Journal of Data Science, Statistics, and Visualisation

March 2023, Volume III, Issue I. *doi: 10.52933/jdssv.v3i1.53*

A Review of Containerization for Interactive and Reproducible Analysis

Gregory J. Hunt
Department of Mathematics
William & Mary

Johann A. Gagnon-Bartsch
Department of Statistics
University of Michigan

Abstract

In recent decades, the analysis of data has become increasingly computational. Correspondingly, this has changed how scientific and statistical work is shared. For example, it is now commonplace for underlying analysis code and data to be proffered alongside journal publications and conference talks. Unfortunately, sharing code faces several challenges. First, it is often difficult to take code from one computer and run it on another. Code configuration, version, and dependency issues often make this challenging. Secondly, even if the code runs, it is often hard to understand or interact with the analysis. This makes it difficult to assess the code and its findings, for example, in a peer review process. In this review, we describe the combination of two computing technologies that help make analyses shareable, interactive, and completely reproducible. These technologies are: (1) analysis containerization, which leverages virtualization to fully encapsulate analysis, data, code and dependencies into an interactive and shareable format, and (2) notebooks, a literate programming format for interacting with analyses. The fusion of these two technologies offers significant advantages over using either individually. This review surveys how the combination enhances the accessibility and reproducibility of code, analyses, and ideas.

Keywords: containerization, notebooks, reproducibility, Docker.

1. Introduction

Before the widespread adoption of peer-reviewed scientific journals, it was not uncommon for scientists to keep their findings secret. Famously, Leonardo Da Vinci wrote in mirrored handwriting to obfuscate his notebooks and Isaac Newton kept hidden his development of calculus for nearly forty years (National Academy of Sciences et al. 2009). Modern science, however, advances through a rich process of open and timely sharing. Today, there are a plethora of ways to share results such as talks at conferences, proceedings, seminars, posters, peer-reviewed literature and pre-print repositories. Open sharing not only allows results to be disseminated and built upon, but also allows scrutiny and verification of the research and is fundamental to the scientific process itself. However, as scientific analysis has progressed, so too has the notion of sharing. In particular, the last several decades have seen the analysis of scientific data become heavily computational. This is especially true of statistical work, where coding has become deeply intertwined with statistical analysis. Correspondingly, the notion of what it means to share research results has also expanded (Ellis and Leek 2018). The modern notion of sharing research encompasses not only sharing prose and proofs, but also sharing code and data.

It is now commonplace for data and the accompanying analysis code to be shared through online repositories. Indeed, many peer-reviewed journals either require or strongly encourage it. For example, most of the journals sponsored by the International Statistical Association and American Statistical Association require data and code be posted along with analysis (Journal of the American Statistical Association 2022). Similarly, many prominent scientific journals have data sharing requirements (Nature 2022; Science 2022). There are many tools that help facilitate this sharing. For general purpose code, a popular sharing platform is GitHub (Github, Inc. 2021). Language specific repositories for software packages also exist, e.g., CRAN for R packages (The R Project for Statistical Computing 2021) or PyPI for Python (Python Software Foundation 2021). However, CRAN and PyPI are intended for software, not to host full analyses for the purposes of reproducibility. Moderately sized datasets may be hosted on GitHub or Kaggle (Kaggle Inc. 2019). Larger datasets may be hosted on scientific data repositories like Figshare (Digital Science 2022) or Zenodo (CERN Data Centre & Invenio 2022). Zenodo is operated by CERN and allows hosting up to 50GB of data while Figshare is operated by Digital Science and has a limit of 20GB. Both platforms assign a DOI so that data may be permanently referenced. This open sharing of analysis code is a growing trend in statistics. Nonetheless, it faces several practical challenges. Among these, two important issues are (1) actually running the shared code, and (2) understanding and interacting with the code.

The first challenge is that code that runs on one computer may not always run on another. For example, the package may not be available for the current version of the language or dependencies of the package may fail to install. Modern analysis often relies on a large and complex collection of interdependent software packages and thus there are many places for such version or dependency issues to arise. Similarly, directory structures across machines may not be identical and, for example, data, code, or other files may not reside where the analysis is expecting. Fixing such problems often entails a significant investment of time and energy. For example, troubleshooting failed installations of dependencies can often lead down a chain of fixing cryptic installation errors which is difficult even for an experienced user.

In addition to the challenges of taking analysis from one computer and running it on another, a second major challenge is difficulty understanding or interacting with code. While it may be impractical or unnecessary to insist on understanding code on a line-by-line basis, a lot can be learned about an analysis by making small modifications to code. For example, one can explore different parameter settings or function arguments and see how output changes. Here, simply sharing raw code is often inadequate. Unless code is particularly well-written, documented, and organized, it can be difficult to understand and explore. Consequently, it is often difficult for third-parties to identify reasonable entry-points into code to modify or scrutinize the analysis.

These issues limit gaining a deeper understanding of shared of statistical and scientific results. In this work, we will review how two computational toolsets can be combined to help address these problems. These toolsets are: (1) analysis containerization, and (2) interactive notebooks. Containerization is a virtualization technology that allows encapsulation of an entire computing environment including data, code, dependencies, and programs into a reproducible, shareable, and self-contained format (Nüst et al. 2020). When a third party takes the container and runs it on their own computer, it will be as if they are instead working in the computational environment where the analysis was originally done. All of the programs, files, code, data, and configurations will be exactly reproduced as they were in that original environment. Containerization is a flexible approach that allows one to encapsulate any format or organization of analysis according to their preferences and assessment of the best way to organize and share the analysis. While there are many good ways of writing shareable analyses, in this work we advocate for containerizing interactive notebooks. Notebooks are an increasingly popular approach to analysis that allow natural interweaving of commentary, code, and output. Containerizing notebooks makes for some of the most clear, concise, and intuitive ways of documenting and interacting with analysis.

In this work, we will review how the merging of containerization and notebook software can be used to create interactive and reproducible analyses. In addition to an overview, we will also make concrete recommendations for what we believe to be the most straight-forward tools and workflows to enhance reproducibility through containerized notebooks. The remainder of this paper is organized as follows: Section 2 reviews barriers to computational reproducibility in statistics, how containerization helps, and the landscape of available tools. Section 3 reviews interactive notebooks and surveys a selection of software options for writing notebooks that can be easily containerized. Finally, Section 4 concludes with a discussion of additional benefits of containerization and notebooks.

2. Containerizing Analyses

The basic computational reproducibility problem is that often code encounters errors when moved from one computer to another. This was emphasized by the American Statistical Association's 2017 recommendations on reproducible research, which noted that "[reproducible code] may initially sound like a trivial task but experience has shown that it's not always easy to achieve this seemingly minimal standard."(Broman et al. 2017) One major source of trouble is ensuring correct code dependencies. The most familiar example of this is installing add-on packages for a language like **ggplot2** for R or **numpy** for Python. While add-on dependencies are easy to install in some cases, this can quickly become complicated, for example, if the original analysis used a now out-of-date version of a package. Furthermore, add-on packages often have their own dependencies. Thus installing a single package may actually require a large network of interrelated packages to be configured. Figure 1 displays the package dependency network for the popular R package **lme4** (Bates et al. 2015) which enables fitting linear mixed-effect models. This package has 35 add-on package dependencies and a system-level library dependency.

