# Interactive Data in R

## Shiny, Leaflet, and what comes after

Zachary Hoylman
Montana Climate Office, University of Montana
zachary.hoylman@umontana.edu

November 9, 2026

## Table of contents

Before class: setup ..... 1
How to use this document ..... 2
1 Why build apps, and why now ..... 2
1.1 The one diagram that matters ..... 2
1.2 The LLM argument ..... 3
2 Shiny from zero ..... 3
2.1 The Old Faithful app ..... 3
3 Shiny for exploring data ..... 5
3.1 A short note on the pipe ..... 7
4 Maps with Leaflet ..... 8
4.1 Two other mapping tools worth knowing ..... 9
5 Leaflet and Shiny together: live earthquakes ..... 10
5.1 First attempt: the obvious way ..... 10
5.2 Second attempt: put each box where it belongs ..... 11
6 Click the map, query the climate ..... 14
6.1 Tool 1: the precipitation downloader ..... 14
6.1.a The same app with an interactive plot ..... 20
6.2 Tool 2: how dry is it right now? ..... 25
7 The other end of the spectrum: drought.climate.umt.edu ..... 29
Sharing what you built ..... 30
Appendix: reading an app you did not write ..... 30
Resources ..... 30

## Before class: setup

You have two files: this document (`interactive_data_in_R.qmd`) and `setup.R`. That is everything. Do these steps before class, not during. They take about ten minutes, most of it waiting.

1. **Install R (4.3 or newer) and RStudio.** On Windows, open a terminal and run:

```
winget install RProject.R Posit.RStudio
```

Or download both from https://posit.co/download/rstudio-desktop/. You do **not** need Rtools. Every package we use comes pre-built for Windows.

2. **Put both files in one folder** that is not inside OneDrive, for example `C:\Users\you\interactive-data`. OneDrive-synced folders make R package installs very slow on Windows.
3. **Open `setup.R` in RStudio and click Source** (top right of the editor). It installs the packages, checks that your laptop can reach the data we use, and draws a map when it is done. Four `OK`s and a map means you are ready.
4. **Open `interactive_data_in_R.qmd` in RStudio.** This is where you will work during class.

If step 3 fails, copy the red text and email it to me before class. If you cannot install software on your laptop at all, make a free account at posit.cloud, click **New Project > New RStudio Project**, upload both files with the **Upload** button in the Files pane, and do steps 3 and 4 there. Everything works the same in the browser.

### How to use this document

Every app is a single code chunk. Put your cursor in it and press **Ctrl+Shift+Enter** (Cmd+Shift+Enter on a Mac), or click the green arrow at the top right of the chunk. The app opens in a new window.

To change an app: close its window (or press Esc in the console), edit the chunk, and run it again. That loop, edit and rerun, is how you will work through every "Try it" box. Break things on purpose. You cannot hurt anything.

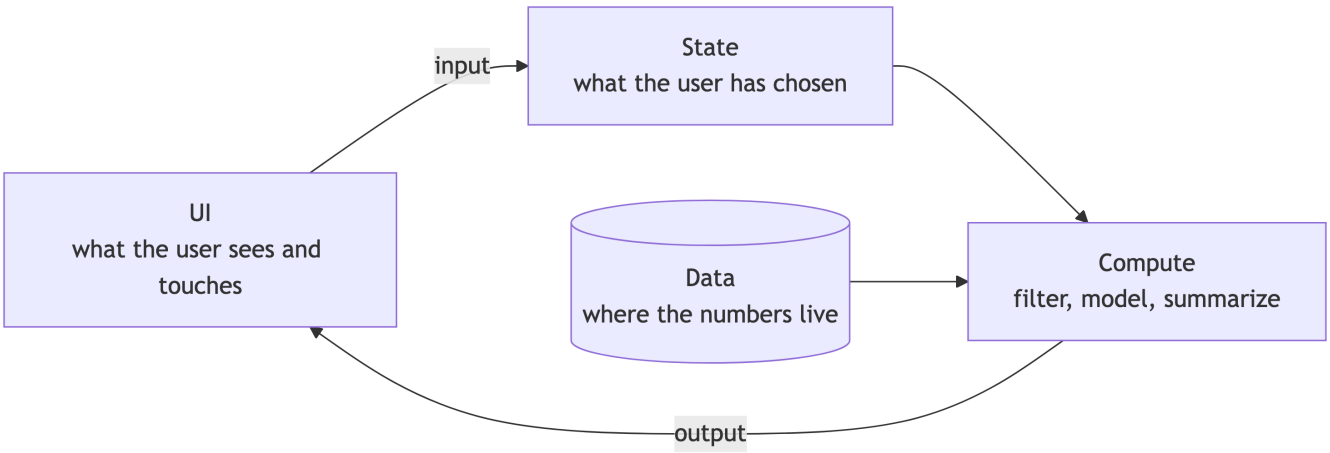
## 1 Why build apps, and why now

For most of your graduate career the way you share research is a paper. That is the right tool for talking to other scientists. It is a poor tool for talking to a water manager, a county commissioner, a rancher, or a reporter. Those people do not want your figure 3. They want to click on their place and see their answer.

Interactive tools close that gap. They also change how you do science: an app that lets you flip between variables and timescales will show you things in your own data that a static plot never will.

### 1.1 The one diagram that matters

Every interactive data tool, in any language, is the same four boxes.



The only interesting design question is *where each box lives*. In a Shiny app, the UI is in the browser and the other three boxes are in an R process on a server. By the end of the hour you will see a tool where all four boxes live in the browser and there is no server at all.

### 1.2 The LLM argument

You will build most of your apps with an AI model writing a lot of the code. Good. That is not a reason to skip this hour, it is the reason for it.

A model will happily give you a Shiny app that re-downloads a 20 MB file every time a slider moves, redraws an entire map to change three dots, or pulls forty years of daily data to show one bar chart. The app will *work*. It will also be slow, fragile, and impossible to share with more than three people. The model cannot tell you that, because it does not know what you are going to do with the app.

If you can draw the four boxes for an app and say where each one lives, you can look at generated code and ask: what runs once, what runs on every click, what crosses the network, and does any of it need a server? Those questions are the difference between using the model and being used by it. Each section below ends with one such question you can take to a model yourself.

## 2 Shiny from zero

Shiny is an R package that turns R code into a web application. An app has two parts:

- a **UI** object that describes the layout and the widgets, and
- a **server** function that reads `input$...`, does work, and writes `output$...`.

```
ui = page_sidebar(...)                          # what the user sees
server = function(input, output, session) {   # what the app does
  output$thing = render...({ ... input$... })
}
shinyApp(ui, server)
```

`bslib` is the current layout package for Shiny. It gives you sidebars, cards, value boxes and a sane theme in a few function calls. `fluidPage()` and `sidebarLayout()` from older tutorials still work, but there is no reason to start there now.

Let's see the simplest possible app, then read it.

```
shiny::runExample("01_hello")
```

### 2.1 The Old Faithful app

`faithful` is a base R dataset of 272 geyser eruptions: how long each lasted, and how long until the next one.

```
head(faithful, 3)
```

```
  eruptions waiting
1     3.600      79
2     1.800      54
3     3.333      74
```

```r
library(shiny)
library(bslib)
library(ggplot2)

# ---- UI: what the user sees ----------------------------------------------
ui = page_sidebar(
  title = "Old Faithful eruptions",
  sidebar = sidebar(
    # Input: a slider. Its value shows up in the server as input$bins
    sliderInput(
      inputId = "bins",
      label = "Number of bins",
      min = 1, max = 50, value = 30
    )
  ),
  # Output: a plot. The server fills it by assigning output$hist
  card(plotOutput(outputId = "hist"))
)

# ---- Server: what the app does -------------------------------------------
server = function(input, output, session) {
  # renderPlot() marks this block as reactive: it re-runs whenever any
  # input$... it reads changes. Here that is input$bins.
  output$hist = renderPlot({
    ggplot(faithful, aes(x = waiting)) +
      geom_histogram(bins = input$bins, fill = "#75AADB", color = "white") +
      labs(
        x = "Waiting time until next eruption (minutes)",
        y = "Count"
      ) +
      theme_minimal(base_size = 16)
  })
}

shinyApp(ui, server)
```

Read it top to bottom once:

1. `sliderInput(inputId = "bins", ...)` creates a widget and a name. In the server that name is `input$bins`.
2. `plotOutput("hist")` reserves a spot in the page. The server fills it by assigning `output$hist`.
3. `renderPlot({...})` marks a block as **reactive**. Shiny notes which inputs the block reads. When one changes, the block re-runs and the browser gets a new picture.

