianca De Paoli.

[1] https://github.com/b-rodrigues/rixpress_demos/tree/master/rbc

> **Source Attribution**
>
> The economic theory and equations for the RBC model are based on the following source. This document serves as a guide for our implementation.
> **Source:** De Paoli, B. (2009). *Slides 1: The RBC Model, Analytical and Numerical solutions*. https://personal.lse.ac.uk/depaoli/RBC_slides1.pdf

First, we need the actual execution environment. Thanks to the `{rix}` package we can declaratively define a `default.nix` file. This file locks down the exact versions of R, Julia, Python, and all their respective packages, ensuring the pipeline runs identically today, tomorrow, or on any machine with Nix installed.

```
#| eval: false
#| code-summary: "Environment Definition with {rix}"
# This script defines the polyglot environment our
 ↪  pipeline will run in.
library(rix)

# Define the complete execution environment
rix(
  # Pin the environment to a specific date to ensure
   ↪  that all package
  # versions are resolved as they were on this day.
  date = "2025-10-14",

  # 1. R Packages
  # We need packages for plotting, data
   ↪  manipulation, and reading arrow files.
  r_pkgs = c(
    "ggplot2",
    "dplyr",
```

```
  "arrow",
  "rix",
  "rixpress"
),

# 2. Julia Configuration
# We specify the Julia version and the list of
 ↪ packages needed
# for our manual RBC model simulation.
jl_conf = list(
  jl_version = "lts",
  jl_pkgs = c(
    "Distributions", # For creating random shocks
    "DataFrames", # For structuring the output
    "Arrow", # For saving the data in a
     ↪ cross-language format
    "Random"
  )
),

# 3. Python Configuration
# We specify the Python version and the packages
 ↪ needed for the
# machine learning step.
py_conf = list(
  py_version = "3.13",
  py_pkgs = c(
    "pandas",
    "scikit-learn",
    "xgboost",
    "pyarrow"
  )
),
```

```
  # We set the IDE to 'none' for a minimal
   ↪ environment. You could change
  # this to "rstudio" if you prefer to work
   ↪ interactively in RStudio.
  ide = "none",

  # Define the project path and allow overwriting
   ↪ the default.nix file.
  project_path = ".",
  overwrite = TRUE
)
```

Assuming we are working on a system that has Nix installed, we can "drop" into a temporary Nix shell with R and `{rix}` available:

```
nix-shell --expr "$(curl -sl
 ↪ https://raw.githubusercontent.com/ropensci/rix/main/inst/ex
```

Once the shell is ready, start R (by simply typing `R`) and run `source("gen-env.R")` to generate the adequate `default.nix` file (or even easier, just `Rscript gen-env.R`). This is the environment that we can use to work interactively with the code while developing, and in which the pipeline will be executed.

We then need some dedicated scripts with our code. This Julia script contains a pure function that simulates the RBC model and returns a DataFrame. The code is a direct implementation of the state-space solution derived from the equations in the aforementioned lecture and is saved under `functions/functions.jl`. I don't show the scripts contents here, as it is quite long, so look for it in the link from before.

At the end of the script, I create a wrapper around `Arrow.write()` called `arrow_write()` to serialize the generated data frame into an Arrow file.

While developing, it is possible to start the Julia interpreter and test things and see if they work. Or you could even start by writing the first lines of `gen-pipeline.R` like so:

```
library(rixpress)

list(
  # STEP 0: Define RBC Model Parameters as
   ↪ Derivations
  # This makes the parameters an explicit part of
   ↪ the pipeline.
  # Changing a parameter will cause downstream steps
   ↪ to rebuild.
  rxp_jl(alpha, 0.3), # Capital's share of income
  rxp_jl(beta, 1 / 1.01), # Discount factor
  rxp_jl(delta, 0.025), # Depreciation rate
  rxp_jl(rho, 0.95), # Technology shock persistence
  rxp_jl(sigma, 1.0), # Risk aversion (log-utility)
  rxp_jl(sigma_z, 0.01), # Technology shock standard
   ↪ deviation

  # STEP 1: Julia - Simulate a Real Business Cycle
   ↪ (RBC) model.
  # This derivation runs our Julia script to
   ↪ generate the source data.
  rxp_jl(
    name = simulated_rbc_data,
    expr = "simulate_rbc_model(alpha, beta, delta,
     ↪ rho, sigma, sigma_z)",
    user_functions = "functions/functions.jl", # The
     ↪ file containing the function
```

```
    encoder = "arrow_write" # The function to use
     ↪  for saving the output
  )
) |>
  rxp_populate(
    project_path = ".",
    build = TRUE,
    verbose = 1
  )
```

Start an R session, and run `source("gen-pipeline.R")`. This will cause the model to be simulated. Now, to inspect the data, you can use `rxp_read()`: by default, `rxp_read()` will try to find the adequate function to read the data and present it to you. This works for common R and Python objects. But in this case, the generated data is an Arrow data file, so `rxp_read()`, not knowing how to read it, will simply return the path to the file in the Nix store. We can pass this file to the adequate reader.

```
rxp_read("simulated_rbc_data") |>
  arrow::read_feather() |>
  head()
```

Now that we are happy that our Julia code works, we can move on to the Python step, by writing a dedicated Python script, `functions/functions.py`.

This Python script defines a function to perform our machine learning task. It takes the DataFrame from the Julia step, trains an XGBoost model to predict next-quarter's output, and returns a new DataFrame containing the predictions. Of course, we could have simply used the RBC model itself for the predictions, but again, this is a toy example.

The Python script contains several little functions to make the code as modular as possible. I don't show the script here, as it is quite long, but just as for the Julia script, I write a wrapper around `feather.write_feather()` to serialize the data into an Arrow file and pass it finally to R. Just as before, it is possible to start a Python interpreter as defined in the `default.nix` file and use to test and debug, or to add derivations to `gen-pipeline.R`, rebuild the pipeline and check the outputs:

```
library(rixpress)

list(
  # ... the julia steps from before ...,

  # STEP 2.1: Python - Prepare features (lagging
   ↪  data)
  rxp_py(
    name = processed_data,
    expr = "prepare_features(simulated_rbc_data)",
    user_functions = "functions/functions.py",
    # Decode the Arrow file from Julia into a pandas
     ↪  DataFrame
    decoder = "feather.read_feather"
    # Note: No encoder needed here. {rixpress} will
     ↪  use pickle by default
    # to pass the DataFrame between Python steps.
  ),

  # STEP 2.2: Python - Split data into training and
   ↪  testing sets
  rxp_py(
    name = X_train,
    expr = "get_X_train(processed_data)",
    user_functions = "functions/functions.py"
```

```
  ),

  # ... more Python steps as needed ...
) |>
  rxp_populate(
    project_path = ".",
    build = TRUE,
    verbose = 1
  )
```

Arrow is used to pass data from Julia to Python, but then from Python to Python, pickle is used. Finally, we can continue by passing data further down to R. Once again, I recommend writing a dedicated script with your functions that you need and save it in `functions/functions.R`. The `gen-pipeline.R` script would look something like this:

```
library(rixpress)

list(
  # ... the julia steps from before ...,

  # ... Python steps ...,

  # STEP 2.5: Python - Format final results for R
  rxp_py(
    name = predictions,
    expr = "format_results(y_test,
     ↪  model_predictions)",
    user_functions = "functions/functions.py",
    # We need an encoder here to save the final
     ↪  DataFrame as an Arrow file
```

```
    # so the R step can read it.
    encoder = "save_arrow"
  ),

  # STEP 3: R - Visualize the predictions from the
   ↪ Python model.
  # This final derivation depends on the output of
   ↪ the Python step.
  rxp_r(
    name = output_plot,
    expr = plot_predictions(predictions), # The
     ↪ function to call from functions.R
    user_functions = "functions/functions.R",
    # Specify how to load the upstream data (from
     ↪ Python) into R.
    decoder = arrow::read_feather
  ),

  # STEP 4: Quarto - Compile the final report.
  rxp_qmd(
    name = final_report,
    additional_files = "_rixpress",
    qmd_file = "readme.qmd"
  )
) |>
  rxp_populate(
    project_path = ".",
    build = TRUE,
    verbose = 1
  )
```

To view the plot in an interactive session, you can use `rxp_read()` again:

```
rxp_read("output_plot")
```

To run the pipeline, start the environment by simply typing `nix-shell` in the terminal, which will use the environment as defined in the `default.nix` and run `source("gen-pipeline.R")`. If there are issues, setting the `verbose` argument of `rxp_populate()` to `1` is helpful to see the full error logs.

8.4.2 ryxpress: The Python Interface

If you prefer working in Python, `ryxpress` provides the same functionality. You still define your pipeline in R (since that's where `{rixpress}` runs), but you can build and inspect artifacts from Python.

To set up an environment with `ryxpress`:

```
rix(
  date = "2025-10-14",
  r_pkgs = c("rixpress"),
  py_conf = list(
    py_version = "3.13",
    py_pkgs = c("ryxpress", "rds2py", "biocframe",
    ↪  "pandas")
  ),
  ide = "none",
  project_path = ".",
  overwrite = TRUE
)
```

Then from Python:

```
from ryxpress import rxp_make, rxp_inspect, rxp_load

# Build the pipeline
rxp_make()

# Inspect artifacts
rxp_inspect()

# Load an artifact
rxp_load("mtcars_head")
```

`ryxpress` handles the conversion automatically:

- Tries `pickle.load` first
- Falls back to `rds2py` for R objects
- Returns file paths for complex outputs

8.4.3 Using rixpress in CI

Running `{rixpress}` pipelines in Continuous Integration is straightforward. The `rxp_ga()` function generates a complete GitHub Actions workflow file that handles everything automatically.

To set up CI for your pipeline, simply run:

```
rxp_ga()
```

This creates a `.github/workflows/` directory with a workflow file that runs your pipeline on every push or pull request.

The generated workflow performs these steps:

1. **Restore cached artifacts**: If previous run artifacts exist, they are restored using `rxp_import_artifacts()` to avoid recomputing unchanged derivations.
2. **Install Nix**: The deterministic Nix installer is used to set up Nix on the runner.
3. **Build the environment**: Your `default.nix` environment is built, with the `rstats-on-nix` cache enabled to speed up R package builds.
4. **Generate the pipeline**: Your `gen-pipeline.R` script is executed.
5. **Visualise the DAG**: Since there's no graphical interface in CI, the workflow uses `stacked-dag` (a Haskell tool) to print a textual representation of the pipeline.
6. **Build the pipeline**: `rxp_make()` executes, building any derivations that aren't already cached.
7. **Archive artifacts**: Build outputs are exported using `rxp_export_artifacts()` and pushed to a `rixpress-runs` branch for reuse in subsequent runs.

In an interactive session, you would use `rxp_ggdag()` or `rxp_visnetwork()` to visualise your pipeline. In CI, the workflow uses `stacked-dag` to produce a text-based DAG:

```
- name: Check DAG if dag.dot exists and show it if
 ↪  yes
  run: |
    if [ -f _rixpress/dag.dot ]; then
      nix-shell --quiet -p
 ↪  haskellPackages.stacked-dag \
        --run "stacked-dag dot _rixpress/dag.dot"
```

```
    else
      echo "dag.dot not found"
    fi
```

When `rxp_ga()` is called, it automatically invokes `rxp_dag_for_ci()` to generate the required `.dot` file. The output looks like this:

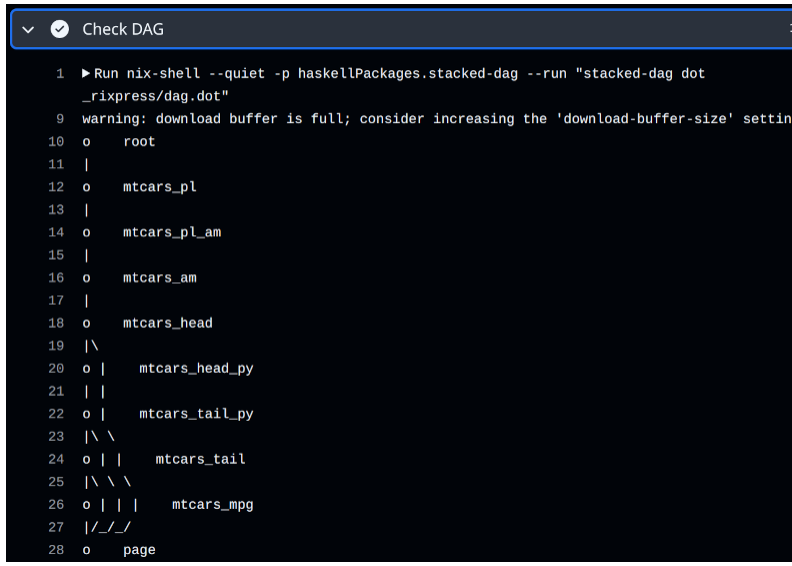


Figure 8.1: Text representation of the DAG in CI.

