INTRODUCTION

Modern data scientists are no longer required to just crunch some numbers and spit out a statistic. Instead, it is kind of expected that we can

- interact with **databases, cloud storages and web APIs**,
- **use AI to find and extract information** from large bodies of text,
- visualize our findings into **beautiful data visualizations** (charts or tables),
- **built automations** that run these calculations on a regular schedule, and then
- wrap all of this into a **nicely formatted website, document, dashboard or web app** that stakeholders can interact with.

To be fair, this is still the good old data data science process:

- Get data
- Clean and transform
- Visualize and then model
- Communicate your findings

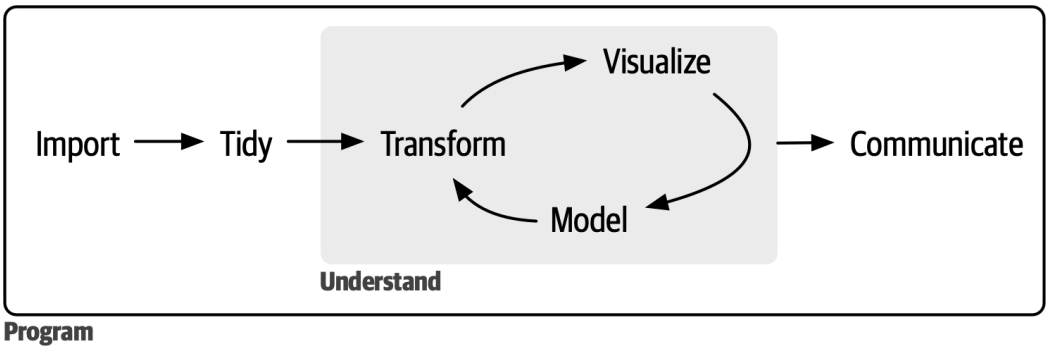


Figure 1: Data science process as depicted in R for Data Science, included here under CC BY-NC-ND 3.0 US.

It's just that **each step of the way got broader**. More technologies, more expectations on robustness and more things to do. I don't expect this to change in 2026. So let's have a look at what the modern R ecoverse has to offer to accomodate these requirements.

FUNDAMENTALS & CODING BEST PRACTICES

The Fundamentals

You want to make sure that you have a good understanding of these before diving into other data journeys. Typically, these basics involve some numbers crunching using tabular data. For example, you could

- take a dataset,
- select specific columns you need,
- compute a new one based on the data,
- and then calculate summary statistics,
- which you sort in a meaningful order.

Sounds like a whole lot? It is. But in the Tidyverse, it's a simple chain of a few fundamental functions:

```
palmerpenguins::penguins |>
  select(
    species, body_mass_g, flipper_length_mm
  ) |>
  mutate(
    ratio = body_mass_g / flipper_length_mm
  ) |>
  summarise(
    mean_weight_per_mm_flipper = mean(
      ratio,
      na.rm = TRUE
    ),
    .by = species
  ) |>
  arrange(mean_weight_per_mm_flipper)
# # A tibble: 3 × 2
#   species   mean_weight_per_mm_flipper
#   <fct>                          <dbl>
# 1 Chinstrap                       19.0
# 2 Adelie                          19.5
# 3 Gentoo                          23.3
```

These simple building blocks are what makes the Tidyverse so versatile. You can chain them together as you please. You can learn more about that here:

Use These 6 Functions For All Data Projects | Step-By-Step Tutorial

Tidyverse fundamental building blocks explained for beginners.

Watch on YouTube →

Once you have these fundamentals covered you'll likely want to wrap your code into reusable functions. After all, copy-and-pasting is what feels easy to do but will leave you with a mess pretty quickly. Nicely designed functions are an antidote for that.