That is the whole cycle: UI → input → server → output → UI.

**Try it**

Change the app so it plots `eruptions` instead of `waiting`, and limit the slider to 10 to 20 bins with a starting value of 15. Two lines change.

Then add a widget: a `selectInput()` in the sidebar that lets the user pick `waiting` or `eruptions`, and use `.data[[input$var]]` inside `aes()` so the plot follows it. Now the server reads two inputs. Which ones make the plot re-run?

**Ask the model**

"In this app, which lines run when the app starts, and which lines run when I move the slider? Rewrite the comments to say so."

## 3 Shiny for exploring data

Now something you would actually use. The `iris` data has four flower measurements for three species. We want to pick any two, fit a polynomial, and see the fit.

```
head(iris, 3)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
```

```
library(shiny)
library(bslib)
library(dplyr)
library(ggplot2)

# Named vector: what the user sees = what the code uses
vars = c(
  "Sepal length" = "Sepal.Length",
  "Sepal width"  = "Sepal.Width",
  "Petal length" = "Petal.Length",
  "Petal width"  = "Petal.Width"
)

ui = page_sidebar(
  title = "Iris explorer",
  sidebar = sidebar(
    selectInput("x", "X variable", choices = vars, selected = "Sepal.Length"),
    selectInput("y", "Y variable", choices = vars, selected = "Petal.Length"),
```

```
    sliderInput("degree", "Polynomial degree", min = 1, max = 5, value = 1, step = 1)
  ),
  layout_columns(
    fill = FALSE,
    value_box(title = "R squared", value = textOutput("r2")),
    value_box(title = "Observations", value = nrow(iris))
  ),
  card(plotOutput("plot"))
)

server = function(input, output, session) {

  # A reactive(): a chunk of computation that caches its result and only
  # re-runs when the inputs it reads change. Both the model and the plot
  # read dat(), so the data is built once per change, not twice.
  dat = reactive({
    iris |>
      select(x = all_of(input$x), y = all_of(input$y), Species)
  })

  model = reactive({
    lm(y ~ poly(x, input$degree), data = dat())
  })

  output$r2 = renderText({
    sprintf("%.3f", summary(model())$r.squared)
  })

  output$plot = renderPlot({
    ggplot(dat(), aes(x = x, y = y)) +
      geom_point(aes(color = Species), size = 2.5, alpha = 0.8) +
      geom_smooth(
        method = "lm", formula = y ~ poly(x, input$degree),
        color = "black", fill = "grey70"
      ) +
      labs(
        x = names(vars)[vars == input$x],
        y = names(vars)[vars == input$y]
      ) +
      theme_minimal(base_size = 16) +
      theme(legend.position = "bottom")
  })
}

shinyApp(ui, server)
```

Two new ideas:

**`reactive()`** builds a value that other reactive blocks can read by calling it like a function: `dat()`. It caches its result and only recomputes when something it reads changes. Here both the model and the plot need the same subset of `iris`. Without `reactive()`, you would compute it twice. With a bigger dataset or a slow query, that matters.

**Several outputs from one input.** Change the X variable and the value box, the model and the plot all update, and Shiny worked out the order on its own by tracking which blocks read which inputs. That web of dependencies is called the **reactive graph**. When an app is slow, the first question is which parts of that graph re-run on every click, and whether they need to.

> **Try it**
>
> Remove the polynomial slider and fix the model to a straight line. You will touch the UI once and the server twice.

> **Ask the model**
>
> Paste the app into a chatbot and ask: "When I change only the Y variable, which parts of this app re-run, and why? Is anything re-running that does not need to?" Check its answer against what you see when you use the app.

## 3.1 A short note on the pipe

The rest of this document uses `|>`, the native R pipe. It passes the result of one line into the first argument of the next, so a nested call reads top to bottom instead of inside out.

```
nested = round(exp(sin(log(iris$Sepal.Length))), 2)

piped = iris$Sepal.Length |>
  log() |>
  sin() |>
  exp() |>
  round(2)

identical(nested, piped)
```

```
[1] TRUE
```

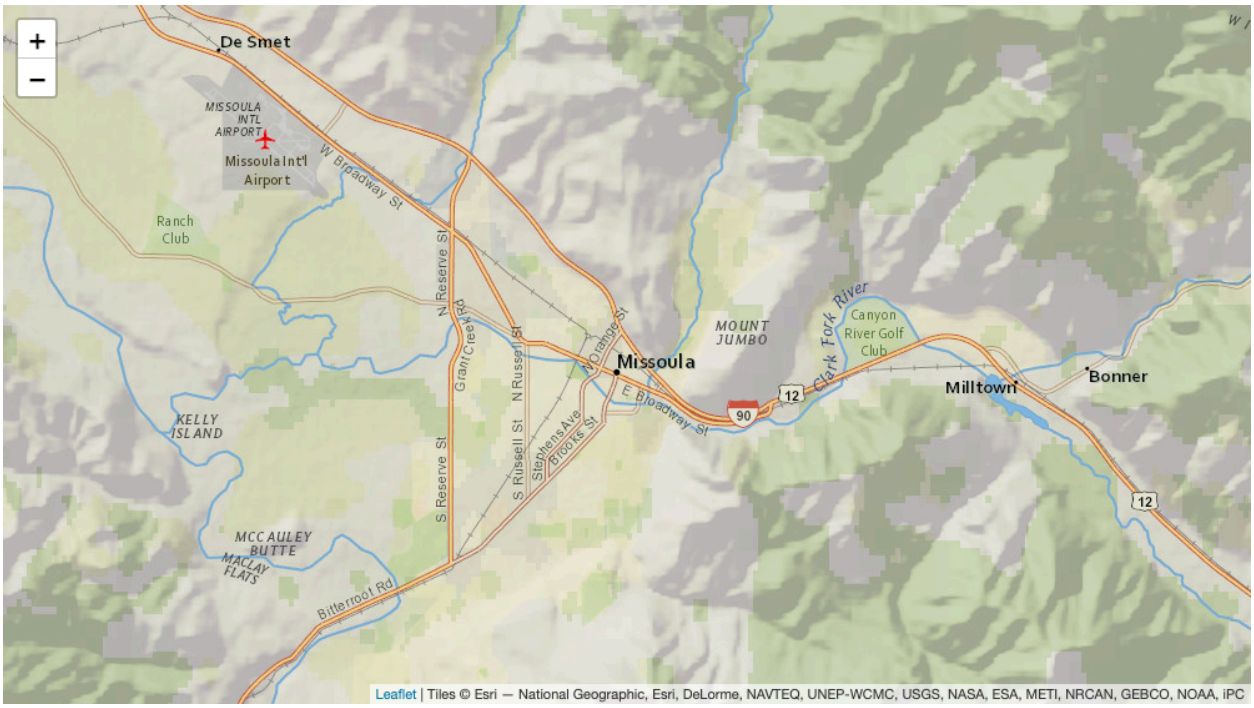
You will still see `%>%` from the magrittr package in older code and on Stack Overflow. For almost everything it is the same thing. Use `|>`; it needs no package.