8.4.3.1 Manual Artifact Management

For custom CI setups or sharing artifacts between machines, you can use the export and import functions directly:

```
# Export all build outputs to a .nar archive
rxp_export_artifacts()
```

```
# On another machine or CI run, import before
 ↪  building
rxp_import_artifacts()
```

By default, artifacts are saved to `_rixpress/pipeline_outputs.nar`. You can specify a different path:

```
rxp_export_artifacts(archive_file =
 ↪  "my_outputs.nar")
rxp_import_artifacts(archive_file =
 ↪  "my_outputs.nar")
```

This approach dramatically speeds up CI by leveraging Nix's content-addressed storage: if an artifact's hash matches what's already in the Nix store, it's instantly available without rebuilding.

8.4.4 More Examples

The rixpress_demos repository includes:

- **jl_example**: Using Julia in pipelines
- **r_qs**: Using `{qs}` for faster serialization
- **python_r_typst**: Compiling to Typst documents
- **r_multi_envs**: Different Nix environments for different derivations
- **yanai_lercher_2020**: Reproducing a published paper's analysis

8.5 Caveats: Consequences of Hermetic Builds

Nix derivations are *hermetic*: they run in an isolated sandbox with no network access and a minimal, reproducible filesystem. This design ensures reproducibility but has practical implications:

No network access during builds. If your analysis requires downloading data from an API or remote source, this must happen *before* the pipeline runs. Use `rxp_r_file()` or `rxp_py_file()` to read pre-downloaded data, or create a separate script outside the pipeline to fetch data into a `data/` directory.

Environment variables must be explicit. Tools that rely on environment variables (like CmdStan for Bayesian modelling) need these set via the `env_var` argument. For example, CmdStanR requires pointing to the CmdStan installation:

```
rxp_r(
  name = model_fit,
  expr = fit_stan_model(data),
  user_functions = "functions.R",
  env_var = c(CMDSTAN =
   ↪  "${defaultPkgs.cmdstan}/opt/cmdstan")
)
```

See `vignette("cmdstanr", package = "rixpress")` for a complete Bayesian workflow example.

Paths are relative to the build sandbox. Files referenced in your code must be explicitly included via `additional_files`

or `user_functions`. The build environment doesn't see your working directory—only what you declare.

8.6 Summary

`{rixpress}` unifies environment management and workflow orchestration:

- **Define** pipelines with `rxp_*()` functions in familiar R syntax
- **Mix** languages freely: R, Python, Julia, Quarto
- **Build** with Nix for deterministic, cached execution
- **Inspect** outputs with `rxp_read()`, `rxp_load()`, `rxp_copy()`
- **Debug** with timestamped build logs
- **Share** reproducible pipelines via Git

The Python port `ryxpress` provides the same experience for Python-first workflows.

By embracing structured, plain-text pipelines over notebooks for production work, your analysis becomes more reliable, more scalable, and fundamentally more reproducible.

9 Robust Pipelines with Monads

In the previous chapters, you learned functional programming fundamentals and how to build reproducible pipelines with `{rixpress}`. Now we'll add another layer of robustness: **monads**.

Monads might sound abstract, but they solve concrete problems involving **Logging** (tracing what each step does without cluttering your functions), **Error handling** (letting failures propagate gracefully instead of crashing), and **Composition** (keeping functions composable even when they need to do "extra" work).

By the end of this chapter, you'll know how to integrate `{chronicler}` and `cronista` into your pipelines for more robust, observable data workflows.

9.1 Why Do We Need Logging?

When you build data pipelines, things inevitably go wrong. A dataset might have unexpected missing values. A computation might produce NaN. A join might drop more rows than expected. When these issues occur, the first question is always: **where did it happen?**

Consider a simple pipeline:

```
result <- raw_data |>
  clean_data() |>
  filter_outliers() |>
  compute_stats() |>
  generate_report()
```

If `result` is wrong or the pipeline crashes, you need to figure out which step failed. A way to do this is to add print statements:

```
result <- raw_data |>
  clean_data() |>
  { print("clean_data done"); . }() |>
  filter_outliers() |>
  { print("filter_outliers done"); . }() |>
  compute_stats() |>
  { print("compute_stats done"); . }() |>
  generate_report()
```

This works, but it's ugly and mixes logging logic with business logic. You have to remember to remove these print statements before sharing your code. Worse, if you want to capture *what* went into each step or *how long* it took, you end up with even more boilerplate.

> **Nix Error Messages**
>
> When using `{rixpress}`, Nix does report errors during builds, but the messages can be hard to read—especially if the error happens deep in your pipeline or involves R/Python stack traces mixed with Nix output. Having

> explicit logging in your functions makes debugging much easier because you get clear, structured information about what happened at each step.

What if your functions could automatically log themselves without any changes to their core logic?

9.2 The Problem: Decorated Functions Don't Compose

Suppose you want your functions to provide logs. You might rewrite `sqrt()` like this:

```
my_sqrt <- function(x, log = "") {
  list(
    result = sqrt(x),
    log = c(log, paste0("Running sqrt with input ",
     ↪  x))
  )
}

my_log <- function(x, log = "") {
  list(
    result = log(x),
    log = c(log, paste0("Running log with input ",
     ↪  x))
  )
}
```

These functions now return lists with both the result and a log. But there's a problem: they don't compose:

```
# This works:
10 |> sqrt() |> log()

# This fails:
10 |> my_sqrt() |> my_log()
# Error: non-numeric argument to mathematical
 ↪  function
```

`my_log()` expects a number, but `my_sqrt()` returns a list. We've broken composition.

9.3 The Solution: Function Factories and Bind

A monad provides two things:

1. **A function factory** that decorates functions so they can provide additional output without rewriting their core logic
2. **A `bind()` function** that makes these decorated functions compose

Here's a simple function factory for logging:

```
log_it <- function(.f) {
  fstring <- deparse(substitute(.f))

  function(..., .log = NULL) {
    list(
      result = .f(...),
      log = c(.log, paste0("Running ", fstring, "
       ↪  with argument ", ...))
```

```
    )
  }
}

# Create decorated functions
l_sqrt <- log_it(sqrt)
l_log <- log_it(log)

l_sqrt(10)
#> $result
#> [1] 3.162278
#> $log
#> [1] "Running sqrt with argument 10"
```

Now we need `bind()` to make them compose:

```
bind <- function(.l, .f) {
  .f(.l$result, .log = .l$log)
}

# Now they compose!
10 |>
  l_sqrt() |>
  bind(l_log)

#> $result
#> [1] 1.151293
#> $log
#> [1] "Running sqrt with argument 10"
#> [2] "Running log with argument 3.16227766016838"
```

This pattern of a function factory plus `bind()` is the essence of a monad.

9.4 A Deeper Look: The Formal Definition

While our function factory approach is practical for R programmers, it's useful to understand how this relates to the formal definition from Haskell, where monads originate:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b  -- This is
 ↪  bind()
  (>>)   :: m a ->  m b       -> m b  -- We won't
 ↪  need this
  return ::   a               -> m a  -- This wraps
 ↪  a value
```

What this means is that a Monad has three operations:

- `>>=` (called "bind"): exactly what we implemented as our `bind()` function
- `>>` (called "then"): not essential for understanding, so we'll skip it
- `return`: a function that wraps a plain value into a monadic value

Don't be confused by the name `return`—this has nothing to do with R's `return()` for exiting functions. In monad terminology, `return` (also called `unit`) takes a value `a` and wraps it into a monadic value `m a`.

While we didn't explicitly implement `return`/`unit`, our function factory accomplishes the same goal. We can easily derive `unit` from `log_it()`:

```
unit <- log_it(identity)

unit(10)
#> $result
#> [1] 10
#> $log
#> [1] "Running identity with argument 10"
```

So `return`/`unit` is just the `identity()` function passed through our factory. In a sense, the function factory is even more fundamental than `return`/`unit`.

9.5 Alternative Perspective: fmap, flatten, and flatmap

You may sometimes read that monads are objects with a `flatmap()` method. This is another valid perspective. To understand it, we need two helper functions: `fmap()` and `flatten()`.

fmap() applies an *undecorated* function to a monadic value:

```
fmap <- function(m, f, ...) {
  fstring <- deparse(substitute(f))

  list(
    result = f(m$result, ...),
    log = c(m$log, paste0("fmapping ", fstring, "
     ↪  with argument ", m$result))
  )
}
```

```
# Create a monadic value using unit()
m <- unit(10)

# Apply an undecorated function
fmap(m, log)
#> $result
#> [1] 2.302585
#> $log
#> [1] "Running identity with argument 10"
#> [2] "fmapping log with argument 10"
```

But what happens if we use `fmap()` with a *decorated* function?

```
fmap(m, l_log)
#> $result
#> $result$result
#> [1] 2.302585
#> $result$log
#> [1] "Running log with argument 10"
#>
#> $log
#> [1] "Running identity with argument 10"
#> [2] "fmapping l_log with argument 10"
```

We get a nested structure—a monadic value containing another monadic value. This is where **flatten()** (sometimes called `join()`) comes in:

```
flatten <- function(m) {
  list(
    result = m$result$result,
```

```
    log = c(m$log, m$result$log)
  )
}

flatten(fmap(m, l_log))
#> $result
#> [1] 2.302585
#> $log
#> [1] "Running identity with argument 10"
#> [2] "fmapping l_log with argument 10"
#> [3] "Running log with argument 10"
```

Now here's the key insight: **flatmap is the composition of flatten and fmap**:

```
# Define a composition operator
`%.%` <- \(f, g)(function(...)(f(g(...))))

# Compose flatten and fmap
# "flatten after fmap"
flatmap <- flatten %.% fmap

# Now we can use flatmap instead of bind:
10 |>
  l_sqrt() |>
  flatmap(l_log)
```

This gives the same result as using `bind()`. I prefer introducing monads via `bind()` because it comes naturally as a solution to the composition problem. But in some contexts, thinking in terms of `fmap()` and `flatten()` is more intuitive, so it's good to know both approaches.

9.6 Lists Are Monads Too

Here's a surprising fact: **lists are monads**. We already have all the necessary ingredients in base R and `{purrr}`:

- `as.list()` is `return`/`unit`
- `purrr::map()` is `fmap()`
- `purrr::flatten()` is `flatten()`

This means we can derive `flatmap` for lists:

```
# Compose flatten and map to get flatmap for lists
flatmap_list <- purrr::compose(purrr::flatten,
 ↪  purrr::map)

# Functions that return lists don't compose
 ↪  naturally...
list_sqrt <- \(x)(as.list(sqrt(x)))
list_log <- \(x)(as.list(log(x)))

# But with flatmap_list, they do!
10 |>
  list_sqrt() |>
  flatmap_list(list_log)
#> [[1]]
#> [1] 1.151293
```

This example illustrates that monads are not some exotic concept—they're a pattern you've likely used without knowing it. Every time you use `purrr` to chain operations over lists, you're doing monadic programming.

9.7 The chronicler Package

The `{chronicler}` package implements this pattern properly for R. It provides:

- `record()`: A function factory that decorates functions
- `bind_record()`: The bind operation
- Automatic logging of all operations

```
library(chronicler)

# Decorate functions
r_sqrt <- record(sqrt)
r_exp <- record(exp)
r_mean <- record(mean)

# Compose them
result <- 1:10 |>
  r_sqrt() |>
  bind_record(r_exp) |>
  bind_record(r_mean)

# View the result
result$value
#> [1] 5.187899

# View the log
read_log(result)
#> [1] "Complete log:"
#> [2] "  sqrt ran successfully"
#> [3] "  exp ran successfully"
#> [4] "  mean ran successfully"
```

Getting LLM assistance with `{chronicler}`

If the functional programming patterns in `{chronicler}` feel unfamiliar, remember that you can use `pkgctx` to generate LLM-ready context. The `{chronicler}` repository includes a `.pkgctx.yaml` file you can feed to your LLM. You can also generate your own:

```
nix run github:b-rodrigues/pkgctx -- r
 ↪  github:b-rodrigues/chronicler >
 ↪  chronicler.pkgctx.yaml
```

With this context, your LLM can help you refactor your existing functions to be monadic using `record()` and `bind_record()`.

9.8 The Maybe Monad: Handling Errors

Another common monad is **Maybe**, which handles computations that might fail. Instead of crashing, functions return either:

- `Just(value)` if successful
- `Nothing` if something went wrong

The `{chronicler}` package uses this under the hood:

```
r_sqrt <- record(sqrt)

# This works
r_sqrt(16)
#>   Value: Just
```

```
#> [1] 4

# This fails gracefully
r_sqrt("not a number")
#>   Value: Nothing
```

When `Nothing` is passed to a decorated function, it immediately returns `Nothing`, the error propagates through the pipeline without crashing.

9.9 Monads in Python: cronista

The same concepts exist in Python. The `cronista` package (Python equivalent of `{chronicler}`) provides the same `record()` decorator and logging capabilities:

```
from cronista import record, read_log, unveil
import math

# Decorate functions
r_sqrt = record(math.sqrt)
r_exp = record(math.exp)

@record()
def compute_mean(values):
    return sum(values) / len(values)