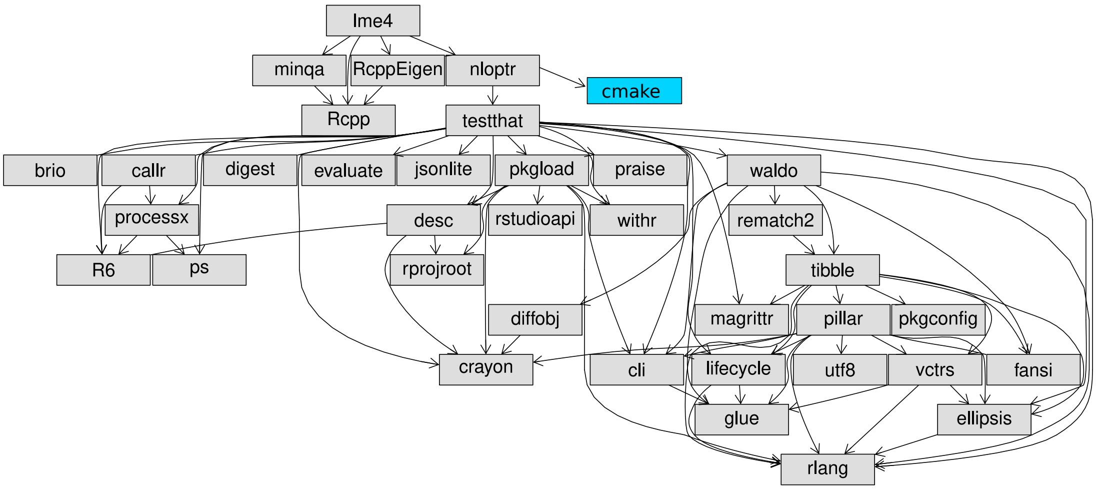


Figure 1: Dependency graph for the R package **lme4**. Grey boxes are R add-on packages. Arrows indicate dependency. The blue box indicates the system-level dependency of the package for Linux OS Ubuntu ver. 20.04.

There has been significant effort in the R community to address some of these add-on dependency issues. The CRAN task view on reproducible research (Blischak and Hill 2021) lists several packages for this purpose like **checkpoint** (Ooi et al. 2021), **groundhog** (Simonsohn and Gruson 2021), and **renv** (Ushey 2021). These tools all enhance reproducibility by maintaining a local archive of packages as used at the time of analysis. This archive can subsequently be distributed with analysis code so that the correct add-on versions are available to third-parties. While R has such archival packages available, other languages have comparatively less support. Furthermore, analyses often have dependencies beyond simple add-on packages that cannot be archived in this way. For example, code typically depends on programming language and operating system versions, and system-level library code (as in Figure 1).

Recently, some have sought to solve these broader dependency issues using virtualization, a well-studied software engineering solution to dependency problems (Nüst et al. 2020; Olaya et al. 2020). Virtualization encapsulates code and all of its dependencies into a virtual computing environment that can be easily disseminated. One can think of virtualization as making a copy of the computer where the code was originally written. This virtual copy can be taken to another computer and run with little to no setup or configuration. While virtualization has been around for decades, containerization is the latest incarnation of the technology and comes with several key advantages over its predecessors. Previous technology virtualized the entire computer from hardware on up. This meant that virtualization was resource intensive and slow to use. Conversely, containerization is incredibly light-weight. Containers only virtualize the high-level components of the operating system (e.g., code, configuration files, software and data) and seamlessly re-use the stable low-level processing components of the host operating system (Turnbull 2014). Indeed, starting up a container doesn't actually start up a second instance of an operating system; it largely just changes all references for resources, system libraries, files, and data, to refer to a particular isolated section of the computer. The light-weight nature of such containers means that the resource footprint is small making them quick to upload, download, and share. Furthermore, since starting a container largely just changes the references to resources in the environment, containers are user-friendly, start up nearly instantaneously, and run code at speeds nearly identical to the host computer (Felter et al. 2015).

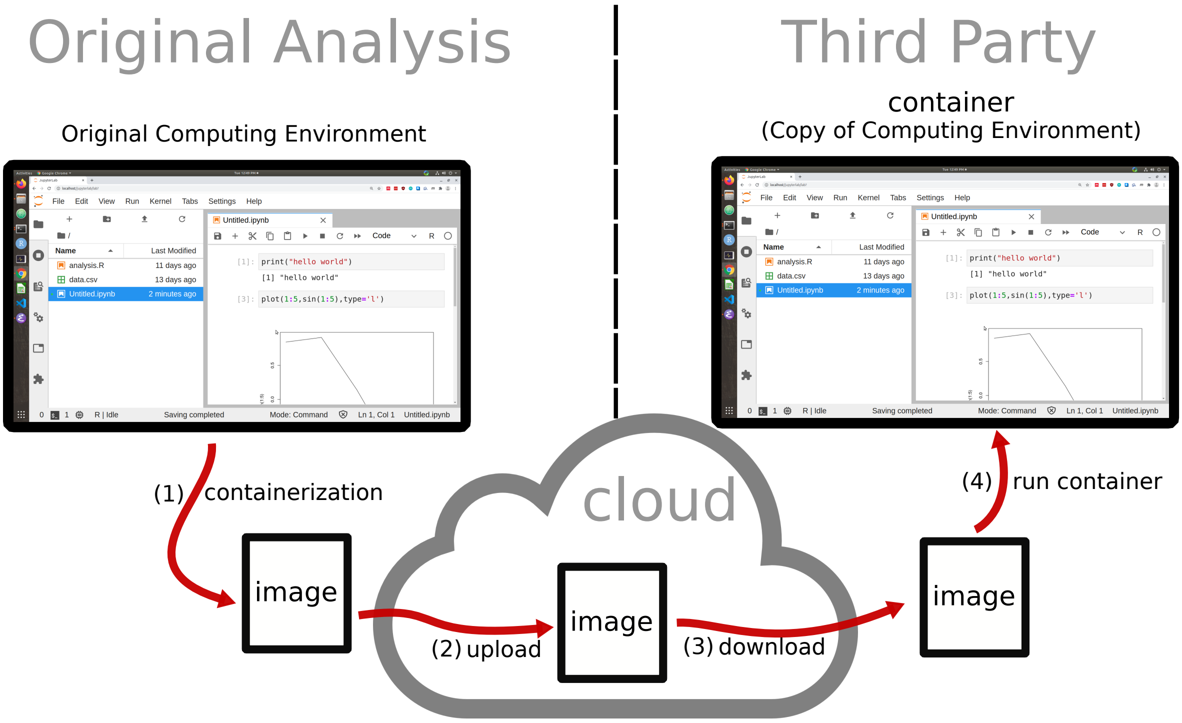
2.1. Containerization in Practice

Containerization has been an increasingly adopted tool for reproducibility widely across the scientific community including areas such as geography, psychology, environmental science, metagenomics and many others (Knoth and Nüst 2017; Wiebels and Moreau 2021; Essawy et al. 2020; Visconti et al. 2018; Nüst et al. 2020; Olaya et al. 2020). To set the stage for a review of containerization technology we will first illustrate how containerization is used in practice. We will present an archetypal example of containerizing and sharing an analysis from three different perspectives: (1) the high-level view of sharing containerized analyses, (2) the end-user experience of interacting with a third-party containerized analysis, and (3) the first-party task of containerizing an analysis for dissemination. These will correspond to Figures 2, 3, and 4, respectively. A detailed explanation of how to use containerization software along with recommended resources may be found in the supplementary material.