You can start writing functions using the basic function skeleton:

```
my_custom_function <- function(x, y) {
  x / y
}
my_custom_function(1:5, 2)
# [1] 0.5 1.0 1.5 2.0 2.5
```

But pretty soon, you'll want to learn how to make your functions more efficient. This can involve simply thinks like thinking about **naming conventions** or go all the way to learn about advanced operators like `{{ }}` **(curly-curly)** or `...` **(dot-dot-dot)**. Here are a few more ideas:

7 Easy Techniques To Write Smooth R Functions

Better functions -> better code

Watch on YouTube →

Or another great strategy to get better at R programming is to think about what to avoid. When I was starting to learn R I sure could have used a list of mistakes to avoid. So here are a few things I wish I'd known sooner:

12 R Programming Sins I Wish I Avoided Sooner

Sometimes it's not about what to learn but about what to unlearn.

Watch on YouTube →

8 Things I Wish I'd Known Before Using R In Business

In a business setting people kind of expect you to know a few particular things about programming that are not often talked about.

Watch on YouTube →

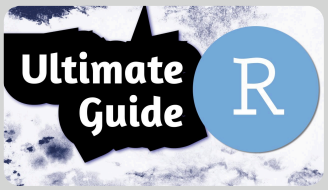
Taking It Further

There are whole bunch of things you can do to improve your general R coding skills. For example, you can think about

- programming productivity in general,
- specific data formats

Productivity

Learning about your IDE, i.e. the tool you use to write code, is a good starting place. If you're using RStudio, then this will teach you what you need:

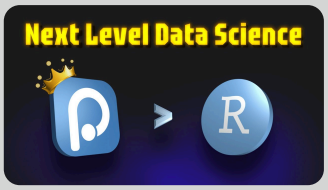


Write R Code Faster With These RStudio Shortcuts and Settings | Step-By-Step Tutorial

RStudio has so many optimizations for R programming. Still one of the best IDEs for data science with R.

Watch on YouTube →

But for a more modern touch, I really enjoy using Positron (which also works well for Python):



Why R Users Will Love Positron (and how to set it up for success)

This is the getting started guide for you if you want to learn how Positron helps you write code.

Watch on YouTube →

Data Scientists Will Love These Positron Productivity Tricks

And this teaches you many more productivity tricks within Positron.

Watch on YouTube →

Data Cleaning Tools

If you want to get better at data cleaning, my go-to approach is to become better at the tools that the `dplyr` package offers you. You can start small and write simple code like this:

```
modeldata::ames |>
  select(
    Neighborhood,
    BsmtFin_SF_1,
    BsmtFin_SF_2,
    Bsmt_Unf_SF,
    Total_Bsmt_SF,
    First_Flr_SF,
    Second_Flr_SF,
    Wood_Deck_SF,
    Open_Porch_SF
  ) |>
  mutate(
    BsmtFin_SF_1   = BsmtFin_SF_1 / 10.7639,
    BsmtFin_SF_2   = BsmtFin_SF_2 / 10.7639,
    Bsmt_Unf_SF    = Bsmt_Unf_SF / 10.7639,
    Total_Bsmt_SF  = Total_Bsmt_SF / 10.7639,
```

```
    First_Flr_SF   = First_Flr_SF / 10.7639,
    Second_Flr_SF  = Second_Flr_SF / 10.7639,
    Wood_Deck_SF   = Wood_Deck_SF / 10.7639,
    Open_Porch_SF  = Open_Porch_SF / 10.7639
  ) |>
  summarize(
    BsmtFin_SF_1   = mean(BsmtFin_SF_1),
    BsmtFin_SF_2   = mean(BsmtFin_SF_2),
    Bsmt_Unf_SF    = mean(Bsmt_Unf_SF),
    Total_Bsmt_SF  = mean(Total_Bsmt_SF),
    First_Flr_SF   = mean(First_Flr_SF),
    Second_Flr_SF  = mean(Second_Flr_SF),
    Wood_Deck_SF   = mean(Wood_Deck_SF),
    Open_Porch_SF  = mean(Open_Porch_SF),
    .by = Neighborhood
  )
```

But that turns into a mess pretty quickly. And it is also super annoying to type this mess. Luckily, `dplyr` allows you to write this quickly and effectively like this:

```
modeldata::ames |>
  select(Neighborhood, contains('SF')) |>
  mutate(
    across(
      .cols = contains('SF'),
      .fns = \(sqft) sqft / 10.7639
    )
  ) |>
  summarize(
    across(
      .cols = contains('SF'),
      .fns = mean
    ),
    .by = Neighborhood
  )
```

I've summarized how to go from this first beginner-level code to pro in this video:

5 Levels of Data Cleaning Every R User Must Master (Beginner to Pro in 5 Steps)

How to write data cleaning code more efficiently.