# Compose them
result = (
    r_sqrt(16)
      .bind_record(r_exp)
```

```
)

# View the result
print(unveil(result, "value"))  # 54.59815...

# View the log
print(read_log(result, style="pretty"))
#> OK `sqrt` at 2025-01-31 12:00:00 (0.001s)
#> OK `exp` at 2025-01-31 12:00:00 (0.001s)
#> Total: 0.002 secs
```

When a computation fails, cronista captures the error as `Nothing`:

```
@record()
def divide_by_zero():
    return 1 / 0

result = divide_by_zero()
print(result.is_ok())  # False (this is a Nothing)
print(read_log(result, style="errors-only"))
#> NOK `divide_by_zero` - ZeroDivisionError:
 ↪  division by zero
```

> **Getting LLM assistance with cronista**
>
> Just like with `{chronicler}`, you can use `pkgctx` to help your LLM understand `cronista`:
>
> ```
> nix run github:b-rodrigues/pkgctx -- python
>  ↪  github:b-rodrigues/cronista >
>  ↪  cronista.pkgctx.yaml
> ```

> This is particularly useful for learning how to chain monadic operations in Python correctly.

The key operations mirror chronicler:

- `record(fn)`: Decorate a function to return a Chronicle
- `.bind_record(rfn)`: Chain with another recorded function
- `read_log(c)`: View the operation log

9.10 Building Robust Pipelines

The real power of monads becomes apparent when you combine them with pipeline orchestration. Consider a typical data pipeline:

```
# Standard pipeline - fragile
raw_data |>
  basic_cleaning() |>
  recodings() |>
  filter_arrivals() |>
  make_monthly() |>
  make_plot()
```

If `recodings()` fails halfway through, the entire pipeline crashes. You get an error message, but no information about what succeeded before the failure.

Now imagine wrapping each function with `record()`:

```
library(chronicler)

# Robust pipeline with logging
r_basic_cleaning <- record(basic_cleaning)
r_recodings <- record(recodings)
r_filter_arrivals <- record(filter_arrivals)
r_make_monthly <- record(make_monthly)
r_make_plot <- record(make_plot)

result <- raw_data |>
  r_basic_cleaning() |>
  bind_record(r_recodings) |>
  bind_record(r_filter_arrivals) |>
  bind_record(r_make_monthly) |>
  bind_record(r_make_plot)

# Now you get:
# - The result (or Nothing if any step failed)
# - A complete log of which steps ran
# - Exactly where and why it failed
read_log(result)
```

This pattern transforms a fragile script into a robust, observable pipeline.

9.11 Integrating with rixpress

You can combine the power of `{chronicler}` with `{rixpress}` for even more robust pipelines. The key insight is that your `user_functions` can use `record()` internally:

```
# functions.R
library(chronicler)

# Create recorded versions of your functions
r_basic_cleaning <- record(function(data) {
  data |>
    select(contains("TIME"), contains("20")) |>
    pivot_longer(cols = contains("20"),
                 names_to = "date",
                 values_to = "passengers")
})

r_recodings <- record(function(data) {
  data |>
    mutate(tra_meas = fct_recode(tra_meas, ...)) |>
    mutate(passengers = as.numeric(passengers))
})

# Export a pipeline function that uses bind_record
process_aviation_data <- function(raw_data) {
  raw_data |>
    r_basic_cleaning() |>
    bind_record(r_recodings)
}
```

Then in your `{rixpress}` pipeline:

```
library(rixpress)

list(
  rxp_r_file(
    name = avia_raw,
    path = "data/avia.tsv",
```

```
    read_function = readr::read_tsv
  ),

  rxp_r(
    name = processed_data,
    expr = process_aviation_data(avia_raw),
    user_functions = "functions.R"
  ),

  # ... more steps
) |>
  rxp_populate()
```

The {rixpress} derivation caches the result while {chronicler} provides logging and error handling within the step.

9.12 Python Integration with cronista

The same pattern works in Python pipelines using cronista:

```
# functions.py
from cronista import record

# Decorate your functions
r_basic_cleaning = record(lambda df:
 ↪  df.dropna().reset_index(drop=True))
r_recodings = record(lambda df: df.assign(
    category=df["category"].map(category_mapping)
))
r_filter_arrivals = record(lambda df:
 ↪  df[df["tra_meas"] == "Arrivals"])
```

```
def process_data(raw_df):
    """Chain operations with bind_record for logging
     ↪  and error handling."""
    return (
        r_basic_cleaning(raw_df)
          .bind_record(r_recodings)
          .bind_record(r_filter_arrivals)
    )
```

Then in your {rixpress} pipeline:

```
rxp_py(
  name = processed_data,
  expr = "process_data(raw_data)",
  user_functions = "functions.py"
)
```

9.13 The Monadic Laws

For the mathematically curious, monads must satisfy three laws. While you don't need to know these to *use* monads effectively, understanding them helps you verify that an implementation is correct.

9.13.1 First Law: Left Identity

Passing a value to `return`/`unit` and then binding it to a function `f` is the same as applying `f` directly to the value:

```
library(chronicler)

a <- as_chronicle(10)  # as_chronicle is
 ↪  chronicler's "return/unit"
r_sqrt <- record(sqrt)

# These should give the same result:
bind_record(a, r_sqrt)$value
#> [1] 3.162278

r_sqrt(10)$value
#> [1] 3.162278
```

9.13.2 Second Law: Right Identity

Binding a monadic value to `return`/`unit` returns the original value unchanged:

```
a <- as_chronicle(10)

# Binding to as_chronicle should return the same
 ↪  value:
bind_record(a, as_chronicle)$value
#> [1] 10

a$value
#> [1] 10
```

9.13.3 Third Law: Associativity

The order of binding operations doesn't matter—applying functions successively or composing them first gives the same result:

```
a <- as_chronicle(10)

r_sqrt <- record(sqrt)
r_exp <- record(exp)

# Method 1: Chain binds sequentially
result1 <- bind_record(a, r_sqrt) |>
  bind_record(r_exp)

# Method 2: Compose binds, then apply
composed_fn <- \(x) bind_record(x, r_sqrt) |>
 ↪  bind_record(r_exp)
result2 <- composed_fn(a)

# These should be equal:
result1$value
#> [1] 6.494743

result2$value
#> [1] 6.494743
```

> **A Note on Logs**
>
> In `{chronicler}`, the *logs* between these equivalent expressions may differ slightly (since they record different paths through the code), but the *values* will always be identical. This is the important property for computational correctness.

These laws ensure that monadic code behaves predictably—you can refactor bind chains without changing semantics, and you can trust that composition works as expected.

9.14 Detecting Silent Failures in Pipelines

Here's the catch: when you use chronicler or cronista functions in a rixpress pipeline, **Nix builds never fail**. Even when a computation produces `Nothing`, it's still a valid object that gets serialized successfully.

Consider this pipeline:

```
library(rixpress)

list(
  rxp_r(
    name = result,
    expr = r_sqrt(-1),  # This produces Nothing, not
     ↪  an error!
    user_functions = "functions.R"
  )
) |> rxp_populate(build = FALSE)
```

When you run `rxp_make()`, the build succeeds! But `result` contains `Nothing`, meaning the computation actually failed. Without checking, you might think your pipeline worked perfectly.

9.14.1 Automatic Chronicle Checking

When `{chronicler}` is available, `rxp_make()` automatically checks pipeline outputs for `Nothing` values after a successful build:

```
rxp_make()
#> Build successful! Run `rxp_inspect()` for a
 ↪  summary.
#> ...
#>
#> This pipeline uses {chronicler}. Here is a
 ↪  summary of chronicle results:
#> Chronicle status:
#>   filtered_mtcars (chronicle: OK)
#>   mtcars_mpg (chronicle: OK)
#>   mean_mpg (chronicle: OK)
#>   sqrt_of_negative (chronicle: NOTHING)
#>     Failed: sqrt
#>     Message: NaNs produced
#>   downstream_of_nothing (chronicle: NOTHING)
#>
#> Summary: 3 success, 0 with warnings, 2 nothing
#> Warning: 2 derivation(s) contain Nothing values!
```

You can also check manually at any time:

```
rxp_check_chronicles()
```

Chronicles can be in one of three states:

Symbol	State	Meaning
	Success	`Just` value, no warnings or errors
	Warning	`Just` value, but warnings were captured
	Nothing	Failed computation, errors captured

9.14.2 Python: cronista + ryxpress

The same pattern works in Python with `cronista` (the Python equivalent of chronicler) and `ryxpress`:

```
# gen-pipeline.R using Python functions
list(
  rxp_py(
    name = sqrt_result,
    expr = "r_sqrt(16)",            # Success:
     ↪  Just(4.0)
    user_functions = "functions.py"
  ),

  rxp_py(
    name = div_by_zero,
    expr = "divide_by_zero()",      # Failure:
     ↪  Nothing
    user_functions = "functions.py"
  )
) |> rxp_populate(build = FALSE)
```

Where `functions.py` uses cronista's `record()`:

```
# functions.py
import math
from cronista import record

r_sqrt = record(math.sqrt)

@record()
def divide_by_zero():
    return 1 / 0  # This error is captured, not
     ↪ raised
```

After building, use ryxpress to check for Nothing values:

```
from ryxpress import rxp_check_chronicles

rxp_check_chronicles()
#> Chronicle status:
#>   sqrt_result (chronicle: OK)
#>   div_by_zero (chronicle: NOTHING)
#>     Failed: divide_by_zero
#>     Message: ZeroDivisionError: division by zero
```

Reading a chronicle with Nothing automatically warns you:

```
from ryxpress import rxp_read

result = rxp_read("div_by_zero")
#> UserWarning: Derivation 'div_by_zero' contains a
 ↪  chronicle with Nothing value!
#> Use cronista.read_log() on this object for
 ↪  details.
```

Getting LLM assistance with cronista

Just like with the R packages, you can generate context for your LLM:

```
nix run github:b-rodrigues/pkgctx -- python
 ↪  github:b-rodrigues/cronista >
 ↪  cronista.pkgctx.yaml
```

This helps your LLM understand how to use `record()`, `bind_record()`, and `read_log()` in Python.

9.15 Summary

Monads add a layer of robustness to your pipelines. In this chapter, we covered:

Core Concepts:

- A monad provides a **function factory** (to decorate functions) and a **`bind()`** function (to compose decorated functions)
- The relationship between `bind()`, `fmap()`, `flatten()`, and `flatmap()`
- The formal Haskell definition with `>>=`, `>>`, and `return`
- Lists as a familiar example of monads (via `purrr::map` and `purrr::flatten`)
- The three **monadic laws** that ensure predictable behaviour

Practical Tools:

- **chronicler** (`record()`, `bind_record()`): Logging and error handling for R

- **cronista** (`record()`, `bind_record()`): The Python equivalent with full logs
- **Just/Nothing**: Graceful failure propagation without crashes
- **Integration with rixpress**: Monads within pipeline steps for observability

Combined with `{rixpress}`:

- Nix provides **hermetic, cached execution**
- Monads provide **logging and error handling**
- `rxp_check_chronicles()` **detects silent failures** in your pipeline
- Together they give you **robust, reproducible, observable pipelines**

This completes our code quality toolkit: environments (Nix), functional code (FP), testing, pipelines (rixpress), and robustness (monads). In the next chapter, we'll learn how to distribute our work with Docker, followed by packaging and continuous integration.

10 Containerisation With Docker

10.1 Introduction

Up until now, we've built reproducible environments with Nix, written testable code with FP principles, orchestrated pipelines with `{rixpress}`, and added robustness with monads. Now we turn to **distribution**: how do we share our work with others who may not have Nix installed?

Nix gives us fine-grained control over every package and dependency in our project, ensuring bit-for-bit reproducibility. However, when it comes to distributing a *data product*, another technology, Docker, is incredibly popular.

A common misconception is that Nix and Docker are alternatives, that you must choose one or the other. This is wrong. They are conceptually different tools that solve different problems. Nix is a **package manager and build system**: it answers "what software do I need and how do I build it reproducibly?" Docker is a **containerisation platform**: it answers "how do I package and run an application in isolation?" Understanding this distinction is key to using them together effectively.

While Nix manages dependencies for an application that runs on a host operating system, Docker takes a different approach: it

packages an application *along with* a lightweight operating system and all its dependencies into a single, portable unit called a **container**. This container can then run on any machine that has Docker installed, regardless of its underlying OS. Being familiar with both tools makes you a more versatile data scientist: you can use Nix for precise, reproducible development environments and Docker for distributing your work to colleagues, servers, or CI pipelines that may not have Nix installed.

10.1.1 Spatial vs Temporal Reproducibility

To understand when and why to use Docker, it helps to distinguish between two types of reproducibility:

- **Spatial reproducibility**: The ability to execute an analysis identically across different machines *right now*. Docker excels here: a container runs the same way on your laptop, a colleague's workstation, and a cloud server.
- **Temporal reproducibility**: The ability to execute an analysis identically *over time*. Docker is weaker here. The imperative commands in a `Dockerfile` (like `apt-get update`) are non-deterministic. Rebuilding the same file at different times can yield different images.

This distinction is crucial. Docker's true reproducibility promise is that a specific, pre-built image will always launch an identical container. It does *not* promise that building the same `Dockerfile` twice will yield an identical image.

As empirical research has shown, achieving deterministic builds requires systems designed for this purpose, like Nix's functional model, where identical inputs always produce identical outputs.

Studies rebuilding historical Nix packages found bit-for-bit reproducibility rates exceeding 90%.

10.1.2 The Best of Both Worlds

This is why we combine Nix and Docker rather than choosing one or the other:

- **Nix** provides strong temporal reproducibility: the guarantee that an environment built today will be identical when rebuilt years later.
- **Docker** provides strong spatial reproducibility and universal distribution: the guarantee that a container runs the same everywhere Docker runs.

The pattern we'll use is simple: use Nix *inside* a Docker container. Start with a minimal base image that has Nix installed. Then, use Nix to declaratively build the precise environment within the image. Docker becomes a portable runtime for this Nix-managed environment, excellent for deployment to systems where Nix isn't installed.

10.1.3 Interactive Development vs Distribution

This approach also clarifies *when* to use each tool:

- **Use Nix directly** for interactive development. It integrates seamlessly with your IDE and filesystem. No volume mounts, no port forwarding, no graphical application headaches.

- **Use Docker** for distributing finished data products. Package your `{rixpress}` pipeline into an image that anyone can run with a single command.

If you've never heard of Docker before, this chapter will provide the basic knowledge required to get started. Let's start by watching this very short video that introduces the core concepts.

The Rocker Project provides a large collection of ready-to-use Docker images for R users.

10.2 Docker Essentials

10.2.1 Installing Docker

The first step is to install Docker. You'll find the instructions for Ubuntu here, for Windows here (read the system requirements section as well!) and for macOS here (make sure to choose the right version for the architecture of your Mac, if you have an M1 Mac use *Mac with Apple silicon*).

After installation, it might be a good idea to restart your computer, if the installation wizard does not invite you to do so. To check whether Docker was installed successfully, run the following command in a terminal:

```
docker run --rm hello-world
```

This should print the following message:

```
Hello from Docker!
This message shows that your installation appears to
be working correctly.

To generate this message, Docker took the following
steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image
 from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from
 that image which runs the
    executable that produces the output you are
    currently reading.
 4. The Docker daemon streamed that output to the
 Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an
Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a
free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

If you see this message, congratulations, you are ready to run Docker. If you see an error message about permissions, this means that something went wrong. If you're running Linux, make sure that your user is in the Docker group by running:

```
groups $USER
```

You should see your username and a list of groups that your user belongs to. If a group called `docker` is not listed, then you should add yourself to the group by following these steps.

10.2.2 The Rocker Project and Image Registries

When running a command like:

```
docker run --rm hello-world
```

what happens is that an image, in this case `hello-world`, gets pulled from a so-called *registry*. A registry is a storage and distribution system for Docker images. Think of it as a GitHub for Docker images, where you can push and pull images, much like you would with code repositories. The default public registry that Docker uses is called Docker Hub, but companies can also host their own private registries to store proprietary images.

Many open source projects build and distribute Docker images through Docker Hub, for example the Rocker Project.

The Rocker Project is instrumental for R users that want to use Docker. The project provides a large list of images that are ready to run with a single command. As an illustration, open a terminal and paste the following line:

```
docker run --rm -e PASSWORD=yourpassword -p
 ↪  8787:8787 rocker/rstudio
```

Once this stops running, go to `http://localhost:8787/` and enter `rstudio` as the username and `yourpassword` as the password. You should login to a RStudio instance: this is the web interface of RStudio that allows you to work with R from a server. In this case, the *server* is the Docker container running the image. Yes, you've just pulled a Docker image containing Ubuntu with a fully working installation of RStudio web!

Let's open a new script and run the following lines:

```
data(mtcars)
summary(mtcars)
```

You can now stop the container (by pressing `CTRL-C` in the terminal). Let's now rerun the container… you should realise that your script is gone! This is the first lesson: whatever you do inside a container will disappear once the container is stopped. This also means that if you install the R packages that you need while the container is running, you will need to reinstall them every time.

Thankfully, the Rocker Project provides a list of images with many packages already available. For example to run R with the `{tidyverse}` collection of packages already pre-installed, run:

```
docker run --rm -e PASSWORD=yourpassword -p
 ↪  8787:8787 rocker/tidyverse
```

10.2.3 Basic Docker Workflow

You already know about running containers using `docker run`. With the commands we ran before, your terminal will need to

stay open, or else, the container will stop. Starting now, we will run Docker commands in the background using the `-d` flag (*d* as in *detach*):

```
docker run --rm -d -e PASSWORD=yourpassword -p
 ↪  8787:8787 rocker/tidyverse
```

You can run several containers in the background simultaneously. List running containers with `docker ps`:

```
docker ps
CONTAINER ID   IMAGE              COMMAND   CREATED
 ↪  STATUS         PORTS                     NAMES
c956fbeeebcb   rocker/tidyverse   "/init"   3
 ↪  minutes ago   Up 3 minutes
 ↪  0.0.0.0:8787->8787/tcp   elastic_morse
```

Stop the container using its ID:

```
docker stop c956fbeeebcb
```

Let's discuss the other flags:

- `--rm`: Removes the container once it's stopped
- `-e`: Provides environment variables to the container (e.g., `PASSWORD`)
- `-p`: Sets the port mapping (host:container)
- `--name`: Gives the container a custom name

Run a container with a name:

```
docker run -d --name my_r --rm -e
 ↪  PASSWORD=yourpassword -p 8787:8787
 ↪  rocker/tidyverse
```

You can now interact with this container using its name:

```
docker exec -ti my_r bash
```

You are now inside a terminal session, inside the running container! This can be useful for debugging purposes.

Finally, let's solve the issue of our scripts disappearing. Create a folder somewhere on your computer, then run:

```
docker run -d --name my_r --rm -e
 ↪  PASSWORD=yourpassword -p 8787:8787 \
  -v
   ↪  /path/to/your/local/folder:/home/rstudio/scripts:rw
   ↪  rocker/tidyverse
```

You should now be able to save scripts inside the `scripts/` folder from RStudio and they will appear in the folder you created on your host machine.

10.3 Making Our Own Images

To create our own images, you can start from an image provided by an open source project like Rocker, or you can start from the base Ubuntu image. Since we are using Nix to set up the reproducible development environment, we can use `ubuntu:latest`. Our development environment will always be exactly the same, thanks to Nix.

10.3.1 A Minimal Dockerfile with Nix

```
FROM ubuntu:latest

RUN apt update -y
RUN apt install curl -y

# Download the default.nix that comes with {rix}
RUN curl -O
 ↪  https://raw.githubusercontent.com/ropensci/rix/main/inst/ex

# Install Nix via Determinate Systems installer
RUN curl --proto '=https' --tlsv1.2 -sSf -L
 ↪  https://install.determinate.systems/nix | sh -s
 ↪  -- install linux \
  --extra-conf "sandbox = false" \
  --init none \
  --no-confirm

# Add Nix to the path
ENV PATH="${PATH}:/nix/var/nix/profiles/default/bin"
ENV user=root

# Configure rstats-on-nix cache
RUN mkdir -p /root/.config/nix && \
    echo "substituters = https://cache.nixos.org
     ↪  https://rstats-on-nix.cachix.org" >
     ↪  /root/.config/nix/nix.conf && \
    echo "trusted-public-keys =
     ↪  cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGsp
     ↪  rstats-on-nix.cachix.org-1:vdiiVgocg6WeJrODIqdprZRUrhi
     ↪  >> /root/.config/nix/nix.conf
```

```
# Copy script to generate environment
COPY gen-env.R .

# Generate default.nix from gen-env.R
RUN nix-shell --run "Rscript gen-env.R"

# Build the environment
RUN nix-build

# Run nix-shell when container starts
CMD nix-shell
```

Every `Dockerfile` starts with a `FROM` statement specifying the base image. Then, every command starts with `RUN`. We install and configure Nix, copy an R script to generate the environment, and then build it. Finally, we run `nix-shell` when executing a container (the `CMD` statement).

10.3.2 Splitting Into a Reusable Base Image

Because the Nix installation step is generic, we can split this into two stages. First, create a base image with just Nix installed:

```
# nix-base/Dockerfile
FROM ubuntu:latest AS nix-base

RUN apt update -y && apt install -y curl

# Install Nix via Determinate Systems installer
RUN curl --proto '=https' --tlsv1.2 -sSf -L
 ↪  https://install.determinate.systems/nix | sh -s
 ↪  -- install linux \
```

```
  --extra-conf "sandbox = false" \
  --init none \
  --no-confirm

ENV PATH="/nix/var/nix/profiles/default/bin:${PATH}"
ENV user=root

# Configure Nix binary cache
RUN mkdir -p /root/.config/nix && \
    echo "substituters = https://cache.nixos.org
     ↪  https://rstats-on-nix.cachix.org" >
     ↪  /root/.config/nix/nix.conf && \
    echo "trusted-public-keys =
     ↪  cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGsp
     ↪  rstats-on-nix.cachix.org-1:vdiiVgocg6WeJrODIqdprZRUrhi1
     ↪  >> /root/.config/nix/nix.conf
```

Build and tag this image:

```
docker build -t nix-base:latest .
```

Now, for any project, simply reuse it:

```
FROM nix-base:latest

COPY gen-env.R .

RUN curl -O
 ↪  https://raw.githubusercontent.com/ropensci/rix/main/inst/ex
RUN nix-shell --run "Rscript gen-env.R"
RUN nix-build
```

```
CMD ["nix-shell"]
```

With `gen-env.R`:

```
library(rix)

rix(
  date = "2025-08-04",
  r_pkgs = c("dplyr", "ggplot2"),
  py_conf = list(
    py_version = "3.13",
    py_pkgs = c("polars", "great-tables")
  ),
  ide = "none",
  project_path = ".",
  overwrite = TRUE
)
```

Build and run:

```
docker build -t my-project .
docker run -it --rm --name my-project-container
 ↪  my-project
```

This drops you in an interactive Nix shell running inside Docker!

10.4 Publishing Images on Docker Hub

If you want to share Docker images through Docker Hub, you first need to create a free account. A free account gives you

unlimited public repositories.

List all images on your computer:

```
docker images
```

Log in to Docker Hub:

```
docker login
```

Tag the image with your username:

```
docker tag IMAGE_ID your_username/nix-base:latest
```

Push the image:

```
docker push your_username/nix-base:latest
```

This image can now be used as a stable base for other projects:

```
FROM your_username/nix-base:latest

RUN mkdir ...
```

10.4.1 Sharing Without Docker Hub

If you can't upload to Docker Hub, you can save the image to a file:

```
docker save nix-base | gzip > nix-base.tgz
```

Load it on another machine:

```
gzip -d nix-base.tgz
docker load < nix-base.tar
```

10.5 What If You Don't Use Nix?

Using Nix inside of Docker makes it very easy to set up an environment, but what if you can't use Nix for some reason? In this case, you would need to use other tools to install the right R or Python packages and it is likely going to be more difficult. The main issue you will face is missing development libraries.

For example, to install and use the R `{stringr}` package, you will need to first install `libicu-dev`:

```
FROM rocker/r-ver:4.5.1

RUN apt-get update && apt-get install -y \
    libglpk-dev \
    libxml2-dev \
    libcairo2-dev \
    libgit2-dev \
    libcurl4-openssl-dev \
    # ... many more
```

Another issue is that building the image is not a reproducible process, only running containers is. To mitigate this, use tagged images or better yet, a digest:

```
FROM
 ↪ rocker/r-ver@sha256:1dbe7a6718b7bd8630addc45a32731624fb7b7
```

This will always pull exactly the same layers. However, at some point, that version of Ubuntu will be outdated. Using Nix, you can stay on `ubuntu:latest`.

10.6 Building Docker Images Directly with Nix

So far, we've used Nix *inside* Docker containers. But there's an even more powerful approach: using Nix to build Docker images directly, bypassing `Dockerfile`s entirely.

Nix provides `dockerTools`, a set of functions that can create OCI-compliant container images. Because these images are built through Nix's deterministic build system, they have stronger reproducibility guarantees than images built with `docker build`.

10.6.1 Why Skip the Dockerfile?

A `Dockerfile` is fundamentally imperative: it's a script of commands executed in order. Commands like `apt-get update` are non-deterministic by nature. Even with careful pinning, you're fighting the tool's design.

Nix's `dockerTools.buildImage` is declarative. You describe what should be in the image, and Nix figures out how to build it reproducibly. The resulting image is a pure function of its inputs.