# 4 Maps with Leaflet

A lot of ecology has a spatial component, and most of us learned to make maps in a desktop GIS. Leaflet is an open source JavaScript library for interactive web maps, and the `leaflet` R package drives it without writing any JavaScript.

```
library(leaflet)

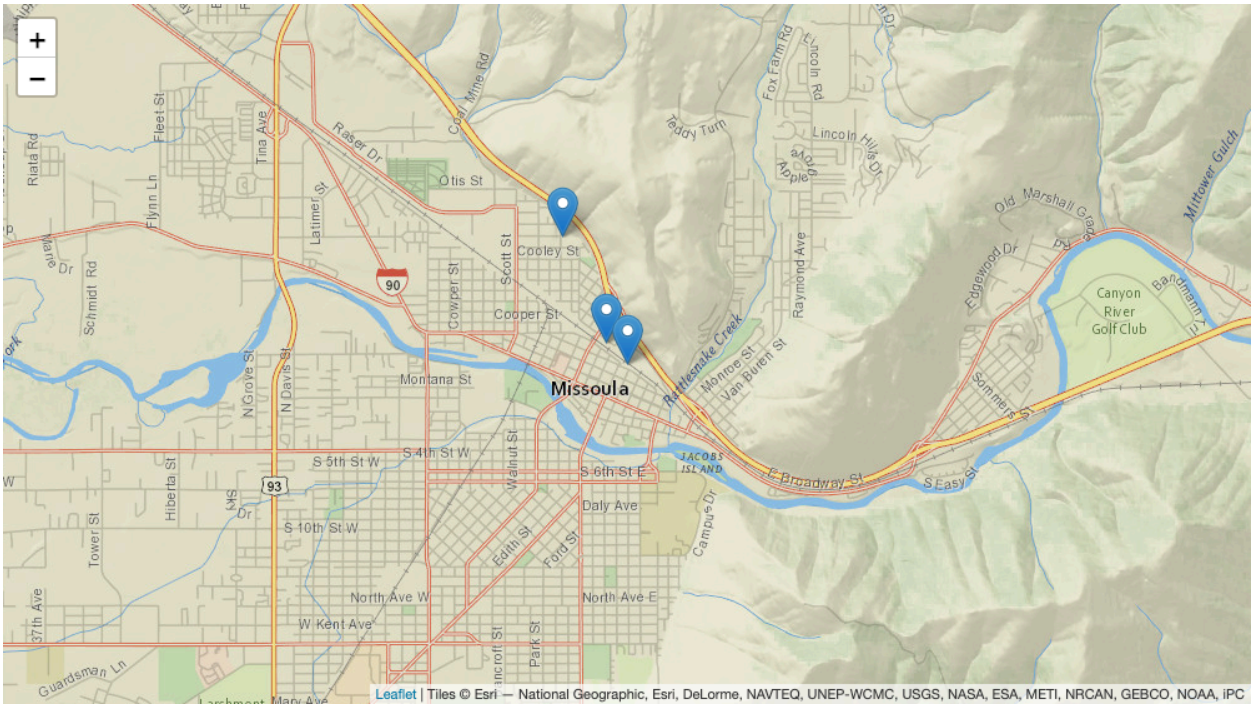
leaflet() |>
  addProviderTiles(providers$Esri.NatGeoWorldMap) |>
  setView(lng = -113.991, lat = 46.876, zoom = 12)
```




That is a complete, pannable, zoomable map. Now add some field sites.

```
sites = data.frame(
  name = c("Site 1", "Site 2", "Site 3"),
  lat  = c(46.875, 46.877, 46.887),
  lon  = c(-113.991, -113.994, -114.000)
)

leaflet(sites) |>
  addProviderTiles(providers$Esri.NatGeoWorldMap) |>
  setView(lng = -113.991, lat = 46.876, zoom = 13) |>
  addMarkers(lng = ~lon, lat = ~lat, popup = ~name)
```




The ~ means "look this up in the data I handed to `leaflet()`".

## 4.1 Two other mapping tools worth knowing

**mapview** is for checking your work. One call, any `sf` or `terra` object, done. Here is the Montana outline with the same 60-day percent-of-normal precipitation raster you will meet in the climate app below.

```
library(sf)
library(terra)
library(mapview)
library(dplyr)

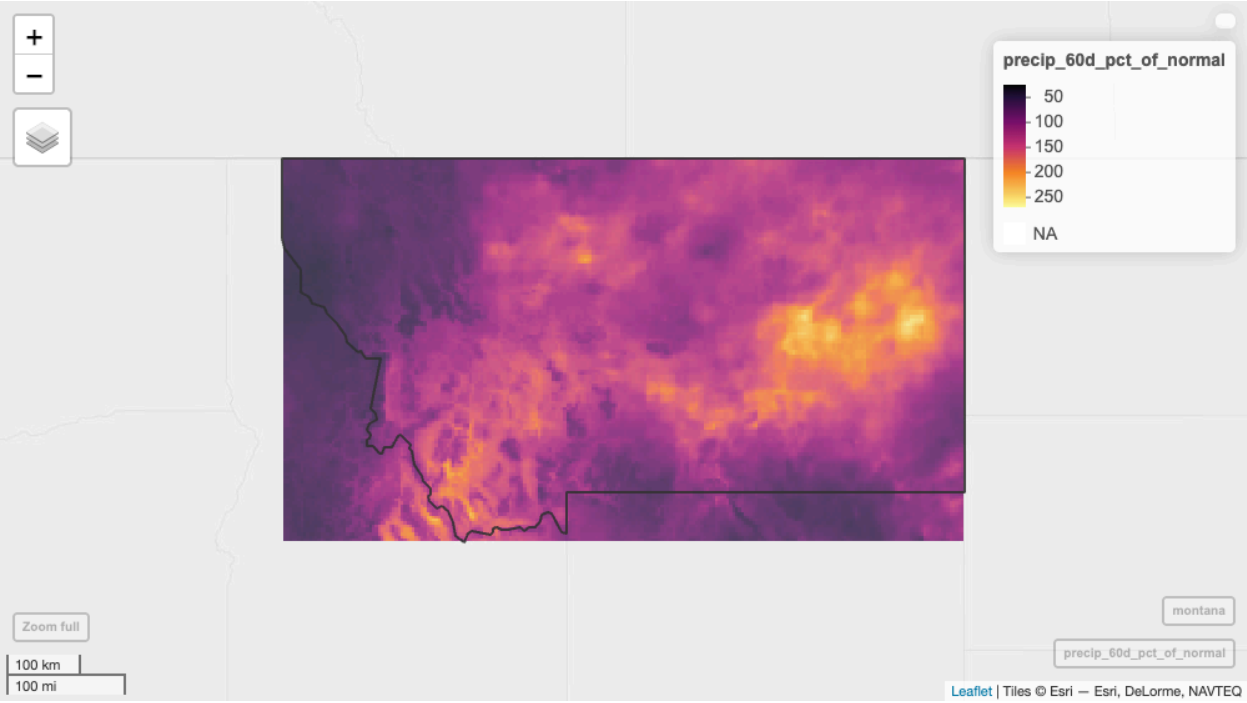
montana = st_read(
  "https://eric.clst.org/assets/wiki/uploads/Stuff/gz_2010_us_040_00_20m.json",
  quiet = TRUE
) |>
  filter(NAME == "Montana")

cog_dir  =  "/vsicurl/https://data2.climate.umt.edu/gridmet/derived/conus_drought_web/
latest/"
pon = rast(paste0(cog_dir, "precip-pon_60d_rolling-30.tif")) / 100

mapviewOptions(
  basemaps = c("Esri.WorldGrayCanvas", "OpenStreetMap", "Esri.WorldImagery")
)

mapview(crop(pon, montana), layer.name = "precip_60d_pct_of_normal",
```

```
        na.color = "transparent") +
  mapview(montana, alpha.regions = 0, lwd = 2, legend = FALSE)
```




**mapgl** wraps MapLibre and Mapbox GL, the vector tile engines behind most modern web maps. Smoother, 3D terrain if you want it, and it is where the web is heading. Same idea as leaflet, different engine.

```
library(mapgl)

maplibre(
  style = "https://tiles.openfreemap.org/styles/positron",
  center = c(-113.99, 46.87), zoom = 11
) |>
  add_markers(c(-113.991, 46.876), color = "#b3202c")
```

# 5 Leaflet and Shiny together: live earthquakes

The USGS publishes every earthquake above magnitude 2.5 in the last 30 days as a CSV, updated every minute. We will read it, map it, and let the user pick a minimum magnitude.