Watch on YouTube →

And there's more that `dplyr` allows you to do. That's why I've spent one whole part in my 5-part data cleaning master class about these kind of productivity tricks. If you're curious, you can check it out here:

Productivity Boosts for Data Transforms

Get a deep understanding of the basics of computing and summarizing columns. Improve your productivity with advanced tricks.

Check out course page →

17 lessons | 2 hours of video material

And if it's not code writing speed that you're worried about but rather computation speed, then R has got something else in store for you. There's the `data.table` package that can handle super large data extremely fast. But there's a caveat. Its syntax takes some time getting used to.

For example, let's use the same dataset from before (but clean its names a bit).

```
library(dplyr)
library(data.table)
ames <- modeldata::ames |>
  janitor::clean_names()
```

As usual, you could chain together steps with `dplyr` code to calculate some stuff:

```
ames |>
  filter(sale_price > 300000) |>
  summarize(
    mean_sale_price = mean(sale_price),
    n_houses = n(),
```

```
    .by = neighborhood
  )
```

Well, here's the exact same operation in `data.table`:

```
df_ames <- as.data.table(ames)
df_ames[
  sale_price > 300000,
  list(
    mean_sale_price = mean(sale_price),
    n_houses = .N
  ),
  by = neighborhood
]
```

See what I mean by having to get used to the syntax? While `dplyr` uses chained together steps using human-readable function names, `data.table` uses very condense notation. Nothing wrong about either of these approaches.

What matters is that `data.table` is insanely fast and can work with lots of data. That's why (depending on your situation) it can be useful to spend some time figuring out the `data.table` syntax. Here's where you could lean back and check out my explanations.

Master R Data Cleaning: dplyr vs data.table

Doing common data cleaning operations with both `dplyr` and `data.table`

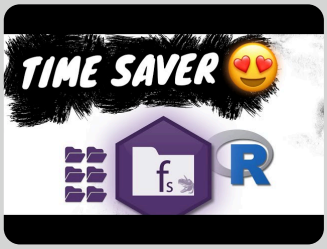
Watch on YouTube →

But in a lot of cases it is not about what tool you use to work with messy data. What often matters more is the file type the data is stored in and the data type itself. To support you with these struggles I've created a bunch of tutorials as well:

How to Clean Up Messy Excel files in R | 7 Easy Strategies

It is a sad fact of life that data stored in Excel files is particularly messy. Here are a few strategies that help to deal with that.

Watch on YouTube →



{fs} Is The R Package You Didn't Know You Needed

When you have to deal with a bunch of files on your hard drive, `fs` becomes invaluable.

Watch on YouTube →

Data Extraction with R & {stringr} | Step-by-Step Tutorial

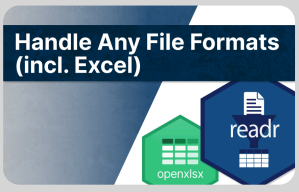
Text data is everywhere so it is worth figuring out how to deal with that

Watch on YouTube →

Of course, no single tutorial can cover it all. But parts of my 5-part data cleaning master class can cover waaay more ground. If you're curious about those, check them out here:

- You can think about how to deal with particular file types…

Get Data Into R Using Any File Format



Learn how to read any file and get it from a messy state into a usable format. This will be particularly useful for Excel files.

Check out course page →

21 lessons | 2.5 hours of video material

- …or you can find out how to deal with text data…

Unlock Your Text Processing Skills

Knowing how to work with text data (and regex in particular) unlocks so many doors for you. I'll show you how that works.