10.6.2 A Basic Example

Here's a Nix expression that builds a Docker image containing R and some packages:

```
# docker-image.nix
let
  pkgs = import (fetchTarball

     ↪  "https://github.com/rstats-on-nix/nixpkgs/archive/202
  ) {};

  # Define the R environment
  myR = pkgs.rWrapper.override {
    packages = with pkgs.rPackages; [
      dplyr
      ggplot2
      quarto
    ];
  };

in pkgs.dockerTools.buildImage {
  name = "my-r-env";
  tag = "latest";

  copyToRoot = pkgs.buildEnv {
    name = "image-root";
    paths = [ myR pkgs.quarto pkgs.coreutils
     ↪  pkgs.bash ];
    pathsToLink = [ "/bin" ];
  };

  config = {
    Cmd = [ "${pkgs.bash}/bin/bash" ];
```

```
    WorkingDir = "/work";
  };
}
```

Build and load it:

```
# Build the image (outputs a .tar.gz file)
nix-build docker-image.nix

# Load into Docker
docker load < result

# Run it
docker run -it --rm my-r-env:latest
```

10.6.3 Using rix with dockerTools

The previous example manually defines the R environment in Nix. If you prefer to use `{rix}` to generate your environment (as we've done throughout this book), you can import the generated `default.nix` into your Docker image definition.

First, create your environment with `{rix}` as usual:

```
library(rix)

rix(
  date = "2025-10-14",
  r_pkgs = c("dplyr", "ggplot2", "quarto"),
  ide = "none",
  project_path = ".",
```

```
  overwrite = TRUE
)
```

This generates a `default.nix`. Now create a `docker-image.nix` that imports it:

```
# docker-image.nix
let
  pkgs = import (fetchTarball

    ↪ "https://github.com/rstats-on-nix/nixpkgs/archive/202
  ) {};

  # Import the shell environment generated by rix
  rixEnv = import ./default.nix;

in pkgs.dockerTools.buildImage {
  name = "my-rix-project";
  tag = "latest";

  copyToRoot = pkgs.buildEnv {
    name = "image-root";
    paths = rixEnv.buildInputs ++ [ pkgs.coreutils
    ↪ pkgs.bash ];
    pathsToLink = [ "/bin" ];
  };

  config = {
    Cmd = [ "${pkgs.bash}/bin/bash" ];
    WorkingDir = "/work";
  };
}
```

This approach lets you keep your familiar `{rix}` workflow for defining environments while gaining the reproducibility benefits of `dockerTools` for image creation. Any changes to your `gen-env.R` → `default.nix` will automatically propagate to the Docker image on the next build.

10.6.4 Benefits Over Dockerfiles

Aspect	Dockerfile	Nix dockerTools
Reproducibility	Imperative, non-deterministic	Declarative, deterministic
Layer caching	Based on command order	Based on content hashes
Image size	Often includes unnecessary packages	Minimal: only what you specify
Composition	Copy-paste between files	Reuse Nix expressions

10.6.5 Layered Images for Efficiency

For faster CI builds, you can create layered images where the base environment is cached:

```
pkgs.dockerTools.buildLayeredImage {
  name = "my-analysis";
  tag = "latest";

  contents = [ myR pkgs.quarto ];
```

```
  config = {
    Cmd = [ "${myR}/bin/R" "--vanilla" ];
  };
}
```

`buildLayeredImage` creates separate layers for each package, so unchanged dependencies are cached between builds.

10.6.6 When to Use This Approach

- **CI/CD pipelines**: When you need reproducible image builds as part of automated workflows
- **Minimal images**: When image size matters and you want only what you need
- **Complex environments**: When the environment is already defined in Nix and you want to deploy it as a container

For simpler use cases, the "Nix inside Docker" approach from earlier sections may be more accessible.

10.7 Dockerising a rixpress Pipeline

We can package our entire `{rixpress}` project into a single Docker image. This image can then be run by anyone with Docker installed, regardless of their host operating system or whether they have Nix.

Assume your project directory has:

```
.
   data/
       mtcars.csv
   gen-env.R
   gen-pipeline.R
   functions.R
   functions.py
   default.nix
   pipeline.nix
```

10.7.1 Step 1: The Dockerfile

```
FROM nix-base:latest
# Or: FROM your-username/nix-base:latest

WORKDIR /app

# Copy all project files
COPY . .

# Build the pipeline during image build
RUN nix-build pipeline.nix

# Export results when container runs
COPY export-results.R .
CMD ["nix-shell", "--run", "Rscript
 ↪  export-results.R"]
```

10.7.2 Step 2: The Export Script

```
# export-results.R
library(rixpress)
library(jsonlite)

output_dir <- "/output"
dir.create(output_dir, showWarnings = FALSE)

message("Reading target 'mtcars_head'...")
final_data <- rxp_read("mtcars_head")

output_path <- file.path(output_dir,
 ↪  "mtcars_analysis_result.json")
write_json(final_data, output_path, pretty = TRUE)

message(paste("Successfully exported result to",
 ↪  output_path))
```

10.7.3 Step 3: Build and Run

Build:

```
docker build -t my-reproducible-pipeline .
```

Run to get results:

```
mkdir -p ./output

docker run --rm --name my_pipeline_run \
```

```
  -v "$(pwd)/output":/output \
  my-reproducible-pipeline
```

Check your local `output` directory. You'll find the `mtcars_analysis_result.json` file containing the exact, reproducible result of your pipeline.

You have successfully packaged a complex, polyglot pipeline into a simple, portable Docker image. This workflow combines the best of both worlds: Nix's power for creating reproducible builds and Docker's universal standard for distributing and running applications.

10.7.4 Alternative: Using dockerTools for rixpress

You can also build your rixpress pipeline image purely with Nix, avoiding Dockerfiles entirely. This gives you fully deterministic image builds.

Create a `docker-pipeline.nix`:

```
# docker-pipeline.nix
let
  pkgs = import (fetchTarball

    ↪ "https://github.com/rstats-on-nix/nixpkgs/archive/2025-
  ) {};

  # Import your rix-generated environment
  rixEnv = import ./default.nix;

  # Build the pipeline as a derivation
```

```
  pipelineResult = pkgs.runCommand "pipeline-result"
    ↪ {
    buildInputs = rixEnv.buildInputs;
  } ''
    mkdir -p $out
    cd ${./. }
    Rscript -e "source('gen-pipeline.R')"
    # Copy results to output
    cp -r _rixpress $out/
  '';

in pkgs.dockerTools.buildImage {
  name = "my-rixpress-pipeline";
  tag = "latest";

  copyToRoot = pkgs.buildEnv {
    name = "image-root";
    paths = [
      rixEnv.buildInputs
      pipelineResult
      pkgs.coreutils
      pkgs.bash
    ];
    pathsToLink = [ "/bin" "/" ];
  };

  config = {
    Cmd = [ "${pkgs.bash}/bin/bash" "-c" "cp -r
      ↪ /pipeline-result/* /output/" ];
    WorkingDir = "/work";
  };
}
```

Build and run:

```
nix-build docker-pipeline.nix
docker load < result
docker run --rm -v "$(pwd)/output":/output
 ↪  my-rixpress-pipeline:latest
```

This approach bakes the pipeline results directly into the image during the Nix build phase. The resulting image is minimal and contains only the outputs, not the full R/Python environment needed to produce them.

10.8 Summary

Docker and Nix complement each other:

- **Docker** provides a universal runtime and distribution mechanism
- **Nix** provides bit-for-bit reproducible environment management
- **Together** they enable truly reproducible, portable data products

Key Docker concepts:

- **Images** are templates; **containers** are running instances
- Containers are ephemeral; use **volumes** for persistence
- Use **registries** (Docker Hub) to share images
- The **Rocker Project** provides ready-made R images

For reproducibility:

- Use `ubuntu:latest` + Nix rather than pinning specific OS versions
- Build a reusable `nix-base` image to share across projects
- Package `{rixpress}` pipelines for distribution

10.9 Further Reading

- Running Your R Script in Docker
- Docker & R Reproducibility
- R Docker Tutorial

11 A Short Intro to Packaging Your Code in R and Python

11.1 Introduction

In the previous chapter, we learned how to prove that our code works through unit testing and how to manage our collaboration with Git. We now have functions, tests, and version control. But there is one more step we can take to truly professionalise our workflow: packaging.

You might think, "I'm a data scientist, not a software engineer. Isn't this overkill?" The answer is a definitive no. Packaging your code, even for an internal analysis project, provides enormous benefits. Instead of copying and pasting your `clean_data()` function from project to project, you can simply `import mypackage` or `library(mypackage)` and use a single, trusted version. Sharing your work with a colleague becomes as simple as sending them a single command rather than emailing a zip file of scripts. Packaging also forces you into a standardised way of documenting your functions, provides a formal framework for running unit tests, and explicitly declares all of your dependencies.

This chapter will walk you through the process of creating a simple package in both R and Python. You will learn how to create, document, and test a basic R package using `{devtools}` and `{usethis}`, how to do the same for a modern Python package using `uv` and `pytest`, and how to install your own packages directly from GitHub so you can share your tools with colleagues and your future self. The goal is not to become an expert package developer, but to understand the structure and benefits so you can apply this powerful "packaging mindset" to all your future projects.

11.2 Part 1: Creating an R Package with `{usethis}` and `{devtools}`

The R community has developed an outstanding set of tools that make package development incredibly streamlined. The two essential packages are:

- `{devtools}`: provides core development tools like `install()`, `test()`, and `check()`.
- `{usethis}`: a workflow package that automates all the boilerplate. It creates files, sets up infrastructure, and guides you through the process.

Let's build a package called `cleanR`, which will contain a function to standardise column names. Create a folder called `cleanR` and `cd` into it.

11.2.1 Step 1: Project Setup

First, make sure you have the necessary tools installed. Create a Nix environment that contains the following R packages: {devtools}, {usethis} and {roxygen2}.

Drop into this Nix shell, and let {usethis} create the package structure for you. From your R console, run:

```
usethis::create_package("~/Documents/projects/cleanR")
```

This will create a new cleanR directory with all the necessary files and subdirectories. It will also open a new RStudio session for that project. The key components are:

- R/: this is where your R source code files will live.
- DESCRIPTION: a metadata file describing your package, its author, licence, and dependencies.
- NAMESPACE: a file that declares which functions your package exports for users and which functions it imports from other packages. You should never edit this file by hand. {roxygen2} will manage it for you.

11.2.2 Step 2: Write and Document a Function

Let's create our function. {usethis} helps with this too:

```
usethis::use_r("clean_names")
```

This creates a new file R/clean_names.R and opens it for editing. Let's add our function, including special comments for documentation. These #' comments are used by the {roxygen2} package to automatically generate the official documentation.

```
# In R/clean_names.R

#' Clean and Standardize Column Names
#'
#' This function takes a data frame and returns a
 ↪  new data frame with
#' cleaned-up column names (lowercase, with
 ↪  underscores instead of spaces
#' or periods).
#'
#' @param df A data frame.
#' @return A data frame with standardized column
 ↪  names.
#' @export
#' @examples
#' messy_df <- data.frame("First Name" = c("Ada",
 ↪  "Bob"), "Last.Name" = c("Lovelace", "Ross"))
#' clean_names(messy_df)
clean_names <- function(df) {
  old_names <- names(df)
  new_names <- tolower(old_names)
  new_names <- gsub("[ .]", "_", new_names)
  names(df) <- new_names
  return(df)
}
```

The key tags here are:

- `@param`: describes a function argument.
- `@return`: describes what the function returns.
- `@export`: this is crucial. It tells R that you want this function to be available to users when they load your package with `library(cleanR)`.

- `@examples`: provides runnable examples that will appear in the help file.

Now, run the magic command to process these comments:

```
devtools::document()
```

This updates the `NAMESPACE` file and creates the help file (`man/clean_names.Rd`). You can now see your function's help page with `?clean_names`.

11.2.3 Step 3: Add Unit Tests

A package without tests is a package waiting to break. `{usethis}` makes setting up tests trivial.

```
usethis::use_testthat() # Sets up the
 ↪  tests/testthat/ directory
usethis::use_test("clean_names") # Creates
 ↪  tests/testthat/test-clean_names.R
```

Now, edit the test file to add your expectations.

```
# In tests/testthat/test-clean_names.R
test_that("clean_names works with spaces and
 ↪  periods", {
  messy_df <- data.frame("First Name" = c("A"),
 ↪  "Last.Name" = c("B"))
  cleaned_df <- clean_names(messy_df)

  expected_names <- c("first_name", "last_name")
```

```
  expect_equal(names(cleaned_df), expected_names)
})

test_that("clean_names handles already clean names",
 ↪  {
  clean_df <- data.frame(a = 1, b = 2)
  # The function should not change anything
  expect_equal(names(clean_names(clean_df)), c("a",
   ↪  "b"))
})
```

To run all the tests for your package, use:

```
devtools::test()
```

11.2.4 Step 4: Check and Install

The final step before sharing is to run the official R CMD check, the gold standard for package quality. This command runs all tests, checks documentation, and looks for common problems.

```
devtools::check()
```

If your package passes with 0 errors, 0 warnings, and 0 notes, you are in great shape. If your package raises NOTEs or WARNINGs during the check phase, you can *most of the time* safely ignore these, especially if the package is only intended for internal usage. However, I would recommend that you still take care of the WARNINGs at the very least.

