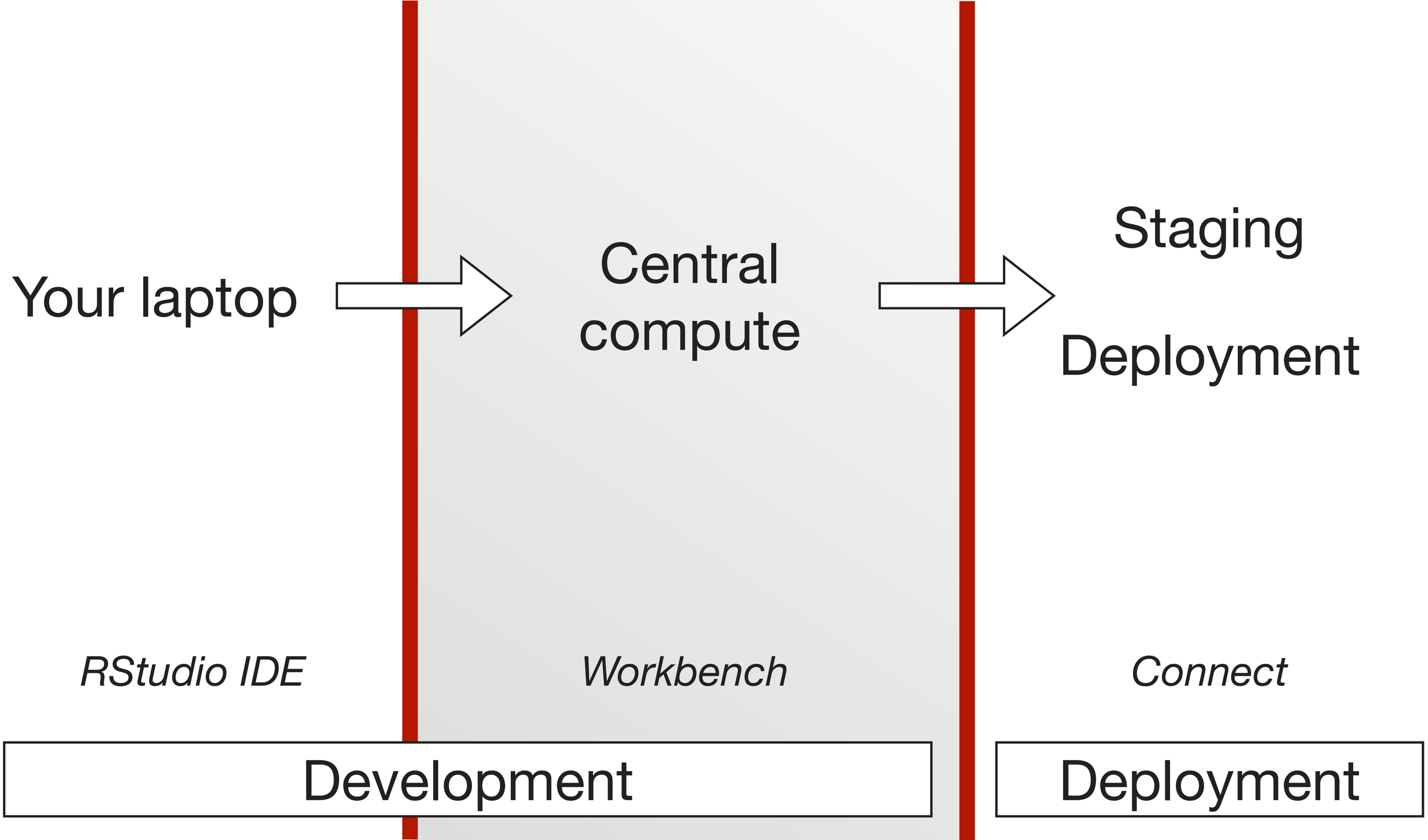
R in production

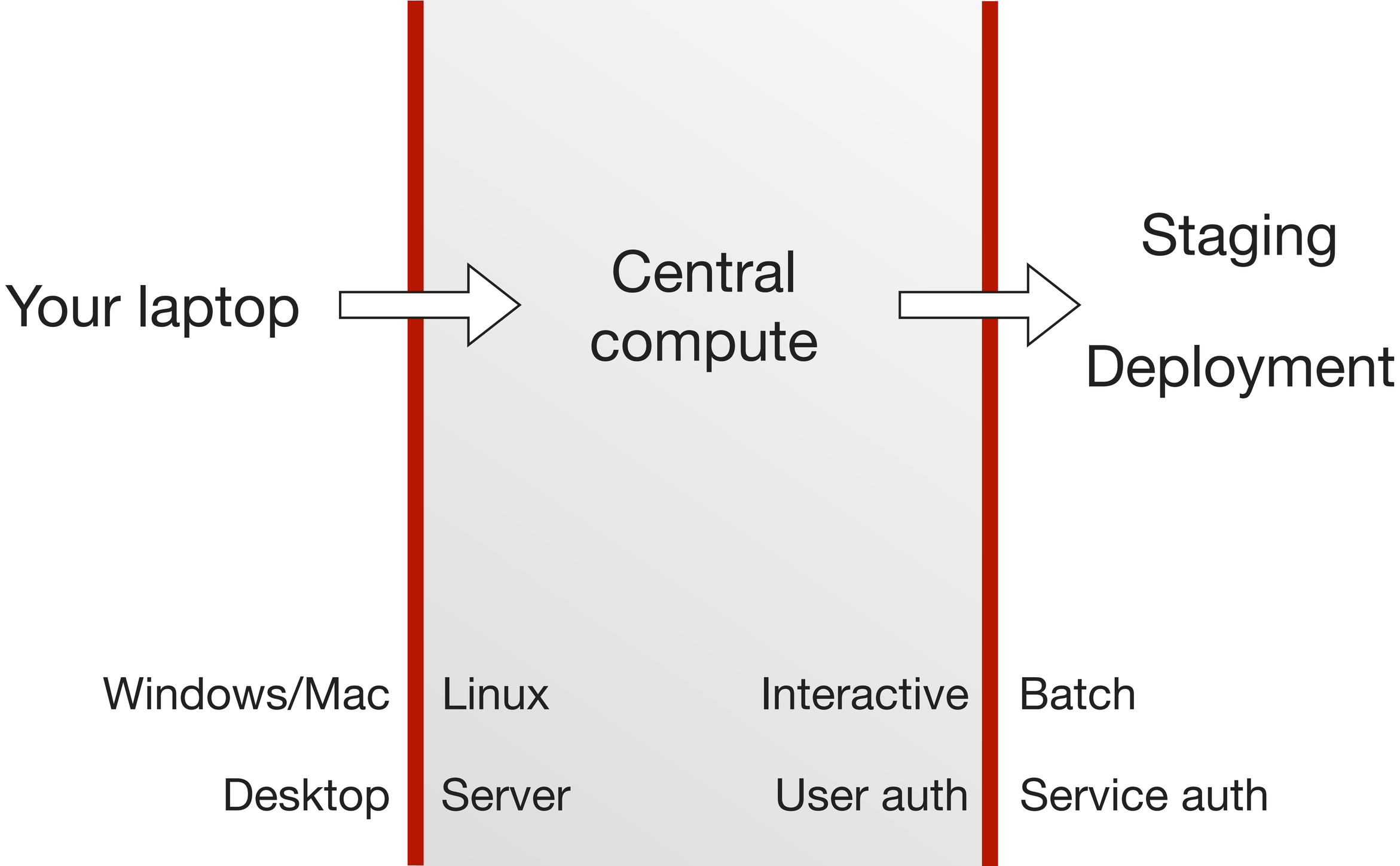
Running code on another machine

Hadley Wickham

Chief Scientist, Posit

August 2024


Your laptop
Central compute
Staging
Deployment
RStudio IDE
Workbench
Connect
Development
Deployment


Your laptop
Central compute
Staging
Deployment
Windows/Mac
Linux
Interactive
Batch
Desktop
Server
User auth
Service auth

1. Minor frustrations
2. Packages
3. Debugging & logging
4. Authentication

Minor frustrations

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	Ask your admin!	
Graphics devices	ragg	

Packages

(we’ll also cover in multiple times and shared responsibility)

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

Library

Important vocabulary

- What’s the difference between a source and binary package?
- What is a system dependency?

A binary package is produced from a source package by:

- Parsing all the R code and saving it as a single .rds file.
- Building the vignettes to generate .html versions.
- Bundling all data files into a single lazy-loaded database.
- **Compiling all C and C++ source code into a platform specific binary code.**

What is a system dependency?