Check out course page →

27 lessons | 3 hours of video material

- …or you can finally learn how to handle temporal data better:

Work With Times And Dates

Working with times and dates is always painful. But your life becomes easier when you use powerful functions from `lubridate`.

Check out course page →

15 Lessons | 2 hours of video material

Finally, if you want to learn a more high-level skills, then it's a good idea to learn more about **functional programming (FP)**. This kind of programming might be weird at first but it is a game changer. For example, imagine that you have a bunch of csv-files

```
csv_files <- fs::dir_ls(
  'data/Part_5/',
  regexp = 'fake_data_.+\\.csv'
)
```

Now, let's bind them together into one data frame. A simple approach would be to write a for-loop:

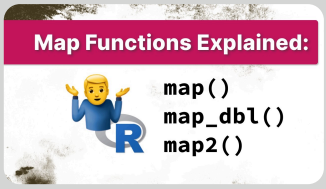
```
csv_list <- list()
for (k in seq_along(csv_files)) {
  csv_list[[k]] <- read_csv(csv_files[k])
}

csv_list |> bind_rows()
```

But with FP, you can do something quicker. Instead of writing a verbose for-loop you can do this:

```
csv_files |> map_dfr(read_csv)
```

That's pretty convenient and this way of programming unlocks many new avenues for you. Here's a primer on what FP can do for you:



Run Many Calculations All at Once With Map Functions | Step-by-Step R Tutorial

Simple things FP can do for you.

Watch on YouTube →

And if you're ready to dive deep into functional programming, there's also more in-depth course:

Level Up Your Workflow

Unlocking the hard but insanely useful functional programming paradigm will make you so much more productive.

Check out course page →

20 Lessons | 2 hours of video material

Data storage and retrieval

If you're working with lots of data, then chances are that they are not available simply as a file on your computer. Most commonly, you will have to grab data via a SQL database. Luckily R has multiple pretty good ways of interacting with such databases.

How to Get Data from SQL Databases With R | Step-by-Step Tutorial

Your primer on how to work with databases.

Watch on YouTube →

And the nice thing is: You might not even have to adjust your beautiful Tidyverse-style code. You see, code that normally looks like this…

```
palmerpenguins::penguins |>
  select(
    species, body_mass_g, flipper_length_mm
  ) |>
  mutate(
    ratio = body_mass_g / flipper_length_mm
  ) |>
  summarise(
    mean_weight_per_mm_flipper = mean(
      weight_per_mm_flipper,
      na.rm = TRUE
    ),
    .by = species
  ) |>
  arrange(mean_weight_per_mm_flipper)
```

… will also work when the dataset is replaced with a database connection. This can look like so:

```
con <- DBI::dbConnect(...) # Fill with
connection settings
tbl(con, 'penguins') |>
  select(
    species, body_mass_g, flipper_length_mm
  ) |>
  # ... rest of code omitted
```

That's pretty neat if you ask me. And naturally, there's a tutorial that will teach you the most common SQL things you need:

How to Easily Solve 10 Common SQL Tasks With R & {dbplyr}

You can do a whole lot with just these 10 basic operations

Watch on YouTube →

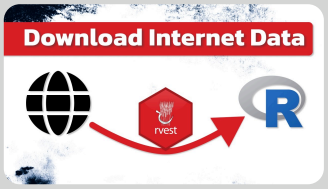
But it's not always a database you have to work with. Sometimes, your data is stored in a cloud storage like AWS S3. For these scenarios, the kind people at Posit have created a package that makes data scientists's life much easier. Check out how `{pins}` work:

Never Lose Data Again: Use R's {pins} package

Cloud data versioning made easy

Watch on YouTube →

Or maybe you need to scrape some web data? In that case, `{rvest}` is your go-to tool:



How to Web Scrape Tables With R & rvest | Step-by-Step Tutorial

Particularly useful for extracting tables.

