R in production

Code is run repeatedly

Hadley Wickham

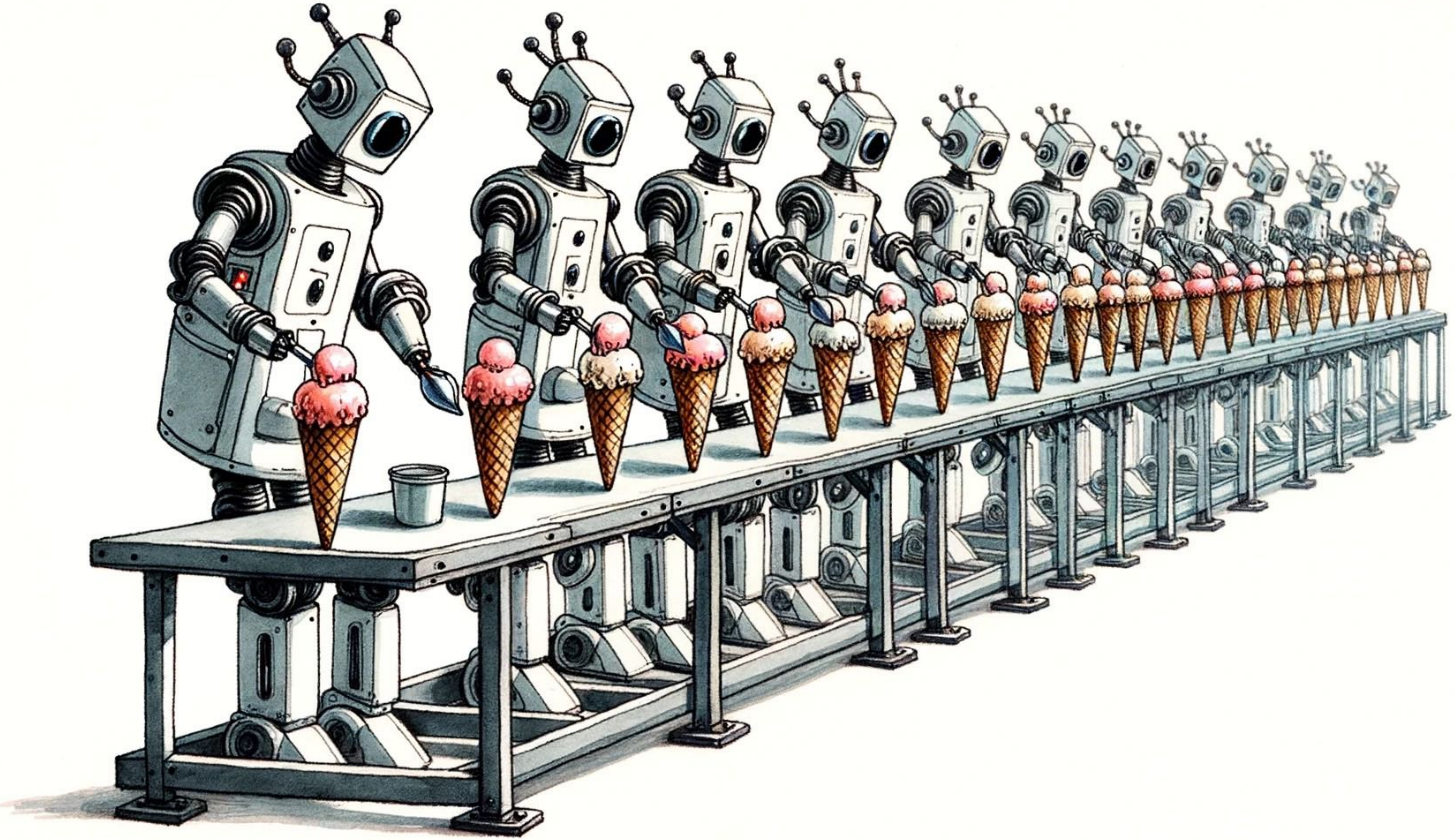
Chief Scientist, Posit

September 2025

The challenges

ICE CEAMI
CE CREAM
ICE

COMITORS SEAMAS
SAMEC
CEAMER
ICE CREAM SALES
CREMER DEAMINS
SEMBSUP.
HUMIDNITY
HOLIDDAYS
CICE MHRTTIOMT
UNTY CACTESSON
IDE ICREREM



