Reproducibility as a Service

by

Joseph Wonsil

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

May 2021

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Reproducibility as a Service

submitted by **Joseph Wonsil** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Margo Seltzer, Professor, Computer Science, UBC
Supervisor

Reid Holmes, Associate Professor, Computer Science, UBC
Supervisory Committee Member

Tiffany Timbers, Assistant Professor of Teaching, Statistics, UBC
Supervisory Committee Member

Abstract

Recent studies demonstrated that the reproducibility of previously published computational experiments is inadequate. Many of these published computational experiments are not reproducible, because they never recorded or preserved their computational environment. This environment consists of artifacts such as packages installed in the language, libraries installed on the host system, file names, and directory hierarchy. Researchers have created reproducibility tools to help mitigate this problem, but they do nothing for the experiments that already exist in online repositories. This situation is not improving, as researchers continue to publish results every year without using reproducibility tools, likely due to benign neglect as it is common to believe publishing the code and data is sufficient for reproducibility. To clarify the gap between what existing reproducibility tools are capable of and this issue with published experiments, we define a framework to distinguish between actions taken by a researcher to facilitate reproducibility in the presence of a computational environment and actions taken by a researcher to enable reproduction of an experiment when that environment has been lost. The difference between these approaches in reproducibility lies in the availability of a computational environment. Researchers that provide access to the original computational environment perform proactive reproducibility, while those who do not enable only retroactive reproducibility. We present Reproducibility as a Service (RaaS), which is, to our knowledge, the first reproducibility tool explicitly designed to facilitate retroactive reproducibility. We demonstrate how RaaS can fix many of the common errors found in R scripts on Harvard's Dataverse and preserve the recreated computational environment.

Lay Summary

One of the pillars of modern science is computation. Research software allows scientists to quickly and accurately analyze large amounts of data. When scientists publish their results, it is critical that their peers can then repeat their computations. Unfortunately, researchers have discovered they cannot reproduce the findings of openly published analyses. This failure is not necessarily due to an incorrect experiment or flawed science. Instead, the software that scientists use to analyze their data fails if the original authors do not take careful steps ahead of time to preserve it. Researchers created new technologies to help facilitate this preservation, but they are not (yet) commonly used. We created a new tool, Reproducibility as a Service, that helps facilitate reproducibility for publicly available analyses that lack this preservation.

Preface

This thesis is based on unpublished work conducted collaboratively in the Systopia Lab at the University of British Columbia. None of the text of this thesis is taken directly from previously published or collaborative articles.

This project originated at Harvard University with Christopher Chen's undergraduate thesis. Chen created the tool containR, which this project considerably extends. Chen is responsible for the original design of a reproducibility web service using Flask, Celery, and provenance for the R language. We started with his original codebase and modified it extensively to become our service, RaaS. The full extent of our design and how it differs from containR is discussed in Chapter 3 and Chapter 4.

In addition to my work on this project, four undergrads assisted on various features. Nichole Boufford implemented the initial static analysis for the R language. Akash Sivaram and Tianhang Cui (Albert) implemented the language-dependent code for the Python language. These three students and I collaboratively designed and implemented the new language-separated design of RaaS. The four of us deliberated and agreed upon a design, Akash wrote the initial implementation, and I updated it to fix bugs and fulfill the tool's changing requirements. Albert also explored the possibility of implementations for different languages such as MatLab. The fourth student, Prakhar Agrawal, implemented the language-dependent code for the Julia language after the previous three students completed their time on the project.

The rest of the project is original work I conducted.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to extend my utmost gratitude to everyone who has helped me while I conducted this research and wrote this thesis. I always knew I could count on the people in my life for support, and these last two years have only cemented this belief as I started my graduate career a few months before a global pandemic.

I am particularly grateful to my supervisor Margo Seltzer, not only for officially guiding me during this degree but for unofficially advising me while I finished my undergraduate degree. I am so glad to have been able to work with you at Harvard Forest originally because I was unsure about graduate school before that summer. So thank you, Margo, and everyone else who worked with me there, Aaron Ellison, Barbara Lerner, Emery Boose, Elizabeth Fong, and Orenna Brand. Thank you, Barb, also for quickly implementing changes and fixes to RDataTracker when I contacted you about bugs while working on this project.

Thank you to committee members Reid Holmes and Tiffany Timbers for providing wonderful feedback on this thesis.

Thank you to all of the undergraduate students who assisted with making this project what it is today, Nichole Boufford, Akash Sivaram, Tianhang Cui (Albert), and Prakhar Agrawal. You all worked so well, despite rarely *if ever* being able to meet face-to-face.

I do not think I can even express how grateful I am to my family for everything they have done for me, including supporting me emotionally and financially, when I decided not only to move across the continent but to a different country for graduate school. Thank you to my parents, who choose to have unwavering, but probably baseless, confidence in my abilities. Thank you to my grandmother, as I have only gotten this far due to your help. Thank you to my sister for your emotional support,

editing my thesis (hi!), and moving to the Pacific Northwest at the same time as me.

I started working on this project initially because of a class project, so thank you, Francis Nguyen, for being my research partner on that project and continuing to work with me on language-level provenance afterward. Especially since working on "L-Vis" led me to containR and, ultimately, this thesis.

I am also incredibly grateful to the people close to me who have helped in so many ways. Thank you, Paulette Koronkevich, for far too many things for me to list here, but you were always available to chat and provide feedback on this paper and my research. Similarly, thank you, Charles Gallagher, for reasons too numerous for me to list. Not only did you provide technical help for my thesis, but emotional support. Not to mention we went through the Hero's Journey together during this thesis.

Finally, I am incredibly fortunate to be part of the most fantastic lab, Systopia. A special thanks to all of my labmates, not only for helping with technical issues but also for everything else, whether it is pad-thai day or ice cream socials.

This research was enabled in part by support provided by Compute Canada (www.computecanada.ca).

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

Nous remercions le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) de son soutien.

Chapter 1

Introduction

The traditional lab notebook has been a powerful tool for the scientific community. The backbone of our scientific system is our ability to evaluate hypotheses repeatedly. A single researcher might arrive at a particular conclusion when they test their hypothesis, but that conclusion will not be accepted into the greater scientific community until it undergoes rigorous repeated experiments. Future researchers must perform the same experiment as similarly as possible to have confidence in the same result. The lab notebook helps future researchers achieve this by acting as a record they can follow.

In the digital age, the usefulness of the lab notebook cannot extend to computation. It is undoubtedly still necessary for recording the methods of experiments and data collection across many disciplines. However, our digital infrastructure for data analysis is too complex for scientists to find the lab notebook helpful. When publishing, some might note the statistical analyses they used or even the software's name that performed the computational experiment. Ellison [18] discusses how this is not enough information for a future researcher to reproduce a computational experiment. In the present wake of a perceived reproducibility crisis in science [4], more researchers are pushing for open and reproducible science [1, 34, 41].

One of the natural responses to this reproducibility crisis is the *computational* notebook. Jupyter [28] is a commonly used computational notebook that allows researchers to analyze data while incorporating text and explanations all in line with the analysis. Unfortunately, these notebooks still encounter many reproducibility

problems due to the complexity of data computation [47]. Even with the digital equivalent of the lab notebook, researchers require different tools and project management techniques to publish reproducible computational experiments.

The shift towards open science is motivated by the desire that research becomes publicly accessible to all. Easy access to research positively impacts reproducibility, as anyone can examine or verify an experiment's results. This movement has led to new ideas and technology focused on facilitating these ideas. Researchers proposed FAIR (Findable, Accessible, Interoperable, and Reusable) data management principles to help scientists make sure their data can be found and used [63]. These principles are, in practice, embodied in research such as the Dataverse Project [58]. Dataverse is a data repository that institutions can manage and host. The largest such repository, Harvard's Dataverse [1], hosts thousands of digital artifacts from publications.

The use of repositories such as Dataverse means that information about computational experiments is now publicly available, and these repositories can begin to assume some of the responsibility the lab notebook used to carry on its own. This acceptance is the right step for reproducibility; however, many challenges remain. Studies of the computational experiments published on services such as Dataverse and GitHub discovered that most could not execute without errors [12, 47]. We confirm these results by scraping Harvard's Dataverse for R [49] scripts and attempting to execute them. The most common errors are related to the computational environment. Scripts import various packages used in the analysis; there might be other dependencies not explicitly identified in the script; and files might have different names or might have moved.

Recognizing these problems, researchers have created various tools to facilitate reproducibility. These tools often address the challenges associated with computational environments by preserving them using virtualization technologies such as Vagrant [23] or Docker [35]. Alternatively, they can virtualize just the language environment with tools such as `renv` [59] or `conda` [3]. However, these tools assume that the computational experiment still has access to a computational environment sufficient to execute it. Therefore, these reproducibility tools cannot as-

[1] dataverse.harvard.edu

sist with all of the already published computational experiments hosted on services such as dataverse. Additionally, some researchers go through lengthy publication processes, and by the time they distribute their computational experiment, they have lost or changed the original computational environment.

We identify the need for a new approach to reproducibility. We define *retroactive reproducibility* as reproducibility for a computational experiment that no longer has access to a sufficient computational environment. In contrast, we define *proactive reproducibility* as reproducibility in the presence of an experiment's computational environment. We have identified a set of challenges explicitly associated with retroactive reproducibility. Most existing reproducibility tools are capable of handling proactive reproducibility only. Some tools have taken steps towards retroactive reproducibility; however, to our knowledge, no researcher has made a tool intended to address the challenges associated with retroactive reproducibility directly. The project that comes closest to addressing this gap is the precursor to this project, Chen's containR [12].

We present Reproducibility-as-a-Service (RaaS), a system that facilitates retroactive reproducibility. We designed RaaS to address the difficulties of reproducing a computational experiment when the original environment is lost. Given the thousands of already published datasets, it can enable future researchers to examine and recompute prior findings. Alternatively, RaaS can act as a piece of the publication pipeline for researchers who did not preserve their computational environment after generating their results. Before publication, they can process their computational experiment with RaaS to ensure that future researchers can reproduce it. Even outside of the scientific domain, companies with internal data processing can benefit from better reproducibility. We built RaaS with common reproducibility technologies such as Docker [35] and data provenance and static analysis techniques to address retroactive challenges specifically. We also designed RaaS with a clean separation between the codebase features that depend on a specific research programming language, allowing developers to easily add support for new languages. RaaS currently has various levels of support for R, Python, and Julia.

RaaS preprocesses scripts from a computational experiment by locating and attempting to fix common retroactive reproducibility bugs. It uses static analysis to collect the information needed to re-create a computational environment within

a Docker image. Upon building this image, RaaS executes the computational experiment and collects provenance at the language-level. It generates a report from the provenance that allows a user to verify that that the computational experiment was successful. Once verified, users can retrieve and distribute their Docker image either through Docker itself, using the service's `push` and `pull` commands, or as a direct download of a compressed file.

We evaluate the usability of RaaS as means to facilitate retroactive reproducibility by processing R scripts from Harvard's Dataverse. First, we attempt to execute the computational experiments in a new environment with just the R language. We observed that only 10.1% of R scripts on Harvard's Dataverse could execute independently without errors. Furthermore, errors occur independent of the length of time since publication and the subject of the experiment. We identify the most common errors: missing library, incorrect working directory, and missing file errors, in that order. Using RaaS, we fix 85% of library errors, the most common error, and 86% of working directory errors. We cannot fix missing files, but RaaS can detect missing files at upload time, allowing users to fix them before publishing their experiments. Despite these promising results, we also observe that these errors tend to mask additional errors. When we analyze only those datasets that completed both with and without RaaS, only 11.8% of scripts ran without error without using RaaS, and 31.0% of the scripts ran without error when using RaaS. However, even though 69% of scripts still encountered errors, 75.3% of those errors are in a different category of error from when we executed the scripts without RaaS. This result demonstrates that RaaS is capable of automatically fixing many errors commonly found in research scripts.

The contributions of this work are:

- a framework to distinguish between actions taken by a researcher to facilitate reproducibility in the presence of a computational environment and actions taken by a researcher to enable reproduction of an experiment when that environment has been lost.
- an evaluation of reproducibility of R scripts hosted on the Harvard Dataverse.
- a new service, RaaS, that explicitly addresses retroactive reproducibility.

- an evaluation demonstrating RaaS's ability to address issues that directly relate to retroactive reproducibility.

- a set of reproducibility best practices, based on our evaluation and experience working with published computational experiments.

Our work identifies a need for a retroactive reproducibility service, given the results we observed from our evaluation of Harvard's Dataverse. We demonstrated that our service, RaaS, effectively deals with many common shortcomings inhibiting reproducibility in published datasets. Furthermore, by reducing common errors, RaaS can highlight potentially more complex errors that users need to fix. This ability positions RaaS as a potentially critical piece of a paper publication pipeline. This service can act as a "format checker" for computational experiments to ensure that they execute without errors.

In Chapter 2, we start by discussing reproducibility and the definition we adopt. We then introduce the concepts of retroactive and proactive reproducibility. We then clarify the style of programming our work is suited to address. With these definitions in mind, we discuss the FAIR principles researchers follow for open science, and we identify the challenges associated with reproducibility. We then wrap up the chapter with an overview of existing reproducibility technologies.

We present RaaS in Chapter 3. We begin with a use case, followed by an overview of RaaS and the challenges associated with retroactive reproducibility. From there, we present RaaS's architecture, the language-independent implementation details, and the language-dependent implementation details. We then discuss the evolution from prior work, containR, to Raas. Then we discuss how the experiences we encountered with reproducibility while working on RaaS shaped its design.

We present our evaluation of RaaS in Chapter 4. We begin by evaluating the current state of reproducibility of R scripts hosted on Harvard's Dataverse. Next, we evaluate how RaaS can increase these scripts' overall reproducibility. We then delve deeper into the most common error, library errors, and potential ways to mitigate them.

We discuss RaaS's limits, future work, and a set of best practices for reproducibility in Chapter 5. We examine RaaS's limitations and offer some potential

fixes. We discuss more in-depth methods to fix some of RaaS's limitations and add helpful new features. Lastly, based on our experiences with RaaS, we present a set of reproducibility best practices.

We examine how RaaS fits into the current reproducibility landscape in Chapter 6. We present platforms that provide reproducibility for new computational experiments. Then we discuss tools that can preserve existing computational environments. Finally, we examine other approaches that have started to take steps towards retroactive reproducibility.

We conclude in Chapter 7.

Chapter 2

Background

There are already numerous tools that assist with the reproduction of computational experiments. They use various techniques such as:

1. hosting computational environments on remote servers to help archive and distribute computational experiments.
2. virtualization to increase a computational environments' portability.
3. tracing system calls to identify all necessary files and programs used by a computational experiment.

These tools benefit scientists who create a computational experiment and are actively trying to make sure it is reproducible in the future. In practice, most researchers do not do this. According to Stodden [57], some researchers think it takes too much time and effort to prepare a computational experiment for release. Even those who release their analysis and data believe sharing only the scripts and data files is sufficient for reproducibility. Unfortunately, researchers including Pimentel et al. [47] and Chen [12] found that thousands of previously published computational experiments do not successfully execute. This problem has a significant impact, as it is harder for the scientific community to verify contributions they cannot reproduce. We address reproducibility challenges, particularly when authors published the computational experiment without reproducibility in mind.

In this chapter, we explore the current reproducibility landscape. We identify reproducibility standards and existing reproducibility technologies. These technologies vary from virtualization for environment preservation to provenance as a means of verification. While these methods vary in their approach, they fall short at addressing the reproducibility of already published computational experiments in their current state. We address this shortcoming with our work; however, it is critical to define our understanding of reproducibility, which aspect of reproducibility we address, and the researchers for whom we are designing a solution.

2.1 Defining Reproducibility

Our focus is on the challenges associated with reproducibility rather than replicability. Given the confusion between the two words and the different definitions depending on the discipline, we use the National Academy of Sciences' definitions [40]. They define *reproducibility* as "obtaining consistent results using the same input data; computational steps, methods, and code; and conditions of analysis." Therefore, reproducibility always deals with only one analysis and the ability to execute it at any point in the future. The critical difference between reproducibility and replicability is whether or not new data are collected. *Replicability* is "obtaining consistent results across studies aimed at answering the same scientific question, each of which has obtained its own data." [40]. Automating an entire study's design to answer a hypothesis is out of scope for this project, where we focus primarily on reproducibility. The question that remains is what does it mean *to reach the same result?*

Reproducibility: obtaining consistent results using the same input data; computational steps, methods, and code; and conditions of analysis [40].
Replicability: obtaining consistent results across studies aimed at answering the same scientific question, each of which has obtained its own data [40].

We examine reproducibility failures rather than successes due to the vague meaning of successful reproduction. "Achieving the same results" has different

meanings, not only across subjects, but within them. Even in a subject such as numerical simulation, which does not rely on measuring potentially inconsistent natural phenomena, Diethelm [16] discusses how the numerical simulation field should accept the common occurrence that running the same computational experiment more than once can lead to different outputs. Random number generation, floating-point arithmetic, and stochastic processes all can lead to deviations on repeated executions and executions on different devices. Researchers should tolerate some deviations, but how much is dependent on many factors. A social sciences study might find an error range of 15% sufficient due to how they sample data representing a human population, which naturally has high variability. Meanwhile, a metrology[1] study might tolerate an error of only 0.1% in the same situation due to their subject's precise nature. It might seem this problem is more relevant to replicability than reproducibility. However, we already know from Diethelm that simply running a computational experiment twice might produce different results in numerical simulations. Ellison [18] notes that "standard" statistical methods vary across different software and even across different versions of the same software, in addition to numerical changes in arithmetic due to varying operating systems and chipsets. With all these potential variations in mind, we leave the final stamp of approval for "successfully reproduced" to our users. Instead of addressing successful reproductions, we examine failed reproductions where a computational experiment does not finish due to an error.

We further categorize reproducibility into two approaches, which is the basis of our new framework for approaching reproducibility. Many researchers now publish their computational experiments alongside their results. Unfortunately, these experiments are often missing a critical piece, the computational environment. This environment includes the directory structure of the experiment and the other software that a computational experiment depends on to execute. In terms of reproducibility, they are missing the "conditions". Someone attempting to rerun a computational experiment is still using the original data and computational steps, so this situation still falls under the umbrella of reproducibility. However, the approach they have to take is different from writing an original computational ex-

[1] Metrology is the study of measurement.

periment and trying to ensure its future reproducibility. We present two new definitions to clarify these different approaches, retroactive reproducibility and proactive reproducibility. *Retroactive reproducibility* is the process of reproducing a computational experiment without the initial computational environment. *Proactive reproducibility* is the method of ensuring the future reproducibility of an experiment that is currently executable within its initial computational environment. A researcher is successful with proactive reproducibility if they eliminate the need to attempt retroactive reproducibility. A common misconception is that using a scripting language such as R or Python and providing open access to the scripts and data is sufficient proactive reproducibility. Unfortunately, that is unlikely to be the case, because it is common to have missing dependencies that constitute the computational environment. This dependency problem is similar to what software developers face when distributing their projects.

Retroactive reproducibility: the process of reproducing a computational experiment without the initial computational environment.
Proactive reproducibility: the method of ensuring the future reproducibility of an experiment that is currently executable within its initial computational environment

2.2 Research Programming

We can draw from the knowledge used to solve software engineering problems, but it is essential to distinguish between research programming (writing code for the purpose of conducting an experiment) and software development. *Research programming* is gaining insight from data using computer programming [22]. Research programming is also often a form of exploratory programming. Kery and Myers define *exploratory programming* as a "task with two properties," [27] which are:

Property 1 "The programmer writes code as a medium to prototype or experiment with different ideas".