## 5.1 First attempt: the obvious way

This is roughly the app from the 2023 version of this lecture, and it is roughly what a model will give you if you ask for "a Shiny app that maps earthquakes with a magnitude slider."

```
library(shiny)
library(leaflet)
library(dplyr)
```

```
url = "https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/2.5_month.csv"

ui = fluidPage(
  titlePanel("Earthquakes"),
  sliderInput("min_mag", "Minimum magnitude",
              min = 2.5, max = 8, value = 4.5, step = 0.1),
  leafletOutput("map", height = 600)
)

server = function(input, output, session) {
  output$map = renderLeaflet({
    quakes = read.csv(url) |>
      filter(mag >= input$min_mag)
    pal = colorNumeric("YlOrRd", domain = quakes$mag)
    leaflet(quakes) |>
      addProviderTiles(providers$Esri.WorldGrayCanvas) |>
      addCircleMarkers(lng = ~longitude, lat = ~latitude, fillColor = ~pal(mag),
                       radius = 6, weight = 1, color = "black", fillOpacity = 0.7) |>
      addLegend(pal = pal, values = ~mag, title = "Magnitude")
  })
}

shinyApp(ui, server)
```

It works. Now go back to the four boxes from the diagram at the start and place this app's pieces in them:

- **UI:** the slider and the map, in the browser.
- **State:** `input$min_mag`, the slider's current value.
- **Compute:** the `filter()` and the color palette, inside `renderLeaflet()`.
- **Data:** the USGS CSV, read by `read.csv()`, also inside `renderLeaflet()`.

The trouble is in those last two. Compute and data both sit inside a block that reads the slider, so every time the slider moves:

- The CSV is **re-downloaded** on every move, because `read.csv()` is inside `renderLeaflet()`.
- The **whole map is rebuilt**: tiles reload, the user loses their zoom, and the legend changes range because the palette is recomputed from the filtered data.

None of that is a bug. All of it is architecture.

## 5.2 Second attempt: put each box where it belongs

```
library(shiny)
library(bslib)
library(leaflet)
library(dplyr)
```

```
# ---- Runs ONCE, when the app starts -----------------------------------------
# Anything up here is shared by every user session. The CSV is fetched one
# time, not once per slider move.
quakes_all = read.csv(
  "https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/2.5_month.csv"
) |>
  select(time, place, latitude, longitude, mag, depth)

# Fix the color scale to the full dataset so the legend does not jump around
# every time the filter changes.
pal = colorNumeric("YlOrRd", domain = quakes_all$mag)

ui = page_sidebar(
  title = "Earthquakes, last 30 days (USGS)",
  sidebar = sidebar(
    sliderInput(
      "min_mag", "Minimum magnitude",
      min = 2.5, max = ceiling(max(quakes_all$mag)),
      value = 4.5, step = 0.1
    ),
    textOutput("n")
  ),
  leafletOutput("map")
)

server = function(input, output, session) {

  # ---- Runs every time input$min_mag changes ------------------------------
  quakes = reactive({
    quakes_all |> filter(mag >= input$min_mag)
  })

  output$n = renderText({
    paste(nrow(quakes()), "earthquakes shown")
  })

  # One helper that draws the dots. We use it twice below.
  add_quakes = function(map, data) {
    map |>
      clearMarkers() |>
      addCircleMarkers(
        data = data, lng = ~longitude, lat = ~latitude,
        radius = ~mag * 1.5, weight = 1, color = "black",
        fillColor = ~pal(mag), fillOpacity = 0.7,
        popup = ~paste0("<b>M ", mag, "</b><br>", place, "<br>", depth, " km deep")
      )
```

```
  }

  # ---- Runs ONCE per session: base map, legend, and the first set of dots --
  # isolate() reads quakes() without depending on it, so this block does NOT
  # re-run when the slider moves. Nothing else in here reads an input.
  output$map = renderLeaflet({
    leaflet() |>
      addProviderTiles(providers$Esri.WorldGrayCanvas) |>
      setView(lng = -100, lat = 35, zoom = 3) |>
      addLegend(
        position = "bottomleft", pal = pal, values = quakes_all$mag,
        title = "Magnitude", opacity = 0.8
      ) |>
      add_quakes(isolate(quakes()))
  })

  # ---- Runs when quakes() changes: swap the dots only ----------------------
  # leafletProxy() talks to the map that already exists in the browser.
  # The basemap tiles, zoom and legend stay exactly where the user left them.
  observe({
    leafletProxy("map") |>
      add_quakes(quakes())
  })
}

shinyApp(ui, server)
```

What moved:

| Piece | Before | After | Why |
|---|---|---|---|
| Download | inside `renderLeaflet` | top of file | runs once per app, shared by every user |
| Palette | from filtered data | from all data | legend stays stable while you filter |
| Base map | rebuilt per change | `renderLeaflet` reads no inputs | runs once per session |
| Markers | rebuilt with the map | `leafletProxy()` in `observe()` | only the dots change |
| Marker code | written once | `add_quakes()` helper, called twice | first draw and updates stay identical |

`leafletProxy()` is the single most useful thing to know about Shiny and Leaflet. It sends instructions to the map already drawn in the browser instead of replacing it. The user's zoom, pan and tile cache all survive.

One small thing to notice: `isolate(quakes())` inside `renderLeaflet()`. Reading a reactive normally creates a dependency. `isolate()` reads the current value without one, so the map block gets the first set of dots but does not re-run when the slider moves. That is exactly the "runs once" behavior we want, stated explicitly.

> **Try it**
>
> Add a second slider for maximum depth (the `depth` column, in km) and filter on both. Then ask: which blocks in the server re-ran when you moved the new slider, and which did not?

> **Ask the model**
>
> Paste the first attempt and ask: "This app re-downloads the CSV and redraws the whole map on every slider move. Restructure it so the download happens once and only the markers update. Explain which lines you moved and why." Then compare its answer to the second attempt. If it did not reach for `leafletProxy()`, ask it why not.

# 6 Click the map, query the climate

The two questions I get most at the Climate Office are "how much precipitation does my place get?" and "how dry is it at my place right now?" They sound alike. They call for opposite designs, and building both is the clearest way to see why the compute and data boxes matter.

## 6.1 Tool 1: the precipitation downloader

Click anywhere in the lower 48, get a plot of the last three years of monthly precipitation there, and download every day since 1979 as a spreadsheet.

The data is gridMET: daily precipitation at 4 km for the lower 48, from 1979 to yesterday, produced at the University of Idaho. It lives on a THREDDS server as one giant NetCDF file. Think of it as a **hypercube**: a three-dimensional array with a day axis, a latitude axis and a longitude axis. A click picks one latitude and one longitude, and we want the whole day axis at that spot.