Figure 2 displays a high-level overview of how statistical analyses are containerized and shared. First, the entire computing environment in which the analysis was originally run is encapsulated into a single file. This file, called an image, is essentially a copy of the system on which the analysis was conducted. The image file may be shared, for example, by uploading it to the cloud. From there, the image may be downloaded by a third party and with just a few keystrokes, the third party is placed into a duplicate of the original computing environment (called a "container"[1]). All of the data, code, dependencies, configurations and software are precisely set up as in the original environment, and thus set up to reproduce the analysis exactly. The goal of containerization is to ensure that if the code worked when containerized, it will work when the image is run by a third party. This figure emphasizes that containerizing and sharing analyses is a simple process akin to uploading code to GitHub. However, unlike uploading code to GitHub, containerizing analyses ensures exact computational reproducibility and enables natural interaction with the shared analyses.

Figure 3 shows in detail what using a shared containerized analysis looks like from the viewpoint of an end-user. First, the container is downloaded and started with a single

[1] An "image" refers to the actual file that may be uploaded, downloaded or shared, while a "container" refers to an ephemeral instance running on the computer.

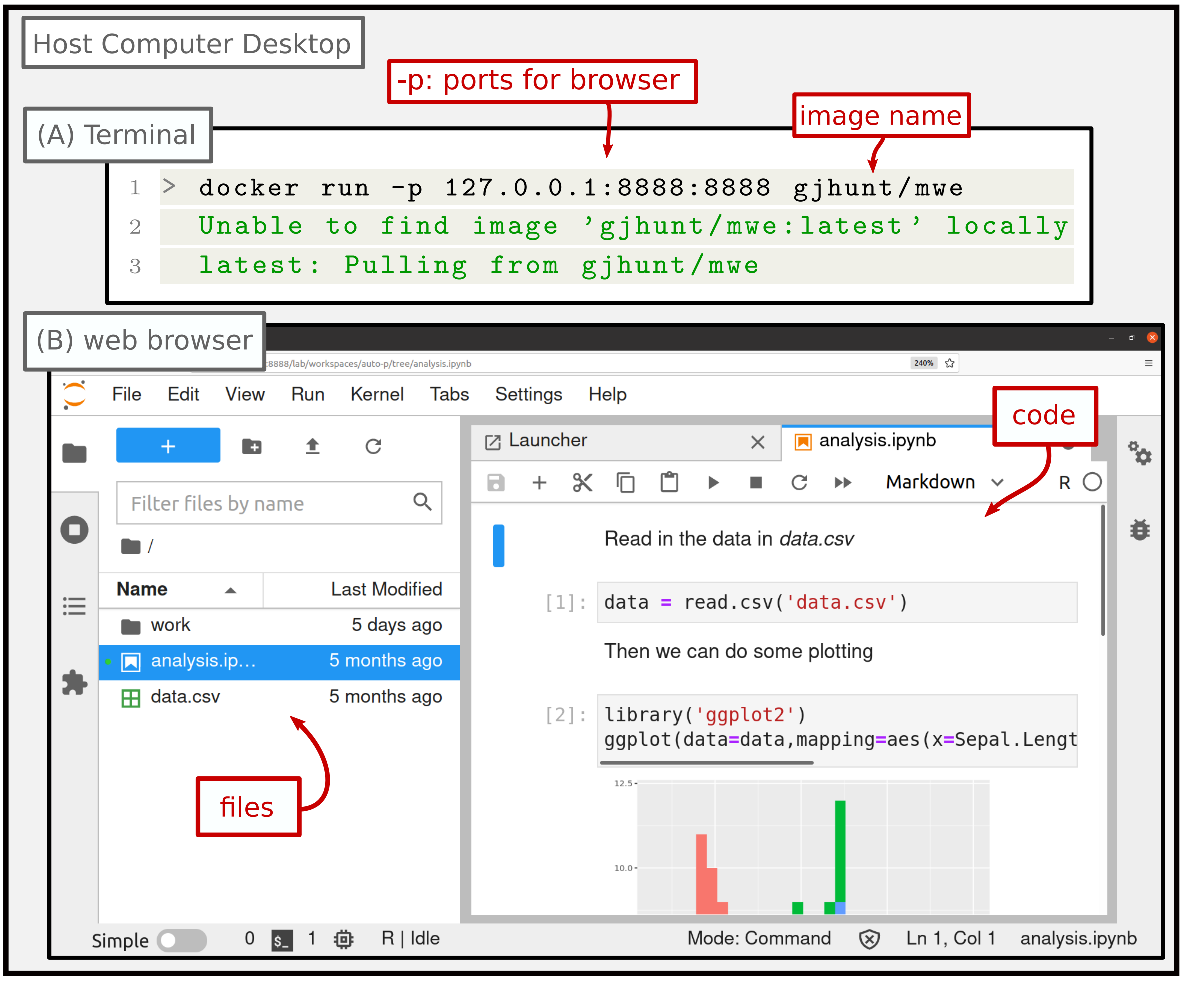


```
[1]: print("hello world")
     [1] "hello world"
[3]: plot(1:5,sin(1:5),type='l')
```

Figure 2: Typical sharing of containerized analysis. (1) The computing environment is containerized, creating a self-contained image file. (2) This image file may be uploaded to the cloud and then (3) downloaded by a third party. (4) From there, the third party may use the image to re-create the original computing environment.

command. The default interface to a container is through the command line. However, as shown in Figure 3, when combined with notebook software the analysis is accessible via an interactive graphical interface through the computer's web browser. Alone, the containerization ensures exact reproducibility but it is not user-friendly. Conversely, notebook software alone provides a user-friendly environment but does not guarantee exact reproducibility. The combination achieved by containerizing notebooks gets the best of both worlds. This will be explored in more detail in Section 3.

We can see from Figure 3 that the container's environment contains all of our files necessary for analysis including data and code scripts. However, in addition to merely allowing inspection of the data or scripts, the container also comes with an installation of R so that the user can actually run the code and analysis through the interactive notebook interface. It is important to keep in mind that while the end-user accesses the container and its contents through the web browser on the host computer, the data, code, software installations and back-end to the interface all actually reside in the container. The web browser merely provides a window into the running container through which one may use the tools installed in the container and interact with the code and data it contains. Indeed, none of these programs or files need to be installed on the host computer in order to use the web browser to interactively access the versions running in the container. This is the power of sharing containerizing analyses – it allows



```
> docker run -p 127.0.0.1:8888:8888 gjhunt/mwe
  Unable to find image 'gjhunt/mwe:latest' locally
  latest: Pulling from gjhunt/mwe
```

Name	Last Modified
work	5 days ago
analysis.ip…	5 months ago
data.csv	5 months ago