Property 2 "The programmer is not just attempting to engineer working code to match a specification. The goal is open-ended, and evolves through the process of programming."

The critical difference is the second property. Software developers tend to write code to match a specification, whereas exploratory programmers' goals are not always clear from the beginning. In some cases, their goal is entirely open-ended; in others, they might try to support or refute a hypothesis, but how they will do that is not known at the start. Alternatively, after analyzing data, they might encounter unexpected results and pivot to exploring a different aspect. The impact of this different programming style manifests in code quality.

Research programmers tend to trade code quality for the ability to iterate on ideas quickly [27]. Kery and Myers discuss how even experienced software engineers admit to sacrificing code quality when engaging in exploratory programming. In reality, most research programmers are domain experts in a field other than computer science, so they often do not even have the software experience that could help increase code quality. One significant reason for research code quality is the perception that they will not use exploratory code again [27]. If these programmers perceive that they will re-write the code later or need it only once, they are less motivated to implement stricter practices such as documentation, modularity, and version control. Even for the researchers who might have a more concrete idea of their goal, the nature of research programming can lead to code quality issues.

Even though someone might write a research script in the same language as a general-purpose software tool, its inherent structure differs due to their differences in goals. Research scripts are highly specialized lists of computational steps that are often not intended to be maintainable, robust, or production quality [22]. Each script ultimately tests a specific hypothesis for a specific input even if the programmer does not know the hypothesis at the beginning. This structure leaves less room for reuse than general-purpose programs. Therefore, the control flow in computational experiments tends to be a linear set of transformations to data. Additionally, the specialization for a specific data set leads to many hard-coded strings representing file paths, working directories, and other objects. This style is often not robust enough to work across environments. When the environment does change, these strings might no longer represent their intended target. Researchers must be cognizant of this behavior when writing a computational experiment to avoid future reproducibility issues. Otherwise, even with the assistance of existing reproducibility tools, it will be challenging to reproduce their work.

2.3 FAIR Guiding Principles

In 2014, a group interested in data reuse and discoverability met for a workshop to discuss a set of community principles around data management. This workshop, "Jointly Designing a Data Fairport", concluded with some foundational principles that would later become the FAIR Guiding Principles. *FAIR* means Findable, Accessible, Interoperable, and Reusable [63]. According to Wilkinson et al. [63], these principles are not a standard but a guide to help achieve accessible, open, and reusable data management. It is important to note we did not create our project to satisfy all principles of FAIR. Instead, we noticed shortcomings in the overall scientific community with fulfilling these principles, specifically when it comes to "Reuse." As discovered by the research of Pimentel et al. [47] and Chen [12], previously published experiments are unlikely to execute successfully. Therefore, our project supplements existing reproducibility platforms by strengthening the reusable aspect of the FAIR principles.

2.4 Reproducibility Challenges

While there are many challenges to reproducibility, the most common and problematic involve losing one or more parts of the computational experiment. If we consider a computational experiment's composition, computational reproducibility requires three components: input data, computational steps, and computational environment. Previously, scientists felt it was sufficient to describe the steps they took with their data. They did not publish any components from their computational experiment. Without any of the components, it is impossible to reproduce their experiments, only to replicate them. As the concept of open science has become more popular, an increasing number of researchers publish components of their computational experiments online. However, many publish only their data or only their computational steps. There are legitimate reasons why researchers might not publish everything. Some data are sensitive and restricted, such as personal health data, and some data are too large to share on regular archival sites such as the data CERN generates. On the other hand, some scientists might not be able to track their workflow or might lose track over long-term studies. Whatever the case may be, it is common for one or two of the three computational experiment

```r
# Load needed packages
library(sf)
library(preText)

# Set working directory
setwd("/home/seq/data_analysis")

data1.df <-
 ↪  read.csv("/home/seq/data_analysis/data/data1.csv")
data2.shp <-
 ↪  st_read("/home/seq/data_analysis/data/data2.shp")

# Contains print_data function definition
source("scripts/function.R")

print_data(data1.df)
```

Listing 1: An R script with reproducibility problems.

components to be missing.

In our experience, losing the computational environment is an understated problem. Many archival sites host files using FAIR principles. This practice supports computational steps and input data reasonably easily. Most computational steps are simply a file for a scripting language such as R, Python, or Julia. Input data are often an individual or collection of files, such as a CSV or database file. Sharing a file in our current digital ecosystem is relatively easy with many options such as e-mail and file-sharing services. However, it is more complex and challenging for a file-hosting site to sufficiently archive a computational environment.

Consider the computational environment for a single R script that uses two different R packages to analyze some input data. We show an example such script in Listing 1. We want this script to be reproducible, so we need to define its environment to upload it to a FAIR archival repository. An obvious initial assessment tells us we certainly need an environment with the R language, including the two packages we use in the script, `sf` and `preText`. So, our first attempt at reproducibility is to write a README with the instructions to install R on a machine and

install the required packages. Unfortunately, one of the R packages (preText[15]) is no longer available on the popular repositories, CRAN [25] and BioConductor [26]. We now need to find a way to provide the package itself with the script, so we export the package source code and provide instructions in the README to install an R package from a source file. However, then we discover that these two R packages depended on multiple other R packages. We recursively find all of the dependencies for the two main R packages in our script and compile a list of more packages that a future user must install. Then we discover that we wrote our script in R 3.6.3 and relied on R's default behavior to read strings as factors. New users are installing the latest R (4.0.4), which does not default strings to factors anymore. Also, we discover some of our R packages (`sf` [44]) will not work, because they rely on system libraries (`gdal` [19]) that we must separately install. Quickly, identifying what constitutes our computational environment has become a nightmare. As we discussed previously, Ellison [18] noted these issues and consequently advocates for preserving the environment for computational experiments.

This example highlights many of the problems we face with computational environments. An older version of software, either the language or a package, might behave differently than the most recent version. Some functions might become deprecated; some entire libraries might become deprecated in favor of newer supported ones. An operating system might not sufficiently support the computational experiment's system requirements, because it is too old, too new, or less popular. All of these interactions between the host system and the computational experiment complicate easily sharing a computational environment.

Another challenge with the computational environment includes how scripts interact with it. Most scripts likely import data from a file. They have to reference a path from their current working directory to the file they wish to import. If a user moves the file or the working directory, the script no longer executes, because it cannot find the data it needs to execute. A typical example is when users hard-code a file path from the root directory. If another researcher reproduces their computational experiment, they will be running on a different file system, with a different path from the root to the file, potentially even a different operating system with different hierarchies. Therefore computational experiments that switch working directories or do not reference file paths relatively can cause reproducibility

problems in the future.

While we discuss retroactive reproducibility in terms of published computational experiments, it is common for issues to arise before then. When a researcher prepares a computational experiment for publication, it might already be in a state where retroactive reproducibility is necessary. They might have written their computational experiment months or even years prior to their final publication date. In that time, they have changed and lost their computational environment. They might not even notice this and upload the computational experiment with a variety of unnoticed problems. The need for retroactive reproducibility starts even before distribution. If publication pipelines included a reproducibility service, similar to format checks, common in the paper submission pipeline, they could detect these errors and fix them ahead of time.

2.5 Existing Reproducibility Technology

Many technologies currently exist to solve some aspects of reproducibility, including project management tools, online archival platforms, virtualization, system call tracing, and provenance. Project management tools help users create a project by managing packages and the environment for them and can also assist in rebuilding these environments. Online archives are a pillar of open science and fulfill almost all principles of FAIR. Technologies such as virtualization help researchers capture the environment used in their computational experiments, while provenance collection provides a fine-grain record of execution. These additional digital artifacts supply the missing components many published experiments lack. Project management tools and provenance can describe a necessary environment. Virtualization allows researchers to publish the environment alongside the code and data. Researchers choose different methods according to their needs; each method has its strengths and drawbacks.

2.5.1 Project Management

Various tools assist research programmers with managing and reproducing computational experiments. These tools can help ensure research code is reproducible, manage computational environments, and help select correct dependencies.

A language-independent example of one of these tools is `GNU make`[52] (or just `make`). Users typically interact with `make` by writing a Makefile with instructions on executing their computational experiment. This tool can track dependencies or "targets" in a project and only build or execute the files that have changed since the last execution. Thus, `make` can help cut down on execution time by only executing scripts that have changed. Also, users can supply make with command-line arguments for their scripts, so they do not have to enter them by hand each time.

Some tools create "virtual" environments so that users can keep each of their projects' dependencies separate from one another. Some common examples include `renv` [59] for R and conda [3] for multiple languages, including R and Python. They can also help with reproducibility by creating snapshots of the environment that a user can reload later. However, in some cases, packages are not available after a researcher initially installed them. For example, if a researcher installs a package from CRAN and takes a snapshot with `renv`, it might be impossible to reload from that snapshot on a different device if CRAN stops hosting that package. In these situations, the Microsoft R Application Network (MRAN) [36] is useful. Microsft hosts snapshots of CRAN at various dates and allows users to install packages from these snapshots. These snapshots can help mitigate problems if CRAN stops hosting a package.

Finally, there are language packages that help improve code quality. For example, the R package `targets`[30] is similar to make, except specific to the R language. Additionally, the `here`[39] package minimizes problems with R projects and working directories by building file paths for the user. This method also helps prevent bad practices such as writing file paths beginning at a computer's root directory.

2.5.2 Online Repositories

Online FAIR-focused repositories promote open science and distribution. Currently, multiple services provide hosting with rich features for metadata and identification. In this context, a *dataset* is all of the digital artifacts from a particular computational experiment: scripts, data, README, and anything else a user up-

loads. These repositories host files, assign each file a unique DOI, and assign each dataset a DOI. These repositories handle and create metadata about datasets, including publication dates, subject, and file versioning. They excel at FAIR data management for computational steps and input data, but fall short at hosting a computational environment.

While there are a few of these online repositories, we focus on one in particular, Harvard's Dataverse[2]. The Dataverse Project produces open-source software that any institution can use to archive datasets [58]. Harvard both manages the Dataverse Project and has the largest Dataverse repository. Dataverse originated as a means to archive quantitative social science data [2]. It has since evolved, expanding and addressing more data management challenges for all subjects.

Other online repositories exist that are commonly used by researchers and can support computational environments. Unfortunately, these services do not support FAIR practices. For example, Docker Hub [17] hosts Docker image repositories. Docker's default behavior when looking for an image to run that does not exist locally is to check Docker Hub first. Any researcher can publish their images on Docker Hub, making distribution incredibly simple. Unfortunately, Docker Hub is not an archival site and will not freely host images indefinitely [37].

2.5.3 Virtualization

Virtual Machines (VMs) are entire operating systems running within a sandboxed environment on a host machine. VMs have great reproducibility potential. A researcher can write an analysis entirely within a VM and, once it is complete, save the current machine state as a VM image. They can distribute this VM image and anyone who loads it has the entire operating system the initial researcher used. This approach is advantageous, because it works even if a researcher wrote a script with poor reproducibility practices, such as hard-coded file-paths that start at the root. Unfortunately, this solution is more heavyweight than others, as it requires running an entire operating system on top of another. Consequently, VM images can be quite large, depending on the data within and the operating system used. While reproducibility potential is relatively high, distribution and storage are more

[2] dataverse.harvard.edu

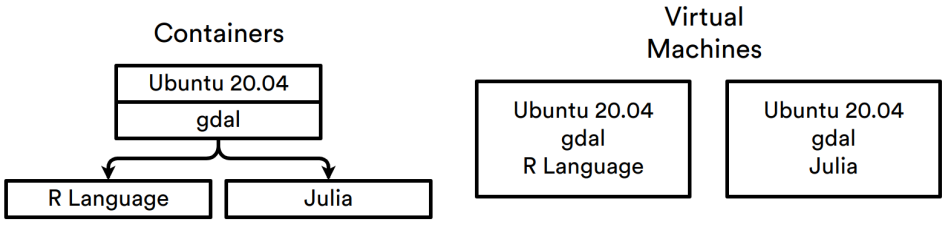

Figure 2.1: Containers versus VMs. This figure shows the structure of two Docker images compared to two VM images.

complicated. A lighter-weight alternative to VMs is containers.

Containers are similar to VMs; however, they share the host's operating system and provide an isolated computational environment and file system. Since containers do not have to emulate an entire operating system, they can be faster to start and lighter weight. Docker [35] is one of the most popular container solutions, and development teams often use Docker containers to run their software in production. Docker users create *Docker images*, which are snapshots of computational state similar to a VM image. They can then run them as *Docker Containers*, executable instances of the Docker image. Unless otherwise specified, we refer to Docker images and Docker containers as images and containers, respectively. A key distinction here is that containers are ephemeral mutable processes, and images are their immutable templates. Running a container might cause the state of the container to change; for example, a program might create a new file in the container's environment. However, once that container stops running, all those changes are gone. If a user runs a second container from the same image, it will be as if the first container never existed. They could even run two or more instances of the same container simultaneously based on one image. This feature alone is not necessarily unique as users working with VMs can also save images and distribute them.

The critical difference between VMs and containers is that containers focus on applications rather than systems. Each Docker image usually houses only a single service. Consequently, Docker images provide more reuse of shared application resources than do VM images. Docker constructs images in layers, and two images that contain the same stack of layers will share the layers rather than replicate them. For example, a user creates an analysis composed of multiple containers executing

```
FROM r-base

COPY . /home/docker/

CMD ["Rscript", "/home/docker/run_analysis.R"]
```

Listing 2: Simple Example Dockerfile.

in parallel. Each container does something different; however, they are all based on Ubuntu 20.04 with `gdal` [19] installed. If the user created two images for this service, both images would share these layers, as seen in Figure 2.1. Additionally, the user can publish their images by pushing them to an online registry. Any other user who already has a copy of the Ubuntu 20.04 image and pulls the published images from the registry will need to download only the layers they do not already have. In contrast, the same setup with VMs requires that each VM have a copy of Ubuntu 20.04, and users would always have to download the entire thing.

Users can create images in two ways, interactively or with a Dockerfile. When a user creates an image interactively, they run a container, change the state, and save a snapshot of the container as an image. Unfortunately, this method can lead to problems, especially if the user needs to rebuild the image. Rebuilding the image would require remembering each step they took before they snapshotted the container. Alternatively, they can use Dockerfiles. A Dockerfile is a set of instructions that direct Docker in building the image. Listing 2 shows a simple example Dockerfile. Typical Dockerfiles such as this one start from an existing image and build upon it. In this case, the file instructs Docker to copy a dataset into the container and then sets the command that executes when the container runs. When the user builds this image and then executes the container, the container will execute the command `Rscript /home/docker/run_analysis.R` in the environment of `r-base`. This method of "containerizing" software fits effectively into a reproducibility pipeline.

The computational experiment shown in Listing 2 now has a portable environment decoupled from the host machine. The experiment consistently executes within the original environment in which the researcher wrote it, even if another

user runs it on a different device. They can even choose to write a new computational experiment using the same environment by creating a new image that utilizes layers from the previous image. Unlike with a VM, the only storage cost incurred comes from the new code and data. A user running the container can tweak the experiment, changing inputs and algorithms, without fear of losing the original due to Docker images' immutability. Unfortunately, one drawback to containers and VMs is the lack of support for FAIR data management. Storing the computational steps and input data within an image can make it challenging to find the computational experiment without proper metadata. One potential solution is uploading the dataset with a Dockerfile, which is the set of instructions necessary to build their image. Nevertheless, this can be problematic if the Dockerfile relies on specific online resources to install dependencies and versions change, or repositories stop hosting the needed version. For example, this occurs in the situation we presented in Section 2.4. If a Dockerfile installs an R package from CRAN, but CRAN drops support for that package, Docker is no longer capable of recreating the environment based on the instructions in the Dockerfile.

Finally, Docker supports tagging images. *Tagging* is the process of a user assigning an alias to an image. When Docker builds an image, it automatically assigns the image an ID by hashing configuration information from the image using SHA256. This ID (or usually a subset of its characters) is how a user can interact with the image. When a user tags an image, they provide an alias corresponding to the image's ID. With tags, instead of a user running the command `docker run fb1782c4e3b7` they can swap the ID for the tag and run `docker run rocker/tidyverse:3.6.3`. This method is a much more descriptive way of referencing an image. This example also follows the standard tagging style used on Docker Hub, `username/image_name:tag`. So in this scenario, Docker runs rocker's tidyverse image tagged with the version "3.6.3" which, in this case, refers to the version of R within this image. Image maintainers can help their users quickly identify desired images by providing descriptive names and tags. Lastly, if a user is referencing an image or tagging an image if they leave out the "tag" portion, it will default to `:latest`. The `latest` tag simply references the most recent image with the specified name. For example, `docker run r-base` becomes `docker run r-base:latest`. This con-

cept is important, as issuing a command using an image's name without the tag can result in different images if performed later after the maintainer creates a new version.

2.5.4 System Call Tracing

Some systems use dynamic analysis of a computational experiment to identify all necessary dependencies. For example, ReproZip [13] and CDE [21] record the files a computational experiment uses through system call tracing. They can then use that information to package the experiment into a reproducibility-friendly version that includes all necessary inputs, including data and system libraries. ReproZip also allows users to convert this package into a container or virtual machine. CDE is a more lightweight solution that supports only Linux. It grabs all of the necessary files and creates a CDE package that can run on different Linux devices.

2.5.5 Data Provenance

Data provenance is a formal record or history of some digital object. These records contain the transformations applied to some data and the agents involved with those transformations. Consequently, a provenance record validates the history of the data it tracks. In the context of data science, researchers can use fine-grain language-level provenance (LL-Prov) collection tools to provide further validation of their experiments. RDataTracker [32] and noWorkflow [38] are two examples of LL-Prov collection for R and Python, respectively. These tools track data processed by a computational experiment, creating a record of how the inputs change and transform into the results.

In the context of reproducibility, provenance serves two purposes. First, LL-Prov acts as a record that researchers can use to validate an experiment. If a publication claims to use a script containing a particular statistical test with specific parameters, the provenance acts as proof of this claim. Each computational step appears in the provenance allowing researchers to verify they executed as intended. Second, LL-Prov assists with retroactive reproducibility since provenance collection tools record the computational environment. Researchers attempting to reproduce a computational experiment can use this record to re-create a sufficient

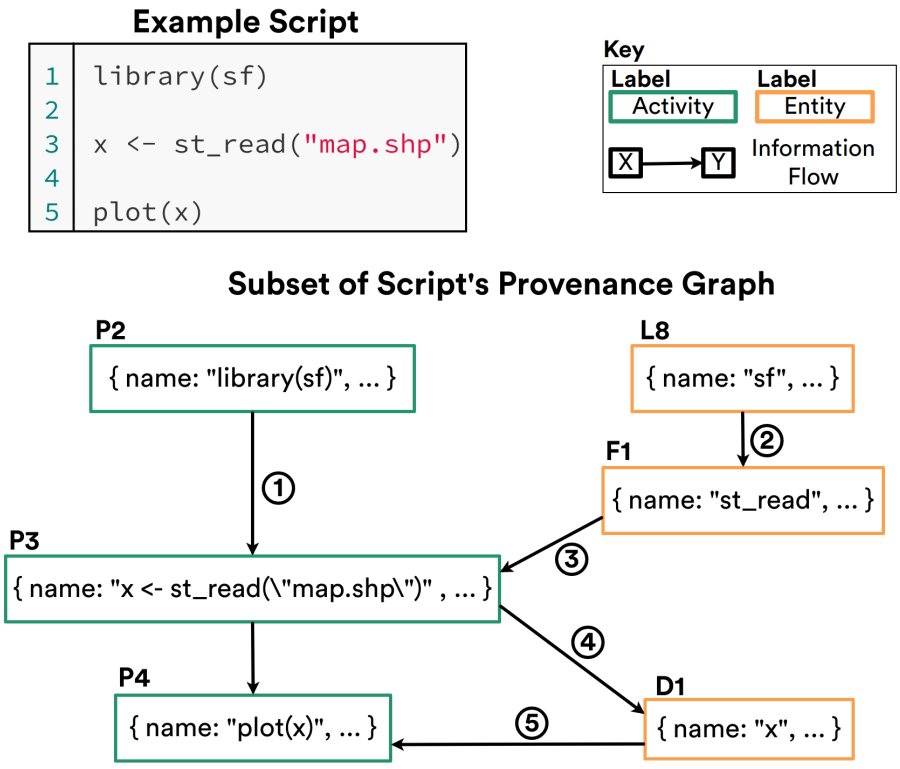

Figure 2.2: This figure shows a simple script and a subset of its resulting language-level provenance graph since the full provenance graph is a 322 line JSON file. Each graph element only shows the name attribute rather than the entire node's metadata to preserve space. This figure also draws arrows "forward" to represent information flow, not "backward" for dependencies.

environment for that experiment. LL-Prov frequently contains records of the software used and their versions, allowing for a more accurate recreation of the original environment. However, this record is also limited since if a repository stops hosting the packages, it can be challenging to reinstall them. Once collected, LL-Prov is a valuable source of information for reproducibility and more.