THREDDS speaks a protocol called OPeNDAP. You describe the slice of the cube you want in the URL, and the server sends back only that slice. Adding `.ascii` to the URL asks for plain text, so `readLines()` is all we need. No NetCDF library to install.

```
library(shiny)
library(bslib)
library(leaflet)
library(dplyr)
library(ggplot2)

# ---- Runs ONCE: learn the shape of the gridMET precipitation hypercube -----
# gridMET daily precipitation, 1979 to yesterday, 4 km, the lower 48, served
# by a THREDDS server at the University of Idaho. It is one giant NetCDF
# "hypercube" with three dimensions: day x lat x lon.
```

```
#
# THREDDS speaks OPeNDAP: you ask for a slice of the cube in the URL and it
# sends back just that slice. Adding ".ascii" asks for plain text, so all we
# need is readLines(). No NetCDF library required.
thredds    =    "https://thredds.northwestknowledge.net/thredds/dodsC/agg_met_pr_1979_
CurrentYear_CONUS.nc"

read_dap = function(query) {
  txt = readLines(paste0(thredds, ".ascii?", query), warn = FALSE)
  txt[-(1:grep("^-+$", txt))]               # drop the header above the dashes
}

grid = read_dap("lat,lon")                  # the cube's coordinates
grid_lat = as.numeric(strsplit(grid[grep("^lat\\[", grid) + 1], ", ")[[1]])
grid_lon = as.numeric(strsplit(grid[grep("^lon\\[", grid) + 1], ", ")[[1]])

# Those are cell CENTERS. The cell EDGES sit half a cell (1/48 degree) either
# side. We use the edges to draw the grid when the user zooms in.
half = abs(grid_lon[2] - grid_lon[1]) / 2
lon_edges = c(sort(grid_lon) - half, max(grid_lon) + half)
lat_edges = c(sort(grid_lat) - half, max(grid_lat) + half)

dds = readLines(paste0(thredds, ".dds"), warn = FALSE)   # how many days so far
n_days  =  as.integer(sub(".*day  =  ([0-9]+).*",  "\\1",  grep("day  =  ",  dds,  value  =
TRUE)[1]))

# ---- The function a click calls ----------------------------------------------
get_precip = function(lat, lon) {
  i = which.min(abs(grid_lat - lat)) - 1    # nearest pixel (OPeNDAP counts from 0)
  j = which.min(abs(grid_lon - lon)) - 1

  # Every day, one pixel: precipitation_amount[day 0 to the last][i][j]
  vals = read_dap(sprintf(
    "precipitation_amount.precipitation_amount[0:1:%d][%d][%d]", n_days - 1, i, j
  ))
  vals = as.numeric(sub(".*, ", "", grep("^\\[", vals, value = TRUE)))

  tibble(
    date = as.Date("1979-01-01") + seq_along(vals) - 1,   # day 0 is Jan 1, 1979
    precip_mm = ifelse(vals == 32767, NA, vals * 0.1)     # stored as mm x 10
  )
}

# Unit choices: what the user sees = what the code uses
units = c("Millimeters" = "mm", "Inches" = "in")
```

```
# ---- UI ------------------------------------------------------------------
ui = page_sidebar(
  title = "How much precipitation falls here?",
  sidebar = sidebar(
    width = 360,
    p("Click anywhere in the lower 48. The app downloads every day of gridMET
       precipitation since 1979 for that 4 km pixel, then summarizes it."),
    textOutput("where"),
    selectInput("units", "Units", choices = units),
    downloadButton("dl_daily", "Daily data (1979 to now)"),
    downloadButton("dl_monthly", "Monthly totals (1979 to now)")
  ),
  leafletOutput("map", height = 440),
  card(plotOutput("plot", height = 260))
)

# ---- Server --------------------------------------------------------------
server = function(input, output, session) {

  output$map = renderLeaflet({
    leaflet() |>
      # Three basemaps, each in its own "group". The layers control in the
      # corner switches between them, and the first one is the default.
      addProviderTiles(providers$Esri.WorldTopoMap, group = "Topographic") |>
      addProviderTiles(providers$USGS.USTopo, group = "USGS topo") |>
      addProviderTiles(providers$Esri.WorldImagery, group = "Satellite") |>
      # Satellite has no place names, so lay a labels-only layer on top of it.
      # Same group name, so the two switch on and off together.
      addTiles(
        urlTemplate = paste0(
          "https://server.arcgisonline.com/ArcGIS/rest/services/",
          "Reference/World_Boundaries_and_Places/MapServer/tile/{z}/{y}/{x}"
        ),
        attribution = "Esri", group = "Satellite"
      ) |>
      addLayersControl(
        baseGroups = c("Topographic", "USGS topo", "Satellite"),
        options = layersControlOptions(collapsed = FALSE)
      ) |>
      addScaleBar(position = "topleft") |>
      setView(lng = -97, lat = 39.5, zoom = 4)
  })

  clicked = reactive({
    req(input$map_click)
    input$map_click
```

```
})

# ---- Runs on every click: one request for ~17,000 days, about 6 seconds --
daily = reactive({
  pt = clicked()
  withProgress(message = "Downloading 45+ years of daily data...", {
    get_precip(pt$lat, pt$lng)
  })
})

monthly = reactive({
  daily() |>
    mutate(month = as.Date(format(date, "%Y-%m-01"))) |>
    summarise(precip_mm = sum(precip_mm, na.rm = TRUE), .by = month)
})

# ---- Units: runs when input$units changes, never re-downloads -------------
# daily() and monthly() stay in millimeters and do not read input$units, so
# switching units only re-draws the plot. The data is already here.
per_mm = reactive({
  if (input$units == "in") 1 / 25.4 else 1      # 25.4 mm in an inch
})

# Convert a table to the chosen units, and rename the column to match.
in_units = function(df) {
  df[[paste0("precip_", input$units)]] = df$precip_mm * per_mm()
  if (input$units != "mm") df$precip_mm = NULL
  df
}

observe({
  pt = clicked()
  i = which.min(abs(grid_lat - pt$lat))     # the pixel the data comes from
  j = which.min(abs(grid_lon - pt$lng))
  leafletProxy("map") |>
    clearMarkers() |>
    clearGroup("cell") |>
    addMarkers(lng = pt$lng, lat = pt$lat) |>
    # Outline that one 4 km cell
    addRectangles(
      lng1 = grid_lon[j] - half, lat1 = grid_lat[i] - half,
      lng2 = grid_lon[j] + half, lat2 = grid_lat[i] + half,
      group = "cell", color = "#d7301f", weight = 2, fillOpacity = 0.1,
      options = pathOptions(interactive = FALSE)   # let clicks pass through
    )
})
```

```
  # ---- Grid lines: runs whenever the user pans or zooms --------------------
  # Drawing all ~800,000 cells would freeze the browser, so draw only the
  # lines inside the visible window, and only once cells are big enough to
  # see (zoom 9 and up). Leaflet reports the view as input$map_zoom and
  # input$map_bounds.
  observe({
    req(input$map_zoom, input$map_bounds)
    proxy = leafletProxy("map") |> clearGroup("grid")
    if (input$map_zoom < 9) return()

    b = input$map_bounds
    lons = lon_edges[lon_edges >= b$west & lon_edges <= b$east]
    lats = lat_edges[lat_edges >= b$south & lat_edges <= b$north]
    if (length(lons) == 0 || length(lats) == 0) return()   # outside the grid

    # Each line runs across the visible part of the grid. NA breaks one line
    # from the next, so all of them go to the browser as a single layer.
    south = max(b$south, min(lat_edges)); north = min(b$north, max(lat_edges))
    west  = max(b$west,  min(lon_edges)); east  = min(b$east,  max(lon_edges))
    proxy |>
      addPolylines(
       lng = c(rep(lons, each = 3) + c(0, 0, NA), rep(c(west, east, NA), length(lats))),
          lat = c(rep(c(south, north, NA), length(lons)), rep(lats, each = 3) + c(0,
0, NA)),
        group = "grid", color = "#333333", weight = 1, opacity = 0.35,
        options = pathOptions(interactive = FALSE)
      )
  })

  output$where = renderText({
    pt = clicked()
    sprintf("Location: %.3f, %.3f", pt$lat, pt$lng)
  })

  output$plot = renderPlot({
    monthly() |>
      slice_tail(n = 36) |>                 # the last three years
      mutate(precip = precip_mm * per_mm()) |>
      ggplot(aes(x = month, y = precip)) +
      geom_col(fill = "#2c7fb8") +
      scale_x_date(date_breaks = "3 months", date_labels = "%b\n%Y") +
      labs(
        x = NULL,
         y = paste0("Precipitation (", tolower(names(units)[units == input$units]), "
per month)")
```

```
      ) +
      theme_minimal(base_size = 14)
  })

  # The file name carries the location, so downloads do not overwrite each other.
  file_stub = reactive({
    pt = clicked()
    sprintf("%.4f_%.4f.csv", pt$lat, pt$lng)
  })

  output$dl_daily = downloadHandler(
    filename = function() paste0("daily_precip_", file_stub()),
    content = function(file) write.csv(in_units(daily()), file, row.names = FALSE)
  )

  output$dl_monthly = downloadHandler(
    filename = function() paste0("monthly_precip_", file_stub()),
    content = function(file) write.csv(in_units(monthly()), file, row.names = FALSE)
  )
}

shinyApp(ui, server)
```

