Setting Up Multi-Language Quarto Documents on macOS

Configuring R, Python, Julia, and Observable JS to coexist in a single rendered document

Ronald 'Ryy' G. Thomas

2026-04-30

Table of contents

Figure 1: Quarto logo: the open-source publishing system that ties multiple languages together in a single document.

Quarto provides a unified authoring framework for scientific and technical publishing.

1 Introduction

The plumbing behind a multi-language Quarto document is more extensive than the Quarto website suggests. The documentation makes it look effortless: drop an R chunk here, a Python chunk there, maybe some Julia for good measure, and hit Render. The reality involves an afternoon of chasing PATH variables, fixing reticulate configurations, and troubleshooting why Julia refuses to find its own packages.

The problem is not that any single language is hard to install. Each one has a clean installer and decent documentation. The difficulty is making them all coexist inside a single rendering pipeline where R orchestrates Python through reticulate, Julia through JuliaCall, and Observable JS runs natively in the browser. Small misconfigurations in any layer can cascade into cryptic errors.

This post documents the lessons learned getting all four languages to cooperate in a single Quarto document on macOS, written from the perspective of working through the process rather than having mastered it.

More formally, this post documents the polyglot capability of the Document layer of the Workflow Construct described in post 52. Quarto is the construct's recommended document layer, subsuming the roles previously divided between R Markdown, Jupyter, and standalone LaTeX. This post addresses the specific case in which a single document spans R, Python, Julia, and Observable, which is the configuration that surfaces the most layer-interaction subtle failures.

1.1 Motivations

The following considerations motivated this exploration:

- Comparing plotting libraries across languages side by side, using the same dataset, in a single rendered document.
- Collaborators who work in Python need a way to view analyses that mix R and Python code without installing R themselves.
- Julia's speed for numerical computation makes it worth exploring, particularly when called from inside an R session.
- Observable JS examples in Quarto documentation suggest useful integration patterns worth understanding.
- Documenting the setup process forces understanding of each component rather than blindly following a tutorial.
- A reproducible multi-language environment can be reused across future projects.

1.2 Objectives

1. Install and configure R, Python, Julia, and Observable JS to work within a single Quarto document on macOS.
2. Verify each language integration independently before combining them.
3. Render a minimal multi-language Quarto document that executes code in all four languages.
4. Document common setup pitfalls and their solutions so future attempts take minutes rather than hours.

Errors and better approaches are welcome; see the Feedback section at the end.

Figure 2: A workspace ready for multi-language exploration.

2 Prerequisites and Setup

Before starting, make sure you have the following available on your macOS system:

- **macOS** (tested on Monterey and later)
- **At least 8 GB RAM** (each language runtime consumes memory; running all four simultaneously is demanding)
- **Roughly 5 GB free disk space** for all installations
- **Admin privileges** for Homebrew and installer packages
- **Internet connection** for downloading packages

Background. This guide assumes familiarity with the macOS Terminal and basic comfort installing software from the command line. You do not need prior experience with all four languages — that is the whole point.

3 What is Multi-Language Quarto?

Quarto is an open-source scientific publishing system that extends the idea behind R Markdown to multiple languages. A single `.qmd` file can contain code chunks written in R, Python, Julia, and Observable JS. When you render the document, Quarto routes each chunk to the appropriate language engine, collects the output, and weaves everything into a unified HTML, PDF, or Word document.

The key insight is that R acts as the orchestration layer. Python chunks are executed through the reticulate package, Julia chunks through JuliaCall, and Observable JS chunks run natively in the browser when producing HTML output. This means your R installation needs to know where Python and Julia live on your system, and each language needs its own packages installed separately.

4 Getting Started: Basic Setup

The setup proceeds in four stages. Completing and verifying each stage before moving to the next is recommended. Debugging a four-language environment is much harder than debugging a single-language problem.

4.1 Stage 1: R and RStudio

1. Install R from CRAN
2. Install RStudio
3. Install Quarto

Then install the R packages that bridge to other languages:

```
install.packages(c(
  "dplyr",
```

```
  "ggplot2",
  "scales",
  "reticulate",
  "JuliaCall"
))
```

The first three packages are for data work. The last two are the bridge packages: reticulate connects R to Python, and JuliaCall connects R to Julia.

4.2 Stage 2: Python

1. Install Anaconda (recommended) or Miniconda
2. Install required Python packages:

```
conda install pandas seaborn matplotlib
```

3. Verify Python configuration in R:

```
library(reticulate)
py_config()
```

The output of `py_config()` should show your Anaconda Python path. If it does not, you need to tell reticulate where to look.

Common Python Issues

- If `py_config()` does not show your Anaconda installation, specify it explicitly:

  ```
  use_python("/path/to/anaconda/bin/python")
  ```

- Seaborn styling requires specific syntax:

  ```
  sns.set_theme(style="whitegrid")
  ```

  Do not use `plt.style.use()` for Seaborn themes.