LL-Prov can be challenging to understand in its raw form. For example, RData-Tracker outputs Extended PROV-JSON [3], a JSON serialization of the PROV data

[3] https://github.com/End-to-end-provenance/ExtendedProvJson

model (PROV-DM) [6] extended for LL-Prov. The PROV-DM, and by extension Extended PROV-JSON, is a model for representing provenance through the following core structures: activities, entities, and agents. Figure 2.2 shows an example script and a subset of its corresponding Extended PROV-JSON graph. Throughout the rest of this section, a letter and number pair in parentheses (XN) refer to a graph label, and numbers in parentheses (N) refer to the graph relationships indicated in the figure. In the context of LL-Prov, *entities* are digital objects such as variables, files, or language packages. For example, elements (L8), (F1), and (D1) are entities. *Activities* are actions that take place over time and use or generate entities. In the context of LL-Prov, activities are often program statements, such as assigning a value to a variable. As an example, (P2), (P3), and (P4) are activities that each represent an evaluated program statement. *Agents* are things that bear responsibility for activities occurring or entities existing. The provenance collection tool records itself and its metadata as an agent in LL-Prov.

This representation inherently produces a directed-acyclic graph (DAG). Activities, entities, and agents are the nodes of the graph. Directed edges between these nodes represent the relationships between the core structures. For example, the edge (1) is a "wasInformedBy" relationship representing control flow in LL-Prov. The edge (2) is a "hadMember" edge that connects a library node (L8) to a function node (F1). This relationship indicates that the function `st_read` is from the package `sf`. These relationships can also indicate when an expression uses some data and generates data, such as (3) and (4). A "used" edge such as (3) indicates the function `st_read` was called in the procedure (P3). A "wasGeneratedBy" edge such as (4) indicates (P3) generated the variable `x`, represented by data node (D1). Similar to (P3) using (F1), (P4) also uses (D1) as indicated in the "used" edge in (5). Unfortunately, when provenance collection tools serialize this model to JSON, all of these relationships are difficult to find and follow, especially if a user is not already familiar with the PROV-DM.

Tools that use provenance can help users interpret the valuable data provenance contains. Visualizations, debuggers, and other exploratory software can use provenance to help users further understand computational experiments. Visualizations are helpful as they visually encode the data lineage, allowing users to view the data relationships and control flow in their scripts. DDGExplorer for RDataTracker [32]

and the built-in visualization for noWorkflow [38] are both appropriate examples. However, DAG visualizations can quickly become difficult to interpret as the number of elements in the graph increases. Alternatively, command-line debugging tools such as provDebugR [10] present the provenance as if the user were using a traditional command-line debugger such as gdb. This post-mortem debugging method is a more familiar and traditional interface, but with extra features such as tracing the lineage of variables and backward execution (or unexecution). Researchers have also created provenance-based tools to compare executions [42], provide a textual summary of an execution [9], and more [4]. Whether or not a user ultimately chooses to collect and use provenance depends on their needs and the availability of tools.

Researchers currently utilize these technologies and practices to assist with reproducibility and FAIR data management. Scientists who publish their computational experiments on archival services help contribute to open science and facilitate reproducibility by enabling future researchers to find their datasets easily. Virtualization through VMs or containers helps distribute the complex computational environment. Researchers combining a container with archival services can provide a high level of reproducibility by ensuring their computational experiment is not only easy to find but likely to execute. Provenance provides another layer of verification by acting as a record of the computational steps the initial researcher executed. Even if a researcher fails to capture their computational environment, provenance can provide extra guarantees and be used to help recreate the lost computational environment. Researchers who wish to create reproducible computational experiments likely need to be familiar with most, if not all, of these practices. Unfortunately, that can be challenging for researchers whose primary goal is to test a scientific hypothesis. Proactive reproducibility platforms can assist researchers with these technologies, but there is much less support for anyone attempting to reproduce a computational experiment retroactively. We created RaaS to address this need.

[4] A full suite of tools built for Extended PROV-JSON is available at github.com/End-to-end-provenance.

Chapter 3

Reproducibility as a Service

Reproducibility-as-a-Service (RaaS) is a retroactive reproducibility tool designed for computational experiments that never recorded their original computational environment. Therefore, we assume that computational experiments uploaded to RaaS cannot necessarily execute [1]. Instead, we use a collection of static analysis tools to determine the computational experiment's requirements. RaaS uses these derived requirements to generate a Dockerfile with instructions to re-create a computational experiment, execute the computational experiment, collect provenance, and generate a report about the whole process. Users can verify the computational experiment results through the report. If they wish to examine the results further, they can directly check the provenance and other files in the Docker image. RaaS is not a FAIR service. It does not provide archival or other data management services. Its only goal is to facilitate the reproducibility of previously published datasets.

In the rest of this chapter, we present our retroactive reproducibility service, RaaS. We start with an example use case based on common retroactive reproducibility problems. We provide a concrete list of these challenges next, along with an overview of RaaS. We then describe how we designed the architecture of RaaS to address these challenges. Finally, we provide the implementation details for the service.

[1] As demonstrated in Chen's evaluation and our evaluation in Chapter 4

3.1 Example Use Case

Sequoyah is an ecologist attempting to reproduce published results from a conference, The Real Ecology Exhibition (TREE). They have a dataset, which is a set of R [49] scripts and corresponding data. Sequoyah tried to run these computational experiments, but the scripts failed due to missing dependencies and incorrect file paths. It would be possible for them to go through each script one at a time to try and fix all of the errors. However, this would be a time-consuming process, and anyone who wanted to work with these computational experiments in the future would have to go through the same process.

Sequoyah chooses to use a retroactive reproducibility service, RaaS, instead. They navigate to the RaaS website and register a new account. With their new account created, they navigate to the build page and upload a dataset, as shown in Figure 3.1. Once they click on submit, RaaS begins to process their dataset. Sequoyah can check its status as shown in Figure 3.2. This page updates each time RaaS moves on to another step in its pipeline (1) until it completes (2). If the dataset encounters an error, RaaS displays it on this page (3). Once RaaS has completed its processing and built the Docker image, Sequoyah can download it.

Sequoyah finds all of the results, reports, and provenance from the scripts executed during the build process upon running the container. They are free to run any additional exploratory analyses inside the running container without worrying about losing the original. They also push the image to a public registry such as Docker Hub, and any other TREE researcher is now free to pull the image and examine the computational experiments themselves. Some researchers examine the provenance and identify a bug in a script where a data frame column's type changed from a factor to a string. They create a new image with the bug fixed and send it back to Sequoyah, who updates their image to this version. Sequoyah sends the Docker image directly to another researcher as a compressed file. This researcher has collected new data to replicate the study and uses the original study's (now bug-free) scripts with their new data to further support the original publication's results.

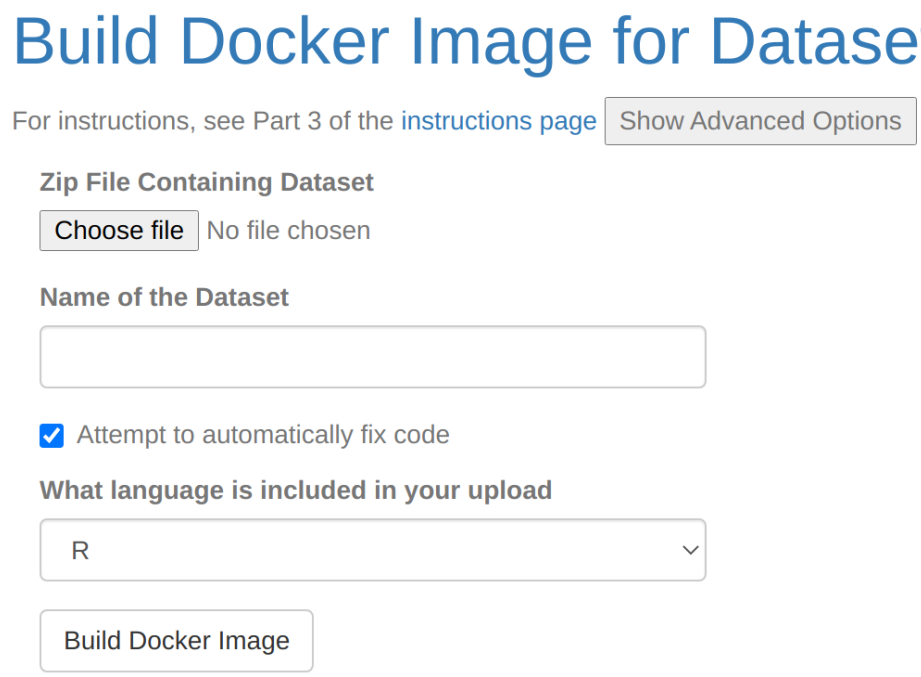

Figure 3.1: RaaS Build Page.

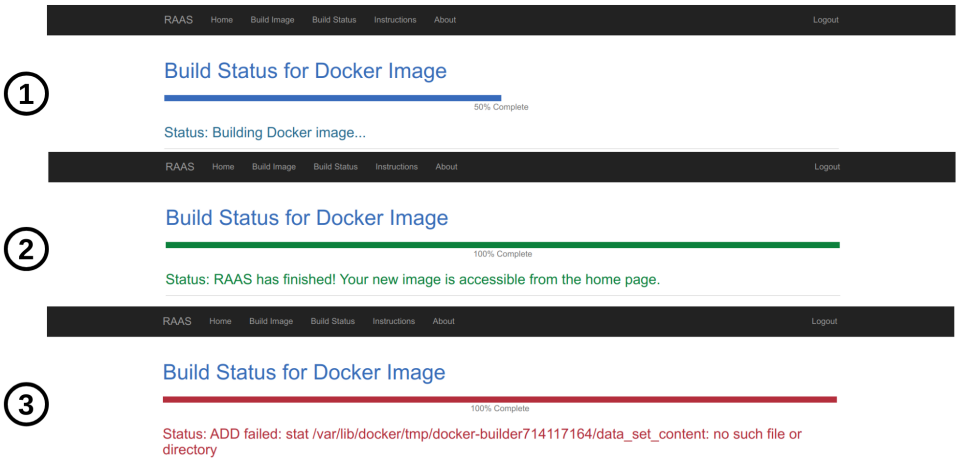

Figure 3.2: RaaS Status Page. The status updates as RaaS executes each step in its processing (1), until it succeeds and displays the finished status (2), or encounters an error and displays the error message (3).

3.2 Overview

Facilitating retroactive reproducibility requires addressing the following challenges.

R.1 Scripts might reference files using paths that no longer exist or were on a different file system.

R.2 Scripts might attempt to change the working directory mid-execution, potentially to a directory that does not exist on a different system.

R.3 Some datasets might have language-dependent challenges. For example, some R scripts use the `source` function on other scripts, and any sourced script should not run independently.

R.4 Datasets might have missing files.

R.5 Scripts might need to execute in a specific order.

R.6 Datasets need access to the original language packages they used in the initial computational environment.

R.7 Language packages often need access to the original system libraries used in the initial computational environment.

R.8 Datasets will need to preserve any re-created computational environment to stay reproducible.

Many of these challenges are language-dependent. However, the sequence of checks and transformations and, in many cases, the actual implementation of large parts of the process are the same, regardless of scripting language. We designed RaaS with a clean separation between language-dependent and language-independent features. This separation allows us to provide retroactive reproducibility with the same service for multiple languages. The language-independent architecture serves as the service's platform and specifies the language-dependent features. Anyone adding a new language has to implement only the language-dependent functions, for which we have identified and created a guide that describes the arguments they take and their return values. While we present RaaS

in the context of retroactive reproducibility for the R language, it currently has varying degrees of support for R [49], Python [60], and Julia [7].

In the following sections, we discuss RaaS's design. In Section 3.3, we present an overview of the service. Regardless of language implementation, RaaS always processes datasets in a predefined way. This section describes each of these steps and presents the data flow between them. In Section 3.4 we describe the implementation that controls the language-independent features and enables data flow in our system. While the features in this section do not directly address any reproducibility challenges, the language-dependent implementation does so by using this section as a platform. In Section 3.5 we describe our solutions for each reproducibility challenge for the R language. We wrote a preprocessing script to address challenges **R.1**, **R.2**, and **R.3** as well as to identify instances of **R.4** (since it cannot be fixed by a tool). This preprocessing can also address **R.5** depending on the dataset's structure. With a collection of static tools we address challenges **R.6** and **R.7**. Finally, we discuss writing the Dockerfile. Once this Dockerfile is complete, RaaS builds a Docker image, addressing challenge **R.8**.

3.3 Architecture

We created RaaS as a web service, so everything starts at the user interface. Figure 3.3 shows the user interface hosted on the web server. Throughout the rest of this section, numbers in parentheses (N) refer to the steps indicated in the workflow. Once a user logs in, they can see a table of all of the datasets they have uploaded, their corresponding reports, an option to download the corresponding Docker image, and in some cases, a link to a Docker Hub URL for the image. All of this information is pulled from a database when the user loads the page (1). If they choose, they can also upload a new dataset as a zip file. A dataset is typically a set of scripts and the data they analyze. At each step in RaaS's computation, it updates a status screen to let the user know what tasks are currently in progress. Regardless of the languages uploaded, users always see the same home page, status page, and options to build a new image, as our web server is entirely language-independent. Additionally, users can choose to include anything else they might find helpful in the dataset, such as READMEs or figures explaining the analysis and data. RaaS

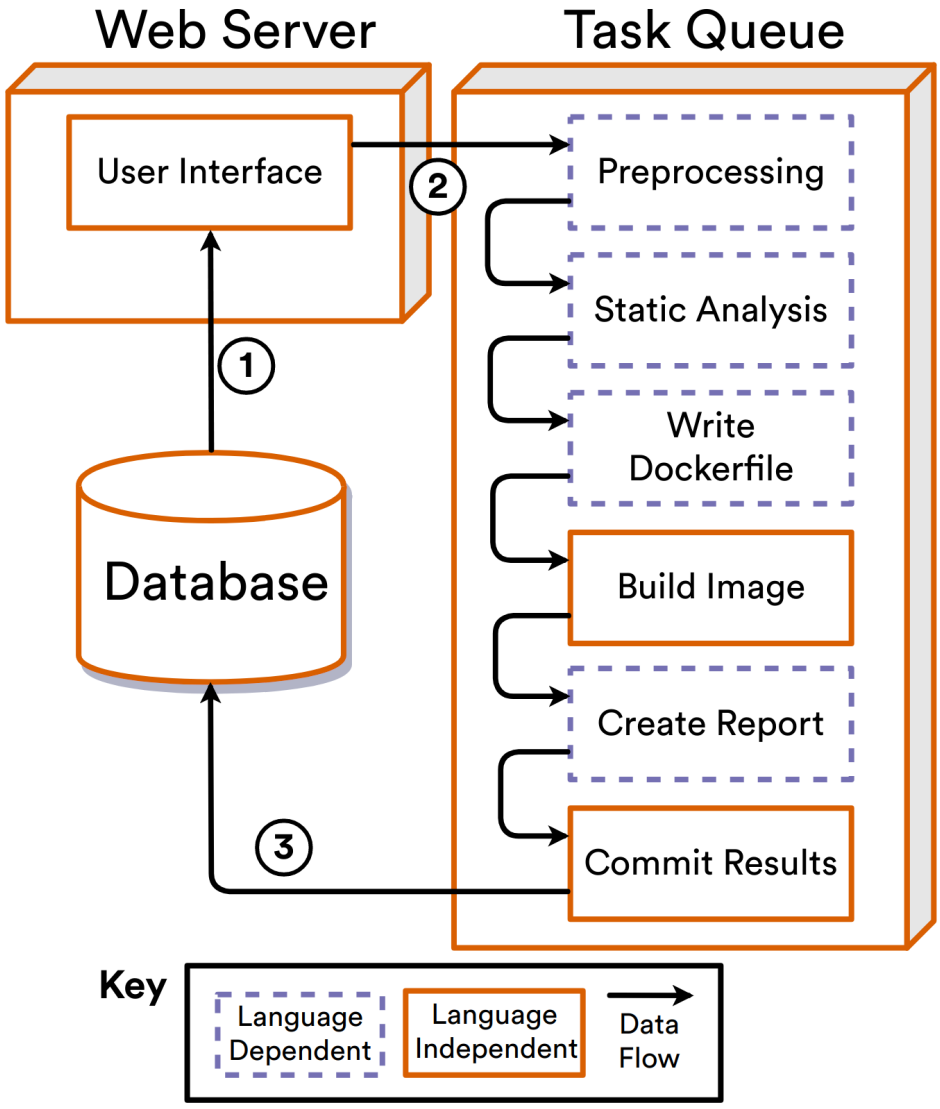

Figure 3.3: Overview of RaaS Architecture displaying the separation between language-dependent and language-independent sections. Each dataset RaaS processes will go through these steps, but the language-dependent sections contain different implementations in the code for each language RaaS supports. In the event of an error, the task ends, and RaaS adds nothing to the database.

ignores these files during processing but includes them in the final Docker image. However, if a user includes a Makefile or has some other similar method for executing the scripts that is not a script in the chosen language, RaaS will not use it. Similarly, RaaS does not currently provide command-line arguments to scripts automatically.

Once RaaS receives a new zip file, it hands the dataset over to a task queue that asynchronously performs the computation necessary to process it. The first step in our pipeline is typically the language-dependent preprocessing (2). In reality, not everyone wants an automated service making changes to their scripts, even to assist with reproducibility. To account for this, users can opt-out of preprocessing on the dataset upload page. The language-dependent code checks if the user opted-out, and if they have not, continues with the preprocessing. Next, RaaS analyzes the scripts for references to files and working directories. If it finds any, it can take the appropriate steps to ensure these lines execute. We commonly remove references to working directory changes and ensure that the script references each file using a relative path starting from the directory containing the script. While fixing pathnames, we might encounter an R `source` statement, which acts as a macro substitution, inserting the sourced script into the one we are processing. We make a note of this fact and then exclude that sourced script from standalone execution. We discuss an example of this scenario in the R language implementation in Section 3.5. RaaS's next step after preprocessing is static analysis, and since the same function handles preprocessing and static analysis, we do not need to provide a standard method by which we route data through our language-independent architecture between these features.