Read in the data in *data.csv*

```
[1]: data = read.csv('data.csv')
```

Then we can do some plotting

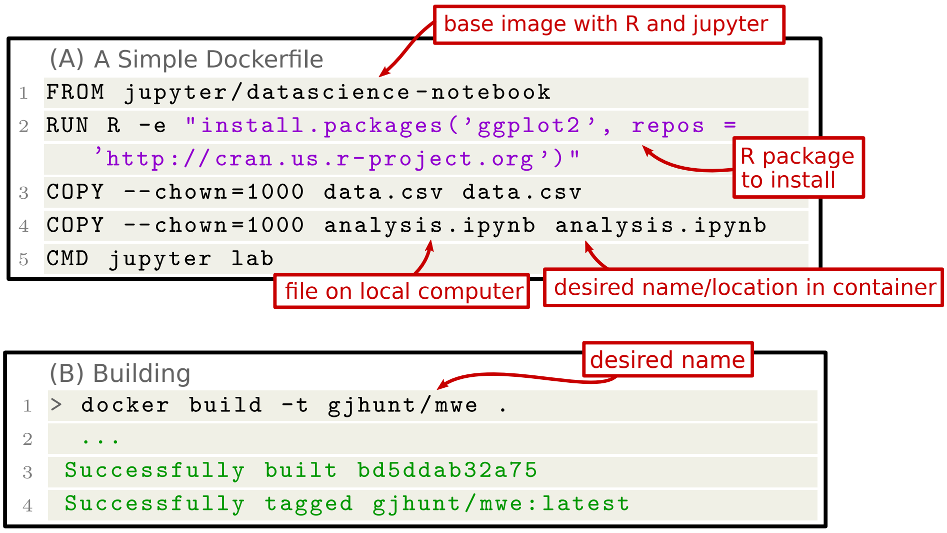
```
[2]: library('ggplot2')
     ggplot(data=data,mapping=aes(x=Sepal.Lengt
```

Figure 3: Example of interacting with a containerized analysis. (A) Here we use the containerization software Docker to launch the container. The command `docker run` starts the container. The flag `-p` allows us to specify the port forwarding to enable interaction through the web browser. (B) The container may now be interacted with through the web browser on the host computer via a graphical interface running from the container. Here, the interface is the Jupyter lab integrated development environment (IDE). The container has the necessary data and code files and an installation of R to run the analysis through this web interface. While the end-user may interact naturally with the analysis through a web-browser on the host computer, all of the code, files, and software reside in the container's pre-configured environment.

users to bring to bear the full power and convenience of popular graphical interfaces to fully encapsulated analysis environments with a single command.

To set up an image, a configuration file must be written giving instructions of which files and programs to be copied and installed. Images need not be built from the ground up but instead, one can simply add on to existing pre-configured images to create new ones. For example, the container repository Dockerhub (Docker Inc. 2021a) contains more than 100,000 images freely available and usable by all major containerization software. Such repositories make containerizing analyses simple as one can choose a nearly-complete image, with desired software like Jupyter lab and R already installed, and simply add a small amount of project-specific code, data, and documentation.

Figure 4 (A) displays the configuration file used to build the image from Figure 3. In five lines the configuration specifies a base image with R and Jupyter already installed, installs a desired R add-on package, copies over data and analysis code, and starts the Jupyter lab interface. Such a simple configuration file is quite typical for containerizing statistical analyses. Most of the heavy lifting is done by the base image which sets up a nearly complete environment. On top of this base image one needs only to install the necessary software packages or language add-ons and copy over the data and code. Once the configuration file has been written, the image needs to be built, after which, it may be run or shared. Building the image is illustrated in Figure 4 (B).



(A) A Simple Dockerfile

```
1 FROM jupyter/datascience-notebook
2 RUN R -e "install.packages('ggplot2', repos =
     'http://cran.us.r-project.org')"
3 COPY --chown=1000 data.csv data.csv
4 COPY --chown=1000 analysis.ipynb analysis.ipynb
5 CMD jupyter lab
```

base image with R and jupyter

R package to install

file on local computer

desired name/location in container

(B) Building

```
1 > docker build -t gjhunt/mwe .
2   ...
3  Successfully built bd5ddab32a75
4  Successfully tagged gjhunt/mwe:latest
```

desired name

Figure 4: (A) Example configuration file for building an image using Docker. **(Line 1)** `FROM` specifies the base image named `jupyter/datascience-notebook` to get a container with R and Jupyter. **(Line 2)** `RUN` executes code which calls R and installs **ggplot2**. **(Line 3-4)** Copies the data and code. First argument to `COPY` is location on host, second argument is desired location in container, the flag `-chown` sets the ownership of the file to the container's user. **(Line 6)** `CMD` sets the command executed when the container starts, here starting Jupyter lab. (B) Building image from Dockerfile. Flag `-t` specifies the image name as `gjhunt/mwe`. "." specifies necessary files to copy are in the current directory.

2.2. The Containerization Landscape

While virtualization can trace its roots all the way back to early mainframe computers, modern lightweight containerization was largely popularized with the software Docker starting in 2013 (Graziano 2011; Docker Inc. 2021b). While other tools have been developed since then, the present space of user-friendly containerization software for statisticians and scientists has two major players: (1) Docker (Docker Inc. 2021b), and (2) Singularity (Sylabs 2021). In the remainder of this section we will briefly compare these options, summarizing findings in Table 1.

Portability is paramount to reproducibility. Docker and Singularity are both free and open source and built off a Linux base. Consequently, they both work on Linux.

However, Singularity does not have native support on Windows or MacOS while Docker has both support and a graphical interface for these systems. Nonetheless, Singularity is partially inter-operable with Docker and can run Docker images or use them as a base image. Conversely, Docker can only work with Docker images.

A significant distinction is that Docker requires administrator privileges to run, while Singularity does not. This makes Singularity capable of deploying software on high-performance computing clusters where users do not have these rights. If one wishes to run Docker on a cluster they may consider using Podman instead. Podman (Red Hat, Inc. 2021) is a re-implementation of Docker that doesn't require administrative privileges. Podman is available on Linux or available on Windows using the Windows Subsystem for Linux.

In addition to required privileges, there are differences in system isolation. Singularity does not by default isolate the host computer's file-system or network interface from the container while Docker does. This makes Singularity's default behavior less secure for running unverified third-party analyses but more amenable for deploying non-interactive code to clusters. Singularity's default configuration also locks containerized analyses as read-only unlike Docker. This makes it relatively difficult to explore and edit third-party analysis code with Singularity.

All of the containerization software we recommend in this manuscript is free and open source software (FOSS). As containerization is fundamentally a refinement of older existing FOSS virtualization technology (itself built upon the FOSS Linux kernel) the core software defining Docker, Singularity, and Podman are publicly available under copyleft/permissive licenses. This is important as we want to make sure that the software will remain freely available in the future.

While containerization software like Docker is FOSS this may not hold for repositories like Dockerhub or other peripheral services. Dockerhub is a service provided by Docker Inc. that allows sharing of images, but there is no guarantee that this serivce will indefinitely provide free, long-term archival of data-heavy images. This leaves open the question of where to store images for the purposes of reproducibility. We suggest Zenodo, a general repository for scientific data operated by CERN (CERN Data Centre & Invenio 2022). Zenodo allows hosting of up to 50GB of data and creates a permanent DOI that can be referenced. As images are simply files, users may upload images created using Docker, Podman or Singularity to Zenodo and share them with the community. Other researchers will simply need to locate the files using the DOI and download/run the images.

Similarly note that Docker maintains a non-FOSS tool called Docker desktop. This software is primarily useful for managing multiple containers. However Docker desktop is not necessary for using the core containerization software.

Table 1 summarizes the discussion of this section. For containerizing shareable and reproducible analyses we recommend Podman or Docker as they are widely used containerization software with cross-platform support, a user-friendly interface, and a huge ecosystem of base images off of which one may build. Nonetheless, for deploying containerized analyses to high-performance computing environments Singularity has substantial strengths.

While all of the containerization tools we discuss in this section can help provide a

Table 1: Comparison of Docker, Singularity, and Podman for containerization of reproducible analyses.