4.3 Stage 3: Julia

1. Install Julia from julialang.org
2. Add Julia to PATH (usually automatic with the installer)
3. Install required Julia packages:

```
using Pkg
Pkg.add([
  "UnicodePlots",
```

```
  "DataFrames",
  "CSV",
  "Statistics"
])
```

4. Set up the Julia-R connection:

```
library(JuliaCall)
julia_setup()
```

> **Julia Gotchas**
>
> - Some Julia plotting backends have system dependencies that may not be present on macOS by default.
> - Use UnicodePlots for the most reliable terminal-based results.
> - Ensure the Julia working directory has write permissions.

4.4 Stage 4: Observable JS

No separate installation is needed. Observable JS runs in the browser when Quarto produces HTML output. Your Quarto document must include the dependency declaration in the YAML header:

```
dependencies:
  - name: "@observablehq/plot"
    version: latest
```

That is it. Observable chunks will render automatically when you build to HTML.

Figure 3: Python logo: the second language engine in the multi-language Quarto setup.

Python integration through reticulate is often the first multi-language feature Quarto users encounter.

5 Verifying Each Language

Before combining all four languages in a single document, verify that each integration works independently. This saves considerable debugging time.

5.1 R Environment Check

```
sessionInfo()
```

This confirms your R version and loaded packages. Look for the version number and ensure the base packages are listed.

5.2 Python Environment Check

```
library(reticulate)
py_config()
```

Confirm that the Python path points to your Anaconda installation, not the system Python. The output should show the version number and the path to the Python binary.

5.3 Julia Environment Check

```
library(JuliaCall)
julia_eval("versioninfo()")
```

This should print the Julia version and system information. If JuliaCall cannot find Julia, check that the Julia binary is on your PATH.

6 Testing a Minimal Example

Once all four verifications pass, test a minimal multi-language document. Create a new `.qmd` file with these chunks:

```
print("Hello from R!")
```

```
print("Hello from Python!")
```

```
println("Hello from Julia!")
```

```
md`Hello from Observable JS!`
```

If all four chunks produce output when you render, your environment is properly configured. If any chunk fails, go back to the verification step for that specific language.

7 Full Example: Palmer Penguins Visualization

Below is a complete, documented version of a multi-language visualization example. Each section creates the same scatterplot — bill depth versus bill length by species — in a different language, showcasing how the same data can flow through four ecosystems.

7.0.1 Data Preparation in R

R loads the data and exports a CSV that the other languages can read:

```r
library(dplyr)
library(ggplot2)
library(scales)

data <- penguins |> na.omit()

write.csv(data, "penguins.csv", row.names = FALSE)

glimpse(data)
```

R prepares and exports the clean data. The other languages read from this shared CSV, which ensures consistency across all four visualizations.

7.0.2 Visualization in R (ggplot2)

```r
ggplot(data, aes(x = bill_length_mm,
                 y = bill_depth_mm,
                 color = species,
                 shape = species)) +
  geom_point(size = 3, alpha = 0.7) +
  geom_smooth(method = "lm",
              se = FALSE,
              linewidth = 0.7,
              alpha = 0.5) +
  scale_color_brewer(palette = "Set1") +
  labs(
    title = "Bill Depth vs Length (R)",
    x = "Bill Length (mm)",
    y = "Bill Depth (mm)",
    color = "Species",
    shape = "Species"
  ) +
  theme_minimal() +
  theme(
    plot.title = element_text(
      hjust = 0.5, size = 14, face = "bold"
    ),
    legend.position = "bottom",
    panel.grid.minor = element_blank(),
    axis.text = element_text(size = 10),
    axis.title = element_text(size = 12)
  )
```

The R version uses ggplot2 with a colorblind-friendly palette. The `geom_smooth()` layer adds per-species trend lines that reveal Simpson's paradox: the overall negative correlation between bill length and depth reverses when you condition on species.

7.0.3 Visualization in Python (Seaborn)

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_theme(style="whitegrid")
sns.set_palette("Set1")

penguins = pd.read_csv("penguins.csv").dropna()

plt.figure(figsize=(10, 6))

sns.scatterplot(
    data=penguins,
    x='bill_length_mm',
    y='bill_depth_mm',
    hue='species',
    style='species',
    s=100,
    alpha=0.7
)

sns.regplot(
    data=penguins,
    x='bill_length_mm',
    y='bill_depth_mm',
    scatter=False,
    color='gray',
    line_kws={'alpha': 0.5}
)

plt.title(
    'Bill Depth vs Length (Python)',
    pad=20, size=14, weight='bold'
)
plt.xlabel('Bill Length (mm)', size=12)
plt.ylabel('Bill Depth (mm)', size=12)

plt.legend(
    title='Species',
    bbox_to_anchor=(0.5, -0.15),
    loc='upper center',
    ncol=3
)
```