After preprocessing, RaaS uses static analysis to identify the necessary dependencies for a dataset. These dependencies are additional software the dataset needs to execute. This process is responsible for identifying both the language packages required by a dataset and the system libraries required by those language packages. Language packages are the extensions to a language that the language itself is responsible for installing, such as the R `sf` package [44] or the Python `pandas` library [61]. System libraries are software dependencies for the language packages that the language cannot or will not install, such as `gdal` [19]. We install these libraries with the Linux tool `apt-get`, since our Docker images are on Debian-

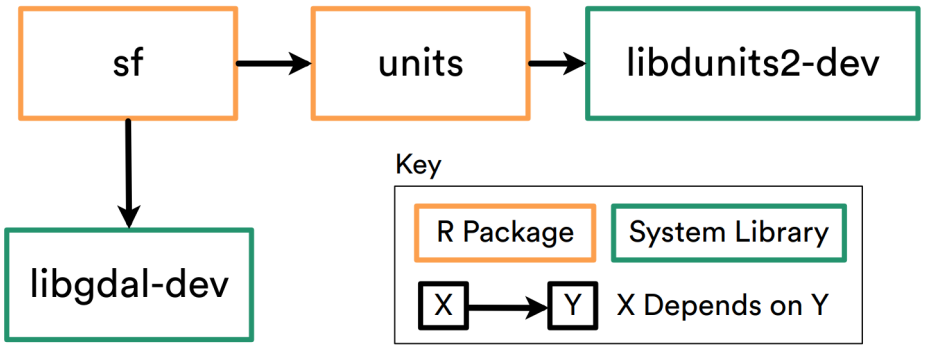

Figure 3.4: A simple example of the dependencies required for an R package. Installing the package `sf` in a clean environment on Ubuntu with only R, such as with the r-base Docker image, will fail due to these missing dependencies.

based systems. Figure 3.4 shows an example set of dependencies RaaS identifies when a script uses the `sf` package. This example shows how one package can fail during installation due to its dependencies. This package fails because it directly relies on `gdal` (`libgdal-dev`). However, even if the user installs `gdal`, the installation fails because `sf` relies on the `units` package [45] and `units` relies on the `udunits` system library (`libdunits2-dev`). RaaS needs to identify and install each of these dependencies, or installation fails. Since RaaS cannot (yet) execute the scripts, we use static analysis to collect this information. This feature is language-dependent as developers tend to write static analyzers in the language they analyze. Once this function completes, it is responsible for returning three essential lists: a list of scripts to ignore during the execution of the computational experiment, a list of the language packages, and a list of the system dependencies. The language-independent code then passes these lists to the Dockerfile creation piece of the RaaS pipeline.

RaaS uses the data collected from preprocessing and static analysis to write a Dockerfile. Our current implementation for writing Dockerfiles is entirely language-dependent, as there are different instructions that RaaS writes, depending on the language. They all start from a base container with the language pre-loaded, potentially with some common language packages. Next, RaaS writes the instructions to

```
{ "Container Information":{
    "System Libraries" :
        [["name","version"],["name2","version2"]],
    "Language Packages" :
        [["name","version"],["name2","version2"]],
    "Language Version" : "X.X.X" },
  "Individual Scripts":{
    "Script1": {
      "Input Files":["name", "name2"],
      "Output Files":["name", "name2"],
      "Warnings":["warning1", "warning2"],
      "Errors":["error1", "error2"] }},
  "Additional Information":{
    "Container Name" : "repo/tag-name"
    "Build Time" : 127.5 }}
```

Listing 3: RaaS report example

instruct Docker to install all system requirements and then language dependencies. It then provides the instructions to copy the dataset into the image and execute all the necessary scripts in it. This step has to return only the directory of the Dockerfile, although mainly as a precaution since this directory is typically the same as the one RaaS creates for the dataset in the beginning.

Once RaaS completes the Dockerfile, it passes control back to the language-independent code to build the Docker image. All Docker needs to build an image is a correctly specified Dockerfile. Once that Dockerfile exists, RaaS sends it to the Docker daemon, which constructs the dataset's image regardless of its language.

With the Docker image completed, RaaS generates a report about the process. We designed a standardized language-independent report structure in JSON to provide information about the build to users. Listing 3 shows the entire report structure. The report includes a description of the computational environment by listing the language and system packages present in the image. It also contains data about the scripts that RaaS executed, their inputs and results, and any warnings and errors generated. Lastly, the report has a section for any miscellaneous information that,

at a minimum, contains the build time and tag for the image. An image *tag* is a string that references a specific Docker image. The report is the last entry necessary to fully commit the results of building the dataset to the RaaS database (3).

The last step RaaS takes when processing a dataset is committing the results to a database and potentially pushing the image to Docker Hub. If a user wishes, they can choose to instruct RaaS to push their newly created image to Docker Hub. However, users should be careful with this process as they might feel tempted to keep the image on Docker Hub indefinitely. Docker Hub is not an archival site, and soon its default behavior for free accounts is to delete unused images [37]. Instead, Docker Hub can be a method by which users can download or distribute the image. Regardless of whether they use Docker Hub, RaaS commits this dataset to its database (3). Each entry it adds to the database has a unique ID, a user-chosen name, a URL to Docker Hub if the user chooses to upload the image, a timestamp of when the image completed building, and the dataset report. These entries are the data that the web server pulls to display users' datasets on the RaaS home page. At the moment, RaaS will keep images indefinitely until we manually delete them. However, future versions will automatically delete images after a pre-determined amount of time to save space since RaaS is not an archival service.

3.4 Language-Independent Implementation

RaaS's language-independent architecture forms its foundation. These language-independent components are the interface to the user, the transfer of selected data between language-dependent pieces, and the process of building Docker images - all the sections bordered by solid orange in Figure 3.3. RaaS is a web service, so we need to start from a web framework that can support multiple users, manage each user's data, and still compute its reproducibility tasks, while being responsive to users. We use the Flask framework [53], which helps facilitate these web service activities.

Flask is the core of RaaS' language-independent architecture. It is a web framework written in Python that allows us to quickly iterate on a design as many standard features of a web service, such as an object-relational mapper (ORM), user accounts, and an asynchronous task queue, already exist as extensions to its frame-

work. We also use Python for all other parts of RaaS's language-independent implementation, so Flask interfaces neatly with the other code. Flask and the software we use with it, including SQLalchemy [5], Celery [56], and Redis [50], are the foundation upon which we build our service. They allow us to perform the engineering necessary for retroactive reproducibility, even though they are not specialized software for reproducibility.

We use SQLAlchemy with Flask to interact with a SQLite [24] database that stores user and dataset information. RaaS stores a new user's information with a unique ID in the database. Whenever a user uploads a new dataset, RaaS stores information about the final Docker image in a "dataset" table with a foreign key to the user who uploaded it. The link to the original user allows us to filter datasets so that each user sees only the datasets they uploaded. They can continue to browse this information and download previous datasets even while processing a new computational experiment.

RaaS uses the asynchronous task package Celery to stay available to users while it processes datasets. Celery is an asynchronous, distributed task queue. We use Celery, because the computation that RaaS performs on datasets can be time-intensive. Celery allows us to perform those computations in parallel or asynchronously with the web service itself. The code that processes the uploaded datasets executes as an asynchronous Celery task. When a user uploads a dataset, Flask uses the key-value store Redis as a message broker to initiate a Celery task to process the dataset. This Celery worker is the process that then executes the function that ultimately builds the user's new Docker image. After RaaS builds the image, it will tag the image with the user's account name prepended to a user-provided name for the image. RaaS does not tag a version to the image since RaaS will not change any previously created image. Nevertheless, if a user wishes to update the image, they can manually add version tags themselves. This processing, including preprocessing, analysis, and building, happens independently from Flask. However, users can view their dataset's status from the RaaS website, Figure 3.2. Once Celery completes the build process, it commits all necessary information to the database, notifies the user, and exits.

A user can choose to download a newly built image or push it to Docker Hub. We provide two ways for a user to obtain their image once RaaS creates it. They

can download it directly or pull it from Docker Hub. If they choose to download, RaaS compresses the image in a tar.gz and makes it available for download. This compressed version of the image typically requires much less space than the original and does not need to be stored in any special registry. The user could upload it to any digital archival service that accepts general digital artifacts. However, they do not have to worry about losing the image's functionality, because they can run the image straight from the tar.gz file using the `docker load` command. Alternatively, the user can instruct RaaS to push the image to Docker Hub. This process involves Docker pushing each layer individually to Docker's repository. If they choose this option, any user with the image's name can get it by simply issuing the `docker pull` command.

We designed the RaaS code that processes datasets such that we can reuse pieces that are not language-dependent. We identified all the processing steps that must occur for retroactive reproducibility, independent of language, in a computational experiment. We always have to perform some initial static analysis, write a Dockerfile, build a docker image, generate a report, commit and push the results, and clean up. Currently, the only language-dependent portions of RaaS are the static analysis, writing the Dockerfile, and generating the report. To avoid duplicating code and encourage easier adoption for RaaS, we created a new Python class called the `language_interface`. This class contains abstract methods for the language-dependent portions of RaaS, while the language-independent methods are fully defined. Any developer who adds a new language to RaaS has to create a new class that inherits from `language_interface` and then implements each of the abstract methods. The Flask server handles receiving the dataset, calling the language object's methods, cleaning up, and allowing the user to download the image.

3.5 Language-Dependent Implementation

Implementing the language-dependent architecture might vary between languages, but the high-level actions are the same. This section addresses the challenges we defined at the beginning of Section 3.2. We correct lines of code that reference incorrect file locations or directories, challenges **R.1**, **R.2**, and **R.3**. We also identify

Original Script

```r
# Set working directory
setwd("/home/seq/data_analysis")

data.df <-
 ↪  read.csv("/home/seq/data_analysis/data/data.csv")

# Contains print_data function definition
source("scripts/function.R")

print_data(data.df)
```

Preprocessed Script

```r
# Set working directory

data.df <- read.csv("../data/data.csv")

# Contains print_data function definition
print_data <- function(to.print){
    print("Your data is: ")
    print(to.print)
}

print_data(data.df)
```

Listing 4: An R script with reproducibility problems before and after RaaS preprocessing.

potential missing files, challenge **R.4**. In some cases, scripts might need to execute in a certain order. When they enforce this order with `source` statements, we solve this issue by inlining sourced scripts, challenge **R.5**. We determine a dataset's dependencies, challenges **R.6** and **R.7**. Finally, by constructing the container, we facilitate the future reproducibility of the computational experiment, challenge **R.8**. In this section, we discuss how we implemented these features for the R language.

Script Preprocessing RaaS can preprocess scripts to increase their likelihood of executing without error by modifying lines of code that are known to be problematic. Listing 4 shows an example R script that is a subset of Listing 1 from Section 2.4. The top script has several reproducibility problems, and RaaS has preprocessed the bottom one. Some of the largest sources of errors in R scripts come from incorrectly referenced files or directories. It is common for R users to reference a file by its full path from the root directory, as seen in line 4 of the original script in Listing 4. Unfortunately, this means that as soon as they move that computational experiment at all, even within the same file system, it is likely that the path no longer points to the file or directory they intended.

We wrote a Python script that preprocesses R code and converts it to a more reproducibility-friendly style while also keeping a backup of the original. This style assumes that the working directory is always the directory where the script resides. Any reference to a file in the script should always be relative to the script itself. Our preprocessing script uses regular expressions to search for code that references commands such as reading a file or setting a working directory. If it finds a reference to a working directory change, it simply deletes the line (see the first two lines of each script in Listing 4). When we write the instructions to execute the computational experiment, we set the working directory to the script location, so any change at runtime will likely fail.

In addition to working directories, our preprocessor searches for instances of a script reading a file. First, we search for indicators that the script is reading or writing a file by matching on patterns such as `file=` keyword argument, a write file `w+` flag, or even text within quotes. Next, we search for commonly used functions for reading files such as `read`, `load`, `fromJSON`, `import`, or `scan`. If our preprocessor finds a reference to a file, it searches for that file in the dataset directory. If it finds the relevant file, it determines the relative path between the R script and the file and replaces the code's file path with the relative path. Listing 4 line 4 in the original script and line 3 in the preprocessed script shows how reading a file is changed by the preprocessor. If it cannot find the file, it simply leaves the line unchanged. We use this feature to identify instances when we cannot find a file, indicating a potential missing file, **R.4**.

RaaS contains an important language-dependent feature for preprocessing R

scripts. R has a commonly-used feature that we handled explicitly in our preprocessing, the `source` function. This function allows a user to execute another script from within another. Consider line 7 of the original script in Listing 4. This script will act as if `function.R` has been copied into this script at line 7. Since the sourced file could exist in a different location from the calling script, any files referenced in the sourced script can have different relative paths. Since the scope between the two scripts is the same, we choose to replace calls to `source` with the contents of the file being sourced, which is simply the function's behavior. Lines 6 through 9 in the preprocessed script in Listing 4 show that we copied the function defined in `function.R` into the script, replacing the `source` call. We do this preprocessing first so that any other changes we make to the script include the sourced scripts' code. All of this preprocessing code addresses the challenges we identified in **R.1**, **R.2**, and **R.3**.

Static Analysis Static analysis allows RaaS to collect the necessary information about a script to generate the appropriate computational environment. This information includes the language packages used and the system dependencies they have. For our initial analysis, we use the R package `CodeDepends` [31]. This tool provides us with the names of any R packages that a script loads. It also searches the script for any potential errors or warnings that can occur at runtime. This process is similar to what many development environments do to code that a user has written but not executed. Unfortunately, `CodeDepends` cannot provide us with the system dependencies that these R packages might require. These are the dependencies that we install through `apt-get` rather than R itself. To determine these dependencies, we use the r-hub sysreqs database. This database is an open-source mapping of R packages to system dependencies. Since there is no official documentation that ships with packages identifying their dependencies, this method is currently the most effective for gathering these requirements. To use it, we pass the list of R packages `CodeDepends` found to their public API, which returns a list of system libraries that we can install with the `apt-get` command-line tool. With all this information, we can usually install all of the necessary dependencies in our Docker environment addressing challenges **R.6** and **R.7**.

```
① FROM rocker/tidyverse:latest

② RUN sudo apt-get install <system libraries>

③ COPY install__packages.R /home/rstudio/
  RUN Rscript /home/rstudio/install__packages.R

④ COPY data_set_content /home/rstudio/datasets/
  COPY get_dataset_provenance.R /home/rstudio/datasets/
  COPY create_report.R /home/rstudio/datasets/

⑤ RUN Rscript /home/rstudio/datasets/get_dataset_provenance.R

⑥ RUN Rscript /home/rstudio/datasets/create_report.R
```

Listing 5: Example of a Dockerfile that RaaS generates. For readability, this is a simple example rather than a copy of a Dockerfile RaaS has produced.

Dockerfile Creation After analyzing a dataset, RaaS has the necessary information to construct a Dockerfile. The Dockerfile must contain all of the correct instructions for Docker to install the required system dependencies, install the necessary R packages, execute the computational experiment, and collect provenance as shown in Listing 5. If one of these steps fails, the build process could fail to produce an image, or the resulting image could fail when it tries to run the computational experiment. Throughout the rest of this section on the Dockerfile creation, numbers in parentheses (N) refer to the steps indicated in the Dockerfile in Listing 5,

The first step in the Dockerfile is to instruct Docker to base our new image off of the rocker/tidyverse:latest[2] image (1). Docker builds images in layers, so rather than start from scratch, it is common to use an already constructed image and add more layers. We chose to start from this image for two reasons: its interface and pre-installed libraries. The interface is likely familiar to R users since this image hosts RStudio Server. Researchers who run one of these images can use the familiar RStudio interface by navigating to a provided local port in their web browser. They can do anything they would normally do in RStudio, including run and edit R

[2]hub.docker.com/r/rocker/tidyverse

scripts and use the Linux bash terminal. Additionally, the rocker/tidyverse image has many commonly used R packages pre-installed. Any package pre-installed avoids RaaS specific installation, saving time during the potentially lengthy build process. There is also a slight chance that our static analysis could miss identifying a package but not cause an issue since the package is pre-installed. Finally, we currently use the :latest tag when choosing this image because it is impossible for us to know which version of R a user might want. Because we use this tag, a Dockerfile RaaS creates today will be no different from a Dockerfile it writes for the same dataset in three years. The only line of the Dockerfile that might change is the system libraries installed since we query an online database for that information. If a package adds or loses a system dependency, that change will appear in the libraries the Dockerfile installs.

Once we have our base image, we write the instructions to install the computational experiment's system dependencies (2). We must install system libraries first, as the R packages we install might depend on their existence. Without specific system libraries, installing some R packages fails. For example, the R package `rgdal` [8], used by the popular geospatial packages `sp` [46] and `sf` [44], requires the `gdal` system library [19]. Installing `rgdal`, `sf`, or `sp` through R does not install `gdal` on the system. We also allow the users to specify system libraries they wish to install in case our static analysis might not identify them or that they want for any other reasons. We also install these libraries before the statically identified R packages. We install all statically identified libraries with one call to `apt-get` and all user-inputted libraries with another call to `apt-get`. We separate these calls because it is easier to determine where a failed package came from in case of a failed build. Additionally, if the second call to `apt-get` fails and the user reruns it with a fix, the previous `apt-get` layer is cached, saving build time. We have these installations grouped rather than installing each library on its own because each line of a Dockerfile becomes one of the limited number of layers an image can have [3].

After the system libraries' installation instructions, we can write the necessary

[3]There was previously a 42 layer limit. Currently, there appears to be a 127 layer limit, which we reached when testing before optimizing the number of layers. For more information, see github.com/docker/docker.github.io/issues/8230

R packages' installation instructions. Like the system libraries, to ensure we do not reach the image layer limit, we need to keep the number of lines in a Dockerfile to a fixed number rather than scale by the number of needed packages. Therefore, before we write the Dockerfile, we write an R script, `install__packages.R`, that installs all of the identified R packages the computational experiment uses. This script has two unique features for efficiency and to increase the likelihood of success. First, it installs any given R package only if that package is not present in the current environment. Consequently, the build time potentially shortens as we can take advantage of any pre-installed packages by not reinstalling them. Second, the script also tries to install the package first from CRAN [25] using the standard `install.packages` function, and if that does not work, it tries to install from BioConductor [26]. We chose to install from either given that support for packages on these repositories can be somewhat volatile, and in most cases, if a package is not on CRAN, it is likely on BioConductor. While packages such as `renv` [59] exist to help provide proactive reproducibility when installing packages, we chose not to use it in this case because it felt redundant given the reproducible nature of Docker.

This script is then copied into the container and executed, a total of two lines in the Dockerfile every time, ensuring we do not reach the layer limit (3). If a user needs to install packages from other sources such as GitHub, we allow them to provide R code with these instructions. We place user-input installation instructions before the `install_instructions.R` script in case anything they choose to install manually is a dependency of a package we identify statically.

Once we write all of the installation instructions for the computational experiment's dependencies in the Dockerfile, we can provide instructions to execute the experiment. In preparation for executing the computational experiment, we write instructions to copy the dataset, a `get_dataset_provenance.R` script, and a `create_report.R` script into the image (4). The first script to execute, `get_dataset_provenance.R` (5) finds all R scripts in a dataset and executes them while collecting provenance using the R package rdtLite[33]. Since we use this script, we can ensure we do not have a Dockerfile where its number of lines scales with the number of R scripts in a directory. The `create_report.R` script uses the provenance to write a report in JSON format that RaaS stores in its

database.

