

R in production

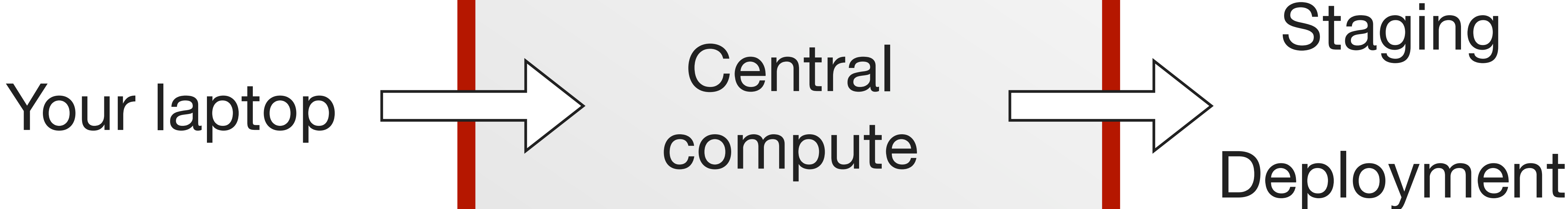
Code is run on another server

Hadley Wickham

Chief Scientist, Posit

September 2025





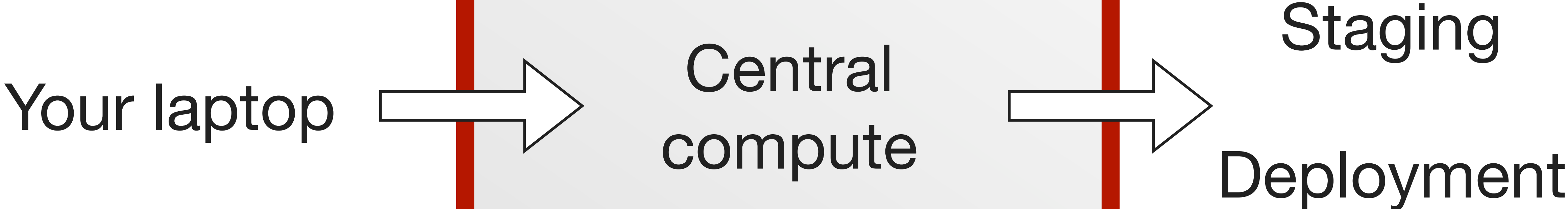
RStudio IDE

Workbench

Connect

Development

Deployment



Windows/Mac

Linux

Interactive

Batch

Desktop

Server

User auth

Service auth

1. Paper cuts

2. Packages

3. Logging

4. Authentication

Paper cuts

Your laptop

Central compute

	Windows	Linux
Line endings	\r\n	\n
Character encoding	Native	UTF-8
Path separator	\	/

Your laptop

Central compute

	Desktop	Server
Time zone	varies	UTC
Locale	various	C
Fonts	Many	Few
Graphics devices	quartz / windows	cairo

Your laptop

Central compute

	Desktop	Server
Time zone	lubridate / clock	
Locale	readr / stringr / forcats	
Fonts	systemfonts	
Graphics devices	ragg	

Packages

1. Understand the difference between desktop and server package installs.
2. Understand how to get the same package versions in development and deployment environments.

Package installation

Use pak

- Regardless of where you're installing packages, we recommend using pak
- It's **safe**: works out a plan and tells you about it, and it protects you against common failure modes
- It's **convenient**: pak can also install from GitHub, GitLab, Bioconductor; can install older versions of packages.
- It's **fast**: downloads and installs packages in parallel, and caches packages locally to make switching versions fast

Use CRAN

- For mac and windows, CRAN provides self-contained binary packages for the latest version of R:
 - **Self-contained:** you don't need to install anything else
 - **Binary:** CRAN has done as much work as possible so installing the package is fast.
- Life is easy!
- But what about Linux?