```
plt.tight_layout()
plt.show()
```

The Python version uses Seaborn, which provides a statistical plotting interface on top of Matplotlib. The `regplot()` adds an overall trend line in gray, making Simpson's paradox visible: the gray line slopes downward while species-specific patterns slope upward.

7.0.4 Visualization in Julia (UnicodePlots)

```
using UnicodePlots
using DataFrames
using CSV
using Statistics

penguins = CSV.read("penguins.csv", DataFrame)
dropmissing!(penguins)

plt = scatterplot(
    penguins.bill_length_mm,
    penguins.bill_depth_mm,
    name = string.(penguins.species),
    title = "Bill Depth vs Length (Julia)",
    xlabel = "Bill Length (mm)",
    ylabel = "Bill Depth (mm)",
    canvas = DotCanvas
)

plt
```

Julia's UnicodePlots creates text-based scatter plots that render reliably in any terminal or HTML environment. The trade-off is fewer aesthetic options compared to ggplot2 or Seaborn, but the performance advantage for large datasets can be substantial.

7.0.5 Visualization in Observable JS

```
import { Plot } from "@observablehq/plot"

penguins = FileAttachment("penguins.csv").csv()

Plot.plot({
  color: {scheme: "category10"},
  marks: [
    Plot.dot(penguins, {
      x: "bill_length_mm",
      y: "bill_depth_mm",
```

```
      stroke: "species",
      fill: "species"
    })
  ],
  x: {label: "Bill Length (mm)"},
  y: {label: "Bill Depth (mm)"},
  title: "Bill Depth vs Length (Observable JS)"
})
```

Observable JS produces interactive plots that respond to mouse hover and pan events natively in the browser. No server is required once the HTML is rendered.

7.0.6 Language Comparison Summary

Each language brings distinct strengths to the table:

- **R (ggplot2)**: Publication-quality static plots with deep customization and a grammar of graphics approach.
- **Python (Seaborn)**: Clean integration with statistical functions and strong support for categorical variables.
- **Julia (UnicodePlots)**: Fast performance and text-based output that works anywhere.
- **Observable JS**: Interactive web-native visualization with no server requirements.

8 Checking Our Work

After verifying the minimal example, check that all languages can access the same files and share data. Working directory consistency is the most common source of subtle bugs.

8.1 Working Directory Alignment

All languages need to access the same files. Check working directories in each:

```r
getwd()
```

```python
import os
os.getcwd()
```

```
pwd()
```

If any of these return different directories, you will get "file not found" errors when one language tries to read data written by another.

8.2 Memory Monitoring

Running four language runtimes simultaneously is memory-intensive. Monitor usage if you experience slowdowns:

```
gc()

import psutil
psutil.Process().memory_info().rss / 1024 / 1024
```

8.3 Things to Watch Out For

1. **Python path confusion.** macOS ships with a system Python that is not the one you want. Always verify that reticulate points to your Anaconda or Miniconda installation, not `/usr/bin/python3`.
2. **Julia first-run compilation.** The first time you call `julia_setup()`, Julia compiles its package cache. This can take several minutes and may look like the session has frozen. Be patient.
3. **Observable JS is HTML-only.** Observable chunks do not execute when rendering to PDF or Word. If you need all four languages in PDF output, you will need a different approach for the JavaScript visualizations.
4. **Working directory drift.** Each language engine may start in a slightly different working directory. Use absolute paths or verify `getwd()` / `os.getcwd()` / `pwd()` in each language before reading shared files.
5. **Memory pressure.** Running R, Python, and Julia simultaneously in a single Quarto render can consume 4+ GB of RAM. Close other applications and monitor Activity Monitor if you experience crashes.

9 Daily Workflow

Once all four languages are configured, these commands become routine:

Command	Action
`quarto render doc.qmd --to html`	Render multi-language document
`quarto preview`	Live preview with auto-reload
`Rscript -e "reticulate::py_config()"`	Verify Python binding
`Rscript -e "JuliaCall::julia_setup()"`	Verify Julia binding
`conda activate quarto-env`	Activate the Python environment

10 Uninstall / Rollback

To remove individual language integrations without affecting the others:

```
# Remove Julia integration
Rscript -e "remove.packages('JuliaCall')"
brew uninstall julia        # macOS