The `get_dataset_provenance.R` script we wrote to execute the computational experiment and collect provenance has unique responsibilities. This script ensures that it executes each necessary R script present in the dataset in an independent environment. Identifying "necessary" scripts is not as simple as it might seem. There are two sets of exceptions specifically, causing us to implement a feature to ignore a provided list of scripts. When we preprocess scripts, we save backups of each file we change. These backups are stored in the container, but could cause problems if executed. Therefore we add all backups of the original scripts to our "do not execute" list. The other exception is R-specific, sourced scripts, scripts called by the `source()` function. While sourced scripts get independently preprocessed, since we know they execute within the script that calls them, we ignore the sourced script itself. Besides identifying "necessary" scripts, the `get_dataset_provenance.R` script ensures each script executes in an independent environment. Since we use an R package to execute the scripts, rdtLite's `prov.run` function specifically, we could write that function call directly in `get_dataset_provenance.R`. Unfortunately, this could cause issues, as our script and all the scripts in the dataset share the same global environment. To avoid conflicts between variables and options, we instead use R's `system` function, which executes a shell command passed to it. For each script, we write a command that starts a new R instance, loads rdtLite, and executes the script while collecting provenance. This method prevents our script's environment from sharing a dataset script's environment and ensures each script in the dataset has a unique environment.

While it might not be entirely necessary to execute the datasets as part of the build process, this method has multiple benefits. Most importantly, if we did not execute the computational experiment, we would have no way of knowing whether we had created an adequate computational environment. We could still execute the experiment after building the image; however, we would lose any of the provenance generated. There is a wealth of information about the computational experiment stored in the provenance, and we install packages that help users parse the raw PROV-JSON: provParseR and provDebugR. We also use these packages to help write a report that details information about the computational experiment.

The last instruction in the Dockerfile generates a report for the dataset (6). We created a standard report structure shown in Listing 3. We populate this template with information from the system and provenance, using the `create_report.R` script. The script collects system libraries and their versions from the apt tool. It records all of the R packages installed using R's `installed.packages` function. It also gets the R version using the `R.Version` function. All of the data for the report under "Individual Scripts" the `create_report.R` script collects from provenance. There will be a PROV-JSON file for each R script stored on the Docker image when this command executes. We use the R provenance parsing package provParseR to read each provenance file and collect the package name, input files, output files, warnings, and errors. The script then stores the report in a pre-determined location in the image. The last portion of the report, "Additional Information," is populated later. The last line of the Dockerfile instructs Docker to execute this script. Anyone who uses this image receives a computational environment where the last thing that occurred was `create_report.R` writing the report to the file system.

Build Image Once RaaS writes the Dockerfile, it will pass control back to the language-independent architecture to build the Docker image. Given that building an image always consists of calling the build process on a written Dockerfile, there is no need for any language-dependent code, and we have addressed challenge **R.8**. However, there is one more language-independent function that must execute.

Create Report The last language-dependent function to execute retrieves and finalizes the dataset's report. The vast majority of the report information RaaS writes inside of the image. However, to provide this information to the user, RaaS now needs to retrieve that report and add the final section. This function runs the container and copies out the report before loading it into a Python dictionary. Now it can add the last two pieces, the name of the image tag, and the build time. For the last time, the language-dependent architecture switches control to the language-independent code. The process is nearly complete; all that is left is for the dataset information to be committed to the database.

3.6 containR

RaaS is a successor to Chen [12]'s containR. ContainR contained many of the core fundamentals of a retroactive reproducibility service. Chen identified the need to facilitate reproducibility for previously published computational experiments, and the result was containR. The difference from RaaS is that containR still contained aspects of a proactive reproducibility tool. Rather than focus on creating an environment around a dataset and then executing it, containR preprocessed datasets in an attempt to prevent them from failing on a host machine. If successful, containR collected provenance on the dataset and used that to create an environment in Docker where it deposited the datasets and provenance. While this approach could often work, there are areas in its design that needed revision from a reproducibility standpoint.

One of the most critical pieces was the disconnect between the environment and the dataset. ContainR never actually executed the dataset within the container. Figure 3.5 shows a comparison between RaaS and containR displaying which processes take place in which environment. Since previous reproducibility tools assumed a working dataset, it makes sense for an initial retroactive reproducibility tool to try and make the scripts work first and then apply standard proactive reproducibility techniques such as provenance. As we discussed in Chapter 2, sometimes just changing environments is enough to causes errors in a script. For example, containR ran on Ubuntu, and rocker/tidyverse is Debian. While Ubuntu is a fork of Debian, they have different release cycles, which ultimately could lead to mismatches with system libraries between the two. Additionally, containR always pulled the most recent version of rocker/tidyverse. Without specifying a version, there was no guarantee that the R version on the host machine was the same as that in the image. Researchers have previously discussed software versions as a weak point for reproducibility [18].

The other aspect of this environment difference is that the dataset's provenance did not match its environment. While having the provenance was beneficial since it still validates the execution, the disconnect with the Docker environment leaves it with less reusability potential. While the host and the Docker image's environment would likely still be quite similar, we encountered firsthand the effects of software

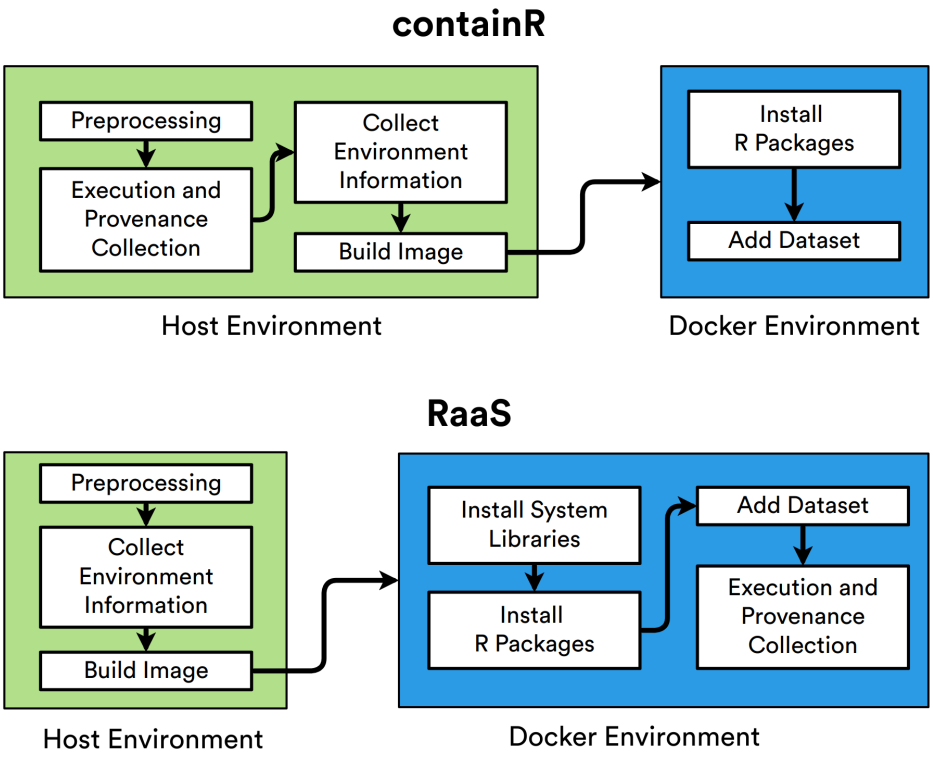

Figure 3.5: Comparison of which processes happen in which environments between containR and RaaS.

rot during the time we created RaaS, so understand its surprisingly quick effects. Given this tendency for software rot, coupling the computational experiment with its computational environment protects against this degradation. While we also pull the latest rocker/tidyverse, RaaS only ever runs the dataset in that version and within the recreated environment. This choice to use the latest version of the rocker/tidyverse image might initially seem at odds with most reproducibility practices; however, it is the best choice within the context of retroactive reproducibility.

Most proactive reproducibility tools attempt to preserve the computational environment where a computational experiment executes. This design leads to most tools trying to maintain whatever version of the software the computational experiment used rather than installing the latest because a proactive reproducibility tool attempts to preserve a computational experiment's result rather than recreate it.

Therefore they are trying to maintain as much of the environment as possible, down to the versions. This approach was something containR did as well. It installed the exact R package versions from the provenance into the container. This choice is appropriate from the proactive angle from which Chen approached containR. If it succeeded on the host, it makes sense to want to match that environment in the Docker image as much as possible. However, containR was not working with the original computational environment the way other proactive reproducibility tools usually do. Instead, containR had re-created an environment on its host machine using the latest versions in the first place. So when it parsed the provenance for specific package versions, it was the same as if it had just installed the packages straight from the repository.

Since we assume we do not have provenance for the datasets, we cannot know what software versions they originally used. This knowledge gap is an unfortunate reality of retroactive reproducibility and leads to the decision to install the latest version of R and its packages. We do not have any other options. There is an argument for using the Dataverse metadata to determine which version of the packages existed at the time of publication. Still, there is no guarantee that the authors wrote the computational experiment close to the time of publication. They can update a dataset at any time, meaning that the packages might be newer than the publication date. We concluded that if we had to guess anyway, we should use the versions that should, in theory, be the most stable. So while both RaaS and containR install the most up-to-date packages and version of R, RaaS installs new software and packages only inside the Docker image and never on a system with an unknown environment. Since we removed host environment execution, we needed a new way to collect the necessary package installation data.

Rather than rely on provenance to install the correct dependencies in our Docker images, RaaS relies on static analysis. This method solves the problem of differences in the host and Docker environments and allows us to execute the dataset only once. In an earlier iteration of RaaS, we ran each dataset twice, once as containR did on the host and again on the Docker image as part of the build process. However, this process was lengthy and inefficient, and there was always the chance it worked on the host but not the image. We used this iteration only briefly before we switched to static analysis to address this performance problem and prevent

potential mismatched environment problems.

Our decision to take RaaS in a multilingual direction led to our most significant architectural redesign. ContainR's design guided us towards our eventual multilingual design. We examined the structure of containR and identified pieces that we could abstract to all of the language-independent features discussed in Section 3.4. After another analysis of our current design, we identified more pieces that we can abstract to be language-independent without sacrificing simplicity; we discuss this change in Section 5.2.

3.7 Reproduciblity Design Influences

The first problem we encountered when we tried to build on containR was software rot. In the relatively short time since Chen finished work on containR, critical pieces of the project were failing. The R provenance collection tool provR became deprecated, the Dataverse API changed, and we could not install dependencies at their designated versions. These were obvious changes we would have to make. We switched the provenance collection tool to rdtLite and fixed our API integration and dependencies. However, throughout the project, these types of problems ultimately helped us shape how we created RaaS. Two of the most significant manifestations of this are in writing R Dockerfiles and how we distribute RaaS.

While we were creating this project, CRAN removed a package used by our R static analysis tool, `CodeDepends`. This package, `graph` [20], must now be installed from Bioconductor[4]. We encountered this type of problem during various dataset tests a few times since we were installing packages only from CRAN at this point. Up to this point, we addressed this problem by giving users the ability to input installation code for packages. Once we realized that one of the dependencies from a critical portion of our service faced this issue, we knew it was time to change. This event cemented a decision we had previously considered; we need to install packages from more places than CRAN *by default*, specifically Bioconductor. These problems also led us to another critical design choice.

We realized RaaS needed to simplify its installation process after several dependency issues, including the previously mentioned `graph` problem and conflict-

[4] As of April 2021, CRAN states this here: https://CRAN.R-project.org/package=graph

ing Python packages. We chose to take our own advice and turn RaaS into a Docker service. The process for facilitating the reproducibility of RaaS was similar to how RaaS makes datasets reproducible. We identified all of the dependencies required and wrote a Dockerfile (manually) to create the necessary environment. This image is the `raas-env` image. Things are slightly more complicated after that, as RaaS is a service composed of three containers, not one. RaaS relies on two running instances of `raas-env`, each with the source code copied into it. In one of these instances, `wa01`, is the Flask web server. The other `raas-env` instance, `cl01`, runs the Celery worker. The final container is a Redis instance. Developers commonly use Flask, Redis, and Celery with Docker, so this integration did not take long to achieve. We use docker-compose, a tool for controlling multiple containers, to manage this configuration. When we installed RaaS on our lab server and Compute Canada virtual machines, we simply pulled our `raas-env` image from Docker Hub and issued a docker-compose command to start the service. We could have built the `raas-env` image from scratch, as we include the Dockerfile with the project, but it was faster to pull it. At the moment, RaaS uses only these three containers, but we can provide opportunities for scaling by dynamically adding more Celery containers if necessary.

Chapter 4

Evaluation

Although we built RaaS to be language-agnostic, we focus on R scripts hosted in datasets on Harvard's Dataverse, which has a rich repository of R scripts with a public API that Chen [12] previously examined. While any institution can host an instance of Dataverse, and many do, for the sake of brevity in this paper, when we refer to Dataverse, we are referring to the Dataverse hosted by Harvard University[1]. Chen discovered that the vast majority of the computational experiments on Dataverse contained errors (85.6%), motivating the need for retroactive reproducibility. Since this project builds off Chen's ContainR, we decided that replicating his original experiments with Dataverse could help us determine if the trends and issues he discovered have changed since 2018. The second part of our evaluation consists of running these same datasets after RaaS processes them. We examine if scripts processed by a retroactive-oriented service follow the same trends and error rates as executing scripts as-is. Then we look at how errors changed, if at all, from running scripts without RaaS versus with RaaS[2].

4.1 Research Questions

We compiled a list of questions that help determine the need for a retroactive reproducibility service such as RaaS and its efficacy.

[1] dataverse.harvard.edu

[2] Our data analysis is at github.com/jwons/raas-eval

Q.1 How reproducible are R scripts published on Dataverse?

Q.2 How has reproducibility changed since 2018?

Q.3 What are the barriers to reproducibility?

Q.4 How effectively does RaaS deal with these errors?

Q.5 Of the remaining errors, what fraction could have been remedied if users uploaded their datasets to Dataverse via RaaS instead of simply via direct upload?

4.2 Methodology

We start by identifying the target for our evaluation. We chose Dataverse as it is an actively supported and used repository with a rich set of data. Dataverse was also the target of prior work, Chen's containR. During our evaluation, we found areas for improvements in RaaS, but we actively chose not to make any changes to preserve the validity of these results. We enacted a code freeze on the RaaS codebase[3] starting January 26, 2021, the only exception being any changes needed for the experimental setup.

While some researchers might have used proactive reproducibility technologies such as `renv` [59] or `make` [52], we actively chose not to look for or use these artifacts for any part of our evaluation. While these technologies could supplement RaaS effectively in the future, we first need to establish how effective a retroactive reproducibility system can be on its own. Since we defined retroactive reproducibility to mean a missing computational environment, we adhere to a strict interpretation of this term and assume there are no proactive digital artifacts. In reality, we do not believe these artifacts would have had a significant effect on the results anyway. For example, a researcher who used `renv` can snapshot their package environment in a `renv.lock` file, and there are only five `renv.lock` files on Dataverse at the time of this writing.

On February 13, 2021, we scraped the DOIs of 11819 R scripts from Dataverse. Each script is a discrete test case in our evaluation. Still, we cannot download each

[3] Our codebase, including experimental setup, is at github.com/jwons/raas

script and run it independently, because scripts rely on input data unique to their parent dataset. By examining each script's metadata, we compiled a list of DOIs for each dataset that contained these scripts, leading us to a final list of 2900 DOIs that map to datasets with R scripts[4]. It is essential to compile this list once and always reuse it since these repositories are constantly changing. We use this list of datasets as the input to all other data collection.

First, we evaluated the current state of reproducibility in Dataverse. For each dataset on our list, we downloaded it and ran all the scripts in it. We ensured that each dataset executed in a pristine computational environment. This environment represents the bare minimum necessary to execute scripts in the language. Additionally, each script ran in a unique language session or a separate instance of the language runtime. This method prevented the resulting environment from one dataset affecting the next and the variables from one script from affecting the next. We recorded the status of each script, whether it succeeded or the error if it failed. We imposed a one-hour time limit on all datasets. The activities included in this time are: copying the dataset into a controlled environment, executing each script, and saving the results. We discarded any dataset that did not accomplish all those within the allotted time frame. We also recorded how long it took for each dataset to perform these tasks.

Next, we repeated the process using our reproducibility framework, RaaS. We downloaded each dataset again, and RaaS created a computational environment for each one. RaaS then executed the scripts in a dataset and generated a report for it. Once again, we imposed a one-hour time limit. In this case, all of RaaS's preprocessing, computational environment creation, copying the dataset into the environment, script execution, and report generation counted towards the limit. We discarded any dataset that did not execute within this timeframe.

Once we collected our data, we analyzed the results. For the execution without RaaS, we examined the error rate of *scripts* and *datasets* across the whole repository. We also found the fraction of datasets that could execute all their scripts without errors. Next, we analyzed the content of the error messages to categorize errors. To determine the reproducibility trend over time, we then calculated

[4]These numbers have likely increased by now since Dataverse is a live repository with actively contributing researchers.

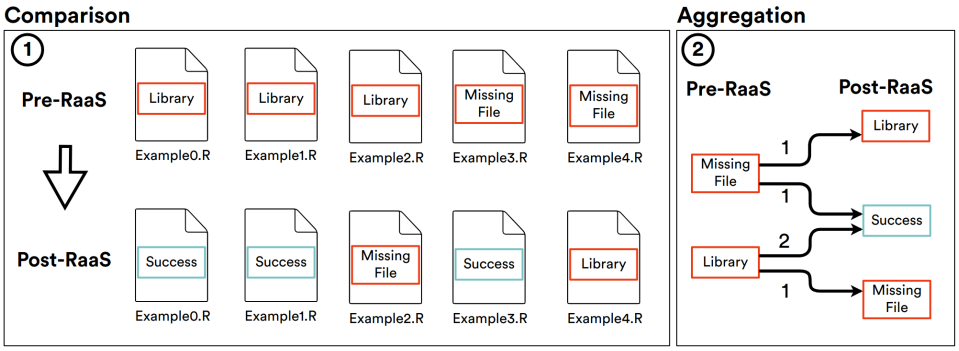

Figure 4.1: Simple example of our process to calculate the change in errors from the pre-RaaS execution to the post-RaaS execution. This first stage consists of examining the status of a script when it executed without RaaS and then comparing it with how it executed with RaaS. Then we aggregate the scripts based on their recorded status. For example, two scripts threw library errors pre-RaaS but were successful post-RaaS.

the error rates of scripts per year. Finally, we looked for trends in error rates of scripts across subjects to determine if certain subjects have better reproducibility practices. These results for the overall reproducibility of Dataverse replicated the evaluation Chen conducted in 2018, although our methodology's implementation differs. This replication simultaneously serves as a baseline for our reproducibility framework while also allowing us to determine if the trends Chen discovered in 2018 persist.

Next, we evaluated the efficacy of our reproducibility framework. We analyzed the RaaS reports generated for each dataset to calculate the error rate for both scripts and datasets after RaaS processing. Once again, we categorized errors. Now that we had the error information from both executions, we compared them.