Watch on YouTube →

And sometimes you have to send HTTP requests to some web API. Sounds complicated, doesn't it? But it's actually pretty easy once you understand two things:

1. All of this "HTTP requests" and "web API" stuff just means that you throw a bunch of code at an internet service and you get a bunch of code back (usually JSON-formatted data).
2. There is a pretty nice package that makes these request things pretty easy.

And I know that this might still sound intimidating. So let's see this in action. I've created a tutorial for going from very simple API requests to very hard ones (which require authentication). Check it out here:

How to Get Data From APIs with R & {httr2} | Ultimate Step-By-Step Tutorial

From first steps with APIs to complicated OAuth flows.

Watch on YouTube →

Programming Patterns / Software engineering practices

Finally, even though we're mainly talking about data science, it can't hurt to pick up a few tricks from software engineering. For starters, we should document our package versions to make our code more future-proof and reproducibe. The `renv` package helps you with that.

Robust R Code That Will Work Forever With {renv}

Because you still want to use your code in 3 years.

Watch on YouTube →

Or we can package parts of our code into objects. The S7 frameworks makes that pretty easy. And it will help you write better functions.

4 Reasons Why Every R Users Should Know The S7 Package

Simple objects without the overhead of R6 classes.

Watch on YouTube →

DATA VISUALIZATION

The Fundamentals

In this section, we want to talk about **data visualizations**. This includes not only charts but also tables (interactive and static ones). First, you want to learn about the three fundamental packages that power these things:

1. The ultimate package for creating charts is `{ggplot2}`.
2. For static tables `{gt}` is my go-to package.
3. And when it comes to interactive tables, nothing beats `{reactable}`.

Beautiful Charts with R & ggplot2 (Step-By-Step Tutorial for Beginners)

If you think ggplot is complicated, then this video is for you.

Watch on YouTube →

How to Make Great Tables with R & {gt} | Step-by-Step Tutorial

Everything you need to know about {gt}.

Watch on YouTube →

How To Create Interactive Tables With R & reactable | Step-By-Step Guide

A quick start for {reactable}.

Watch on YouTube →

Taking It Further

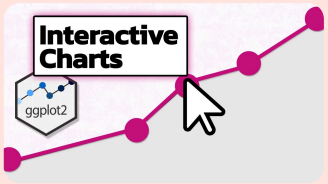
Once you've understood the basics, it's time to learn more about these packages. For charts, this can mean learning to build the most common chart types.

5 Essential Charts Every R User Should Make With ggplot2

If you can't do these, then you can't do dataviz yet.

Watch on YouTube →

And once you can do that, you can take it up a notch and make your charts interactive:



How to Make Any ggplot Interactive With {ggiraph} | Step-By-Step Tutorial

This is surprisingly easy.

Watch on YouTube →

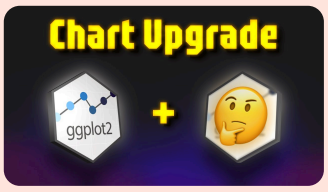
This can even be used nicely in connected plots. All you need to know for that is how to compose plots using `{patchwork}`.

How to Combine ggplots with {patchwork} | A Step-By-Step Tutorial

Thank god, there's patchwork. Compositing multiple plots used to be really tedious.

Watch on YouTube →

Of course, that's just one of many routes you can take. Inside the ggplot ecoverse, there's lots to discover. You could experiment with one of the many other ggplot packages:



9 R Packages That Make ggplot2 Even Better

Yes, ggplot can become even better.

Watch on YouTube →

And for finding out how to create specialized charts, there's always a dedicated video to teach you how to do that:

- 🎥 How to Create **Dumbbell Plots** with R & ggplot2
- 🎥 How to Create **Donut and Pie Charts**
- 🎥 How to Create **Diverging Bar Charts** With {ggplot2}
- 🎥 How To Create **Interactive Maps** with R
- 🎥 How to Create **Correlation Heat Maps** With {ggplot2}
- 🎥 Visualize **Patterns With Calendars** & ggplot2
- 🎥 How to Create **Upset Charts** With {ggplot2}
- 🎥 Create **Raincloud Plots** with ggplot2 and {ggist}
- 🎥 A SIMPLE Guide to Create **Bump Charts** with ggplot2 & {ggbump}

As for getting better at interactive tables, why don't you try recreating this elaborate table?