	Docker	**Singularity**	**Podman**
O/S Support	Linux, Mac, Windows	Linux	Linux
Image Type Support	Docker	Docker, Singularity	Docker
Admin. Privileges	Required	Not Required	Not Required
Host/Container Isolation	Yes	No	Yes
Container Mutability	Read/Write	Read Only	Read/Write

powerful approach for ensuring exact reproducibilitiy of results, this would ideally also be done in a user-friendly manner. Otherwise, the added effort of interacting with analyses through a container has the potential to hinder the accessibility of the analysis and code. In Section 3 we discuss notebooks and how they can be used to help provide an accessible and intuitive graphical interface to writing and interacting with containerized analyses.

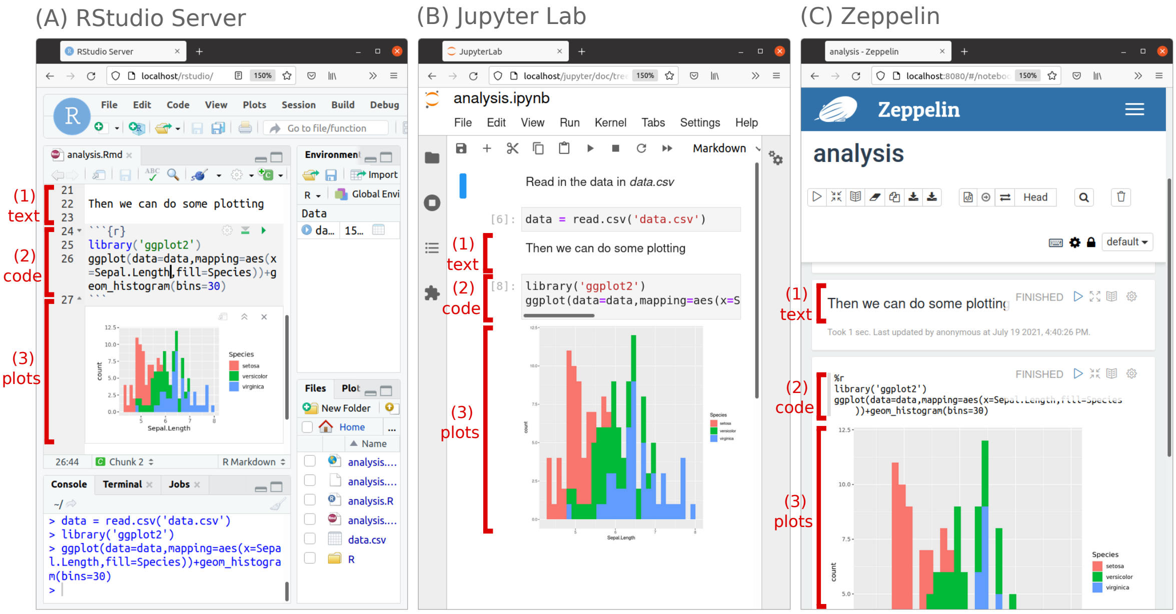
3. Notebooks and Interactivity

Notebooks are a document format that allow interweaving of rich commentary, code, and output all together. As such, notebooks are an increasingly popular way to structure and share analyses. Notebooks can also be especially useful when containerizing analyses as they are a great way to make containerized analyses interactive and user-friendly, and thus ultimately more shareable. In the remainder of this section we will describe notebooks, highlight some of their advantages, and review popular options.

Figure 5 displays three examples of popular notebook formats which will be reviewed in Section 3.1. While there are several variants of notebooks, they all structure analysis as a sequence of "chunks" that can be edited and evaluated one at a time. Each chunk can either be text or code. Text chunks typically allow writing in `markdown` (Matt Cone 2021) which allows website-like formatting with headers, lists, embedded click-able links, images/figures, tables, and typeset LaTeX mathematics. Running code chunks displays the output inline.

This chunked structure can help promote good programming practices. Good coding practices are an important component of producing clear and reproducible analyses. Two important practices are (1) logically organizing code into blocks, and (2) interweaving meaningful comments into the code. In the software development community there is a long history of discussion of best coding practices and development of tools aimed at promoting them. For example, paradigms like literate programming (Knuth 1984) and documentation generation tools like Doxygen (Doxygen 2022) have long been popular.

Notebooks build upon this history, promoting chunked code organization and rich commentary, but go a step further and embed analysis output along the code and commentary. This allows one to comment not only on the code, but also on the output. Consequently, notebooks not only encourage good coding practices, but also facilitate a rich discussion of the code, its relation to the output, and the bearing of this output



(A) RStudio Server

~~~
Then we can do some plotting

```{r}
library('ggplot2')
ggplot(data=data,mapping=aes(x=Sepal.Length,fill=Species))+geom_histogram(bins=30)
```
~~~

```
> data = read.csv('data.csv')
> library('ggplot2')
> ggplot(data=data,mapping=aes(x=Sepal.Length,fill=Species))+geom_histogram(bins=30)
>
```

(B) Jupyter Lab

Read in the data in *data.csv*

```
[6]: data = read.csv('data.csv')
```

Then we can do some plotting

```
[8]: library('ggplot2')
     ggplot(data=data,mapping=aes(x=S
```

(C) Zeppelin

Then we can do some plotting

```
%r
library('ggplot2')
ggplot(data=data,mapping=aes(x=Sepal.Length,fill=Species
    ))+geom_histogram(bins=30)
```

Figure 5: Interactive notebook environments run through the web browser using (A) RStudio Server, (B) Jupyter Lab, and (C) Zeppelin. While different notebook formats and software tools exist, all notebooks share the feature of organizing analysis as a sequence of chunks of (1) text or (2) code and its associated (3) output.

on the broader scientific questions.

Indeed, embedding output directly alongside the code allows one to document the entire analysis pipeline including expository plots such as diagnostic and exploratory plots that may inform small decisions made in the course of analysis. These types of plots are often not included in manuscripts or supplementary materials because they are difficult to motivate and connect to the analysis when divorced from the actual code. Nonetheless, documentation of these types of micro-decisions is important for properly documenting an analysis pipeline and is necessary for transparent and reproducible research (National Academies of Sciences and Medicine 2019).

A separate advantage of producing output alongside code is that notebooks immortalize output directly alongside the code that generated it. This can help ensure, for example, that figures are directly linked to their source code. This can be useful in a research context where both code and output evolve over time and it is easy to mismatch versions of results/figures to the correct versions of the underlying analysis. Notebooks provide a mechanism to help avoid such a mismatch. However, it should be noted that notebooks do not prevent running chunks in a non-sequential order nor do notebooks prevent editing code without re-running the chunk. Both of these practices have the potential to make confusing notebooks where re-running the code sequentially does not reproduce the immortalized output (or may even produce errors). Some notebook software attempts to alert users to these issues. For example, Jupyter lab (which will be discussed in the next section) highlights code in orange if it has been edited but not run. Similarly, Jupyter lab maintains a numeric label for each code chunk to indicate the order in which the chunks have been run. Nonetheless, we still recommend that before sharing notebooks they are re-run sequentially to ensure they work as intended and, in particular, containerizing the analysis provides a good opportunity to do this.