- Many R packages use code from existing C and C++ libraries. e.g. httr2 -> curl -> libcurl, rvest -> xml2 -> libxml.
- On mac and windows, these are *statically linked*, which means that they're bundled with the packages that you download.
- On linux, they're either *dynamically linked* (if you use P3M) or you need to build from source (if you use CRAN). In either case, you have to install them.
- (But a typical server user won't have permissions to do that)

Package installs on Linux servers

- Use https://packagemanager.posit.co to get binaries.
- You'll also need system libraries. Your admin will need to install, but pak will tell you what's missing.

```
pak::pkg_sysreqs("tidyverse")
pak::pkg_sysreqs("tidyverse", sysreqs_platform = "ubuntu")
pak::pkg_sysreqs("devtools", sysreqs_platform = "centos")
```

But pak will report automatically

```
→ 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
```

Desktop

Server

	Desktop	**Server**
Package type	Binary (CRAN)	Source (CRAN) Binary (PPM)
System dependencies	Bundled in package	Outside of package Must be installed
Libraries	One per R version	One per user per R version

Three approaches to match deployment and development packages

- DESCRIPTION + pak::pak(".") — live life on the edge!
- rsconnect::writeManifest() — capture dependencies when you deploy
- renv::snapshot() — lock dependencies for eternity. We'll come back to that a bit 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 rsconect::writeManifest().
  What additional information does it capture about Github packages?

Debugging & logging

Your turn

```
# Deploy a quarto doc containing the following code
# More steps on following slide
# What do you learn when it fails?

```{r}
f <- function() g()
g <- function() h()
h <- function() i()
i <- function() stop("Error")
```

```{r}
f()
```
```

Your turn with more steps

```
create_project("~/Desktop/quarto-fail")
use_git()
use_github()
# create qmd file and save it and commit it
rsconnect::writeManifest()
# push to GitHub
# deploy to connect cloud
```

rlang::back_trace() vs traceback()

```
# rlang::back_trace()

Error in `i()`:
! Error
  ▉
 1. └─global f()
 2.   └─global g()
 3.     └─global h()
 4.       └─global i()
```

```
# traceback()

5: stop("Error") at #1
4: i() at #1
3: h() at #1
2: g() at #1
1: f()
```

Backtrace recommendations

- I recommend using rlang’s backtraces instead of traceback().
- You can add this to errors automatically by calling rlang::global_entrace() — this is done for you in knitr and positron.
- You may want/need options(rlang_backtrace_on_error = "full")
- Sometimes useful to do print(rlang::back_trace())

Lazy evaluation makes life more complicated

```
f ← function() g()
g ← function() h()
h ← function() i()
i ← function() stop("Error")

a ← function(x) b(x)
b ← function(x) c(x)
c ← function(x) x + 1

a(f())
```

Overall strategy

- 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.
- https://github.com/daroczig/logger/pull/171

Why log?

- 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

```r
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(\n"This 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 it 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
}

# Full disclosure: I wrote this with Claude
```

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")
```

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

Authentication

There are two basic approaches to authentication

Encrypted env vars	**Federated auth**
Everywhere	Posit Connect
You	Your IT department
Sys.getenv()	🪄✨

How to set an environment variable

Your laptop	usethis::edit_r_environ()
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)
req <- request("https://newsapi.org/v2/everything") |>
  req_url_query(
    q = '`"data science"`',
    from = Sys.Date() - 1,
    pageSize = 10,
    apiKey = Sys.getenv("NEWS_API_KEY")
  )

resp <- req_perform(req)
resp_body_json(resp)
```

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.
- **Stretch goal:** add logging
- **Stretch goal**: make it download all new records since the last time it was run.
- **Stretch goal**: create a shiny app that lets the user select the search term and deploy it to connect cloud.

What if you need a file?

```r
# Create a unique key
key <- httr2::secret_make_key()
key
# That's the environment variable you're going to use

# Then check in an encrypted version of the file
httr2::secret_encrypt_file(path, "MY_KEY")

# And decrypt when needed
decrypted <- httr2::secret_decrypt_file(path, "MY_KEY")
```

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 to authenticate with user credentials? This is hard and varies from service to service. Posit Connect is working on this sort of seamless user-auth for Databricks and Snowflake.