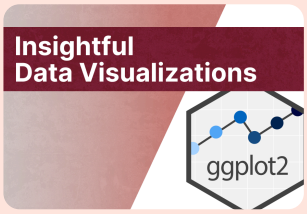
How to Master Interactive Tables With R & Reactable

This was really tricky to pull off.

Watch on YouTube →

And, of course, if you want to get better at data visualization, there's always my self-paced video course.

Master Data Visualization With ggplot



In '**Insightful Data Visualizations for "Uncreative" R Users**', I show you how to Leverage {ggplot2} to make **charts that communicate effectively without being a design expert**.

Check out course page →

31 lessons | 6 hours of video material

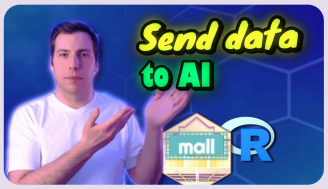
AI WITH R

Nowadays, everyone want to use AI by which they usually mean large language models (LLMs). And for good reason. LLMs are pretty powerful and allow for many cool analyses and automations on text data.

Of course, they are not the magic bullet the hype makes them out to be. But they are nevertheless pretty powerful. And a lightweight way to get started using LLMs is given by the `{mall}` package. It allows you to run sentiment analysis on data frames. This can look like so:

```
tib <- tibble::tibble(
  id = 1:4,
  adjective = c(
    'painful',
    'hungry',
    'cheerful',
    'exciting'
  )
)
tib |>
  llm_sentiment(col = adjective)
## # A tibble: 4 × 3
##      id adjective .sentiment
##   <int> <chr>     <chr>
## 1     1 painful   negative
## 2     2 hungry    negative
## 3     3 cheerful  positive
## 4     4 exciting  positive
```

More details on what `{mall}` can do for you can be found inside this video:



AI Data Science with R: Analyze data.frames with the {mall} package

Your entry into text data analysis with R.

Watch on YouTube →

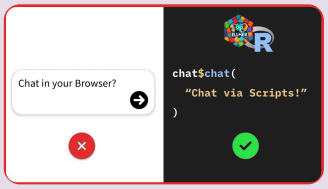
But you'll probably soon outgrow this package. Instead, you might like to use the `{ellmer}` package. It's a bit more complicated to set up but it allows you to do soooo much. Nowdays, it's my premier way of interacting with LLMs through R.

You can use it to

- set up a "chat" with an LLM, and then
- use some of {ellmer}'s chat methods.

This can look like this:

```
library(ellmer)
chat_oai <- chat_openai(
  system_prompt = '
    You are an AI bot that ONLY REPLIES WITH
"BANANA". Reply only this AT ALL COSTS'
)
chat_oai$chat(
  "What's your favorite fruit?"
)
## BANANA
```



Master Multiple AI APIs With R And The {ellmer} Package

Getting started with {ellmer}.

Watch on YouTube →

Now, this sounds like a pretty boring thing but it's incredily helpful. This is the case because there are many more methods inside of a `chat` object. For example, you can tell your LLM about an R function that you wrote. And now it can "use" this function based on some text data input.

This is pretty convenient in cases where you can't use traditional programming to clean data so that it can be sticked into a function. But you can let an LLM

extract the correct input for your functions for you. And because it uses regularly programmed functions after that, you can limit the risk of hallucinated output. There are some steps involved to set this up, though. So here's a video that shows you how it's done:

How to Enhance R Functions With AI Using {ellmer}

Equipping LLMs with regular functions.