Now place it in the four boxes. The **UI** is a map, a plot and two buttons. The **state** is the clicked point and the chosen units. The **data** is the remote hypercube, sliced on every click. The **compute** is the monthly summary, done in R while the user waits. Every click moves about 17,000 numbers across the network and takes around six seconds.

That is the right design here. The user asked for the whole record, so every one of those 17,000 numbers is the answer. Nobody is going to precompute a spreadsheet for each of the 800,000 pixels.

Now try the **Units** dropdown after a click. The plot changes instantly, with no six-second wait. The download happens in `daily()`, which never reads `input$units`, so Shiny has no reason to run it again. Only the plot and the download buttons read the units. Deciding which blocks read which inputs is how you decide what is slow and what is instant.

Also zoom in. Past a certain zoom the map draws the gridMET cells, and the red box is the cell your numbers come from. Click downtown Missoula and the cell reaches up the side of Mount Jumbo, so its precipitation is not the valley floor's. Every gridded dataset averages over a cell like this. Seeing the cell keeps you honest about what one number means.

> **i** Try it
>
> Change the plot to show the last ten years instead of three. One number changes, and the date labels will need thinning.
>
> Then add a third download button for annual totals. You need three new pieces: an `annual` reactive built like `monthly`, a `downloadButton()` in the sidebar, and a `downloadHandler()` in the server.

### 6.1.a The same app with an interactive plot

Version 1 draws a static picture of the last three years. **plotly** draws charts in the browser with JavaScript, so the user can hover, zoom and drag. This version sends every month since 1979 to the browser once. It opens on the last three years, and the range slider under the plot lets the user explore the whole record.