The **data** changes

The **schema** changes

date	temperature
05/01/2024	64.4
05/02/2024	68.0
05/03/2024	71.6
05/04/2024	66.2
05/05/2024	69.8
05/06/2024	73.4
05/07/2024	68.0
05/08/2024	71.6

The **schema** changes

date	**temp**
2024-05-01	18
2024-05-02	20
2024-05-03	22
2024-05-04	19
2024-05-05	21
2024-05-06	23
2024-05-07	20
2024-05-08	22

What changed? How is it likely to affect your code?

date	temperature
05/01/2024	64.4
05/02/2024	68.0
05/03/2024	71.6
05/04/2024	66.2
05/05/2024	69.8
05/06/2024	73.4
05/07/2024	68.0
05/08/2024	71.6

date	temp
2024-05-01	18
2024-05-02	20
2024-05-03	22
2024-05-04	19
2024-05-05	21
2024-05-06	23
2024-05-07	20
2024-05-08	22

Change	Impact
column name	probably errors
date format	might just work might error
temperature units	garbage predictions

A **package** changes

The **platform** changes

R/Python
System libraries
Operating system
Architecture

The **universe** changes

Concept drift
Model drift
Data drift

A **requirement** changes

Hard to mitigate
Universe
Schema
Requirements
Unlikely to affect you
Likely to affect you
Packages
Platform
Data
Easy to mitigate

1. Platform
2. Packages
3. Schema
4. Requirements

Platform

Insulate yourself from platform changes with a container

- Defines operating system + system dependencies
- Isolated
- Portable
- Immutable
- Scalable

Useful containers to know about

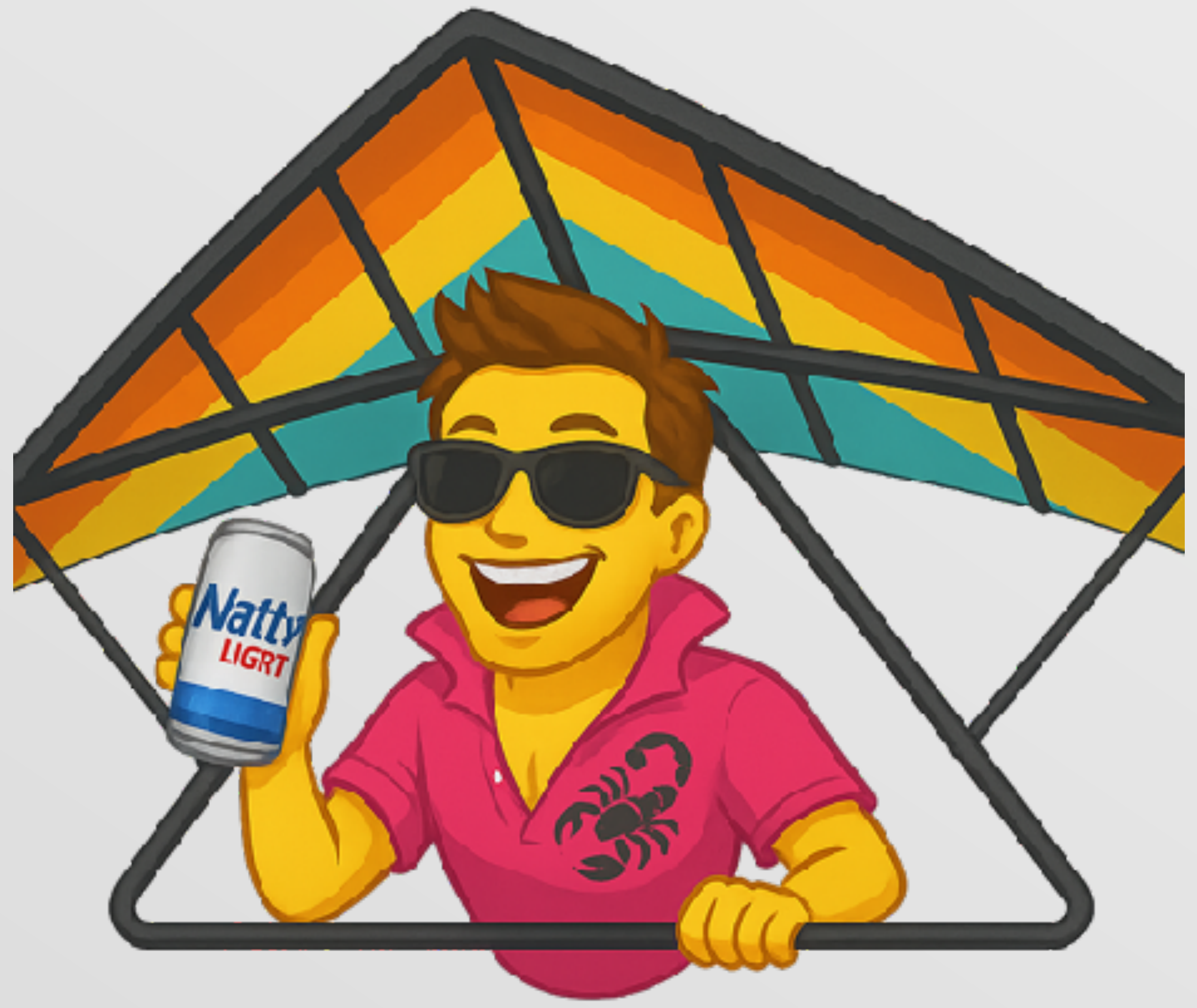
- GitHub action containers: https://docs.github.com/en/actions/writing-workflows/choosing-where-your-workflow-runs/choosing-the-runner-for-a-job
- https://github.com/rocker-org/rocker
- https://github.com/r-hub/rhub
- https://github.com/r-hub/evercran