Watch on YouTube →

And my favorite method from an `{ellmer}` chat object allows you to extract data from inputs and turn those into a data.frame. For example, you might

- extract all text from a PDF (like an invoice)…

```
txt_pdf <- pdftools::pdf_text('dat/
invoice.pdf')
```

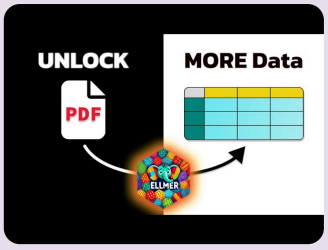
- …define all of the things you want to extract…

```
type_invoice_position <- type_object(
  'A single invoice position',
  position = type_string('Description of the
position'),
  quantity = type_number('Quantity/Number of
hours'),
  rate = type_number('Price per piece/Hourly
rate'),
  total = type_number('Total amount of
invoice positon'),
  currency = type_enum(
    'currency of invoice position',
    values = c('USD', 'EUR', 'Other')
  )
)
```

- … and then let your favorite LLM find the reqired information.

```
chat <- chat_openai(
  system_prompt = '
  You area an invoice extraction assistant.
The user
  will provide you with text from an invoice
and you are
  to extract every position from it.'
)
## Using model = "gpt-4o".
chat$chat_structured(
  txt_pdf,
  type = type_array(
    'List of all available invoices',
    items = type_invoice_position
  )
)
##        position quantity rate total
currency
## 1    Areca palm       20  2.5  50.0
USD
## 2  Majesty palm       35  3.0 105.0
USD
## 3 Bismarck palm       15  2.5  37.5
USD
```

And if you now think *"that's pretty cool but I still need more details to understand what's going on"*, then I've got you covered:



How To Turn Messy PDFs Into Clean Data Frames With R & {ellmer}!

Extracting data from texts with LLMs.

Watch on YouTube →

COMMUNICATING DATA INSIGHTS

We have already covered data visualization techniques with charts and tables. But creating these is often not the end of the story. Usually, you want to embed them into one of these:

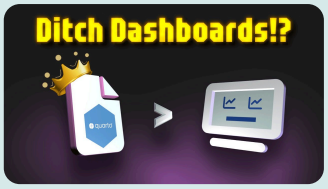
- Websites
- Dashboards
- PDF reports
- Shiny Web Apps
- Word documents
- Powerpoint slide decks

Most of them can be quite easily facilitated via Quarto. This allows you to mix code (and its output) with text effortlessly. Let's go through the different formats one-by-one.

Websites

Let's start with creating websites. And you might think that the best place to start would be to create a dashboards. After all, they are super common and a lot of people think that you need this kind of thing to communicate your findings to lots of different audiences. But that's not necessarily the case.

In fact, static websites are easier to set up and can work just as well. Especially if you combine that with Quarto's parameterized reporting engine. Check out how that works here:



How to Automate Data Reports with Quarto (Beginner's Guide)

Making HTML data reports really easy.

Watch on YouTube →

Of course, if you insist, you can also create a static dashboard with Quarto quite fast.

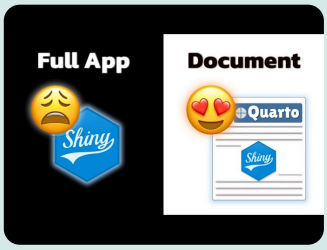
How to Create Dashboards with R, Python, OJS or Julia | Step-By-Step Guide

Dashboards made easy.

Watch on YouTube →

These beware that both of these options create static websites. This means that all outputs are pre-calculated and the the content isn't dynamically calculated. For that you usually need a web app.

In R, this means creating a Shiny app. However, that's usually a lot of work because you have to set up a user interface. But with Quarto you can also get the best of both worlds: Create a website and enhance it with interactive Shiny elements.



Build Interactive Apps in Minutes (Not Days) Using Quarto & Shiny

Speed of setting up a website with Quarto, versatility of a Shiny app. Nice!