Another advantage of using notebooks for explaining and showcasing analysis is that they give users the option to run the code and explore it interactively. Since chunks can be edited and run one at a time, each chunk provides a natural entry-point into a small portion of the analysis. For example, one can pick a segment of the analysis they wish to explore, edit the chunk of code, run it, and observe the subsequent change in output. This encourages one to experiment with small changes to code, e.g., testing different tuning parameters or optional arguments to functions, and immediately observe the changes to local output. This can be used to provide a natural way to play with code in order to build up an understanding of how the code works and test the robustness of the analysis to alterations.

While notebooks can provide a nice way to interact with analyses generally, they are particularly powerful as a tool for interacting with containerized analyses. By default, a containerized analysis requires the user to interact with the code entirely through the command line. This may be a barrier to the adoption of containerization for many potential users. However, if we containerize notebook software in addition to the code, data, and other dependencies, then we can bring the full power of popular coding environments as an interactive interface to our containerized analysis. This is particularly easy if the notebook software is accessible through a web browser. In this case, we can run the notebook back-end from within the container but access the interactive computing interface from the host computer's browser. This is illustrated in both Figure 3 and Figure 5. This combination is the best of both worlds as it brings the native feel of doing analysis on one's own computer to completely self-contained and reproducible analyses.

3.1. Options for Interactive Notebooks

There are three major notebook types that are simple to containerize: (1) RStudio, (2) Jupyter, and (3) Zeppelin. Aside from these three, there is other proprietary notebook software like Wolfram Mathematica or MATLAB Live Scripts; however these are closed-source and difficult to containerize. Conversely, third-party software like Pycharm or VSCode can write and run Jupyter notebooks but are more complicated to containerize as they lack a native web interface. Consequently, this section compares RStudio, Jupyter, and Zeppelin, all three of which have an easily containerizable web interface along with official images and support on Dockerhub. A summary of this comparison is presented in Table 2 and examples of the software interfaces are illustrated in Figure 5.

All three notebook options support a large array of languages popular for data science like R, Python, Julia, Octave, and many others. RStudio boasts over 55 language interpreters, Jupyter lists over 150, and Zeppelin has support for 37 (with a focus on languages for clusters like Hive, Pig, Spark and BigQuery). Zeppelin can also create chunks that effectively run RStudio or Jupyter as a backend and thus directly borrow the features and languages they support. While Jupyter requires that all code chunks in a notebook use the same language, both RStudio and Zeppelin allow notebooks to mix and match languages across chunks. Furthermore, RStudio has extensive support for **reticulate**, which allows analysis using both R and Python at the same time in a shared computing environment.

An important distinction is the format of the notebook file and how it interacts with third-party software. Jupyter and Zeppelin serialize the notebook and save it in a single densely encoded file. Conversely RStudio saves input code and markdown in a plain-text "R Markdown" file and renders output into a separate HTML file. Each format has its advantages and drawbacks. The encoding used by Jupyter/Zeppelin allows them to save output text or plots in one file alongside input code, commentary, and rendered markdown. This is useful for showcasing results because, unlike a traditional code scripts, the notebook has output embedded and one need not re-run the code to view the results. However, the file format is densely encoded which can cause difficulties when combined with other software. In particular, one cannot easily track changes to these notebooks in a human-readable format using version control software like Git since small changes to output can prompt a cascading change to hundreds of lines of the dense encoding.

Alternatively, RStudio uses a combination of plain-text R Markdown for input and HTML for rendered output. An advantage of such an approach is that the input code and commentary are saved in a human-readable format which is more versatile for editing by general software and can be meaningfully tracked by version control schemes. The shareable HTML rendering also contains an embedded downloadable copy of the R Markdown file if one wishes to download the underpinning R Markdown code. In addition to rendering to HTML, RStudio allows rendering notebooks into a host of output types for display like PDF, Word or Powerpoint. Jupyter can also render its notebooks into to a slightly smaller selection of similar display formats however Zeppelin does not have such support. Despite these format differences, from the viewpoint of interacting and exploring analyses all three of RStudio, Jupyter and Zeppelin have broadly similar behavior, and allow users to edit and run code chunks one at time, viewing output in-line in the editor.

To allow conversion between formats, Jupyter has the Jupytext plugin which allows one to conduct analysis using Jupyter, whilst maintaining a simultaneous synchronized version in R markdown or as a simple executable script. This allows the best of both Jupyter and R Markdown notebooks, and in particular makes Jupyter compatible with version control software. Zeppelin only supports converting their notebooks into the Jupyter format while RStudio does not natively support conversion of R Markdown to other formats, nonetheless Jupytext can enable this conversion.

All three of Jupyter, RStudio and Zeppelin have the ability to embed interactive widgets into notebooks using popular interactive libraries in languages like R and Python. As embedding widgets typically takes extra configuration it makes a strong case for containerization which will ensure the back-end software is correctly set up to support such interactivity. While Jupyter has support for interactive elements in both Python and R, RStudio primarily supports these through its R **Shiny** (Chang et al. 2021) platform for building web appplications. As Zeppelin can create notebook chunks running the backend language interpreters of both RStudio (including R **Shiny**) and Jupyter it can create notebooks that naturally embed interactive R **Shiny** applications or Jupyter widgets. Zeppelin also has its own interactive visualizations backend via Apache Spark (The Apache Software Foundation 2022).

A common challenge when using notebooks is that chunks need to be run sequentially and so to explore chunks later in the analysis one needs to run earlier time-intensive code. To facilitate entering the analysis at arbitrary points it is good practice to save the output of time-intensive chunks. This allows subsequent chunks to simply load the precomputed intermediate results instead of requiring a preceeding time-intensive chunk to be run. This practice, called results caching, can be done manually by reading/writing serialized objects to/from the disk e.g., using `pickle` in Python or `read/writeRDS` in R. Containers are well-suited for this as one can distribute notebooks together with cached results. While there is some native support for caching results using notebooks, it is language and IDE specific. For example, while RStudio can natively cache and retrieve serialized R objects when writing in the R markdown format, this does not work if writing code in Python or Julia. Similarly, Jupyter has plugins to enable caching results, but primarily for writing in Python.