We have never experienced a problem caused by using ubuntu-latest on GitHub

Packages

Version management

We mentioned three strategies earlier



YOLO

Just use CRAN

Pack it & ship it

`rsconnect::writeManifest()`

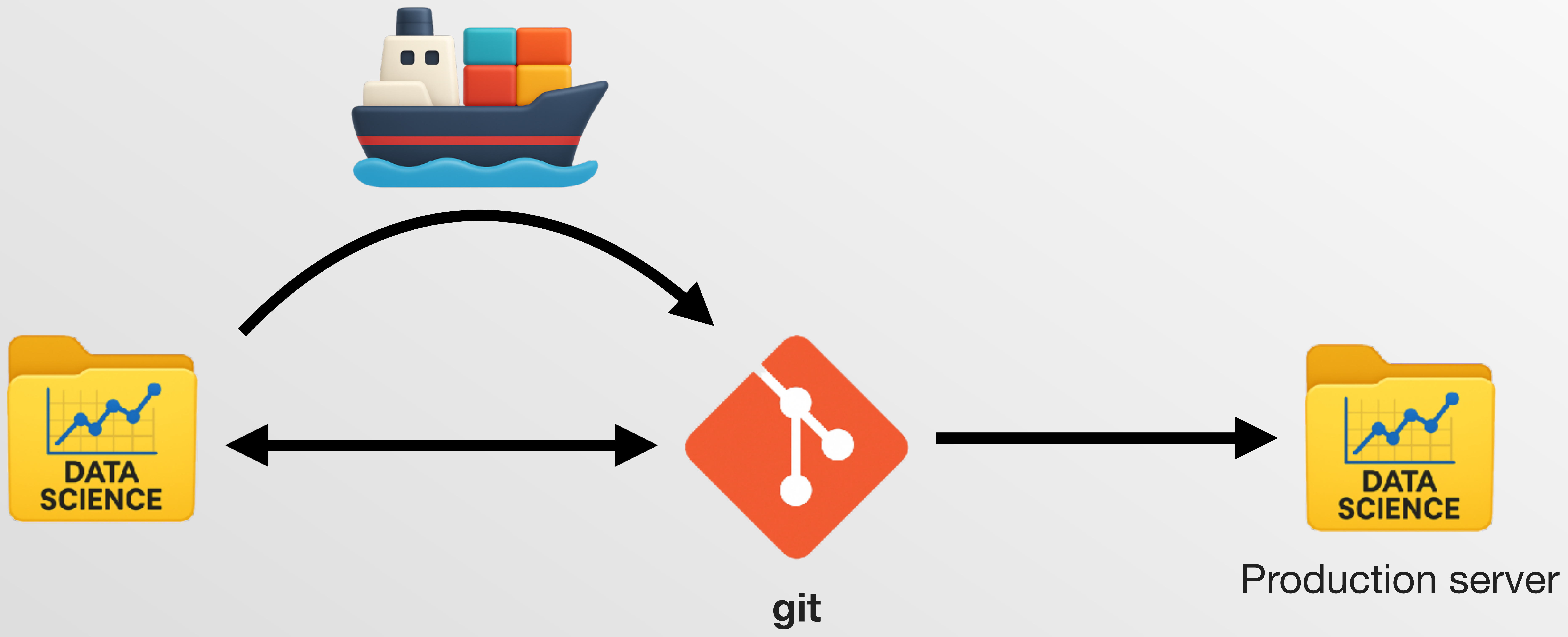
Freeze it

`renv::snapshot()`

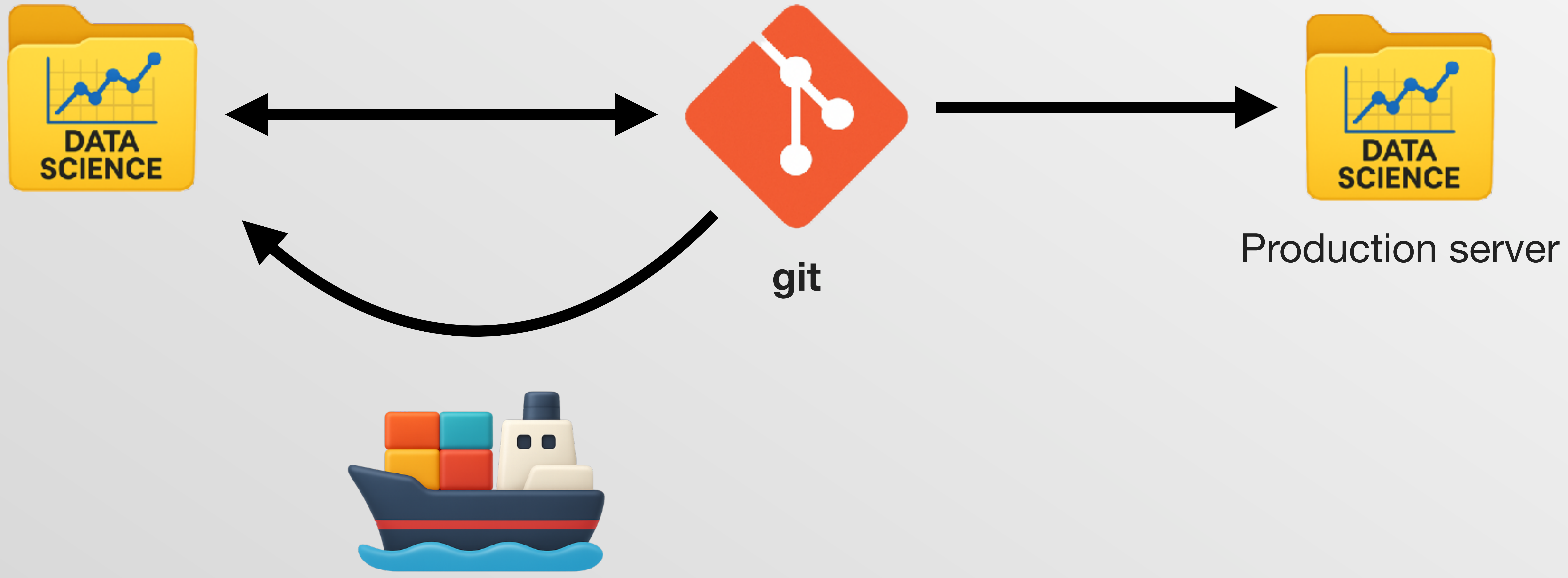
Pack it & ship it

- Ensures that deployed code continues to work regardless of how packages change. 100% needed.
- Handled automatically with click-button deploy, and manually with rsconnect::write_manifest() for other cases.
- But what about your colleagues? Or what happens when your code production code needs a change after a year and you can no longer even get it to run?

Pack it & ship it only goes one way



How do you get those versions back to your computer?



Solving this problem requires us to remember some vocab

Package

Library

Each library can only have a single version of an R package

- 📕 ggplot2 4.0.0
- 📗 dplyr 1.1.4
- 📘 tidyr 1.3.1

So if you want multiple versions you need multiple libraries

- 📕 ggplot2 4.0.0
- 📗 dplyr 1.1.4
- 📘 tidyr 1.3.1

- 📕 ggplot2 3.5.2
- 📗 dplyr 1.0.10
- 📘 tidyr 1.2.1

Then those libraries need names

.libPaths()[1]	~/myproject/library
📕 ggplot2 4.0.0	📕 ggplot2 3.5.2
📗 dplyr 1.1.4	📗 dplyr 1.0.10
📘 tidyr 1.3.1	📘 tidyr 1.2.1