# Remove Python integration
Rscript -e "remove.packages('reticulate')"
conda remove --name quarto-env --all

# Remove Quarto itself
brew uninstall quarto        # macOS
sudo apt remove quarto       # Ubuntu / Debian
```

R and RStudio remain functional without any of the above.

{.img-fluid}

The best technical configurations, like the best libraries, are built on solid foundations.

11 What Did We Learn?

11.1 Lessons Learnt

Conceptual Understanding:

- Multi-language Quarto documents are not just "running different languages"; they are an orchestration problem where R acts as the conductor.
- Data sharing between languages requires a common serialization format (CSV works reliably; RDS and pickle do not cross language boundaries).
- Each plotting library embodies a different philosophy: grammar of graphics (R), statistical defaults (Python), performance (Julia), interactivity (Observable).
- Simpson's paradox appeared naturally in the Palmer Penguins data, demonstrating why multi-view analysis across tools is valuable.

Technical Skills:

- Configuring reticulate to use the correct Python installation requires explicit `use_python()` calls on most macOS systems.
- JuliaCall's `julia_setup()` performs first-run compilation that can take several minutes; this is normal, not an error.
- Observable JS chunks require no installation but only work in HTML output format.
- The `{verbatim}` chunk type in Quarto is useful for showing YAML and code examples without execution.

Gotchas and Pitfalls:

- The macOS system Python and Anaconda Python will conflict if reticulate is not configured explicitly.
- Julia package installation inside a Quarto render can time out; install packages separately beforehand.
- Observable Plot's API changes between versions; pin the version in your YAML dependencies.
- Memory pressure from running four runtimes simultaneously can cause silent failures on machines with 8 GB RAM.

11.2 Limitations

- This guide was tested on macOS only. Linux and Windows setups involve different PATH configurations and installer behaviours.
- Julia integration through JuliaCall is less mature than Python integration through reticulate. Some Julia features may not work inside Quarto chunks.
- Observable JS chunks cannot be rendered to PDF or Word output. Multi-format publishing requires fallback strategies for JavaScript content.
- The guide assumes Anaconda for Python management. Users of pyenv, virtualenv, or system Python will need to adapt the reticulate configuration steps.
- Memory requirements (8+ GB) may exclude older hardware from running all four languages simultaneously.

- Package version compatibility across four language ecosystems is fragile. A major update to any one language can break the integration chain.

11.3 Opportunities for Improvement

1. **Add Docker containerization** to freeze all four language environments and eliminate host configuration variability.
2. **Create a setup validation script** that checks all four language integrations and reports status in a single command.
3. **Explore Quarto's native Julia engine** (currently experimental) as an alternative to JuliaCall for more direct integration.
4. **Add Plotly or Altair** as alternatives to Seaborn for Python, enabling interactive plots that match Observable JS's capabilities.
5. **Build a project template** with pre-configured `renv.lock`, `requirements.txt`, and Julia `Project.toml` files for one-command environment setup.
6. **Test on Linux and Windows** to expand the guide beyond macOS-specific instructions.

12 Wrapping Up

Setting up a multi-language Quarto environment is less about any single installation and more about making four separate ecosystems aware of each other. R sits at the center, with reticulate and JuliaCall acting as bridges to Python and Julia, while Observable JS runs independently in the browser.

The process demonstrates that environment configuration is its own skill, distinct from programming in any one language. The time spent getting PATH variables, package versions, and working directories aligned pays off every time a document is rendered that would otherwise require four separate scripts and manual assembly.

For those starting from scratch, the strongest recommendation is to verify each language independently before combining them. Debugging a four-language pipeline is exponentially harder than debugging a single-language problem.

Main takeaways:

- R orchestrates the other languages through reticulate (Python) and JuliaCall (Julia); Observable JS is browser-native.
- Verify each language integration independently before combining them in a single document.
- Use CSV as the common data exchange format between languages for maximum compatibility.
- Budget extra time for first-run compilations (Julia) and PATH configuration (Python) on macOS.

13 See Also

Related posts:

- Quarto Documentation: the official guide covers multi-language support in depth.

Key resources:

- Reticulate Documentation: R-Python bridge package reference.
- JuliaCall Documentation: R-Julia bridge package reference.
- Observable Plot API Reference: Observable JS plotting library.
- R Markdown Cookbook: comprehensive reference for literate programming in R.

14 Reproducibility

14.1 Environment Requirements

- **R**: Version 4.4 or later
- **Python**: Anaconda distribution recommended
- **Julia**: Version 1.9 or later
- **Quarto**: Version 1.4 or later
- **macOS**: Monterey or later

14.2 Version Checks

```
R.version.string

library(reticulate)
py_config()

library(JuliaCall)
julia_eval("VERSION")

quarto::quarto_version()
```

14.3 Session Information

```
sessionInfo()
```

15 Let's Connect

Have questions, suggestions, or spot an error? Let me know.

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** Contact form

I would enjoy hearing from you if:

- You spot an error or a better approach to any of the code in this post.
- You have suggestions for topics you would like to see covered.
- You want to discuss R programming, data science, or reproducible research.
- You have questions about anything in this tutorial.
- You just want to say hello and connect.