As a next step, you could edit the DESCRIPTION file. This is where you will list yourself as the package author, list the dependencies of the package and so on. I won't got into detail here, but learning how to edit the DESCRIPTION file is important for actual package development (especially listing dependencies is key).

To use your package within a project, the simplest way is to host it on GitHub or build a `.tar.gz` file and install it locally.

11.2.5 Step 5: Install from GitHub

If you can publicly host your package, hosting it on GitHub is a good way to easily share your code, and install the package in your projects without needing to publish it on CRAN.

1. Create a new, empty repository on GitHub (e.g., `cleanR`).
2. In your local project, follow the instructions GitHub provides to link your local repository and push your code. This usually involves commands like:

```
git remote add origin
 ↪  git@github.com:yourusername/cleanR.git
git branch -M main
git push -u origin main
```

3. Now, anyone (including you on a different machine) can install your package with a single command (**if you don't use Nix**):

```
# You might need to install {remotes} first
# install.packages("remotes")
remotes::install_github("yourusername/cleanR")
```

If you want to create an environment using Nix that includes this package, use the `git_pkgs` argument of `rix::rix()` to generate the right `default.nix` file.

Congratulations, you have created and shared a fully functional R package!

11.2.6 Step 5bis: Install it locally

If you can't share your package on GitHub, the alternative is to build it locally using:

```
devtools::build()
```

which will create a `.tar.gz` package. You can then install it using either `devtools::install_local()` if you don't use Nix or the `local_r_pkgs` argument of `rix::rix()`.

11.3 Part 2: Creating a Minimal Python Package with uv

There are many different ways to build packages in Python, and what I propose here is just one way to do it. While we will use Nix to manage our overall environment, we still need to define the metadata and structure for our Python package. We will use `uv`, an extremely fast and modern tool, for one specific purpose: initialising our project's configuration file. We will not use `uv` to manage a virtual environment, as Nix already handles that for us (unless you absolutely want to: however, you should

then make sure that `uv` itself is being managed by Nix to ensure reproducibility).

Let's build a Python package called `pyclean`, the equivalent of our R package.

11.3.1 Step 1: Project Setup with uv

First, ensure `uv` is installed in your Nix environment:

```
library(rix)

rix(
  date = "2025-10-07",
  py_conf = list(
    py_version = "3.13",
    py_pkgs = c("pytest", "pandas")
  ),
  system_pkgs = "uv",
  ide = "none",
  project_path = ".",
  overwrite = TRUE
)
```

Then, create a directory for your new package and initialise it:

```
mkdir pyclean
cd pyclean
uv init --bare
```

The `--bare` flag is perfect for our Nix workflow. It creates only the essential `pyproject.toml` file without creating a virtual

environment or extra directories. This leaves us with a clean slate.

Now, we must create the source and test directories manually. We'll use the standard `src` layout:

```
mkdir -p src/pyclean
mkdir tests
touch src/pyclean/__init__.py
```

Your project structure should now look like this (check it using the `tree` command):

```
pyclean/
   pyproject.toml
   src/
       pyclean/
           __init__.py
   tests/
```

11.3.2 Step 2: Write a Function and Declare Dependencies

Let's create our `clean_names` function inside a new file, `src/pyclean/formatters.py`.

```
# In src/pyclean/formatters.py
import pandas as pd

def clean_names(df: pd.DataFrame) -> pd.DataFrame:
    """Clean and standardize column names of a
     ↪ DataFrame.
```

```
    Args:
        df: The input pandas DataFrame.

    Returns:
        A pandas DataFrame with standardized column
↪   names.
    """
    new_df = df.copy()
    new_cols = {col: col.lower().replace(" ",
↪   "_").replace(".", "_") for col in
↪   new_df.columns}
    new_df = new_df.rename(columns=new_cols)
    return new_df
```

To make this function easily importable, we expose it in `src/pyclean/__init__.py`:

```
# In src/pyclean/__init__.py
from .formatters import clean_names

__all__ = ["clean_names"]
```

Next, we must declare our dependencies by manually editing `pyproject.toml`. We need `pandas` for our function and `pytest` for our tests.

```
# In pyproject.toml
[project]
name = "pyclean"
version = "0.1.0"
description = "A simple package to clean data."
```

```
dependencies = [
    "pandas>=2.0.0",
]

[project.optional-dependencies]
test = [
    "pytest",
]

[tool.pytest.ini_options]
pythonpath = [
  "src"
]
```

The `pythonpath = ["src"]` line is very important. Without it, you'd first need to install your `pyclean` library in editable mode using `pip` before running the tests. By adding this block, simply running `pytest` from the command line will work.

11.3.3 Step 3: Add Unit Tests

Create a new test file, `tests/test_formatters.py`, and add your tests.

```
# In tests/test_formatters.py
import pandas as pd
from pyclean import clean_names

def test_clean_names_happy_path():
    messy_df = pd.DataFrame({"First Name": ["Ada"],
 ↪  "Last.Name": ["Lovelace"]})
```

```
    cleaned_df = clean_names(messy_df)
    expected_cols = ["first_name", "last_name"]
    assert list(cleaned_df.columns) == expected_cols

def test_clean_names_is_idempotent():
    clean_df = pd.DataFrame({"first_name": ["a"],
 ↪  "last_name": ["b"]})
    still_clean_df = clean_names(clean_df)
    assert list(still_clean_df.columns) ==
     ↪  list(clean_df.columns)
```

Since your Nix environment provides all the tools, you can run tests directly from your terminal:

```
pytest
```

11.3.4 Step 4: Build and Install

To package your code, you need a build tool. It turns out that `uv` bundles a build tool with it, so we only need to call `uv build`:

```
# In your terminal, from the root of the 'pyclean'
 ↪  project
uv build
```

This creates a `dist/` directory containing a source distribution (`.tar.gz`) and a compiled wheel (`.whl`). The wheel is the modern standard for distribution.

Outside of a Nix shell, to use your package during development, you can install it in "editable" mode. This creates a link to your

source code, so any changes you make are immediately reflected without needing to reinstall.

```
# Install the package and its test dependencies
pip install -e .[test]
```

But we are working from a Nix shell. Instead, we will simply edit our `default.nix` to update the `PYTHONPATH` environment variable, so our package can easily be found:

```
shellHook = ''
  export PYTHONPATH=$PWD/src:$PYTHONPATH
'';
```

To generate this with `{rix}`, use the `shell_hook` argument:

```
library(rix)

rix(
  date = "2025-10-07",
  py_conf = list(
    py_version = "3.13",
    py_pkgs = c("pytest", "pandas")
  ),
  system_pkgs = "uv",
  shell_hook = "export
   ↪  PYTHONPATH=$PWD/src:$PYTHONPATH",
  ide = "none",
  project_path = ".",
  overwrite = TRUE
)
```

With this, dropping into the Nix shell, starting the Python interpreter and then typing `import pyclean` will work without any issues. You may need to adapt the path depending on where you're developing the package.

11.3.5 Step 5: Install from GitHub

Sharing via GitHub is the most common way to distribute packages that aren't on the official Python Package Index (PyPI):

1. Create a new, empty repository on GitHub.
2. Push your local project to the remote repository.
3. Now, anyone can install your package directly from GitHub using `pip`, which is smart enough to find and process your `pyproject.toml` file: `bash pip install git+https://github.com/yourusername/pyclean.git`

For Nix environments, add this to your `default.nix`:

```
pyclean = pkgs.python313Packages.buildPythonPackage
rec {
  pname = "pyclean";
  version = "0.1.0";
  src = pkgs.fetchgit {
    url = "https://github.com/b-rodrigues/pyclean";
    rev =
    "174d4d482d400536bb0d987a3e25ae80cd81ef3c";
    sha256 =
    "sha256-xTYydkuduPpZsCXE2fv5qZCnYYCRoNFpV7lQBM3LMSg=";
  };
  pyproject = true;
  propagatedBuildInputs = [
  pkgs.python313Packages.pandas
  pkgs.python313Packages.setuptools ];
```

```
  # Add more dependencies to propagatedBuildInputs
  as needed
};
```

You need to add the `rev`, which corresponds to the commit that want, and the `sha256`. To find the right `sha256`, start with an empty one (`sha256 = "";`) and try to build the package. The error message will give you the right `sha256`. Also not that this isn't the the most idiomatic way to build a Python package for Nix, but it's good enough for our purposes.

Finally, add `pyclean` to the `buildInputs` of the shell:

```
  buildInputs = [ rpkgs pyconf pyclean tex
↪  system_packages github_pkgs ];
```

This process is naturally more involved than simply calling `pip install`, but it has the advantage of being entirely reproducible.

11.4 Conclusion: The Packaging Mindset in the Age of AI

You have now successfully created, tested, documented, and shared a basic package in both R and Python. While there is much more to learn about advanced package development, you have already mastered the most important part: the packaging mindset.

From now on, when you start a new analysis project, think of it as a small, internal package.

- Put your reusable logic into functions.
- Place those functions in the `R/` or `mypackage/` source directory.
- Document them.
- Write a few simple tests to prove they work.
- Manage dependencies formally in `DESCRIPTION` or `pyproject.toml`.

Adopting this structure will make your work more robust, easier to share, and fundamentally more reproducible. It is the bridge between writing one-off scripts and building reliable, professional data science tools.

This packaging mindset becomes even more powerful when you introduce a modern collaborator: the LLM. The structured, component-based nature of a package is the perfect way to interact with AI assistants.

A package provides a clear contract and a well-defined structure that LLMs thrive on. Instead of a vague prompt like, "Refactor my messy analysis script," you can now make precise, targeted requests:

- "Here is my function `clean_names`. Please write three `pytest` unit tests for it, including one for the happy path, one for an empty DataFrame, and one for names that are already clean."
- "Generate the roxygen2 documentation skeleton for this R function, including `@param`, `@return`, and `@examples` tags."
- "I need a function in my `pyclean/utils.py` module that calculates the Z-score for a pandas Series. Please generate the function and its docstring."

This synergy is a two-way street. Not only does the structure help you write better prompts, but LLMs excel at generating

the very boilerplate that makes packaging robust. Tedious tasks like writing standard documentation headers, creating skeleton unit test files, or even generating a first draft of a function based on a clear description become near-instantaneous.

This elevates your role from a writer of code to an architect and a reviewer. Your job is to design the components (the functions), prompt the LLM to generate the implementation, and then, most critically, use the testing framework you just built to rigorously verify that the AI-generated code is correct, efficient, and robust. You are the final authority, and the package structure gives you the tools to enforce quality control.

By combining the discipline of packaging with the power of LLMs, you lower the barrier to adopting best practices like comprehensive testing and documentation. This combination doesn't just make you faster; it makes you a more reliable and professional data scientist, capable of producing tools that are truly reproducible and built to last.

While a full guide to package development is beyond the scope of this course, it is the natural next step in your journey as a data scientist who produces reliable tools. When you are ready to take that step, here are the definitive resources to guide you:

- For R: the "R Packages" (2e) book by Hadley Wickham and Jennifer Bryan is the essential, comprehensive guide. It covers everything from initial setup with `{usethis}` to testing, documentation, and submission to CRAN. Read it online here.
- For Python: the official Python Packaging User Guide is the place to start. For a more modern and streamlined approach that handles dependency management and publishing, many developers use tools like Poetry or Hatch.

Treating your data analysis project like a small, internal software package, complete with functions and tests, is a powerful mindset that will elevate the quality and reliability of your work.

12 Continuous Integration with GitHub Actions

12.1 Introduction

We are almost at the end of our journey. In the previous chapters, we built reproducible environments with Nix, organised our code into pure functions, added robustness with monads, proved correctness with unit tests, managed our collaboration with Git, and bundled everything into shareable packages. We can now run our pipelines in a 100% reproducible way.

However, all of this still requires manual steps. And maybe that's not a problem; if your environment is set up and users only need to drop into a Nix shell and run the pipeline, that's already a huge improvement. But you should keep in mind that manual steps don't scale. Imagine you are part of a team that needs to quickly ship products to clients. Several people contribute to the product, and you might need to work on multiple projects in the same day. You and your teammates should be focusing on writing code, not on repetitive tasks like building images or running tests. Ideally, we would want to automate these steps. That is what we are going to learn in this chapter.

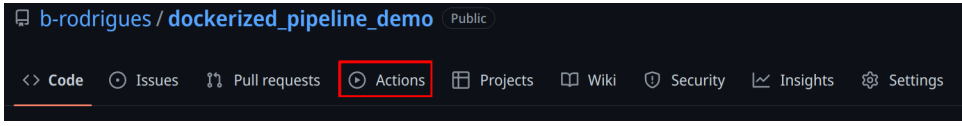
This chapter will introduce you to **Continuous Integration (CI)** with GitHub Actions. You will learn how to set up workflows that automatically run your tests when you push code, how

to build Docker images and recover artifacts, and how to run your pipelines directly from GitHub's servers. Because we're using Git to trigger all the events and automate the whole pipeline, this approach is sometimes called GitOps.