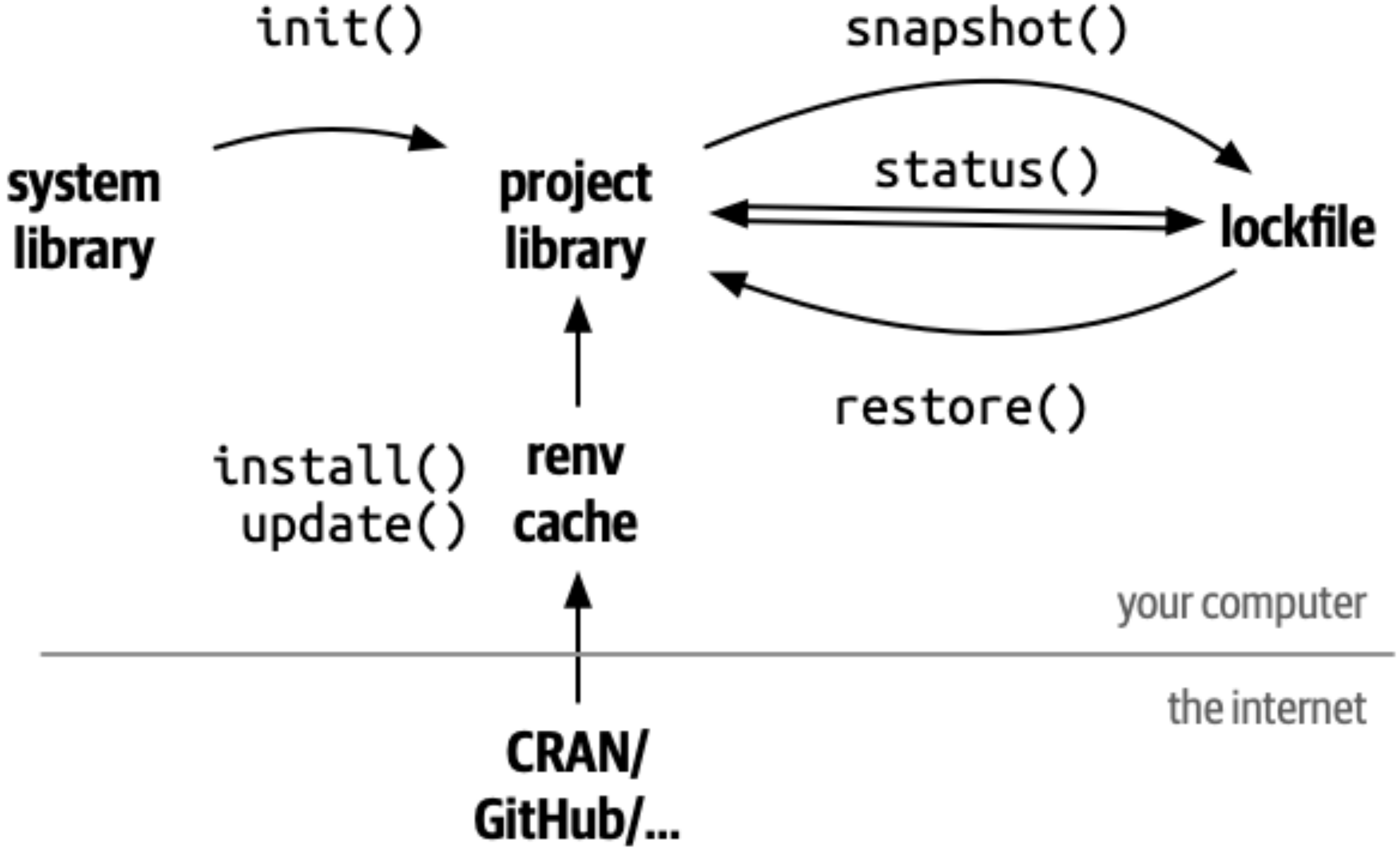
renv makes this as easy as possible

Using renv

```r
# Turn it on for a project once
# This will capture all current dependencies
renv::init()

# install/update packages as usual
pak::pak("tidyverse")
# OOPS YOU DIDN'T INSTALL PAK YET
install.packages("pak")
pak::pak("tidyverse")

# Record which versions are currently used
renv::snapshot()
```


init()
snapshot()
system library
project library
status()
lockfile
restore()
install()
update()
renv cache
your computer
the internet
CRAN/GitHub/...

Your turn

- Create a new project called penguins
- Initialize renv. Look at renv/library. Look at renv.lock
- Create a plot of the palmerpenguins dataset.
  - You'll need to install some packages.
  - After each install look at renv/library & renv.lock.
- Snapshot. Look at renv/library & renv.lock.

Your turn

Return to the **diamonds** project.

Call renv::init() to create a private library. Look at renv/library. Look at renv.lock.

Change setup-r-dependencies action to setup-renv.

Redeploy and check that your code works.

Add a new chunk that uses dplyr to show the five most expensive diamonds. What do you need to do to get this to work in production?

Project-specific libraries are a hassle

- Have to install package in EVERY project.
- Have to update packages in every project. This means it's easy to keep living with buggy or insecure packages.
- You have to remember which package features are available in which project.
- Unlike other programming languages, in R you can mostly survive with a single library because CRAN ensures that all packages work together.

So I think you should use them temporarily

```
# manifest.json captures package versions at some point in time
# If it doesn’t work on your computer:
renv::renv_lockfile_from_manifest("manifest.json", "renv.lockfile")
renv::restore()

# I highly recommend updating if you have the time
renv::update()
# Check/fix the code
renv::deactivate()
rsconnect::writeManifest()
```

How do you know if your code is still correct?

(Well unit testing, obviously, but what does that mean for analysis code?