Only three things change from Version 1, and each is marked `# PLOTLY` in the code: `library(plotly)`, `plotlyOutput()` in the UI, and `renderPlotly()` with `plot_ly()` inside it in the server.

```
library(shiny)
library(bslib)
library(leaflet)
library(dplyr)
library(plotly)      # PLOTLY: replaces ggplot2

# ---- Runs ONCE: learn the shape of the gridMET precipitation hypercube -----
# gridMET daily precipitation, 1979 to yesterday, 4 km, the lower 48, served
# by a THREDDS server at the University of Idaho. It is one giant NetCDF
# "hypercube" with three dimensions: day x lat x lon.
#
# THREDDS speaks OPeNDAP: you ask for a slice of the cube in the URL and it
# sends back just that slice. Adding ".ascii" asks for plain text, so all we
# need is readLines(). No NetCDF library required.
thredds    =    "https://thredds.northwestknowledge.net/thredds/dodsC/agg_met_pr_1979_
CurrentYear_CONUS.nc"

read_dap = function(query) {
  txt = readLines(paste0(thredds, ".ascii?", query), warn = FALSE)
  txt[-(1:grep("^-+$", txt))]              # drop the header above the dashes
}

grid = read_dap("lat,lon")                 # the cube's coordinates
grid_lat = as.numeric(strsplit(grid[grep("^lat\\[", grid) + 1], ", ")[[1]])
grid_lon = as.numeric(strsplit(grid[grep("^lon\\[", grid) + 1], ", ")[[1]])

# Those are cell CENTERS. The cell EDGES sit half a cell (1/48 degree) either
# side. We use the edges to draw the grid when the user zooms in.
half = abs(grid_lon[2] - grid_lon[1]) / 2
lon_edges = c(sort(grid_lon) - half, max(grid_lon) + half)
```

```
lat_edges = c(sort(grid_lat) - half, max(grid_lat) + half)

dds = readLines(paste0(thredds, ".dds"), warn = FALSE)   # how many days so far
n_days = as.integer(sub(".*day = ([0-9]+).*", "\\1", grep("day = ", dds, value =
TRUE)[1]))

# ---- The function a click calls ----------------------------------------------
get_precip = function(lat, lon) {
  i = which.min(abs(grid_lat - lat)) - 1    # nearest pixel (OPeNDAP counts from 0)
  j = which.min(abs(grid_lon - lon)) - 1

  # Every day, one pixel: precipitation_amount[day 0 to the last][i][j]
  vals = read_dap(sprintf(
    "precipitation_amount.precipitation_amount[0:1:%d][%d][%d]", n_days - 1, i, j
  ))
  vals = as.numeric(sub(".*, ", "", grep("^\\[", vals, value = TRUE)))

  tibble(
    date = as.Date("1979-01-01") + seq_along(vals) - 1,   # day 0 is Jan 1, 1979
    precip_mm = ifelse(vals == 32767, NA, vals * 0.1)     # stored as mm x 10
  )
}

# Unit choices: what the user sees = what the code uses
units = c("Millimeters" = "mm", "Inches" = "in")

# ---- UI ----------------------------------------------------------------------
ui = page_sidebar(
  title = "How much precipitation falls here? (interactive plot)",
  sidebar = sidebar(
    width = 360,
    p("Click anywhere in the lower 48. The app downloads every day of gridMET
       precipitation since 1979 for that 4 km pixel, then summarizes it."),
    textOutput("where"),
    selectInput("units", "Units", choices = units),
    downloadButton("dl_daily", "Daily data (1979 to now)"),
    downloadButton("dl_monthly", "Monthly totals (1979 to now)")
  ),
  leafletOutput("map", height = 440),
  card(plotlyOutput("plot", height = 320))   # PLOTLY: plotlyOutput, not plotOutput
)

# ---- Server ------------------------------------------------------------------
server = function(input, output, session) {

  output$map = renderLeaflet({
```

```
  leaflet() |>
    # Three basemaps, each in its own "group". The layers control in the
    # corner switches between them, and the first one is the default.
    addProviderTiles(providers$Esri.WorldTopoMap, group = "Topographic") |>
    addProviderTiles(providers$USGS.USTopo, group = "USGS topo") |>
    addProviderTiles(providers$Esri.WorldImagery, group = "Satellite") |>
    # Satellite has no place names, so lay a labels-only layer on top of it.
    # Same group name, so the two switch on and off together.
    addTiles(
      urlTemplate = paste0(
        "https://server.arcgisonline.com/ArcGIS/rest/services/",
        "Reference/World_Boundaries_and_Places/MapServer/tile/{z}/{y}/{x}"
      ),
      attribution = "Esri", group = "Satellite"
    ) |>
    addLayersControl(
      baseGroups = c("Topographic", "USGS topo", "Satellite"),
      options = layersControlOptions(collapsed = FALSE)
    ) |>
    addScaleBar(position = "topleft") |>
    setView(lng = -97, lat = 39.5, zoom = 4)
})

clicked = reactive({
  req(input$map_click)
  input$map_click
})

# ---- Runs on every click: one request for ~17,000 days, about 6 seconds --
daily = reactive({
  pt = clicked()
  withProgress(message = "Downloading 45+ years of daily data...", {
    get_precip(pt$lat, pt$lng)
  })
})

monthly = reactive({
  daily() |>
    mutate(month = as.Date(format(date, "%Y-%m-01"))) |>
    summarise(precip_mm = sum(precip_mm, na.rm = TRUE), .by = month)
})

# ---- Units: runs when input$units changes, never re-downloads -------------
# daily() and monthly() stay in millimeters and do not read input$units, so
# switching units only re-draws the plot. The data is already here.
per_mm = reactive({
```

```
  if (input$units == "in") 1 / 25.4 else 1      # 25.4 mm in an inch
})

# Convert a table to the chosen units, and rename the column to match.
in_units = function(df) {
  df[[paste0("precip_", input$units)]] = df$precip_mm * per_mm()
  if (input$units != "mm") df$precip_mm = NULL
  df
}

observe({
  pt = clicked()
  i = which.min(abs(grid_lat - pt$lat))     # the pixel the data comes from
  j = which.min(abs(grid_lon - pt$lng))
  leafletProxy("map") |>
    clearMarkers() |>
    clearGroup("cell") |>
    addMarkers(lng = pt$lng, lat = pt$lat) |>
    # Outline that one 4 km cell
    addRectangles(
      lng1 = grid_lon[j] - half, lat1 = grid_lat[i] - half,
      lng2 = grid_lon[j] + half, lat2 = grid_lat[i] + half,
      group = "cell", color = "#d7301f", weight = 2, fillOpacity = 0.1,
      options = pathOptions(interactive = FALSE)   # let clicks pass through
    )
})

# ---- Grid lines: runs whenever the user pans or zooms --------------------
# Drawing all ~800,000 cells would freeze the browser, so draw only the
# lines inside the visible window, and only once cells are big enough to
# see (zoom 9 and up). Leaflet reports the view as input$map_zoom and
# input$map_bounds.
observe({
  req(input$map_zoom, input$map_bounds)
  proxy = leafletProxy("map") |> clearGroup("grid")
  if (input$map_zoom < 9) return()

  b = input$map_bounds
  lons = lon_edges[lon_edges >= b$west & lon_edges <= b$east]
  lats = lat_edges[lat_edges >= b$south & lat_edges <= b$north]
  if (length(lons) == 0 || length(lats) == 0) return()   # outside the grid

  # Each line runs across the visible part of the grid. NA breaks one line
  # from the next, so all of them go to the browser as a single layer.
  south = max(b$south, min(lat_edges)); north = min(b$north, max(lat_edges))
  west  = max(b$west,  min(lon_edges)); east  = min(b$east,  max(lon_edges))
```

```
    proxy |>
      addPolylines(
       lng = c(rep(lons, each = 3) + c(0, 0, NA), rep(c(west, east, NA), length(lats))),
          lat = c(rep(c(south, north, NA), length(lons)), rep(lats, each = 3) + c(0,
0, NA)),
        group = "grid", color = "#333333", weight = 1, opacity = 0.35,
        options = pathOptions(interactive = FALSE)
      )
  })

  output$where = renderText({
    pt = clicked()
    sprintf("Location: %.3f, %.3f", pt$lat, pt$lng)
  })

  # PLOTLY: renderPlotly, not renderPlot.
  # R sends the browser every month since 1979 ONCE. After that, zooming,
  # panning the range slider and hovering all happen in the browser, with no
  # trip back to R. That is the compute box moving to the user's side.
  output$plot = renderPlotly({
    d = monthly() |>
      mutate(precip = precip_mm * per_mm())

    last_3_years = c(max(d$month) - 3 * 365, max(d$month) + 31)

    plot_ly(
      d, x = ~month, y = ~precip, type = "bar",
      marker = list(color = "#2c7fb8"),
      # What the hover box says: "Mar 2025, 1.24 in"
     hovertemplate = paste0("%{x|%b %Y}<br>%{y:.2f} ", input$units, "<extra></extra>")
    ) |>
      layout(
        xaxis = list(
          title = "",
          range = last_3_years,                  # start zoomed in on recent years
          rangeslider = list(visible = TRUE)     # drag to see any part of the record
        ),
        yaxis = list(
         title = paste0("Precipitation (", tolower(names(units)[units == input$units]),
" per month)")
        )
      )
  })

  # The file name carries the location, so downloads do not overwrite each other.
  file_stub = reactive({
```

```
    pt = clicked()
    sprintf("%.4f_%.4f.csv", pt$lat, pt$lng)
  })

  output$dl_daily = downloadHandler(
    filename = function() paste0("daily_precip_", file_stub()),
    content = function(file) write.csv(in_units(daily()), file, row.names = FALSE)
  )

  output$dl_monthly = downloadHandler(
    filename = function() paste0("monthly_precip_", file_stub()),
    content = function(file) write.csv(in_units(monthly()), file, row.names = FALSE)
  )
}

shinyApp(ui, server)
```

Back to the four boxes. In Version 1, R draws every version of the plot on the server and sends a picture. In Version 2, R sends the numbers once, and the browser does the zooming and hovering on its own. A small piece of the compute box moved to the user's side. Hold on to that idea: the drought dashboard at the end moves all of it.

## 6.2 Tool 2: how dry is it right now?

"How dry is it?" is a different question. The answer is a drought index, such as the Standardized Precipitation Index (SPI), at a few timescales, and computing one means comparing recent precipitation against 30 years of history. You could do that on every click. The Montana Climate Office drought pipeline instead does the heavy computation once a day and writes the results as **Cloud Optimized GeoTIFFs** (COGs): ordinary GeoTIFFs arranged internally so that a client can ask for just the tiles it needs with HTTP range requests. One COG per index and timescale, about 1.5 MB each, sitting on cloud storage with no server logic in front of them.

So the point query goes from "pull 17,000 days and compute" to "read five numbers."

```
library(terra)

base   =   "/vsicurl/https://data2.climate.umt.edu/gridmet/derived/conus_drought_web/
latest/"
timescales = c("30d", "60d", "90d", "180d", "365d")

spi = rast(paste0(base, "spi_", timescales, "_rolling-30.tif"))
names(spi) = timescales
spi
```

```
class       : SpatRaster
size        : 584, 1386, 5  (nrow, ncol, nlyr)
```

```
resolution  : 0.04166667, 0.04166667  (x, y)
extent      : -124.7875, -67.0375, 25.04583, 49.37917  (xmin, xmax, ymin, ymax)
coord. ref. : lon/lat WGS 84 (EPSG:4326)
sources     : spi_30d_rolling-30.tif
              spi_60d_rolling-30.tif
              spi_90d_rolling-30.tif
              ... and 2 more sources
names       : 30d, 60d, 90d, 180d, 365d
time (days) : 2026-11-07
```

Note that nothing has been downloaded yet. `rast()` read the headers and stopped. Now ask for one pixel:

```
system.time(
  missoula <- terra::extract(spi, cbind(-113.99, 46.87)) / 100
)
```

```
   user  system elapsed
  0.015   0.001   0.085
```

```
missoula
```

```
   30d   60d  90d 180d 365d
1 0.83 -0.21 1.43 1.07 1.59
```