We filtered the scripts to analyze only those that completed within the time limit both with and without RaaS. Figure 4.1 illustrates a simple example of how we then examined the differences between the executions. First, we compared each script's result after RaaS processing, Figure 4.1(1). Then, we aggregated this data to the category of common error messages, Figure 4.1(2). This data allowed us to determine how using a reproducibility framework changed the result of an execution, if at all. These results show what types of error messages a reproducibility

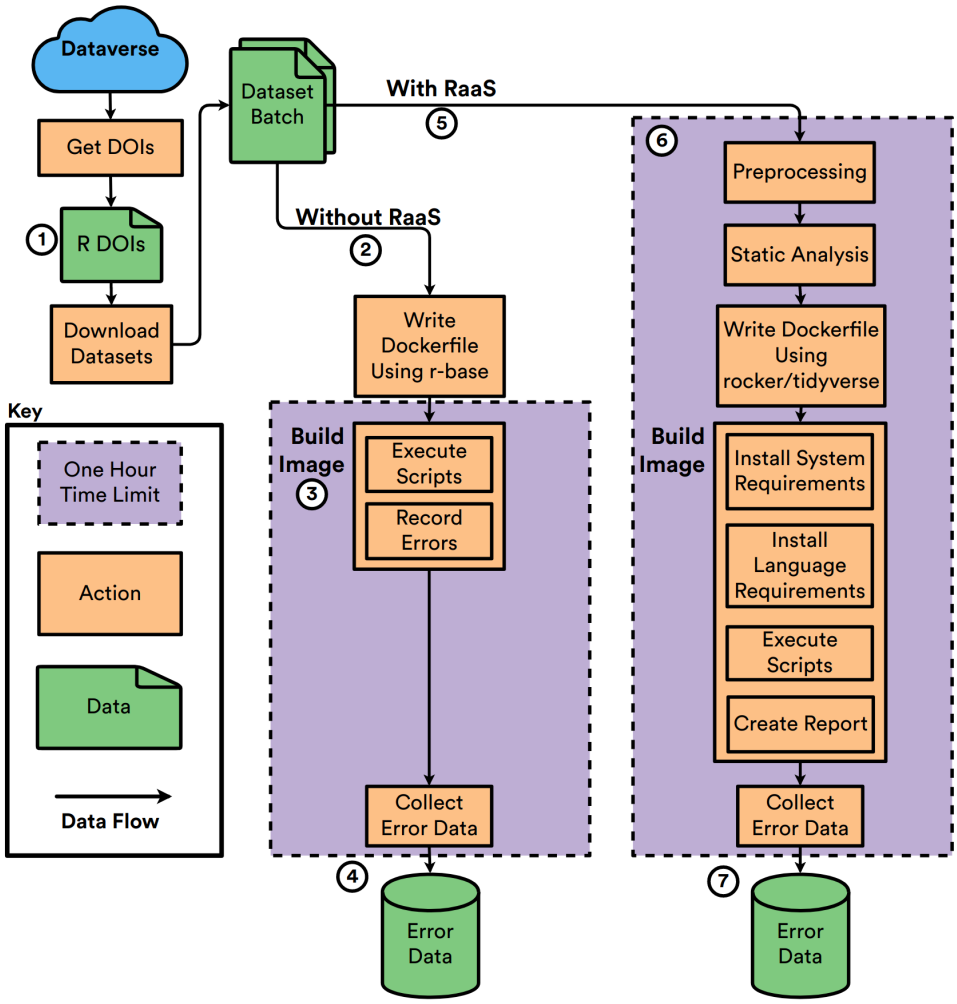

Figure 4.2: Overview of the evaluation workflow.

framework can effectively fix and what it cannot. We also compared the execution times recorded by RaaS for each dataset and the execution times we recorded for the evaluation without RaaS for a rough understanding of the overhead RaaS incurs.

4.3 Evaluation Implementation

We ran our evaluations on a lab server and virtual machines hosted by Compute Canada. Our lab server contains an Intel © Xeon © E5-2407 v2 processor with four 2.40 GHz cores, 32 GB of DDR3 system memory at 1600 Mhz, and about 549 GB of storage. We created eight virtual machines on Compute Canada's Arbutus Cloud. With one exception, all had two VCPUs, 7.5 GB of system memory, 20 GB of root storage, and a secondary storage device with 36 GB. The one exception is a single virtual machine with the same specifications, except it has 15 GB of memory and a secondary storage device of 72 GB.

We used the lab server to scrape DOIs from Dataverse, perform the without RaaS evaluation, and perform a subset of the with RaaS evaluation. We used virtual machines hosted on Compute Canada to perform most of the RaaS evaluation, except the subset executed on the lab server.

We used Python to write a script that scrapes Dataverse for datasets containing R scripts. Figure 4.2 shows how scraping DOIs is the beginning of our data collection. Throughout the rest of this section, numbers in parentheses (N) refer to the steps indicated in this figure. Our script queries the Dataverse API for any file identified as an R script (1). Then we parsed the resulting metadata to determine each script's dataset. This list of dataset DOIs defines our test corpus and is the input to later parts of the evaluation.

4.3.1 Dataverse Reproducibility Data Collection

On our lab server, we execute a Python script that takes as input our DOI list, queries Dataverse one DOI at a time, and downloads each dataset (2). As our lab server cannot store all 2900 datasets, our evaluation script downloads five datasets and then creates a new thread that executes and collects data for them while the original thread downloads the next five datasets. When the data collection thread completes evaluating its datasets, it removes their directories. After the next five datasets finish downloading and the data collection ends, the original thread repeats the process.

When executing a dataset, we place it in a unique Docker image created from r-

base[5]. The r-base Docker image is a Debian image with the version of R available from `apt-get` installed on it; specifically, we chose the image with R version 3.6.3. This method ensures datasets always run in the same environment and do not have extra packages installed. To execute the R scripts in each dataset, we wrote a new R script, `get_dataset_results.R`. This script searches through all of the directories in a dataset to identify all R scripts. Next, it executes each R script it finds using R's `source()` function inside a `try()` block, which attempts to execute code, and in case of an error, returns the error message. We wrap all of this in a `system()` function that calls R to ensure that each script executes in an environment separate from the rest of the dataset's scripts. Once the main script executes all of the dataset's scripts, it writes the results to a file. In this case, the results are either the error a script threw or the string "success."

For each dataset, we write a Dockerfile that places the dataset and the script `get_dataset_results.R` into a new Docker image. The last line of the Dockerfile runs the `get_dataset_results.R` script (3). Once we gather these results, we delete the container, image (all layers except r-base), and dataset and save the results in a SQLite database (4).

4.3.2 RaaS Reproducibility Data Collection

We repeat the previous analysis adding RaaS to the pipeline (5). As we conducted this analysis on our cloud VMs, which had only 20 GB of root storage, we processed them in batches of two instead of five. We further mitigated this problem by downloading all of the datasets directly to the external 36 or 72 GB drive and only copied them to the root for processing.

We limited each dataset's runtime to one hour again for this portion of our evaluation (6). To enforce the one-hour runtime, we set a time limit on the RaaS Celery task that, when reached, threw an error allowing the task to clean up the dataset before exiting. It is important to note that while we use the same one-hour time limit for both parts of the evaluation, the experiments using RaaS do more work than those without RaaS. The RaaS execution time includes preprocessing, static analysis, installing system dependencies, installing packages, and the dataset's ex-

[5]hub.docker.com/_/r-base/

Table 4.1: Our categorization of errors and their causes.

Error Categories	Cause
Library	Loading an R package that is not installed Typo in package name
Working Directory	Setting the working directory to a nonexistent directory Typo in working directory name
Missing File	Reading a nonexistent file Reading a file that exists but changed names or location Writing a file to nonexistent directory Typo in file name
Function	Calling a nonexistent function Calling a function from a library that has not been loaded Typo in function name

ecution. If the dataset finishes within the imposed time, we save the results to RaaS's database (7).

4.3.3 Data Analysis

We wrote our data analysis using a Python notebook in Jupyter Lab [28]. We gathered all of the databases from our data collection steps and created a project repository. Our analysis starts from the raw data collected by our evaluation script and RaaS. The RaaS databases are in the "data" folder with filenames of the form "XXXX-XXXX-app.db" where the first numbers indicate the range of DOIs within the file. The results from datasets executing without RaaS are in a database file named "results.db."

We calculated the total number of errors and the total number of scripts from these data, which produced an overall error rate for Dataverse. We then determined common error messages by starting from the most common error messages in Chen's evaluation [12]. To categorize them, we looked for substrings in each message. Table 4.1 shows the errors we encountered and how we categorized them. For example, if a script tries to load an R package that does not exist in the environment, the error message has the string `Error in library` within it. Some error messages manifest in different ways, such as a missing file error. It

can appear with the text `unable to open` or `Error in readChar`.

We then examined error occurrences across years and subjects. Using the metadata scraped from Dataverse, we identify error rates from each year and each subject. Some scripts belonged to multiple subjects, and a few were not assigned a subject. There appear to be varying reasons for why there might not be a subject associated with a dataset. We noticed some datasets were never assigned one when we visited their Dataverse webpage, while for another, the author deaccessioned a dataset since they linked it in a different Dataverse.

Since we provide pre-loaded packages for the Docker images we build with RaaS using rocker/tidyverse, we examined how much of a potential impact this had on the evaluation. We compiled a list of all the installed libraries on the rocker/tidyverse image by running a container and using the `installed.packages()` R function. We extracted all of the package names from the error messages generated when running datasets *without* RaaS. We compared these lists to determine how many of these errors we fixed using an image with pre-loaded packages.

Finally, we examined if we could increase the effectiveness of pre-loaded packages. We calculated the distribution of R packages across the images we created with RaaS. The report RaaS generates for each dataset has a list of all of the R packages present in the image. We gathered these data from each report and aggregated them by the number of images they exist inside.

4.4 Results

To determine the current state of reproducibility in Dataverse, we present error rates at the dataset and script level across the repository, by year, and by subject. We also present the results for the most common error messages generated. Next, we look at Raas's preprocessing effects, where we also present error rates for the entire repository and the most common causes of error messages. Then we examine how error messages changed from running natively to running with RaaS. Finally, we take a closer look at the most common cause of errors, missing libraries.

Table 4.2: The occurrences of errors in scripts from Dataverse without processing through a reproducibility framework. The first set of results are from Chen's study in 2018, and the second is ours conducted in 2021. The percents are rounded to the nearest tenth.

	Chen's		Ours	
Result	Count	Percent	Count	Percent
Success	408	14.4%	1035	10.1%
Error	2431	85.6%	9178	89.9%
Total	2839		10213	

Table 4.3: The most common causes of errors in scripts from Dataverse without processing through RaaS. The percents are rounded to the nearest tenth.

	containR		RaaS	
Error Type	Count	Percent	Count	Percent
Library	363	14.9%	5526	60.2%
Working directory	696	28.6%	1265	13.8%
Missing file	802	33.0%	856	9.3%
Function	NA	NA	613	6.7%
Other	569	23.4%	918	10.0%
Total	2431		9178	

4.4.1 Dataverse Reproducibility

We examined the total number of R scripts that failed to run as published on Dataverse, relative to the total number of scripts. Table 4.2 shows that the vast majority of scripts did not successfully execute. Only 10.1% of scripts executed successfully. This result is a decrease by almost one-third of Chen's result of 14.4% of scripts. We also calculated success rates on a per-dataset basis. The total number of datasets that executed all of their scripts without errors is 62, or only 2.4% of the total number of datasets.

Next, we examined the most common error messages to appear. Table 4.3 shows that the same three error messages from 2018 are still the most common

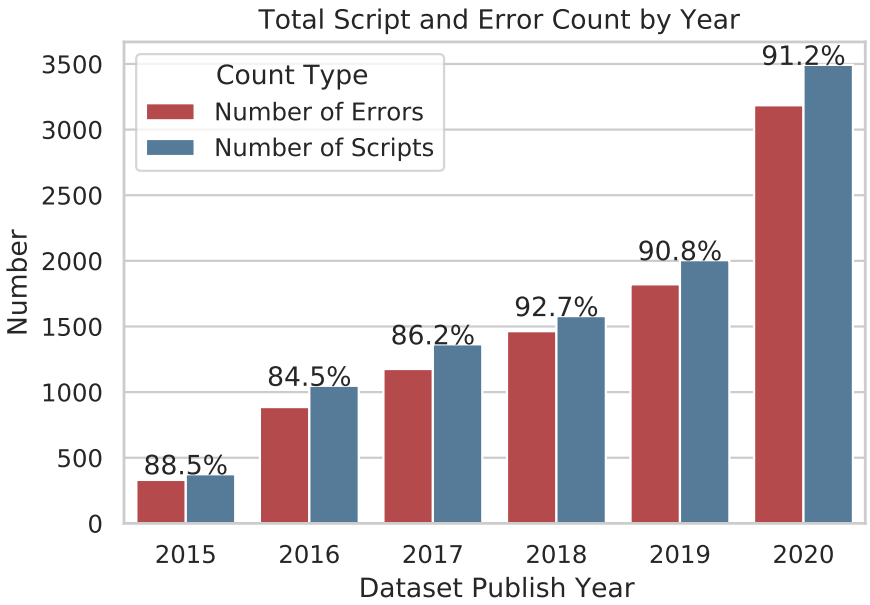

Figure 4.3: Number of scripts with errors by year from 2015 to 2020 (the last full year of data). We executed these scripts in the r-base environment without any processing with RaaS.

in 2021. Missing libraries caused the majority of errors, 60.2% of the total number of errors. This result is significantly higher than Chen's, where errors from loading files were responsible for the highest percentage of errors at 33%. Despite the differences in which root cause was responsible for the most errors, the top three causes remained the same: missing library errors, working directory errors, and reading file errors. However, we also discovered a new category of error not mentioned in Chen's evaluation: missing function errors. These errors occurred when a script tried calling a function that does not exist either because it is from an unloaded package, the user forgot to define it, or a typo.

We completed our replication of Chen's evaluation of Dataverse by breaking down error occurrences by year and subject. These results show the same trend that Chen reported. More datasets are published each year. The number of scripts from 2020 is almost double that of 2019. Figure 4.3 shows this growth. We excluded 2021 due to a low number of scripts due to our analysis's timing, completed

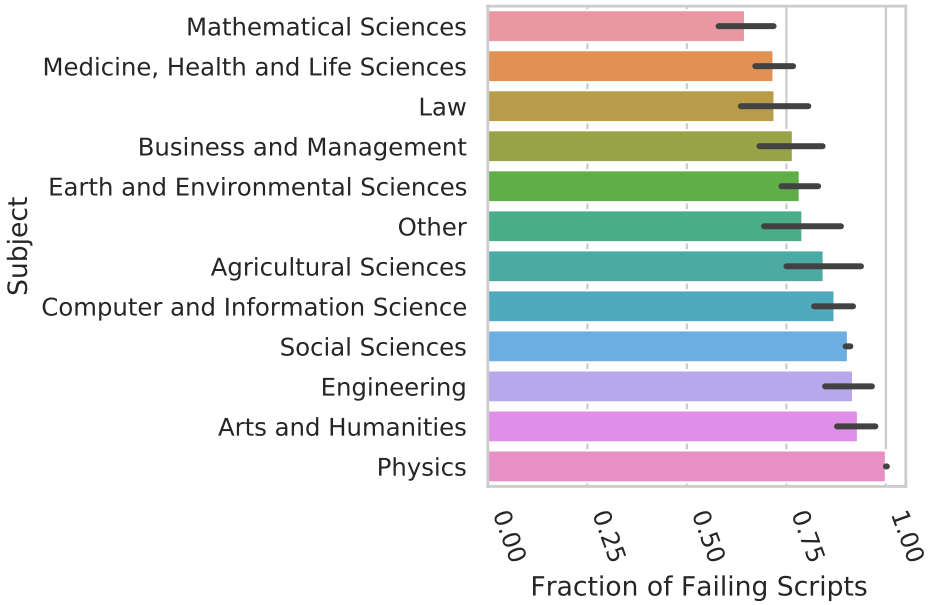

Figure 4.4: Fraction of scripts with errors by subject. Error bars show a confidence interval of 95%. We executed these scripts in the r-base environment without any processing with RaaS.

in March 2021. However, the error rate each year is relatively consistent. The minimum error rate was 84.5% in 2016, and the highest was 92.7% in 2018, a difference of approximately 8.2%. Each year's rate falls within this relatively small range, with the four most recent years of complete data all above 90%.

Finally, we calculated the error rates across different subjects. Figure 4.4 shows our findings[6]. The subjects mathematical sciences; medicine, health and life sciences; and earth and environmental sciences all have lower error rates than other subjects. Chen saw differences in these subjects due to potential better reproducibility practices that warranted further investigation. However, our results show these subjects' error rates moved closer to the mean since Chen's analysis. Physics is the highest, but Table 4.4 shows that it only has a sample size of 21.

[6]We excluded the results for Chemistry, as there was only one R script for it. Additionally, we excluded eight datasets having no subject at all.

Table 4.4: This table contains the breakdown of error occurences in R scripts on Dataverse by subject. Percentages are rounded to the nearest tenth.

Subject	Total Files	Total Error Files	Error Rate
Social Sciences	9623	8706	90.5%
Computer and Information Science	194	169	87.1%
Medicine, Health and Life Sciences	326	234	71.8%
Physics	21	21	100.0%
Engineering	85	78	91.8%
Other	62	49	79.0%
Business and Management	107	82	76.6%
Mathematical Sciences	195	126	64.6%
Arts and Humanities	114	106	93.0%
Agricultural Sciences	64	54	84.4%
Law	118	85	72.0%
Earth and Environmental Sciences	282	221	78.4%

Table 4.5: This table displays the number of datasets that time out (TO) during the evaluations, and the number that completed with and without RaaS. *Note that for the number of scripts that completed in both, the percentage is out of the total number of scripts: 11819. The rest of the percentages are out of the total number of datasets: 2900.

Datasets				Scripts
Total	TO w/o RaaS	TO w/ RaaS	Both Completed	Both Completed
2900	292	1078	1753	5872
100%	10.1%	37.2%	60.4%	50.3%*

4.4.2 RaaS Reproducibility

The most common errors we discovered in Dataverse confirmed that a retroactive reproducibility tool is becoming increasingly critical given the high error rates of even recent datasets. To determine the efficacy of RaaS, we calculated the error rates for scripts after processing through RaaS and determined the most common causes of errors.

As discussed in Section 4.2, RaaS introduces overhead in its processing, and

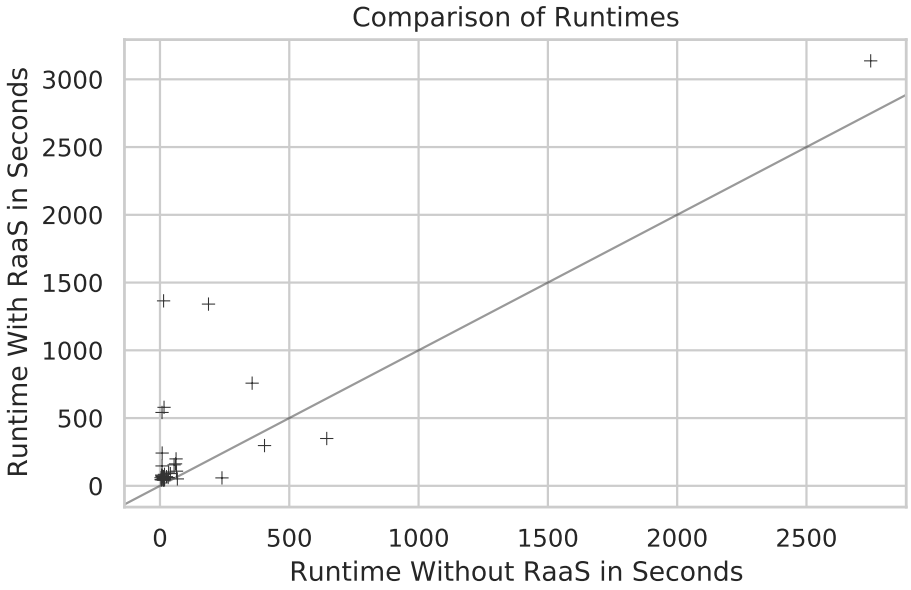

Figure 4.5: Comparison of the runtimes from our evaluation without RaaS and with RaaS. The line depicts a simple linear trend, x = y. Any dataset above the line executed more slowly with RaaS.

since we use the same one-hour **dataset** timeout both with and without RaaS, we expect that this will result in RaaS timing out some datasets (and all the scripts in them) that completed without RaaS. Table 4.5 shows the timeout information from our evaluation with and without RaaS. We observed that more datasets timed out with RaaS, as we expected. Additionally, we determined the number of successful scripts in all datasets that completed without RaaS but timed out with RaaS. There were 344 such scripts, out of a total of 4341 scripts that were a part of datasets that timed out with RaaS, but not without RaaS. Some of these datasets likely timed out due to many scripts in the dataset, the highest being 71 scripts. Others we hypothesize RaaS fixed but had lengthy execution times that cannot complete in an hour. Especially since so many had errors previously, indicating they did not fully execute without RaaS.