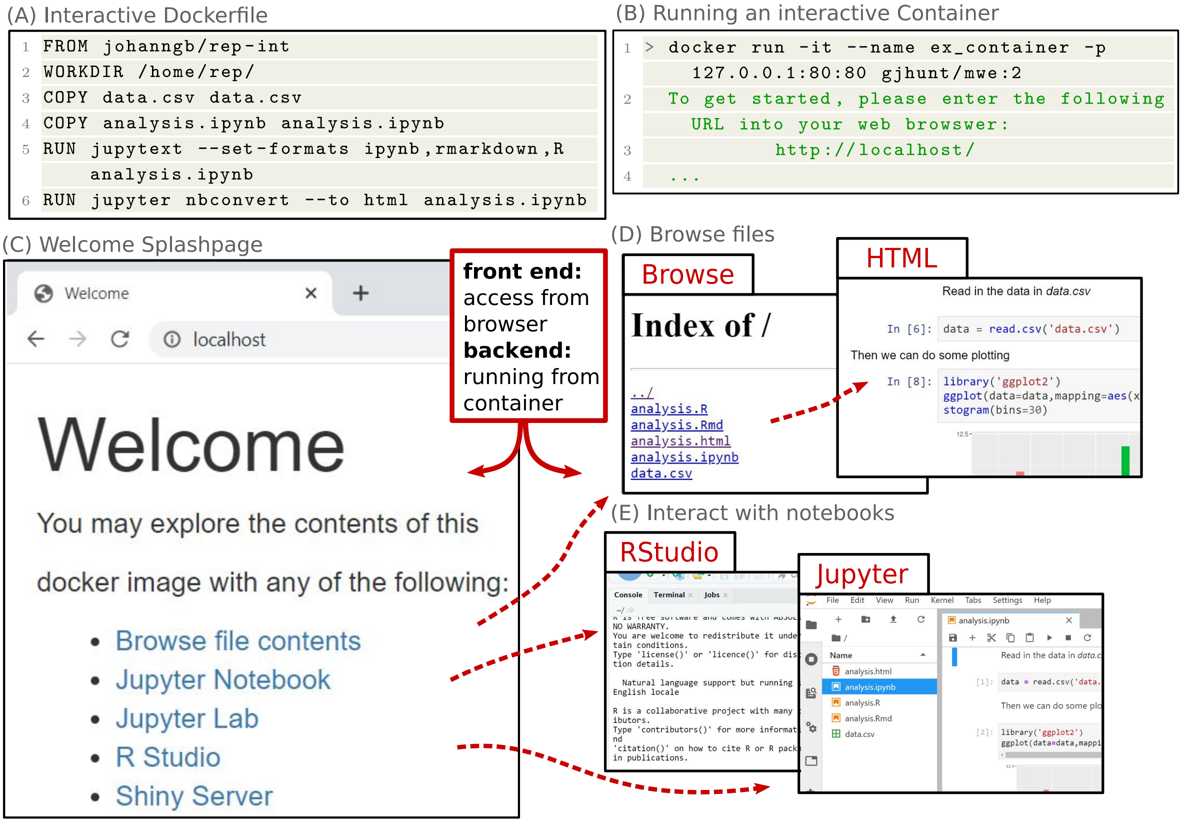
Table 2: Comparison of notebook software options RStudio, Jupyter, and Zeppelin.

	RStudio	**Jupyter**	**Zeppelin**
Notebook Type	R Markdown (text, HTML)	Jupyter (JSON)	Zeppelin (JSON)
Convertible to	None	R Markdown, code script	Jupyter
Language Support	≥56	≥152	≥37
Export Types	≥18	≥9	None
Widgets Backends	**Shiny**, HTML Widgets	Several	Jupyter, **Shiny**
Caching Support	R (native)	Python (add-ons)	None
Ex. Docker Image	`rocker/rstudio`	`jupyter/base-notebook`	`apache/zeppelin`

In summary, for every-day statistical analyses, we recommend either Jupyter or RStudio but also using Jupytext to mirror copies into both formats. Nonetheless, if one needs to connect to cluster architecture, Zeppelin likely a better candidate. Figure 6 displays an example container workflow we find works well for sharing analyses. Here, we conduct analysis with Jupyter and then use Jupytext to mirror the analysis into R Markdown, a code script, and a HTML rendering for showcasing. These files are then containerized by building off a custom base image we have created containing Python, R, Jupyter, RStudio Server, and R **Shiny**. Once running, the container is accessible through the host computer's web browser where a start-page offers several options to interact with the analysis including browsing the files (e.g., to view the HTML rendering) or opening the notebooks in a graphical interface like Jupyter or RStudio.

4. Code Sharing and Beyond

Containerizing code has potential benefits in a wide variety of contexts. For example, in a peer review process journals might ask authors to provide an interactive containerized version of their analysis. Building such an image is relatively easy to do and would allow reviewers to quickly assess a working version of the code. Additionally, containerized analyses might provide a more secure way for reviewers to run third party code. For example, when using Docker, containerized code cannot see, change, affect, or in any way alter other containers or the host system. For security-conscious individuals or



(A) Interactive Dockerfile

```
1 FROM johanngb/rep-int
2 WORKDIR /home/rep/
3 COPY data.csv data.csv
4 COPY analysis.ipynb analysis.ipynb
5 RUN jupytext --set-formats ipynb,rmarkdown,R
      analysis.ipynb
6 RUN jupyter nbconvert --to html analysis.ipynb
```

(B) Running an interactive Container

```
> docker run -it --name ex_container -p
    127.0.0.1:80:80 gjhunt/mwe:2
  To get started, please enter the following
    URL into your web browswer:
            http://localhost/
  ...
```

(C) Welcome Splashpage

Welcome

You may explore the contents of this docker image with any of the following:

- Browse file contents
- Jupyter Notebook
- Jupyter Lab
- R Studio
- Shiny Server

front end: access from browser **backend:** running from container

(D) Browse files

Browse

Index of /

../
analysis.R
analysis.Rmd
analysis.html
analysis.ipynb
data.csv

HTML

Read in the data in *data.csv*

```
In [6]: data = read.csv('data.csv')
```

Then we can do some plotting

```
In [8]: library('ggplot2')
        ggplot(data=data,mapping=aes(x
        stogram(bins=30)
```

(E) Interact with notebooks

RStudio

```
NO WARRANTY.
You are welcome to redistribute it unde
tain conditions.
Type 'license()' or 'licence()' for dis
tion details.

  Natural language support but running
English locale

R is a collaborative project with many
ibutors.
Type 'contributors()' for more informat
nd
'citation()' on how to cite R or R pack
in publications.
```

Jupyter

analysis.html
analysis.ipynb
analysis.R
analysis.Rmd
data.csv

Read in the data in *data.c*

```
[1]: data = read.csv('data.
```

Then we can do some plo

```
[2]: library('ggplot2')
     ggplot(data=data,mappi
```

Figure 6: An interactive containerization workflow. (A) **(Line 1)** An interactive dockerfile built from `johanngb/rep-int` base. **(Line 5)** `jupytext` links the Jupyter notebook to a R Markdown notebook and script. **(Line 6)**
proglangJupyter runs the notebook and saves input/output as a HTML document for showcasing. (B) We build the image and name it adding the tag `:2` to indicate it is version 2 of our previous example. Subsequently, we may run the image interactively with `-it`, naming it with `-name` and correctly mapping ports with `-p`. (C) The start page for the interactive container. Several options for interacting with the analysis files are listed. (D) We may browse the files or (E) open the notebooks with one of several choices of graphical web-based interfaces running from the container.

institutions, this may be attractive. An added benefit is that one need not clutter up their system installing single-use libraries in order to evaluate third party analyses.

Containerized notebooks may also be used as a tool for teaching allowing distribution of identical code, data, and a computing environments to all students. Conversely, student projects in applied statistical courses could be containerized before submission. While a small amount of time would need to be devoted to teaching students some simple mechanics of containerization, in our estimation this is not more complicated than other coding tasks required in many courses and would provide an opportunity for a discussion with students about research reproducibility, replicability as well as good coding practices.

Beyond the direct benefits of making code more easily shareable, the act of containerizing analyses can itself serve as a helpful review step in a scientific pipeline. Preparing analyses for containerization forces one to review the code. This encourages simplification and refactoring of code, as well as writing of the associated documentation and commentary. It also provides an opportunity to re-run the analysis in a hands-off manner to ensure that the notebooks and the entire code pipeline actually correctly produce the results when run sequentially. If the final results included in a manuscript are the output of a container then one can be ensured the results are computationally reproducible. Additionally, software like Docker can be interwoven seamlessly into popular code sharing and versioning workflows. For example, one can connect GitHub and Dockerhub accounts together so that updates to code on GitHub are automatically propagated to Dockerhub where an image is subsequently built. Alternatively, Docker can directly pull and build repositories from GitHub.