Five range requests, a fraction of a second, and we have the Standardized Precipitation Index for Missoula at five timescales. The app is just that call wrapped in a map click.

```
library(shiny)
library(bslib)
library(leaflet)
library(terra)
library(dplyr)
library(ggplot2)

# ---- Runs ONCE: point at the data, do not download it ----------------------
# These are Cloud Optimized GeoTIFFs (COGs) that the Montana Climate Office
# drought pipeline writes every morning. The /vsicurl/ prefix tells GDAL to
# read over HTTP with range requests: only the bytes we ask for are fetched.
base   =   "/vsicurl/https://data2.climate.umt.edu/gridmet/derived/conus_drought_web/
latest/"

timescales = c("30d", "60d", "90d", "180d", "365d")

# One SpatRaster with five layers, one per timescale. Nothing is read yet.
```

```r
spi = rast(paste0(base, "spi_", timescales, "_rolling-30.tif"))
names(spi) = timescales

# One layer to paint on the map: 60-day precipitation as percent of normal.
# This one we do read in full (about 1.5 MB) because we need every pixel.
pon = rast(paste0(base, "precip-pon_60d_rolling-30.tif")) / 100
pon_pal = colorNumeric(
  "BrBG", domain = c(0, 200), na.color = "transparent"
)

ui = page_sidebar(
  title = "Click anywhere in the lower 48",
  sidebar = sidebar(
    width = 380,
    p("Map: 60-day precipitation as percent of normal."),
    p("Click a point to read the Standardized Precipitation Index (SPI)
       at five timescales for that gridMET pixel."),
    plotOutput("bars", height = 280),
    tableOutput("vals")
  ),
  leafletOutput("map")
)

server = function(input, output, session) {

  output$map = renderLeaflet({
    leaflet() |>
      addProviderTiles(providers$Esri.WorldGrayCanvas) |>
      setView(lng = -97, lat = 39.5, zoom = 4) |>
      addRasterImage(clamp(pon, 0, 200), colors = pon_pal, opacity = 0.7) |>
      addLegend(
        position = "bottomleft", pal = pon_pal, values = c(0, 200),
        title = "60-day precip<br>% of normal", opacity = 0.8
      )
  })

  # input$map_click is NULL until the first click. req() stops everything
  # downstream until it exists, so there is no error on startup.
  clicked = reactive({
    req(input$map_click)
    input$map_click
  })

  # ---- Runs on every click: five tiny range requests, then done ------------
  vals = reactive({
    pt = clicked()
```

```
    v = terra::extract(spi, cbind(pt$lng, pt$lat))
    tibble(
      timescale = factor(timescales, levels = timescales),
      spi = as.numeric(v[1, ]) / 100     # stored as integers x100
    )
  })

  observe({
    pt = clicked()
    leafletProxy("map") |>
      clearMarkers() |>
      addMarkers(lng = pt$lng, lat = pt$lat)
  })

  output$bars = renderPlot({
    ggplot(vals(), aes(x = timescale, y = spi, fill = spi)) +
      geom_col() +
      geom_hline(yintercept = 0) +
      scale_fill_distiller(
        palette = "BrBG", direction = 1, limits = c(-2.5, 2.5),
        oob = scales::squish, guide = "none"
      ) +
      coord_cartesian(ylim = c(-3, 3)) +
      labs(x = "Timescale", y = "SPI") +
      theme_minimal(base_size = 14)
  })

  output$vals = renderTable({
    vals() |> mutate(spi = sprintf("%.2f", spi))
  })
}

shinyApp(ui, server)
```

Put the two tools side by side using the four boxes (UI, state, compute, data). The UI and state are nearly identical: a map and a click. The difference is all in compute and data. The downloader keeps both inside the click, because the user wants the raw record. The drought app moved compute out of the app entirely, into a pipeline that runs every morning, and changed the data from one huge cube into small files that are already the answer. That is why it answers in a fraction of a second.

Neither design is wrong. Ask what the user actually needs, then decide where the compute lives. The drought dashboard below is the second design taken all the way.

**Try it**

Swap `spi_` for `spei_` or `eddi_` in the file names. Then add a `selectInput()` so the user can choose the index. Where does the `rast()` call need to live now, and why?

**Ask the model**

"I want a map where users click a point and see a 40 year monthly precipitation time series for that location. Propose two architectures: one that computes on click, and one that precomputes. For each, what happens when 500 people use it the same afternoon?"

## 7 The other end of the spectrum: drought.climate.umt.edu

Every app so far runs on an R process somewhere. Here is the drought app with no R at all. Open the Montana Climate Office drought dashboard now and click somewhere.

https://drought.climate.umt.edu

There is no R and no Shiny. The page is HTML, CSS and JavaScript on GitHub Pages. The map is OpenLayers. The rasters are the same COGs you just read with `terra`, fetched by the browser with range requests as you pan and zoom, and colored by WebGL shaders on your graphics card. Change the index, the timescale or the color ramp, and no server is involved; the browser asks for a different file and recolors it locally. The point query you just built in R is the same operation, done in JavaScript, in milliseconds.

The "convergence of evidence" tool composites several indices per pixel, on the fly, in the shader. That is compute in the browser against data that was precomputed yesterday. Nobody is waiting on a server, and the server bill is the cost of storing some GeoTIFFs.

Here are the two ends of the spectrum in one table.

| | Shiny app | Static site (the dashboard) |
|---|---|---|
| UI | R ⟶ HTML, in the browser | hand-written HTML and JavaScript |
| State | R session on a server | JavaScript, in the URL |
| Compute | R on the server | JavaScript and GPU shaders, in the browser |
| Data | whatever R can open | COGs via range requests |
| Needs a server? | yes | no |
| Cost of a click | network round trip | local |
| Cost of 10,000 users | real money | none |
| What it costs you | little | you write JavaScript |

Neither is the right answer. A Shiny app is the right tool for sharing an analysis with your committee this week. The dashboard took a team months. The point is that you can now look at any interactive tool,

including one a model just generated for you, and place it on this table, and say what would have to move to get it where you need it.

> Ask the model
>
> Give it the URL of the drought dashboard and ask: "Which of the four boxes (UI, state, compute, data) live in the browser here, and what did the authors have to do to the data to make that possible? What would break if I tried to do the same thing with a 45 year daily NetCDF file?"

## Sharing what you built

- **Just the map:** a leaflet or mapview object saved with `htmlwidgets::saveWidget(m, "map.html")` is a single file you can email or drop on any web server.
- **A Shiny app, hosted for you:** copy an app chunk into its own folder as a file named `app.R`, then `rsconnect::deployApp("that_folder")` puts it on shinyapps.io, which has a free tier.
- **A Shiny app, on your own server:** Shiny Server (open source) or Posit Connect.
- **Beyond R:** when you need the far end of the table, the ingredients are OpenLayers or MapLibre, COGs on object storage, and a static host. Ask a model to scaffold it; you now know what to check.

## Appendix: reading an app you did not write

Use this on your own apps, on apps from a paper's supplement, and above all on apps a model writes for you. Six questions.

1. **What runs once, and what runs on every input?** Anything expensive inside a `render*()` or `observe()` block runs on every change. Should it?
2. **What crosses the network, and when?** Downloads inside reactive blocks are the most common cause of a slow app. Precompute, cache, or move them up.
3. **What is precomputed, and what is computed on demand?** If the same calculation is done for every user, do it once in a pipeline and serve the result.
4. **Where does each of the four boxes live?** Browser or server? Could any of them move?
5. **What happens with 100 simultaneous users?** Shared data at the top of the file is loaded once. Everything in `server()` is per session.
6. **What would have to change to need no server at all?** Even if you never do it, the answer tells you where the architecture is bound to the server, and whether that is on purpose.

If you can answer those for an app, you can direct a model to fix it. If you cannot, you are trusting the model's defaults, and the model's defaults are "make it work," not "make it work for your users."

## Resources

- Mastering Shiny, Hadley Wickham. Free, and the reactivity chapters are the best explanation of the reactive graph.
- Shiny gallery: every app comes with source.
- bslib documentation: modern layouts, cards, value boxes, theming.
- Leaflet for R and mapgl.
- Cloud Optimized GeoTIFF: what they are and why they matter.
- MCO drought dashboard source and the data pipeline behind it.