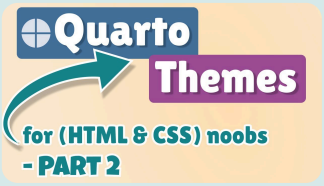
Watch on YouTube →

Naturally, you'll want to make all of your websites look (somewhat) nice. For that you'll need HTML & CSS knowledge. I've set up simple guide for you that teaches you just enough of that to get the job done (in combination with Quarto).

How to style your Quarto docs without knowing HTML & CSS

Styling Quarto themes by changing variables. Easy.

Watch on YouTube →



An easy way to style your Quarto docs without knowing HTML & CSS

Adding custom styles by finding the right HTML & CSS. Not so hard after all.

Watch on YouTube →

PDF Documents

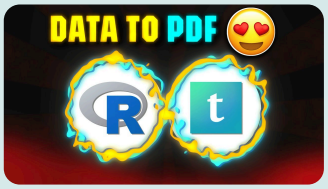
Of course there is also a large demand for PDF outputs. Lots of stakeholders may want to get a data report in such a format. Thankfully, this is also pretty easy to do with Quart. It allows you to switch the output format to PDF files created via Typst.

Quarto & Typst Are Fantastic for High-Quality PDF Reports

Typst creates beautiful PDFs and Quarto weaves together the necessary content from your code.

Watch on YouTube →

And if you like Typst, you could also leave Quarto altogether and generate the Typst file directly from R.



Every R User Can Make Data Reports With Typst (using this technique)

Sometimes it's easier to use Typst directly and inject the data via R manually.

Watch on YouTube →

The main reason why you might want to do that is because sometimes Quarto doesn't translate custom styling well to Typst. You can see this most noticably when trying to embed a `{gt}` table into a Typst PDF. There are some workarounds for making this work that I show you in this video:

How to Finally Bring Your Tables into Typst PDFs (using {gt})

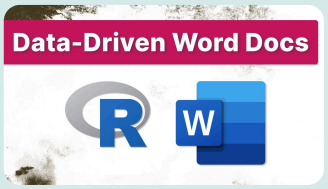
When you want to use your custom tables inside your Typst PDF documents.

Watch on YouTube →

Microsoft Office Documents

And of course there's a format all data scientists dread. It's any output from Microsoft Office. This is never that fun to work with from a programming perspective.

Luckily, the officeverse has a few R packages that make our lifes easier in these situations. They are super useful when it comes to creating Word documents or even Powerpoint slidedecks.



How to Fill Word Files With Data Using R & officer | Step-by-Step Tutorial

Filling word files with content programmatically.

Watch on YouTube →

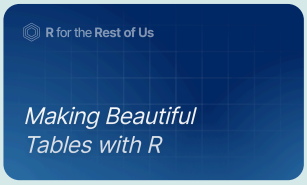
How To Create Data-Driven Slide Decks With R & {officer}

Churning out slide decks faster than McKinsey.

Watch on YouTube →

In both of these settings it's not uncommon to want to include tables as well. This is where I'd recommend a table packages other than `{gt}`. You see, the `{flextable}` package can create nice tables too and is part of the officeverse. And it's also pretty easy to pick up: There's a whole free book online. For an in-depth video course I can recommend this one:

Making Beautiful Tables With R



Learning how to create tables that communicate effectively is just a matter of learning a few important principles (and how to implement them with `{flextable}`.)

Check out course page →

16 lessons | 3 hours of video material

Web Applications

Finally, you can also create web apps using the `{shiny}` package. This is a great way to display data interactively when the previous options are not sufficient. Or you can use it as a means to let the user

- interact with models and other services or
- generate dynamic data outputs on the spot.

But web development is also a complex topic to dive into. That's why I have two separate playlists for that. For starters, there's the R-Shiny playlist:

How to Build Dashboards & Web Apps With R-Shiny (Playlist)

From basic R-Shiny features to elaborate web apps.

Watch on YouTube →

But the more you learn, the more you also want to learn a little bit about the web dev technologies behind this. There's a playlist for that as well:

Web Development For R Users (Playlist)

Getting to know HTML & CSS from an R perspective.

Watch on YouTube →