Use P3M

- P3M = Posit public package manager
- Provides binaries for mac, windows, and 12 linux distros. (For current and four previous versions of R.)
- These are not* self-contained so you still need system dependencies

CentOS 7 (x86_64)
Rocky Linux 9 (x86_64, arm64)
Rocky Linux 10 (x86_64, arm64)
OpenSUSE 15.6 (x86_64)
Red Hat Enterprise Linux 7 (x86_64)
Red Hat Enterprise Linux 8 (x86_64)
Red Hat Enterprise Linux 9 (x86_64, arm64)
Red Hat Enterprise Linux 10 (x86_64, arm64)
SLES 15 SP6 (x86_64)
Ubuntu 22.04 (Jammy) (x86_64)
Ubuntu 24.04 (Noble) (x86_64, arm64)
Debian 12 (Bookworm) (x86_64)

But pak will report needed system dependencies

→ Will install 101 packages.

→ Will download 31 CRAN packages (34.93 MB), cached: 70 (33.19 MB).

[...]

+ yaml 2.3.10 [bld][cmp][dl] (94.76 kB)

✖ Missing 11 system packages. You'll probably need to install them manually:

+ libcurl4-openssl-dev - curl

+ libfontconfig1-dev - systemfonts

+ libfreetype6-dev - ragg, systemfonts, textshaping

+ libfribidi-dev - textshaping

+ libharfbuzz-dev - textshaping

+ libjpeg-dev - ragg

+ libpng-dev - ragg

+ libssl-dev - curl, openssl

+ libtiff-dev - ragg

+ libxml2-dev - xml2

+ pandoc - knitr, reprex, rmarkdown

Use P3M + manylinux

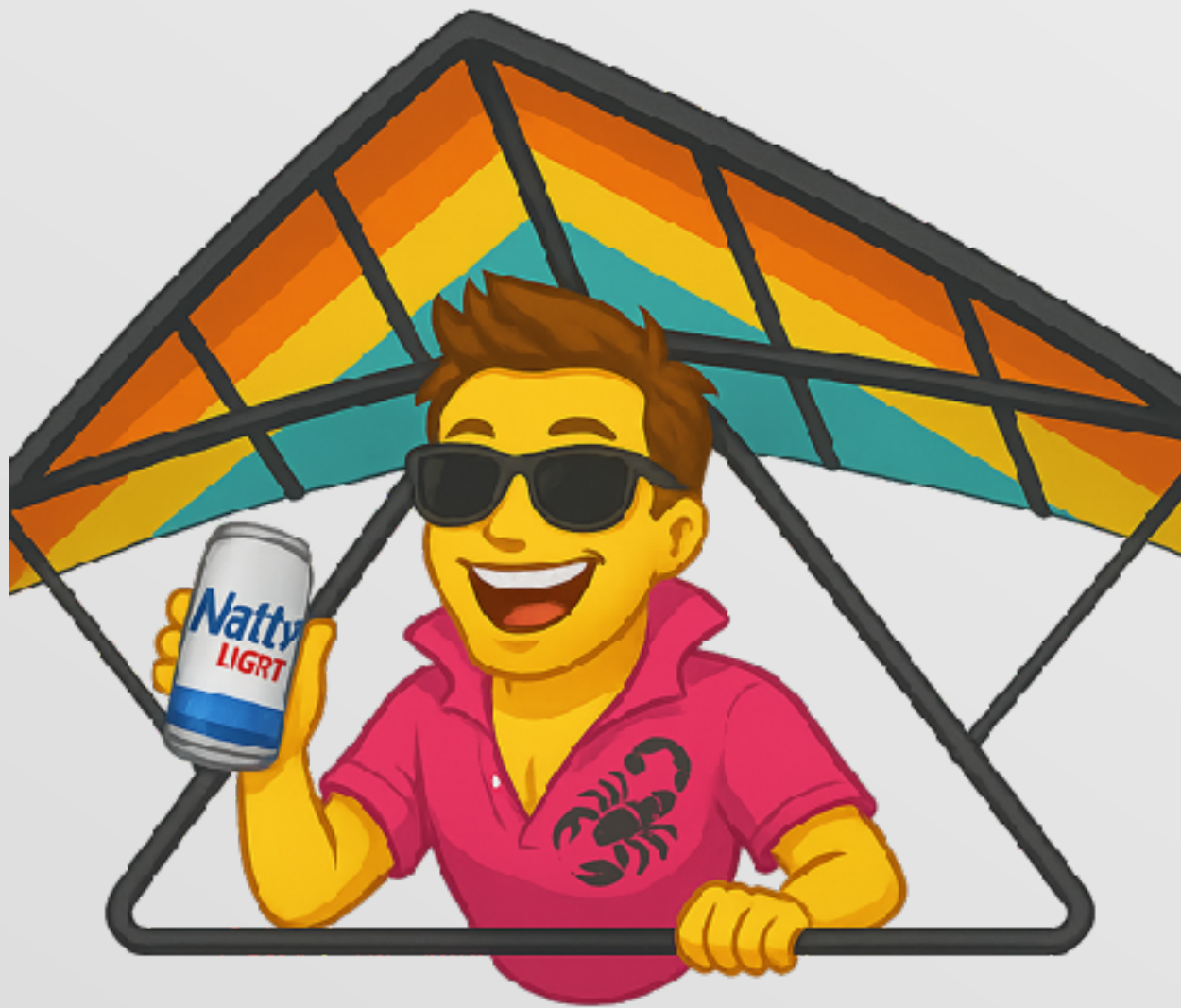
- Experimental new technique launched June 2025
- Self-contained binaries that work on a wide variety of linux platforms
- (Big advantage if you're in an environment where you need to ask a system admin permission to install new system dependencies)
- Still in preview mode, but initial feedback has been positive