In an effort to quantify RaaS overhead, we examined the runtimes for all datasets that completed under both test conditions and for which all scripts exe-

Table 4.6: This table compares the execution results from scripts without RaaS compared to with RaaS. This table contains only the scripts from datasets that completed under both conditions. The "With RaaS" category has fewer total scripts due to RaaS's preprocessing preventing sourced scripts from executing. A dataset error means at least one script in a dataset contained an error.

	Scripts		Datasets	
	Without RaaS	With RaaS	Without RaaS	With RaaS
Total	5872	4949	1753	1753
Successful	691	1535	46	418
Error	5181	3414	1707	1335
Percent Successful	11.8%	31.0%	2.6%	23.8%

cuted successfully. There were 41 such datasets. We then compared their runtimes as shown in Figure 4.5, which shows the runtime with RaaS as a function of the runtime without RaaS. All but four datasets ran more slowly with RaaS. The trend appears to show that running datasets with RaaS results in some overhead, but it varies depending on the dataset. Since the RaaS overhead is affected by multiple factors, including the number of packages to install, the number of scripts to execute, and the content of scripts, it is difficult to determine exact estimates for the overhead. Since these datasets ran without errors both with and without RaaS, the varying overhead likely comes from the script's content and consequently the overhead from rdtLite and static analysis.

Table 4.6 shows a comparison between our previous execution results without RaaS and the execution results with RaaS. The only data included in the table are from datasets that did not time out during both evaluations. We observed 1753 datasets containing 5872 scripts that fit this criterion, as shown in Table 4.5. There are fewer total scripts in the RaaS column as a result of RaaS preprocessing. Since we copy sourced scripts into any script that sources them, we do not run them independently. However, we did examine the number of sourced scripts that were successful without RaaS, a class of scripts that often contain only function definitions that are almost guaranteed to execute without error. We observed that 299 sourced scripts completed without error without RaaS, or 43.3% of all total suc-

Table 4.7: The most common causes of errors in scripts from Dataverse after processing through RaaS compared to containR.

	containR		RaaS	
Error Type	Count	Percent	Count	Percent
Library	8	0.3%	555	15.8%
Working directory	12	0.5%	75	2.1%
Missing file	1400	61.8%	1258	35.8%
Function	NA	NA	307	8.7%
Other	847	37.4%	1320	37.6%
Total Failures	2329		3515	

cessful scripts without RaaS in Table 4.6.

We increased the number of successful scripts by 2.2x. We also calculated success rates on a per-dataset basis, still using datasets that completed under both conditions. The total number of datasets that executed all of their scripts when processed with RaaS without errors is 418. This result amounts to 23.8% of the number of datasets in this set, an order of magnitude higher than without RaaS.

We then examined the most common cause of errors in the RaaS processed scripts. For these results, we use all datasets that completed when executed with RaaS, regardless of their result without RaaS. Table 4.7 shows that no error cause was common enough to make up the majority of errors. The plurality of causes with RaaS was the "other" category, followed closely by "missing file". The "other" category contains any error message that did not fall into one of the defined categories. These messages are highly variable, although some examples are syntax errors, missing object errors, or errors specific to packages. A manual inspection of each of these messages might lead us to discover some of them belong to the common error categories but that we did not categorize them correctly because a package created its own error to report the problem. The following are a few arbitrarily chosen examples of "other" error messages, although these examples are only a small subset of the types of error messages represented in "other."

- `Error in ergm.MCMLE(init, nw, model, initialfit = (initialfit <- NULL), : Unconstrained MCMC sampling`

Table 4.8: How errors changed from running without RaaS, to running with RaaS. The rows indicate errors occuring pre-RaaS. The columns are post-RaaS. For example, 741 scripts encountered a library error pre-RaaS, but ran successfully post-RaaS.

	Post-RaaS					
Pre-RaaS	library	working dir.	missing file	function	other	success
library	431	32	730	70	816	741
working dir.	107	42	207	49	174	190
missing file	3	0	262	26	42	32
function	0	0	12	154	23	36
other	2	0	8	3	240	142
success	0	0	0	0	5	388

```
did not mix at all. Optimization cannot continue.
```

- `Error in tolower(units$county):`
  `invalid multibyte string 1824`

- `Error in myDfm[1]: Subscript out of bounds`

Next, we checked on a per-script basis how errors changed after RaaS processing. This information can help identify the next steps towards successful retroactive reproducibility. Table 4.8 shows our results. Note that if you add up each column's values, they do not necessarily equal the results from Table 4.7, because some datasets failed to record data when executing without RaaS but did so successfully with RaaS. Table 4.8 includes only those scripts that produced data under both conditions.

26.3% of scripts with library errors were *not* masking other errors and succeeded once we installed the correct libraries. We achieved similar results from scripts with working directory errors, 24.7% of these scripts succeeding once processed by RaaS. Unfortunately, in the majority of cases, these errors were masking other errors. However, the third most common cause of errors without preprocessing, missing file errors, behaved differently. Only 8.8% of scripts with missing file errors succeeded after processing, and 71.8% threw another missing file error. Surprisingly, RaaS fixed approximately 35.9% of scripts with errors in the "other"

Table 4.9: This table shows how using the rocker/tidyverse container could potentially fix library errors from scripts since it contains commonly used packages. In this table are the number of unique packages that Dataverse scripts fail to load, as well as the total number of scripts that attempt to load them.

	Unique Packages	Total Scripts
Preloaded In Tidyverse	43	2234
Not Preloaded	398	3236

category that were not masking another error: 10% higher than for the categories of errors for which we intentionally designed RaaS.

We calculated the efficacy of our static analysis compared to using pre-loaded packages. We calculate these results at the dataset level, not the script level, because all the scripts in a dataset share the same computational environment. Table 4.9 shows the results we might have gotten had we only used pre-loaded packages instead of all packages identified via static analysis. We discovered that 43 of the packages that caused errors when executing scripts without RaaS were pre-installed on the rocker/tidyverse image. If we assume that all of those packages would have been successes[7], library errors would still account for 47.4% of our total errors. Even if we assume that these scripts where we fix their library errors solely through pre-loaded packages were only masking other errors, library errors would still make up 35.9% of all errors.

Finally, we examined the likelihood we could increase the effectiveness of pre-loaded packages. We discovered a minimal overlap between packages that datasets use. Figure 4.6 shows the distribution of R packages across images plotted on a log scale. In total, we created 1822 images, one per dataset. Over half of the packages are present in only four or fewer images, and 75% of them are in 26 or fewer images.

[7] Also remembering to remove those from the total, bringing our total number of errors down to 6944.

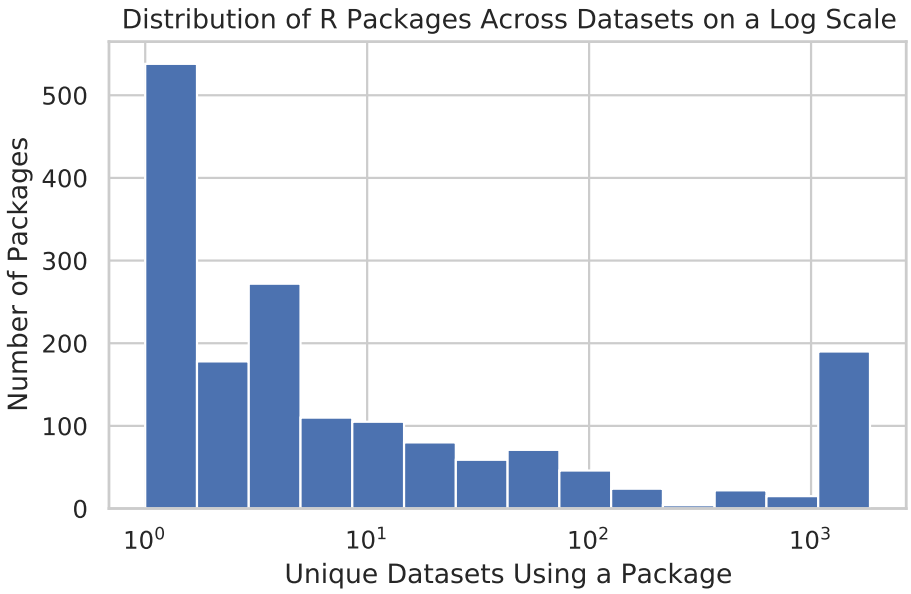

Figure 4.6: Distribution of R packages, the skew shows how most packages are only used in a small number of datasets. The smaller skew on the right side represents the pre-loaded packages.

4.5 Results Discussion

We discuss the potential reasons for some of the results presented in Section 4.4 and explain how they fit into the context of RaaS as a retroactive reproducibility service. We start by discussing our results compared to Chen's evaluation and then proceed to a discussion of RaaS's effects on the datasets.

4.5.1 Dataverse reproducibility

The reproducibility of R scripts in Dataverse is the groundwork for this evaluation and defines the problems a retroactive reproducibility service needs to address. We already have a snapshot of the state of reproducibility in 2018 from Chen [12]. Comparing our results allows us to answer our research questions **Q.1**, **Q.2**, and **Q.3**.

Several of our results are consistent with those of Chen [12]. In particular:

- The majority of scripts on Dataverse cannot be reproduced.
- Library, missing file, and working directory errors are the three most common errors found in scripts that we execute from Dataverse.
- Aging is not the cause of reproducibility failure.
- In most cases, across subjects, the rate of reproducibility is the same.

However, there are also points of departure:

- Our most common cause of errors is "library" and Chen's is "missing file." Our "library" category also accounted for a more significant portion of the number of errors, 60.2% than Chen's "missing file" at 33.0%.
- We added a new category we found significant, the "function" category.
- The subjects that Chen found to be more reproducible have moved closer to the mean in our evaluation.

The increase in "library" errors we found could be due to our experimental setup. Chen downloaded datasets from Dataverse and executed them on a high-performance computing cluster, Harvard's Odyssey cluster. When running R on this cluster, it pre-loads 100 commonly used packages[8]. While the average R user likely builds a collection of packages as they use the language, a computational experiment's reproducibility should not depend on it. Consequently, we chose to execute our datasets without RaaS inside the r-base Docker image, which does not have pre-loaded packages. This method potentially affected the most common error messages, resulting in higher instances of "library" errors.

Second, we observed a relatively significant number of missing function errors, which Chen [12] did not mention. We cannot be sure if they appear in his "other" category. Nonetheless, these errors are intriguing as a manual examination of the error messages showed that many were partially a symptom of missing libraries. For example, one missing function was `readOGR`, likely from the `rgdal` package. However, since this error message is for a missing function, it means the script

[8] https://docs.rc.fas.harvard.edu/kb/r-packages/

never tried to load the `rgdal` package first, or it would have thrown a missing library error message. While we initially wanted to classify this as another missing library error, we cannot. While the library *is* missing, the root cause is the missing library combined with the user mistake of never calling `library(rgdal)` or any package that depends on it.

Finally, we observed that reproducibility is a problem across all subjects. While there were variances in error rates, the range of 64.6% to 100% demonstrates that all subjects suffer in some way. Chen [12] theorized that mathematical sciences; medicine, health and life sciences; and earth and environmental sciences might have better reproducibility practices leading to fewer errors in their scripts. While they still have lower rates, they are no longer vary as much in our evaluation as they did in Chen's. The lower error rates, therefore, could be a result of a smaller sample size. However, some subjects such as mathematics might rely on more basic R functionality than arts and humanities, which might need higher-level functionality present only in different packages. Unsurprisingly, the social sciences dominate Dataverse, with scripts in the thousands, compared to the rest in the tens or low hundreds. We expected this result given Dataverse's origin as a social science repository.

The overall results from this section support our hypothesis that we need a retroactive reproducibility service. Even though each year researchers contribute more datasets to Dataverse, the service's overall reproducibility is relatively constant. Additionally, researchers create proactive reproducibility tools, but datasets on Dataverse rarely use them. For example, there are only five `renv.lock` files on Dataverse as discussed in Section 4.2. As Chen hypothesized, datasets are not only succumbing to software rot; they are simply not reproducible from the start. These results give us a better understanding of the state of reproducibility on Dataverse, **Q.1**; how reproducibility has changed since 2018, **Q.2**; and some of the most significant barriers to reproducibility, **Q.3**.

4.5.2 RaaS Reproducibility

Our results from executing scripts with RaaS demonstrated its efficacy at handling common reproducibility errors, **Q.4**, and its potential as a tool integrated with the

publication pipeline to increase reproducibility quality. We were able to increase the success rate of Dataverse script execution by only 20%; however, many scripts timed out. We also added extra time to the runtime of datasets when running with RaaS. Preprocessing, static analysis, installing dependencies, and collecting provenance all impose additional overhead not present when we ran the evaluation without RaaS. The number of packages a dataset uses, the number of scripts in a dataset, the length of scripts, and the content of the scripts all ultimately affect the overhead with RaaS. Only 15 datasets completed with zero errors without RaaS and timed out with RaaS. While we would have liked to make predictions about what happened to the datasets that timed out based on this information, this small sample size with highly variable features prevents it.

While we cannot analyze any of the data from the timed-out datasets, we are optimistic about their reproducibility, because they executed for so long that we had to move on to the next dataset. Therefore, our evaluation might skew towards not reproducible datasets. Pimentel et al. [47] noted that importing packages into a script tends to occur at the beginning of a script. Since the datasets that timed out were likely executing for a while, they likely did not have missing library errors, the cause for the majority of errors without RaaS.

We manually examined the library errors that remained even after processing with RaaS. We discovered 130 unique packages that threw an error when the user tried to load them. Some packages, such as preText [15], are no longer hosted on CRAN *or* BioConductor. In this situation, a user might search for the package on GitHub or a similar repository site. However, since installing from GitHub requires knowing the name of the user hosting the repository, it is too complex for RaaS to do automatically. Additionally, we found packages that should not have caused an error, but due to the user's method for specifying the library, there was an error. For example, one error message we observed is

> Error in library("stringr", lib.loc =
> "/Library/Frameworks/R.framework/Versions/3.2/Resources/library"):
> no library trees found in 'lib.loc'

The stringr [62] package is pre-loaded on rocker/tidyverse, but the user specified a location to load the library from using `lib.loc` that is version-specific

(`Versions/3.2/`). This argument causes R to throw an error because that directory does not exist in the computational environment. In RaaS's current state, a user could fix this error by simply removing the `lib.loc` argument in the function call. We also could improve RaaS's functionality to handle these scenarios, but we did not discover them until after our code freeze, so we leave it for future work.

Since our preprocessing attempts to delete all instances of a script changing the working directory, we manually examined the error messages from datasets processed with RaaS that threw working directory errors. We observed that since RaaS takes a conservative approach to deleting working directory instances, some manage to slip by the preprocessor. RaaS uses the same method of preprocessing loading files for working directories, which means that when a script uses a variable instead of a hard-coded string as an argument, RaaS does not preprocess it. We elaborate on this in Section 5.1. However, these instances are easy to fix from a user's perspective. Theoretically, they only have to delete the `setwd` function call to fix this error. Once again, these instances were discovered after our code freeze, so as future work, we could make RaaS more robust to these bugs.

It is impossible to determine most of the root causes of the missing file errors when executing datasets with RaaS. Table 4.1 shows that missing files have the largest number of potential root causes, and each of the causes could be due to different reasons. A file might not exist in the dataset because the scripts needed to run in a particular order as another script created the file. Alternatively, the user might have chosen not to upload it for privacy issues. All of these causes manifest in similar error messages. There are some exceptions; for example, we noticed some scripts used the `file.choose` function. This function is interactive, and since we execute all scripts automatically, it throws an error. We observe a similar problem with the function errors. Without domain knowledge, it is impossible to determine if a function is missing due to a failure to load a library, as discussed in Section 4.5.1, or because a user forgot to define the function. Due to the varying nature of these two types of errors, we cannot determine if they would be simple for users to fix.

It might seem troubling that missing files grew from 9.3% of errors to 35.8% of errors, but we expected this result. There are more complex possibilities for miss-

ing file errors compared to missing libraries. Some researchers might not be able to upload all their input data due to privacy or size restrictions. Considering the social sciences account for most of the datasets we examined, it is reasonable to assume that privacy concerns are significant for some of these datasets. Our preprocessing also relies on researchers uploading input data with correct file names that have no duplicates. If there is a typo or they renamed a file for archival purposes without adjusting the script, we cannot correct it with RaaS. These results also demonstrate an interesting phenomenon we experienced throughout this project; there are often complex and almost inexplicable behaviors in these scripts. For example, we fixed a surprising number of errors without even intending to fix them.

One particularly interesting result was the number of scripts that threw errors in the "other" category that succeeded after executing with RaaS. This result could potentially be due to common errors that manifest in unexpected ways. We examined some of the datasets that demonstrated this behavior and, while it is difficult to draw any conclusions, one potential commonality was "missing object" errors. Out of all of the datasets executed without RaaS that did not time out, we discovered 99 scripts had this "missing object" error but succeeded after running with RaaS. We are not sure how missing libraries or other common causes could result in these errors without manifesting directly as a missing library or the other common causes we have found. Hence, this warrants further examination.

We, surprisingly, found five scripts that were successful without any preprocessing but threw "other" errors after processing by RaaS. While we are pleased out of the thousands of scripts processed, only five appeared to break due to RaaS, this issue requires a more in-depth examination to ensure they are not due to a bug in RaaS. Four of the new errors are

```
Error in dev.off(): cannot shut down
 device 1 (the null device)
```

and the fifth is

```
Error in if (is.na(file.param.name))
{: argument is of length zero.
```

We examined these errors manually. The first error appears to be a bug related to running rdtLite using the `system` function. We ran one of the scripts using rdtLite directly, and it did not throw an error; however, if we run it using the command

`system("R -e \"library(rdtLite);prov.run('script.R')\"")` then it does throw the error [9]. Interestingly, there are many calls to `dev.off` in the script we tested, and the one that throws the error is neither the first nor the last. The second error appears to be a bug with rdtLite and the function `trellis.device` from the `lattice` package [55]. If we run the script that throws the error with `source`, there is no error. Since the line with `trellis.device` is one of the last three lines of code out of 225 lines, we commented out the last three lines and ran the script with `prov.run` again, and it was successful. We will investigate these causes and open an issue on the RDataTracker repository.