Reducing version dependency

Right-sizing dependencies

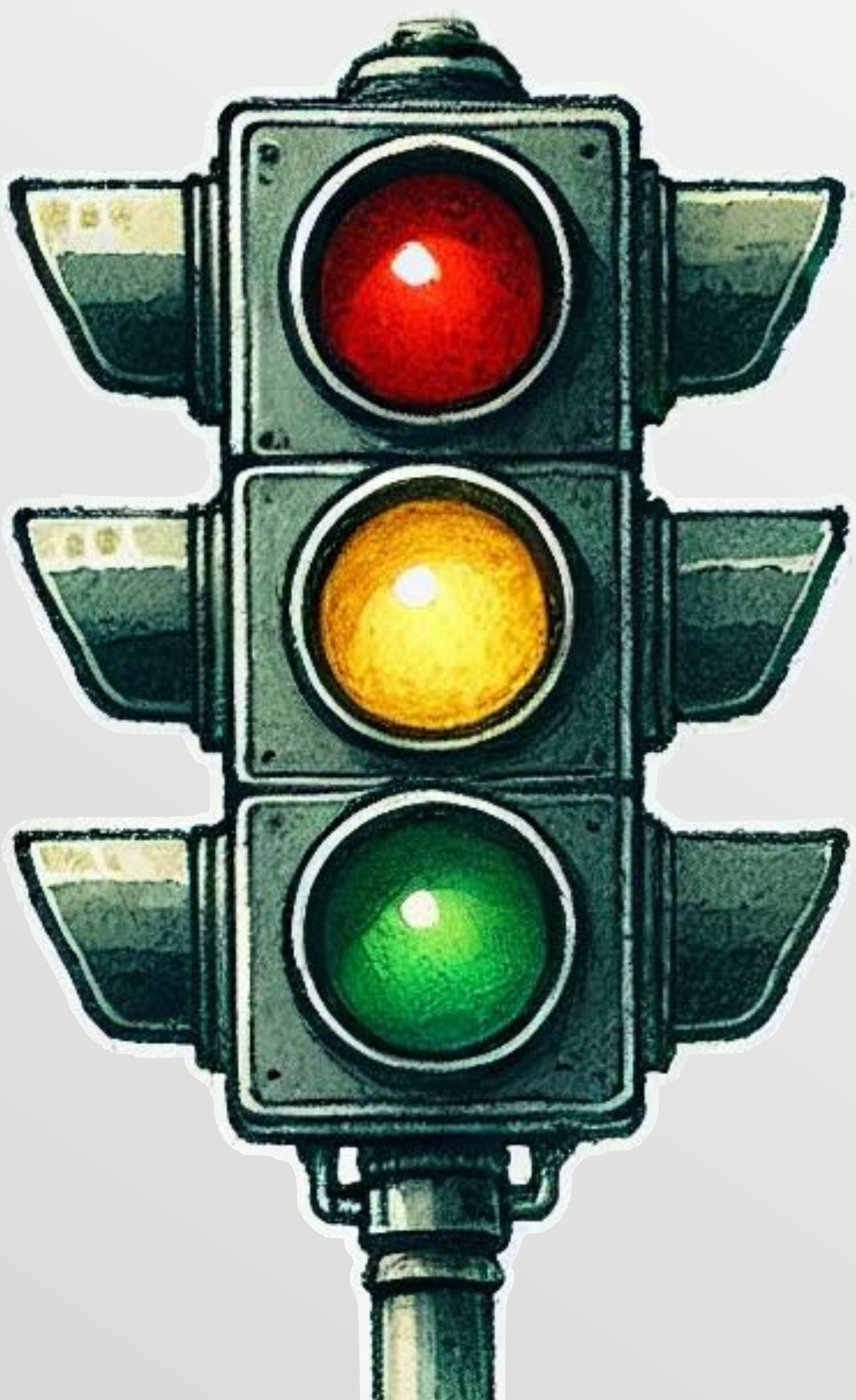
- I strongly believe you shouldn’t care at all about dependencies when you’re prototyping. It’s fine to take a dependency even if it’s one function that saves you 5 minutes.
- But more dependencies can make deployment more challenging, and can make your code more fragile.
- So when you’ve got to the point of having something to deploy it’s worth taking a look at your dependencies to see if there any that you can now shed.
- https://www.tidyverse.org/blog/2019/05/itdepends/

How could you reduce dependencies here?

```r
library(tidyverse)

create_silly_story ← function(name, animal, color, food, place) {
  str_c(
    "Once upon a time, there was a magical ", animal, " named ", name, ".\n",
    name, " had a beautiful ", color, " mane that shimmered in the sunlight.\n",
    "Every day, ", name, " would prance through the fields of ", food, " near ", place,
".\n",
    "One day, ", name, " discovered a secret portal that led to a world made entirely
of ", food, "!\n",
    name, " lived happily ever after, munching on ", food, " and spreading ", color, "
joy wherever ", name, " went.\n"
  )
}
```