To use P3M + manylinux

```
options(repos = c(  
  CRAN = "https://p3m.dev/cran/__linux__/manylinux_2_28/latest",  
))
```

Matching versions

There are three strategies



YOLO

Just use CRAN



Pack it & ship it

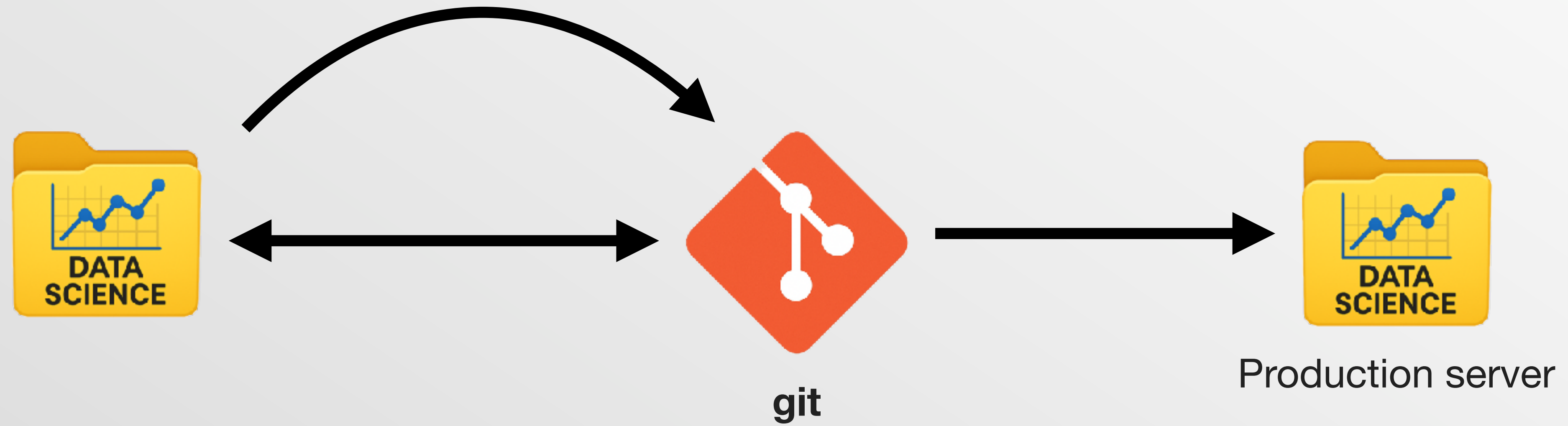
```
rsconnect::writeManifest()
```



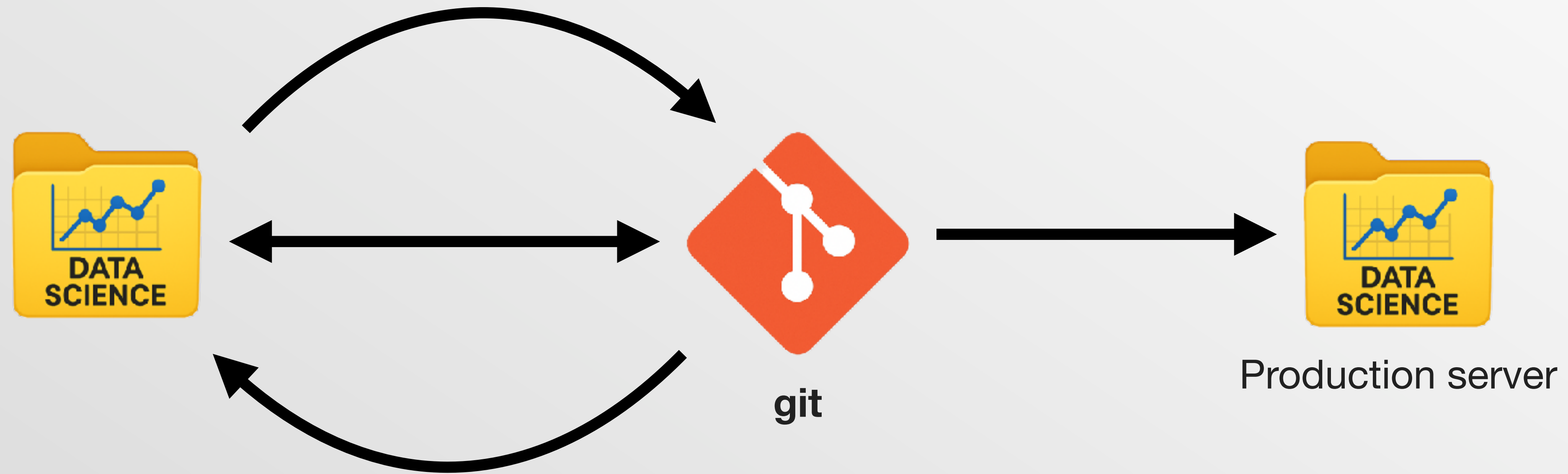
Freeze it

```
renv::snapshot()
```

Pack it & ship it works great if it's just you



But how do you go the other way?



We'll come back to this later

Your turn

- Open manifest.json in your madlibs project.
- What information does it contain about each package?
- What non-package info does it contain?
- Install a package from GitHub
(e.g. `pak::pak("hadley/useself")`)
- Load it in your project, then re-call `rsconnect::writeManifest()`. What additional information does it capture about Github packages?

Logging

Debugging is hard locally, but even worse at a distance



Debugging tips

- Iteration time is long and you can't browser() so you need a different strategy.
- Worth spending some time to brainstorm multiple hypotheses about what is going wrong. Then design an experiment so you can accept/reject multiple at once.
- Utterly mystified? Take a step back and confirm one by one.
- (Can you make it interactive? <https://github.com/r-hub/actions/tree/main/ssh-server>)

Make your life easier with logging

- You know debugging is hard.
- You know your code isn't perfect.
- So maybe you should include some breadcrumbs to make debugging as easy as possible when you inevitably hit a problem?

Logging basics

```
cat("This is a message\n")
```

```
# Or in Rmd/qmd
```

```
cat("This is a message\n", file = stderr())
```

```
# If there's a progress bar, will also need leading \n
```

```
cat("\nThis is a message\n", file = stderr())
```

```
# Useful tips
```

```
cat(strep("-", 100), "\n", file = stderr())
```

```
cat("🐱\n", file = stderr())
```

Your turn

Add logging to your madlibs shiny app. Verify that it works locally. (Have a go at first, but if you get stuck I've included some helper code on the next slide.)