You may have heard the term "CI/CD," where CD stands for Continuous Deployment or Continuous Delivery. We will focus on CI in this chapter. Continuous Deployment (automatically pushing results to a database, dashboard, or API) is highly specific to your organisation and infrastructure. What we cover here, however, gives you the foundation: once your pipeline runs reliably on CI, the "deployment" step is just one more workflow job pointing to wherever your results need to go.

12.2 Getting your repo ready for Github Actions

Obviously, you should use a project that is versioned on GitHub. Use the package we've developed previously. If you go on its GitHub page, you should see an "Actions" tab on top:



This will open a new view where you can select a lot of available, ready to use actions. "Actions" are premade scripts that execute some commands you might need: such as setting up R, Python, running tests, etc. Since we're using Nix, we don't really need to look for any actions to set up our environments. However, we might want to use some pre-made actions to upload artifacts for instance.

To actually configure our repository to run actions, we need to edit a file in our project under the `.github/workflows` directory (create them if needed). In it, write a `yaml` file called `hello.yaml` and write the following in it:

```
name: Hello world
on: [push]
jobs:
  say-hello:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Hello from Github Actions!"
      - run: echo "This command is running from an
 ↪  Ubuntu VM each time you push."
```

Let's study this workflow definition line by line:

`name: Hello world`

Simply gives a name to the workflow.

`on: [push]`

When should this workflow be triggered? Here, whenever something gets pushed.

`jobs:`

What is the actual things that should happen? This defines a list of actions.

`  say-hello:`

This defines the `say-hello` job.

`    runs-on: ubuntu-latest`

This job should run on an Ubuntu VM. You can also run jobs on Windows or macOS VMs, but this uses more compute minutes than a Linux VM (which doesn't matter for public projects. For private projects, the amount of compute minutes is limited).

```
steps:
```

What are the different steps of the job?

```
  - run: echo "Hello from Github Actions!"
```

First, run the command `echo "Hello from Github Actions!"`. This commands runs inside the VM. Then, run this next command:

```
  - run: echo "This command is running from an
  Ubuntu VM each time you push."
```

If we take a look at the commit we just pushed, on GitHub, we see this yellow dot next to the commit name. This means that an action is running. We can then take a look at the output of the job, and see that our commands, defined with the `run` statements in the workflow file, succeeded and echoed what we asked them.

12.3 Nix and GitHub Actions

To set up Nix on GitHub Actions you can use several steps (create a new file called `run-tests.yaml`):

```
- name: Install Nix
  uses: cachix/install-nix-action@v31
  with:
    nix_path:
 ↪  nixpkgs=https://github.com/rstats-on-nix/nixpkgs/archive/

- name: Setup Cachix
  uses: cachix/cachix-action@v15
  with:
    name: rstats-on-nix
```

If you're repository contains a `default.nix` file, the same environment you've been using locally can be used on GitHub Actions just as easily. You can also instead generate the `default.nix` from the `gen-env.R` script:

```
- name: Build dev env
  run: |
    nix-shell --expr "$(curl -sl
 ↪  https://raw.githubusercontent.com/ropensci/rix/main/inst/
 ↪  --run "Rscript -e 'source(\"gen-env.R\")'"
```

You can then use the shell to run whatever you need. For example, if you're developing a package, you could run unit tests on each push:

```
- name: devtools::test() via nix-shell
  run: nix-shell --run "Rscript -e
 ↪  \"devtools::test(stop_on_failure = TRUE)\""
```

`stop_on_failure = TRUE` is needed to make the step fail if there's an error, otherwise, the step would run successfully, even with failing tests.

Of course, if you're developing a Python package, use `nix-shell --run "pytest"` instead to run the tests.

I highly recommend you run tests when pull requests get opened:

```
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
```

This will ensure that if someone contributes to your project, you know immediately if what they did breaks tests or not. If it does, ask them to fix the code until tests pass.

12.4 Running a dockerized workflow

This next example can be found in this repository. This example doesn't use Nix, `{rix}` nor `{rixpress}`, but the point here is to show how a Docker container can be executed on GitHub Actions, and artifacts can be recovered. The process is always the same, regardless is inside the Docker image. If you want to follow along, fork this repository.

This is what our workflow file looks like:

```
name: Reproducible pipeline

on:
  push:
    branches: [ "main" ]
```

```
  pull_request:
    branches: [ "main" ]

jobs:

  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v5
    - name: Build the Docker image
      run: docker build -t my-image-name .
    - name: Docker Run Action
      run: docker run --rm --name
↪   my_pipeline_container -v
↪   /github/workspace/fig/:/home/graphs/:rw
↪   my-image-name
    - uses: actions/upload-artifact@v4
      with:
        name: my-figures
        path: /github/workspace/fig/
```

For now, let's focus on the `run` statements, because these should be familiar:

```
run: docker build -t my-image-name .
```

and:

```
run: docker run --rm --name my_pipeline_container -v
/github/workspace/fig/:/home/graphs/:rw
my-image-name
```

The only new thing here, is that the path has been changed to `/github/workspace/`. This is the home directory of your repository, so to speak. Now there's the `uses` keyword that's new:

```
uses: actions/checkout@v5
```

This action checkouts your repository inside the VM, so the files in the repo are available inside the VM. Then, there's this action here:

```
- uses: actions/upload-artifact@v4
  with:
    name: my-figures
    path: /github/workspace/fig/
```

This action takes what's inside `/github/workspace/fig/` (which will be the output of our pipeline) and makes the contents available as so-called "artifacts". Artifacts are the outputs of your workflow. In our case, as stated, the output of the pipeline. So let's run this by pushing a change, and let's take a look at these artifacts!

After the action done running, you will be able to download a zip file containing the plots. It is thus possible to rerun our workflow in the cloud. This has the advantage that we can now focus on simply changing the code, and not have to bother with boring manual steps. For example, let's change this target in the `_targets.R` file:

```
tar_target(
  commune_data,
  clean_unemp(
    unemp_data,
    place_name_of_interest = c(
```

```
      "Luxembourg", "Dippach",
      "Wiltz", "Esch/Alzette",
      "Mersch", "Dudelange"),
    col_of_interest = active_population)
)
```

I've added "Dudelange" to the list of communes to plot. Pushing this change to GitHub triggers the action we've defined before. The plots (artifacts) get refreshed, and we can download them. Take a look and see that Dudelange was added in the `communes.png` plot!

It is also possible to "deploy" the plots directly to another branch, and do much, much more. I just wanted to give you a little taste of Github Actions (and more generally GitOps). The possibilities are virtually limitless, and I still can't get over the fact that Github Actions is free for public repositories.

12.5 Building a Docker image and pushing it to a registry

It is also possible to build a Docker image and have it made available on an image registry. You can see how this works on this repository. This images can then be used as a base for other reproducible pipelines, as in this repository. Why do this? Well because of "separation of concerns". You could have a repository which builds in image containing your development environment: this could be an image with a specific version of R and R packages built with Nix. And then have as many repositories as projects that run pipelines using that development environment image as

a basis. Simply add the project-specific packages that you need for each project.

12.6 Running a rixpress Pipeline from GitHub Actions

With Nix and `{rixpress}`, running your pipeline directly on GitHub Actions is straightforward. Because Nix handles all dependencies reproducibly, you don't need Docker as an intermediary. The rixpress_demos repository contains several complete examples; here we will walk through the key steps.

The workflow triggers on pushes and pull requests to `main`:

```
on:
  pull_request:
    branches: [main, master]
  push:
    branches: [main, master]
```

After checking out the repository and installing Nix (with Cachix for faster builds), the first step generates or regenerates the development environment from the `gen-env.R` script:

```
- name: Build dev env
  run: |
    nix-shell -p R "rPackages.rix"
 ↪  "rPackages.rixpress" --run "Rscript gen-env.R"
```

Next, the pipeline definition is generated from `gen-pipeline.R`:

```
- name: Generate pipeline
  run: |
    nix-shell --quiet --run "Rscript gen-pipeline.R"
```

You can optionally visualise the DAG to verify the pipeline structure:

```
- name: Check DAG
  run: |
    nix-shell --quiet -p haskellPackages.stacked-dag
 ↪  --run "stacked-dag dot _rixpress/dag.dot"
```

Finally, build and inspect the pipeline:

```
- name: Build pipeline
  run: |
    nix-shell --quiet --run "Rscript -e
 ↪  'rixpress::rxp_make()'"

- name: Inspect built derivations
  run: |
    nix-shell --quiet --run "Rscript -e
 ↪  'rixpress::rxp_inspect()'"

- name: Show result
  run: |
    nix-shell --quiet --run "Rscript -e
 ↪  'rixpress::rxp_read(\"confusion_matrix\")'"
```

12.6.1 Caching Pipeline Outputs Between Runs

While Nix caches derivations, CI runners are ephemeral: each run starts fresh. To avoid rebuilding the entire pipeline every time, `{rixpress}` provides `rxp_export_artifacts()` and `rxp_import_artifacts()` to persist outputs between runs.

Before building, check if cached outputs exist and import them:

```
- name: Import cached outputs if available
  run: |
    if [ -f
 ↪  "../outputs/my_pipeline/pipeline_outputs.nar" ];
 ↪  then
      nix-shell --quiet --run "Rscript -e
 ↪  'rixpress::rxp_import_artifacts(archive_file =
 ↪  \"../outputs/my_pipeline/pipeline_outputs.nar\")'"
    else
      echo "No cached outputs found, will build from
 ↪  scratch"
    fi
```

After building, export the outputs so they can be reused:

```
- name: Export outputs to avoid rebuild
  run: |
    mkdir -p ../outputs/my_pipeline
    nix-shell --quiet --run "Rscript -e
 ↪  'rixpress::rxp_export_artifacts(archive_file =
 ↪  \"../outputs/my_pipeline/pipeline_outputs.nar\")'"
```

Finally, commit the cached outputs back to the repository:

```
- name: Push cached outputs
  run: |
    cd ..
    git config --global user.name "GitHub Actions"
    git config --global user.email
 ↪  "actions@github.com"
    git pull --rebase --autostash origin main
    git add outputs/my_pipeline/pipeline_outputs.nar
    if git diff --cached --quiet; then
      echo "No changes to commit."
    else
      git commit -m "Update cached pipeline outputs"
      git push origin main
    fi
```

This pattern ensures that only changed derivations are rebuilt on subsequent runs. The `.nar` file format is Nix's archive format and contains all the built outputs.

12.6.2 The Easy Way: `rxp_ga()`

If the above seems like a lot of boilerplate, `{rixpress}` provides a helper function that generates a complete GitHub Actions workflow for you:

```
rixpress::rxp_ga()
```

This creates a `.github/workflows/run-rxp-pipeline.yaml` file that handles everything: installing Nix, setting up Cachix, generating the environment and pipeline, importing and exporting artifacts, and storing them in a dedicated `rixpress-runs`

orphan branch. Using an orphan branch keeps your main branch clean while persisting the cached outputs between runs.

For most projects, running `rxp_ga()` once and committing the generated workflow file is all you need to get your pipeline running on CI.

12.7 GitHub Actions without Nix

If you're not using Nix, you'll have to set up GitHub Actions manually. Suppose you have a package project and want to run unit tests on each push. See for example the `{myPackage}` package, in particular this file. This action runs on each push and pull request on Windows, Ubuntu and macOS:

```
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  rcmdcheck:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest,
        macos-latest]
```

Several steps are executed, all using pre-defined actions from the `r-lib` project:

```
    steps:
    - uses: actions/checkout@v4
```

```
- uses: r-lib/actions/setup-r@v2
- uses: r-lib/actions/setup-r-dependencies@v2
  with:
    extra-packages: any::rcmdcheck
    needs: check
- uses: r-lib/actions/check-r-package@v2
```

An action such as `r-lib/actions/setup-r@v2` will install R on any of the supported operating systems without requiring any configuration from you. If you didn't use such an action, you would need to define three separate actions: one that would be executed on Windows, on Ubuntu and on macOS. Each of these operating-specific actions would install R in their operating-specific way.

Check out the workflow results to see how the package could be improved here.

Here again, using Nix simplifies this process immensely. Look at this workflow file from `{rix}`'s repository here. Setting up the environment is much easier, as is running the actual test suite.

12.8 Advanced patterns

Now that you understand the basics, let's look at some more advanced patterns that will make your CI workflows more efficient and informative.

12.8.1 Caching with Cachix

Building Nix environments from scratch on every CI run can be slow. Cachix solves this by providing a binary cache for your

Nix derivations. Once you build something, subsequent runs can download the pre-built binaries instead of rebuilding from source.

To use Cachix, you first need to create a free account at cachix.org and create a cache. Then, generate an auth token and add it as a secret in your GitHub repository settings (under Settings → Secrets and variables → Actions). Call it something like `CACHIX_AUTH`.