R’s condition hierarchy



Error

You must fix this before continuing

Warning

We’ll let it go this time, but you need to fix this

Message

FYI; no action needed

But still worth eliminating to make logs easier to read

Eliminate all warnings and messages

```
options(warn = 1)
options(warn = 2)

# https://lifecycle.r-lib.org/
options(lifecycle_verbosity = "warning")
options(lifecycle_verbosity = "error")

# Tools of last resort
suppressWarnings()
suppressMessages()
```

Eliminate all messages and warnings in this code

```
library(dplyr)
library(readr)

df1 ← data.frame(x = c(1, 1, 2), y = 1:3)
df2 ← data.frame(x = c(1, 2, 2), z = letters[1:3])
left_join(df1, df2)

path ← tempfile()
write_csv(df1, path)
read_csv(path)
```

Schema

Mitigation strategy

- Social
  - Make friends with the folks who provide your data!
  - Establish data contracts
- Technical
  - Aggressively check that your data looks as expected

Pointblank provides a flexible tool for validating data

There are six ways to use it

- Canned data report
- Data quality reporting
- Pipeline data validation
- Expectations in unit tests
- Custom control flow
- Rmd integration

We’ll focus on two

- **Canned data report**
- Data quality reporting
- **Pipeline data validation**
- Expectations in unit tests
- Custom control flow
- Rmd integration

scan_data() produces a handy report

```r
report ← pointblank::scan_data(mtcars)
report
pointblank::export_report(mtcars, "mtcars-report.html")

# Interactions/correlations sections are slow for large datasets
# so you can drop them with this code
pointblank::scan_data(ggplot2::diamonds, sections = "OVMS")
```

Pipeline validation throws an error if data isn’t as you expect

- Check variable types:
  col_is_numeric(), **col_is_character()**, ...
- Check missingness (if you don’t expect any):
  col_vals_not_null()
- Check ranges/valid values:
  col_is_between(), **col_vals_in_set()**
- Special purpose:
  col_vals_expr(), **col_vals_regex()**
- Custom: **specially()**
  e.g. specially(\(df) nrow(anti_join(df, ref) == 0)

What would you validate for the ice cream data?

```r
library(dplyr)

data ← tribble(
  ~date,           ~temperature,
  "2024-05-01",  64.4,
  "2024-05-02",  68.0,
  "2024-05-03",  71.6,
  "2024-05-04",  66.2,
  "2024-05-05",  69.8,
  "2024-05-06",  73.4,
  "2024-05-07",  68.0,
  "2024-05-08",  71.6
)
data ← data ▷ mutate(date = as.Date(date))
```

Some ideas

```
library(pointblank)

data ▷
  col_is_date(date) ▷
  col_is_numeric(temperature) ▷
  col_vals_not_null(date) ▷
  col_vals_between(temperature, 30, 120)
```

Your turn

```
# You can download USD ⟷ EUR exchange range data from this API
url ← "https://data-api.ecb.europa.eu/service/data/EXR/
D.USD.EUR.SP00.A?
format=csvdata&detail=dataonly&startPeriod=2024-08-08"

# Assume you want to download this data daily.
# Write some pointblank code to ensure that you get an
# error if the data format changes
```

Requirements

No great insights, but...

- Adopt the mindset that a successful project will attract changes and will require upkeep.
- If you are using GitHub (or similar) internally, issues and projects are a great way to track desired changes.
- Do your best to **batch** and **time box** projects.
- Plan to regularly invest time in refactoring.

What is refactoring?

- Rewriting code so the reduces are the same but the internals are better: easier to read or easier to maintain.
- Second order benefit is improving your programming skills.
- Common tasks:
  - Enforce common style
  - Fix any kludges you added in the heat of the moment
  - Reduce code by using packages

Your turn

What other techniques for handling changing requirements have you found useful in your career?