Redeploy it and verify that you can view the logs. What happens if you have multiple shiny apps running at the same time? (i.e. open the same app in another tab). Is there one log or one log per app?

Logging sample

```
generate_story ← function(noun, verb, adjective, adverb) {  
  story ← glue::glue(  
    "Once upon a time, there was a {adjective} {noun} who loved to ",  
    "{verb} {adverb}. It was the funniest thing ever!"  
  )  
  cat(story, file = stderr())  
  story  
}
```

You can also use a package

```
library(logger)

log_info("🛩️ Script starting up ... ")
log_info("Processing {nrow(df)} rows")
log_warn("❌ Missing data for {length(problems)} variables")
log_info("🛩️ Completed; wrote {length(files)} files")

# This makes it easier to control verbosity
log_threshold(WARN)
log_info("🛩️ Script starting up ... ")
log_info("Processing {nrow(df)} rows")
```


Logging hints


- knitr chunk labels = free logging for Rmd/qmd
- Use emoji 🤯. They work everywhere and make it easier to quickly skim 💣 for important messages 🙈.
- Log before and after steps that take a long time.
- Log brief description of the data you're working on.
- Example at <https://github.com/hadley/houston-pollen/actions/runs/10101725281/job/27935890005>

Otel demo

2-otel.R

Authentication

There are two basic approaches to authentication

Encrypted env vars	Federated auth
Everywhere	Posit Connect
You	Your IT department
<code>Sys.getenv()</code>	

How to set an environment variable

Your laptop	<code>usethis::edit_r_environ()</code> User specific
GitHub	Go to repository > Settings > Secrets and Variables > Actions > Repository secrets (not environment variables!) AND add to action
Connect cloud	Go to content > Variables sidebar
Connect	Go to content > Settings menu > Vars

Write only

Don't set with `sys.setenv()`

This will be recorded in your `.Rhistory`, which is easy to share accidentally

How to get an environment variable

- You can't ever see these env vars again.
- But you can access them from code with `Sys.getenv()`.
- If you accidentally print a secret, GHA & Connect Cloud will automatically redact it.
- You can deliberately work around this but you shouldn't!

Scraping news data

```
library(httr2)
```

```
date ← Sys.Date() - 1
```

```
req ← request("https://newsapi.org/v2/everything") ▷
```

```
  req_url_query(
```

```
    q = '`"data science"`',
```

```
    from = date,
```

```
    pageSize = 10,
```

```
    apiKey = Sys.getenv("NEWS_API_KEY")
```

```
  )
```

```
req_perform(req, path = paste0("data/", date, ".json"))
```

Your turn

- Sign up for a news api key at <https://newsapi.org/>.
- Record the key in .Renviron and restart R.
- Check that the code from the previous slide works.
- Create a new GitHub action that you can run on demand. It should save the json into data/year-month-day.json.
- Set up a NEWS_API_KEY secret in GitHub
- (See 3-scrape.yaml if you get stuck)

Stretch goals

- Add logging.
- Make the json prettier.
- Make it download all new records since the last time it was run.
- Create a shiny app that lets the user select the search term and deploy it to connect cloud.

Automatic backfill

- Useful to do “automatic backfill” so if your script fails one day, the next day it will try again.
- Two basic approaches:
 - Create a todo vector and a done vector then use `setdiff()`.
 - Use a for loop and skip the iteration if the file already exists (e.g. houston-pollen).
- Use targets for more sophisticated workflows.

Who is authenticating?

- If you supply the env var, the script/app will act on your behalf.
- Often want to use a **service account**, an account that isn't associated with a person, but with a group of users (i.e. your data science team). You'll typically file a ticket with your IT department to get this set up.
- What happens if you want viewer based authentication? This is hard and varies from service to service. Supported in Posit Connect. See e.g. <https://docs.posit.co/connect/cookbook/content/integrations/databricks/viewer/r/>