Here is a workflow that builds your development environment and pushes the results to your Cachix cache:

```
name: Update Cachix cache

on:
  push:
    branches: [main]

jobs:
  build-and-cache:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Install Nix
        uses:
↪   DeterminateSystems/nix-installer-action@main

      - uses: cachix/cachix-action@v15
        with:
          name: your-cache-name
          authToken: '${{ secrets.CACHIX_AUTH }}'
```

```
      - name: Build and push to cache
        run: |
          nix-build
          nix-store -qR --include-outputs
↪   $(nix-instantiate default.nix) | cachix push
↪   your-cache-name
```

The key line here is the `nix-store` command at the end. It queries all the dependencies of your build and pushes them to Cachix. The next time you or anyone else runs this workflow, the `cachix/cachix-action` will automatically pull from your cache, dramatically speeding up the build.

If you want to build on both Linux and macOS (since Nix binaries are platform-specific), you can use a matrix:

```
jobs:
  build-and-cache:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest]
```

12.8.2 Storing outputs in orphan branches

When running a pipeline on CI, you often want to keep the outputs (plots, data, reports) without committing them to your main branch. A clean solution is to store them in an orphan branch. An orphan branch has no commit history and is completely separate from your main code.

Here is the pattern:

```
- name: Check if outputs branch exists
  id: branch-exists
  run: git ls-remote --exit-code --heads origin
  ↪ pipeline-outputs
  continue-on-error: true

- name: Create orphan branch if needed
  if: steps.branch-exists.outcome != 'success'
  run: |
    git checkout --orphan pipeline-outputs
    git rm -rf .
    echo "Pipeline outputs" > README.md
    git add README.md
    git commit -m "Initial commit"
    git push origin pipeline-outputs
    git checkout -

- name: Push outputs to branch
  run: |
    git config --local user.name "GitHub Actions"
    git config --local user.email
  ↪ "actions@github.com"
    git fetch origin pipeline-outputs
    git worktree add ./outputs pipeline-outputs
    cp -r _outputs/* ./outputs/
    cd outputs
    git add .
    git commit -m "Update outputs" || echo "No
  ↪ changes"
    git push origin pipeline-outputs
```

This pattern first checks if the branch exists using `git ls-remote`. If not, it creates an orphan branch. Then it uses `git`

`worktree` to work with both branches simultaneously, copies the outputs, and pushes them. The `{rix}` and `{rixpress}` packages use this pattern to store pipeline outputs between runs.

12.8.3 Creating workflow summaries

GitHub Actions has a built-in feature for creating rich summaries that appear directly on the workflow run page. You write Markdown to a special file path stored in the `GITHUB_STEP_SUMMARY` environment variable.

```
- name: Create summary
  run: |
    echo "## Pipeline Results  " >>
 ↪  $GITHUB_STEP_SUMMARY
    echo "" >> $GITHUB_STEP_SUMMARY
    echo "| Metric | Value |" >>
 ↪  $GITHUB_STEP_SUMMARY
    echo "|--------|-------|" >>
 ↪  $GITHUB_STEP_SUMMARY
    echo "| Tests passed | 42 |" >>
 ↪  $GITHUB_STEP_SUMMARY
    echo "| Coverage | 87% |" >>
 ↪  $GITHUB_STEP_SUMMARY
```

You can also generate the summary dynamically from your R or Python code:

```
- name: Generate summary from R
  run: |
    nix-shell --run "Rscript -e '
      results <- readRDS(\"results.rds\")
```

```
    cat(\"## Analysis Complete\n\n\", file =
↪ Sys.getenv(\"GITHUB_STEP_SUMMARY\"), append =
↪ TRUE)
    cat(paste(\"Processed\", nrow(results),
↪ \"observations\n\"), file =
↪ Sys.getenv(\"GITHUB_STEP_SUMMARY\"), append =
↪ TRUE)
  '"
```

This is particularly useful for:

- Showing test results at a glance
- Displaying key metrics from your analysis
- Providing download links to artifacts
- Reporting any warnings or issues

The summary appears right on the Actions tab, making it easy for collaborators to see what happened without digging through logs.

12.9 Conclusion

This chapter introduced Continuous Integration with GitHub Actions, the final piece of our reproducible workflow.

Key takeaways:

- **Automation removes manual steps**: Every push triggers tests, builds, and deployments without human intervention
- **Nix simplifies CI setup**: The same `default.nix` you use locally works on GitHub Actions, eliminating "works on my machine" problems

- **Cachix speeds up builds**: By caching Nix derivations, subsequent runs avoid rebuilding unchanged dependencies
- **`rxp_ga()` handles the boilerplate**: One function call generates a complete workflow for running `{rixpress}` pipelines on CI
- **Artifact caching persists outputs**: Using `rxp_export_artifact` and an orphan branch, pipeline outputs survive between ephemeral CI runs

With continuous integration in place, your reproducible analytical pipeline is truly automated. Push your code, and GitHub takes care of the rest: running tests, building your environment, executing your pipeline, and storing the results. This frees you to focus on what matters: the analysis itself.

13 Conclusion: Adopting Reproducibility in the Real World

13.1 Introduction

If you have made it this far, you now have a powerful toolkit at your disposal. You understand how Nix provides truly reproducible environments, how functional programming leads to testable and predictable code, how packages bundle your work for sharing, and how CI/CD automates the boring parts. But knowing these tools is only half the battle. The harder question is: how do you actually start using them?

If you work alone, the answer is simple: just start. Pick one project and commit to doing it the "right" way. You will make mistakes, you will get frustrated, and you will learn. But if you work in a team, the challenge is different. You cannot simply mandate that everyone learns Nix by next Tuesday. Change management is a skill in itself, and this final chapter offers some practical strategies for introducing reproducible practices to yourself, your team, and your organisation.

The philosophy here is simple: show, don't tell. Nobody was ever convinced to adopt a new tool by a slideshow. They are

convinced when they see it solve a real problem.

13.2 Start with yourself

Before you try to convince anyone else, you need to become proficient yourself. Pick a small, low-stakes project and use it as your learning ground. A perfect candidate is a personal side project or an internal analysis that nobody else depends on.

Your goal in this phase is to build muscle memory. You want to reach the point where creating a `default.nix` file feels as natural as opening RStudio. You want to be able to troubleshoot common Nix errors without panic. You want to have a working CI pipeline that you understand inside and out.

This investment pays dividends later. When a colleague asks "Why is this so complicated?", you will have a ready answer because you asked yourself the same question two months ago.

13.3 The two-minute demo

Once you are comfortable, look for opportunities to demonstrate value. The best opportunities are problems that everyone recognises but nobody has solved.

"It works on my machine" is the classic. The next time a colleague struggles to run your code, don't just send them a fixed script. Send them a `default.nix` file and a one-liner: `nix-shell --run "Rscript analysis.R"`. When it works on the first try, you will have their attention.

Another opportunity is the dreaded "update everything and pray" scenario. When a project breaks after a package update, you can show how Nix lets you pin exact versions. "This analysis will run the same way in five years" is a powerful statement.

The key is to solve real problems, not hypothetical ones. Do not lecture your colleagues about reproducibility in the abstract. Show them how it saved you two hours of debugging on Tuesday.

13.4 Make it easy for others

If you want your team to adopt these practices, you need to lower the barrier to entry as much as possible. This means:

1. Write excellent documentation. Not for yourself, but for the colleague who has never heard of Nix. Assume they will spend five minutes reading before giving up.

2. Provide templates. Create a repository with a working `default.nix`, `gen-env.R`, CI workflow, and a simple example analysis. When someone starts a new project, they can clone this template and be productive immediately.

3. Automate the setup. If your organisation allows it, provide a script that installs Nix with a single command. The fewer steps, the fewer places where someone can get stuck.

4. Be available. When someone tries your approach and hits a wall, they need help within minutes, not days. If you are not responsive, they will abandon the effort and go back to their old ways.

13.5 Pick your battles

You cannot change everything at once. Be strategic about where you focus your energy.

Some projects are not worth the effort to convert. If an analysis was run once three years ago and will never be touched again, leave it alone. Focus on new projects and on projects that are actively maintained.

Some colleagues will never be convinced. That is fine. You do not need everyone on board. You need a few early adopters who will become advocates themselves. Focus on the curious ones, the ones who ask "how did you do that?" when they see your workflow.

Some organisations have constraints you cannot change. Maybe you cannot install Nix on the production server. Maybe you are stuck with Windows machines. Work within these constraints rather than fighting them. Docker can bridge many gaps. WSL2 makes Nix possible on Windows. Find the path of least resistance.

13.6 The gradual adoption path

Here is a realistic roadmap for introducing reproducibility to a team:

Month 1-2: Personal mastery. Use Nix and the tools from this book on your own projects. Get comfortable. Build your own templates and workflows.

Month 3-4: Show and tell. Start sharing your work. When you give a code review, mention that the project runs in a Nix

shell. When someone asks how to install a package, show them `rix()`. Let people discover the value organically.

Month 5-6: Pilot project. Propose using the full stack (Nix, tests, CI) for one team project. Pick something visible but not critical. Document everything. Make it a success.

Month 7-12: Expand. Use the pilot project as a template for new work. Gradually, "the new way" becomes "the way we do things here." Update onboarding documentation to include Nix setup.

This is slow. It takes patience. But lasting change always does.

13.7 Common objections and how to address them

"This is too complicated."

It is more complicated than doing nothing. But it is simpler than debugging environment issues at 3am before a deadline. Complexity is a trade-off. You are trading upfront learning for long-term reliability. Acknowledge the learning curve, but emphasise that you are there to help.

"We don't have time for this."

You do not have time not to do this. Every hour spent on environment issues is an hour not spent on analysis. Every "it works on my machine" is a risk. Frame reproducibility as a time investment, not a time cost.

"My code works fine without it."

It works today. Will it work in six months? Will it work when you hand it to a colleague? Will it work when the package maintainer pushes a breaking change? Reproducibility is insurance. You hope you never need it, but you are grateful when you do.

"I'll learn it later."

Later never comes. Start with one small project. Commit to using one new tool. You can learn incrementally. You do not need to master everything before you begin.

13.8 LLMs as adoption accelerators

Throughout this book, we have seen how LLMs can assist with writing tests, generating documentation, and even drafting functions. But there is a bigger point to make here: LLMs dramatically lower the barrier to adopting these practices.

The old objection to tools like Nix was that the learning curve was too steep. You had to understand a new language, a new paradigm, and a lot of unfamiliar syntax. This is still true to some extent. But with an LLM at your side, you no longer need to memorise everything. You can ask.

"How do I add a Python package to my `rix()` call?" "What does this Nix error message mean?" "Generate a GitHub Actions workflow that runs my tests in a Nix shell." These are questions that would have required hours of documentation diving or forum searching. Now they take seconds.

This changes the economics of learning. The boilerplate that used to be a barrier is now generated for you. The syntax you kept forgetting is a prompt away. The edge cases you never

encountered are handled by an assistant that has seen them all.

More importantly, LLMs make it easier to help your colleagues. When someone on your team struggles with a Nix expression, you do not need to be available at that exact moment. They can ask an LLM for help. They can generate a first draft and ask you to review it. The bottleneck shifts from "I need to learn everything" to "I need to understand enough to verify the output."

This is why the `pkgctx` tool exists. By feeding package documentation to an LLM, you give it the context it needs to generate idiomatic code for libraries like `{rix}`, `{rixpress}`, or `{chronicler}`. You are not replacing your expertise; you are amplifying it.

The combination of reproducibility tools and LLMs is powerful. The tools provide structure. The LLMs provide accessibility. Together, they make best practices achievable for teams that would never have adopted them otherwise.

13.9 A note on literate programming

This book has focused on the plumbing of reproducibility: environments, packages, pipelines, and automation. But there is another dimension we have not covered in depth: literate programming.

Literate programming is the practice of combining code, prose, and outputs into a single document. Tools like Quarto, R Markdown, and Jupyter notebooks make this possible. The idea is that the analysis and its documentation are the same thing. You cannot have one without the other.

This book itself was written with Quarto. Every code example runs. Every output is generated fresh on each build. This is literate programming in action.

If you are interested in this approach, there are excellent resources available. The Quarto documentation at quarto.org is a great starting point. For R Markdown, "R Markdown: The Definitive Guide" by Xie, Allaire, and Grolemund is comprehensive. The principles you have learned in this book apply directly: pin your environment with Nix, run your document rendering in CI, and your reports become just as reproducible as your analyses.

13.10 Final thoughts

Reproducibility is not a destination. It is a practice. You will never have a perfectly reproducible workflow because requirements change, tools evolve, and you learn new things. The goal is not perfection but continuous improvement.

Start where you are. Use what you have. Do what you can.

If this book has given you one new tool or one new idea, it has done its job. If it has changed how you think about your work, even better. And if you go on to share these practices with your colleagues and build a culture of reproducibility in your team, then you have not only learned but taught. That is the highest compliment.

Good luck. Stay curious. Build things that last.

References

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