Containerization is more than just an approach for preserving passive code archives. It allows rich interaction and exploration of analysis and helps create usable and reproducible analyses. Containerizing interactive analyses can enhance the ability of statisticians to easily share code, analyses, and ultimately ideas.

References

Bates, D., Mächler, M., Bolker, B., and Walker, S. (2015). Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, 67(1):1–48, DOI: `10.18637/jss.v067.i01`.

Blischak, J. and Hill, A. (2021). CRAN Task View: Reproducible Research. `https://cran.r-project.org/web/views/ReproducibleResearch.html`.

Broman, K., Cetinkaya-Rundel, M., Nussbaum, A., Paciorek, C., Peng, R., Turek, D., and Wickham, H. (2017). Recommendations to Funding Agencies for Supporting Reproducible Research . `https://www.amstat.org/asa/files/pdfs/POL-ReproducibleResearchRecommendations.pdf`.

CERN Data Centre & Invenio (2022). Zenodo - Research. Shared. `zenodo.org`.

Chang, W., Cheng, J., Allaire, J., Sievert, C., Schloerke, B., Xie, Y., Allen, J., McPherson, J., Dipert, A., and Borges, B. (2021). *shiny: Web Application Framework for R*, `https://CRAN.R-project.org/package=shiny`. R package version 1.7.1.

Digital Science (2022). figshare - credit for all your research. `figshare.com`.

Docker Inc. (2021a). Build and Ship any Application Anywhere. `https://hub.docker.com/`.

Docker Inc. (2021b). Empowering App Development for Developers | Docker. `https://www.docker.com/`.

Doxygen (2022). Doxygen. `https://www.doxygen.nl/index.html`.

Ellis, S. E. and Leek, J. T. (2018). How to share data for collaboration. *The American Statistician*, 72(1):53–57, DOI: `10.1080/00031305.2017.1375987`, `https://doi.org/10.1080/00031305.2017.1375987`. PMID: 32981941.

Essawy, B. T., Goodall, J. L., Voce, D., Morsy, M. M., Sadler, J. M., Choi, Y. D., Tarboton, D. G., and Malik, T. (2020). A taxonomy for reproducible and replicable research in environmental modelling. *Environmental Modelling & Software*, 134:104753, ISSN: 1364-8152, DOI: 10.1016/J.ENVSOFT.2020.104753.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172. DOI: 10.1109/ISPASS.2015.7095802.

Github, Inc. (2021). Where the world builds software. https://github.com/.

Graziano, C. D. (2011). A performance analysis of xen and kvm hypervisors for hosting the xen worlds project. https://lib.dr.iastate.edu/etd/12215.

Journal of the American Statistical Association (2022). Instructions for authors. https://www.tandfonline.com/action/authorSubmission?show=instructions&journalCode=uasa20.

Kaggle Inc. (2019). Kaggle: Your Machine Learning and Data Science Community. https://www.kaggle.com/.

Knoth, C. and Nüst, D. (2017). Reproducibility and Practical Adoption of GEOBIA with Open-Source Software in Docker Containers. *Remote Sensing 2017, Vol. 9, Page 290*, 9(3):290, DOI: 10.3390/RS9030290, https://www.mdpi.com/2072-4292/9/3/290/htmhttps://www.mdpi.com/2072-4292/9/3/290.

Knuth, D. E. (1984). Literate Programming. *The Computer Journal*, 27(2):97–111, ISSN: 0010-4620, DOI: 10.1093/comjnl/27.2.97, https://doi.org/10.1093/comjnl/27.2.97.

Matt Cone (2021). Markdown Guide. https://www.markdownguide.org/.

National Academies of Sciences, E. and Medicine (2019). *Reproducibility and Replicability in Science*. The National Academies Press, Washington, DC, ISBN: 9780309486163, DOI: 10.17226/25303, https://www.nap.edu/catalog/25303/reproducibility-and-replicability-in-science.

National Academy of Sciences, National Academy of Engineering, and Institue of Medicine (2009). *On being a scientist: A guide to responsible conduct in research: Third edition*. National Academies Press, ISBN: 9780309141352, DOI: 10.17226/12192.

Nature (2022). Initial submission. https://www.nature.com/nature/for-authors/initial-submission.

Nüst, D., Eddelbuettel, D., Bennett, D., Cannoodt, R., Clark, D., Daroczi, G., Edmondson, M., Fay, C., Hughes, E., Kjeldgaard, L., Lopp, S., Marwick, B., Nolis, H., Nolis, J., Ooi, H., Ram, K., Ross, N., Shepherd, L., Sólymos, P., Swetnam, T. L., Turaga, N., Van Petegem, C., Williams, J., Willis, C., and Xiao, N. (2020).

The Rockerverse: Packages and Applications for Containerization with R. *R Journal*, 12(1):437–461, DOI: 10.32614/rj-2020-007, https://arxiv.org/abs/2001.10641v4.

Olaya, P., Lofstead, J., and Taufer, M. (2020). Building Containerized Environments for Reproducibility and Traceability of Scientific Workflows. https://arxiv.org/abs/2009.08495v1.

Ooi, H., de Vries, A., and Microsoft (2021). *checkpoint: Install Packages from Snapshots on the Checkpoint Server for Reproducibility*, https://CRAN.R-project.org/package=checkpoint. R package version 1.0.0.

Python Software Foundation (2021). PyPI · The Python Package Index. https://pypi.org/.

Red Hat, Inc. (2021). Getting Started with Podman . podman.io/getting-started/.

Science (2022). Information for authors. https://www.science.org/content/page/science-information-authors.

Simonsohn, U. and Gruson, H. (2021). *groundhog: The Simplest Solution to Version-Control for CRAN Packages*, https://CRAN.R-project.org/package=groundhog. R package version 1.4.0.

Sylabs (2021). Singularity | Sylabs.io . sylabs.io/singularity/.

The Apache Software Foundation (2022). Apache Spark; - Unified Engine for large-scale data analytics. spark.apache.org.

The R Project for Statistical Computing (2021). The Comprehensive R Archive Network. https://cran.r-project.org/.

Turnbull, J. (2014). *The Docker Book: Containerization is the new virtualization.* James Turnbull, https://dockerbook.com/.

Ushey, K. (2021). *renv: Project Environments*, https://CRAN.R-project.org/package=renv. R package version 0.13.2.

Visconti, A., Martin, T. C., and Falchi, M. (2018). YAMP: a containerized workflow enabling reproducibility in metagenomics research. *GigaScience*, 7(7), ISSN: 2047-217X, DOI: 10.1093/gigascience/giy072, https://doi.org/10.1093/gigascience/giy072. giy072.

Wiebels, K. and Moreau, D. (2021). Leveraging Containers for Reproducible Psychological Research:. *https://doi.org/10.1177/25152459211017853*, 4(2):1–18, DOI: 10.1177/25152459211017853, https://journals.sagepub.com/doi/full/10.1177/25152459211017853.

Affiliation:

Gregory J. Hunt
Department of Mathematics
William & Mary
Williamsburg, VA
23185
E-mail: ghunt@wm.edu

Journal of Data Science, Statistics, and Visualisation https://jdssv.org/
published by the International Association for Statistical Computing
http://iasc-isi.org/

ISSN 2773-0689
March 2023, Volume III, Issue I *Submitted:* 2021-08-04
doi:10.52933/jdssv.v3i1.53 *Accepted:* 2022-06-09