This evaluation demonstrates that common errors are often masking other, potentially more complex, errors. RaaS can automatically fix a portion of scripts bringing the total success rate of scripts to 30%. Based on the results we examined earlier in this section, we can begin to answer our research question **Q.4**; users can *easily* fix 17.9% of the errors that appear when executing datasets with RaaS if they are library and working directory errors. Due to the incredibly varying nature of the other errors, we cannot confidently say how many of them users can fix *easily*. Still, most are likely fixable with extra work spent examining the scripts. This result demonstrates that RaaS's real power lies in deploying it as part of the publication pipeline. RaaS can identify these more complex errors at upload time and require that users fix them. Missing input data, typos, and almost every other problem could be fixed by the user if they realize the problem exists.

[9] The command "R -e" takes an R expression as an input, executes it, and then returns.

Chapter 5

Implications

This chapter discusses RaaS's limits and a potential future for the project. We examine how RaaS differs from its predecessor containR, lessons we learned about reproducibility and reusability from this work, and how those led to the design choices we made. We also propose reproducibility best practices based on our experience and evaluation.

5.1 RaaS Limitations

RaaS currently has various limitations, some due to design and implementation choices, while others exist due to the nature of the problem we are addressing. In this section, we examine these limitations and whether or not we could improve them.

Unfortunately, there exists a class of errors that we cannot correct, because doing so requires dataset-specific information. Examples of these errors include missing data, renamed files, or scripts that require command-line arguments. But, these are still problems we can present to a user at upload time, so if they are using RaaS during their publication process, they can fix the error before they publish.

Any limitation that exists in Docker is naturally a limitation for RaaS. For example, big data processing might require a swarm of containers to compute, which RaaS alone cannot provide. Additionally, if some software-breaking bug caused Docker as a service to break, then RaaS itself would break. In some environments,

users lack sudo access, such as a high-performance computing cluster. Since the Docker daemon requires sudo privileges, running containers there is not an option. As an alternative, Singularity is an open-source container service for Linux [29]. Researchers created Singularity for use in scientific computing on high-performance computing clusters. Currently, Singularity supports Docker images as an input, so it is trivial to use Singularity as an alternative[1].

Our current preprocessing implementation cannot fix certain issues in R scripts. The fact that we use regular expressions to parse R scripts limits our ability to detect certain problems. We currently rely on the dataset authors using the common style of placing the file path directly into a function call, e.g.,

```
read.csv("path/to/file.csv").
```

We can often correct what could be problematic file paths if they do:

```
read.csv("./relative/path/to/file.csv").
```

However, if a user constructs the path dynamically or expects the path as a command-line argument, we cannot fix those potentially incorrect paths.

```
file.path <- "C:/home/user/file.csv"
read.csv(file.path)
```

Since our regular expressions look for file paths within function calls, not variables as we discuss in Section 3.5, we cannot correct these file paths. We discuss potential fixes for this problem in Section 5.2.

If a dataset needs to execute in a particular order, RaaS cannot identify that order, with one exception. In some cases, datasets might use the output from one script as the input of another. While some researchers might use the `source()` function to enforce this ordering, there is no guarantee they will. If there is no ordering enforced with `source()` we could potentially fail with a dataset that would otherwise be reproducible. However, we noticed some researchers add something such as a `run_all.R` script to their datasets. This script uses the `source()` function on each script in the dataset and enforces any necessary ordering. This scenario is particularly beneficial for RaaS. Its preprocessing ensures that the `run_all.R` script is the only script to execute, and that script runs everything in the correct order. If a dataset does not have a script to run the entire

[1]Singularity might have limitations with Docker interoperability. To find the most up-to-date explanation of the support they provide, visit their documentation. https://sylabs.io/docs/

computational experiment, it is challenging for RaaS to determine if an order is required. However, we could potentially extend our preprocessing to identify input and output files and match them together.

RaaS supports multiple languages but assumes all scripts in a dataset are in the same language. RaaS could process a dataset with various languages, but it would only preprocess and execute the scripts for one. Since we currently cannot use RaaS's output as an input to itself, there is no way for RaaS to process all the languages in a multilingual dataset. Unfortunately, due to the current design of RaaS, there is no simple fix we can apply to fix this. However, in Section 5.2 we discuss design changes to RaaS that could fix this issue.

RaaS currently facilitates reproducibility, but it cannot verify a successful reproduction. If a user uploads a dataset to RaaS and receives a Docker image with collected provenance from script executions, this does not mean RaaS successfully reproduced the computational experiment. It is the user's responsibility to examine the report and the results generated by the scripts to determine if the execution successfully reproduced previous results. Currently, there is no method for inputting previous results to RaaS to act as a measure for reproducibility. However, we discuss this as a potential next step in Section 5.2.

While some datasets may have taken some proactive steps, RaaS can only consider steps that are part of the scripts, such as using `source` to run scripts in the correct order. Some researchers use `make` [52], `renv` [59], `conda` [3], or other project management tools. These could help the reproducibility of the datasets that RaaS processes. Unfortunately, we have to create our own workflow pipeline to collect provenance and ensure that we do not execute sourced scripts. This feature prevents us from using artifacts such as Makefiles. We also currently do not support installing packages from `renv.lock` package snapshot files, although supporting this feature is potentially future work.

5.2 Future Work

While RaaS accomplished its goal of facilitating reproducibility, our evaluation results and observations of limitations suggest avenues of further development. These features focus primarily on user interaction and language-independence, al-

though we can also apply some miscellaneous fixes based on our current limitations. Given our results on the large number of datasets that still encounter errors, adding a human-in-the-loop to RaaS's build process could be an effective retroactive reproducibility strategy. While RaaS would no longer be fully automatic, it would still be responsible for most of the labor required to facilitate reproducibility. The user would need only to guide specific decisions and troubleshoot problems that might be difficult for the software to determine. We can also make our codebase more language-independent without adding additional complexity.

5.2.1 Interactivity

Our results demonstrated that while RaaS can fix a large class of errors in scripts, it frequently reveals more complex errors such as missing files, missing objects, or syntax errors. Given that it would be a *significant* engineering effort to try and account for each of these possibilities, a human-in-the-loop might be more helpful. We see a few areas where we could potentially facilitate human-computer interaction: determining error causes, environment re-use, and results verification. Each of these features could benefit from user oversight or could help users with their goals.

Some scripts might require command-line arguments, which RaaS cannot determine automatically. In this case, we can add two features to assist with this problem. First, we can add support for users to specify command-line arguments per script at upload time manually. We have already started to explore this option starting with our Python implementation. Second, we can add support for pipeline toolkits such as make[52] and targets[30]. Not only will this allow us to encourage the use of these proactive tools, but users can use these tools to supply their scripts with command-line arguments.

Errors generated from installing dependencies tend to be lengthy, vague, and non-standard. We have encountered this with both system libraries and language packages. If an installation requires a dependency, it often requires a glance through the output to determine what is missing. This type of parsing would be complex for us to implement automatically. This output often references tens of other libraries' names, and a few of them might all break because of one missing dependency.

While automating this might be possible, doing so correctly remains a challenge. Instead, a more effective approach is to provide better error reporting to the user building a package to develop a solution collaboratively. For example, if an R package fails, we could provide output from the error message displaying that a system library is missing. In some cases, the output provided might specify the library needed, but not necessarily the name required for installation. One such example is a failed installation of the R package `rgdal`. Its output informs a reader that `gdal` is missing. However, a user cannot just issue `apt-get install gdal`. They have to install `libgdal-dev`. A user is more likely to figure this out than our software. Then, through an interface we provide, they could change the installation process to include `libgdal-dev` without re-uploading their dataset. We can implement this small change before re-installing the failed R package and continuing. One extension to this feature is to provide similar functionality with errors at runtime.

We examined a few of the errors in the "other" category manually, and it appears that simple typos caused some of them. For example, we saw comments that the authors likely intended to be multiline, but they forgot the comment character "#" on the second line. These issues occur, and we would rather not force users to re-upload their entire dataset to fix them. Instead, in the event of a runtime error, we could display the error message to the user with an option to open a text editor for that script. They might notice they could resolve the error by fixing a typo such as a missing comment character. So they open the editor, add the comment character and start processing again. We would then record that this edit took place. We could further make this process more efficient by implementing another feature to increase user interactivity: explicitly tagging the environment layer.

We can further assist users with a hybrid approach to reproducibility. Explicitly tagging the Docker image once we have successfully installed the environment and before we copy in the dataset is helpful for a few reasons. The first relates to runtime errors in the script. Suppose we have an explicitly created Docker image for the environment, and the dataset encounters a runtime error. In that case, we know that we can allow the user to rerun the script without rebuilding the environment, which can be a lengthy process. The next reason is this method increases reusability. If a user chooses RaaS for a retroactive reproducibility task and then

wants to create a new computational experiment in the same domain as the one they retroactively reproduced, they could re-use this environment. For example, Sequoyah finds a geospatial analysis they want to reproduce from the TREE conference. This dataset depends on the `sf` package, which in turn has a host of dependencies, both as R packages and system libraries. Sequoyah uses RaaS, which creates a new Docker image, and examines the computational experiment. Sequoyah is intrigued by the results and chooses to do a follow-up experiment with new data. Rather than create a new computational environment from scratch, they can request the reproduced experiment's environment from RaaS. Sequoyah can then use this environment with all of the necessary dependencies such as `sf` pre-installed.

5.2.2 Results Verification

Our evaluation examined whether or not datasets were "not reproducible" instead of "successfully reproducible," because RaaS cannot currently verify a computational experiment's results. The most significant obstacle in this area is the often varying definition of what it means to be reproducible, as discussed in Section 2.1. At the same time, it might be valuable to make attempts to verify results. The most naive and straightforward method would be to attempt bitwise-reproducibility. If a user could somehow provide the initial results in a parsable way, we could execute the computational experiment and compare the results. This technique might require significant effort on a user's part, as they might have to perform actions such as convert numbers from a figure in a paper to a format RaaS can use. Additionally, as mentioned in Section 2.1, the user might still accept the results as long as they fall within a pre-defined range. We could provide the functionality to test this, although that would only automate what might be a simple task for the user to do themselves, as they still have to determine the range themselves.

5.2.3 Langauge Improvements

RaaS currently processes only one language at a time and contains unnecessarily language-dependent code. By separating more language-independent code, we can support more features such as multi-language datasets. We could feasibly call the

preprocessing for each language, then the static analysis for each language, and then write the Dockerfile use all of the pieces collected. Our current language-dependent method of writing the Dockerfile would be difficult to use with this format. Still, we could improve the separation of language-independent code in that function.

While the current function to write Dockerfiles is language-dependent, it is possible to create a new language-independent version. A new design would require developers to implement more abstract methods; however, they would no longer need to know how to write a Dockerfile. We observed that as we improved RaaS, the Dockerfile began to converge on a specific pattern. Language-dependent functionality often works best as a script in the required language. For example, in R, installing packages is performed through the `install_packages.R` script. Executing the dataset and collecting provenance is performed through the `get_dataset_provenance.R` script, and writing most of the report is performed through the `create_report.R` script. The other aspects of the Dockerfile are language-independent. Since new language support requires users to write similar scripts for new languages[2], we can create methods that identify these scripts' locations and provide the command necessary to run them, and the function to write the Dockerfile calls them. Using this design, implementors of a new language have to write these scripts just as they would have with the previous design, but now they do not need to know anything about writing Dockerfiles. The Dockerfile function becomes language-independent, and we can support multilingual datasets.

We can fully support a dataset containing multiple languages by this abstraction of the Dockerfile generation function into the language-independent code. Since our current iteration has a language-independent function for creating the Dockerfile, if we want to support multilingual datasets, we would have to determine a way to combine two (or more) separate Dockerfiles, a daunting task. With the future design, we can instead call the language-dependent functions for each language during the creation of the Dockerfile. For example, we can install all of the system libraries at once since they are language-independent. If a dataset has R and

[2]Note that as discussed in Section 3.5, scripts are preferred over direct commands in the Dockerfile so that we do not reach the layer limit for Docker images.

Python scripts, we could write the instruction to load and run the script to install the R packages; we could then do the same for Python packages. If we continue this pattern, we have a single Dockerfile with mixed languages when the function completes. The only further change we would have to make is a format change for our report structure. For any language-dependent information in the "Container Information" section, we can add a new key: the language name. The same language-dependent data we already collect would be the value of the key. Since we would be moving towards a design of placing scripts in the container to perform language-dependent functionality, we could make one more piece language-independent.

Using this proposed design, we could construct a language-independent version of the report generation function. We cannot avoid developers writing code to collect the language-dependent features of the environment. However, we can reduce the number of features they need to collect, something our current iteration is already poised to do. Our current R implementation has an R script to generate all the report's parts except build time and tag name. The only thing our RaaS function does to write the report is it runs the container, retrieves the report, and adds the tag name and build time. Since it already has to run the container, we could move the collection of language-independent environment information to this function and make it language-independent. All languages can share this code since it just copies a JSON file and adds language-independent data. With this new design, developers would have to implement a higher number of functions, although they would be responsible for less content overall than they are now.

5.2.4 Miscellaneous Fixes and Features

One of the limitations we noticed in Section 5.1 is failures to fix specific file paths in preprocessing. We could attempt to add more regular expressions to find these instances. However, we have found this parsing method encounters problems quickly if you cannot write robust enough expressions. Alternatively, we could use a static analysis tool for preprocessing instead and employ basic symbolic execution to trace the variable a function uses. In practice, for the example in Section 5.1, our preprocessor would note that the script declares the variable `file.path`. It would remember the value the script assigned it, `"C:/home/user/file.csv"`.

When it reaches the `read.csv` function, it notices that its input is not a string literal but a variable. The preprocessor detects it has the value of the variable remembered, and it can apply the preprocessing techniques we described in Section 3.5.

In the absence of, or in addition to, the potential features we can add to RaaS to assist with verifying dataset results as discussed previously, we can use the provenance as a means to communicate information about the dataset to the user. Because of the wealth of information contained within provenance, researchers have built various tools to communicate that information to users. These tools range from visualizations to debuggers, such as DDGExplorer [32] and provDebugR [10], respectively. We can combine these existing tools with RaaS as a way for users to explore the datasets without ever needing to run a Docker container. If we integrated a provenance visualization tool or explorer into RaaS, users could explore a computational experiment's results at a fine-grain level. These tools could help users verify the dataset results or understand the transformation applied to the data by giving them a closer look at the execution details.

Since RaaS is not an archival service, we cannot keep images forever. While RaaS currently does not automatically delete images, it will happen in the future. Images can require a lot of storage space, especially if the dataset it contains is large. Since we do not want to encourage our users to keep the image on RaaS's servers, we will have to implement a time limit for how long their image is available. How long exactly that limit will be is unknown, but the important part is that it exists.

Finally, we will implement more options for users to upload datasets to RaaS. At the moment, RaaS only receives zip files for processing. However, in the future, we can add a way to provide RaaS a link to a GitHub repository or other online repository service that RaaS will then download the dataset. Since many researchers already use online repositories such as GitHub, they will have to do less work to have RaaS process their dataset.

5.3 Reproducibility Best-Practices

Based on our experience developing RaaS and conducting our evaluation, we have compiled a description of simple proactive practices that can help researchers ease

the burden of retroactive reproducibility. These recommendations are not exhaustive and it is not necessary to adhere to all of them. These recommendations are similar to FAIR's purpose, a way to guide those who wish to be reproducible rather than a set of rules. It is always best for researchers to examine information critically and determine what is best for them.

We also realize and accept that many researchers likely will not or perhaps cannot use a guide such as this one. It is for this reason that RaaS exists at all. In this case, we propose integrating a retroactive reproducibility service such as RaaS into the publication pipeline. Doing so is similar to format checkers for the publication itself, except for the computational experiment. Researchers will be unable to publish until they provide a reproducible computational experiment or demonstrate that RaaS successfully reproduces it.

Rec.1 Preserve the Environment. Preserving the computational environment from the beginning saves researchers from many future headaches. Adopting one of the online platforms discussed in Chapter 6 is one method to accomplish this. However, a researcher will likely find that learning Docker is an invaluable skill. For example, we analyzed the data we collected using JupyterLab hosted in a Docker container. While we wrote the analysis, we mounted our dataset directory into the container. When we finished, we added a new layer to the JupyterLab image with our dataset and saved it as a new image. We now have a reproducible computational experiment where the Docker image not only contains a sufficient computational environment, it is the *original* computational environment. Alternatively, researchers can learn lighter-weight solutions such as `renv` or `conda`; however, these tools might not capture all system library dependencies.

Rec.2 Enforce Directory Structure. A consistent and logical directory structure makes it easier for future researchers to navigate a dataset. Additionally, clearly defined directories for scripts, data, and results assists with further recommendations.

Rec.3 Write a README. Even though RaaS might not find it useful, a well-written README goes a long way. This document can contain all sorts

of human-readable information about the dataset. Some potential important and helpful pieces are data collection procedures, any necessary command-line arguments, expected results, and a place to find more information.

Rec.4 Use Relative File Paths. A file path starting at the root of a device is all but guaranteed to break once it changes environments. Using relative file paths helps eliminate this problem, especially when combined with the next recommendation. The `here` [39] package for R is also a good tool that helps facilitate this style.

Rec.5 Use a Consistent Working Directory. Users should not change the working directory of a script. A working directory change is likely unnecessary, and researchers should always use a consistent location as the working directory. Common styles for working directory placement are the root directory of the dataset or the directory where the script resides. Although choosing the root directory for the dataset is a more consistent style. Once again, the `here` [39] package for R can help mitigate working directory pains.

Rec.6 Enforce Script Ordering. Enforce the ordering of scripts in a dataset, even if there is no strict ordering necessary for the dataset to succeed. Future researchers might not know if a dataset has a required order, but if a researcher uses `make` [52], `targets` [30], or has a `run_all.R` script, there is never confusion. Some additional metadata can be encoded in filenames such as: `1-clean_data.R`, `2-transform-data.R`, `3-plot-data.R`.

Rec.7 Identify Dependencies. Some packages might require additional external dependencies, as discussed in previous sections. In the best-case scenario, a researcher is actively preserving their environment using Docker or a similar service. Then, all that is necessary is to extend their image with a new layer containing the dependency. If they do this with a Dockerfile, they can always go back and add or remove dependencies as necessary. If a user cannot use Docker, the next best thing is to write a script that installs the necessary dependencies, `0-install-dependencies.R`.

Rec.8 Run Computations from Start to Finish. It is an easy mistake to write a successful computation and then create a `run_all.R` script that executes

everything but crashes. If a `run_all.R` uses the `source()` function, then all scripts executes *in the same environment.* Variables and functions could potentially interfere with each other. Alternatively, especially with notebooks, sometimes scripts only work because of the variables in the existing environment. As a sanity check, it is highly advantageous to run an entire dataset from start to finish often. Using make or targets is the preferred solution and helps automate this process, separate language environments, and does not repeat unnecessary computation. If a computation takes too long, so this is not feasible, consider breaking the analysis into pieces so that lengthy steps do not have to repeat.

Rec.9 **Collect Provenance**. A tool such as RDataTracker or noWorkflow can help verify results later and provide debugging information while writing a computational experiment. If a computational environment breaks or becomes unusable at some point, it can also help recreate a new one.

Chapter 6

Related Work

There are a few reproducibility tools that researchers have already created and are in general use. Notably, they all focus on proactive reproducibility, with some minor exceptions. These tools either create a computational environment, help preserve a computational environment, or in rare exceptions, create a new computational environment without addressing retroactive reproducibility issues. We present this previous work and discuss how RaaS addresses challenges they cannot.

6.1 Reproducibility Platforms

Before scientists start writing a computational experiment, they can choose a proactive reproducibility platform to preserve and maintain the computational environment throughout the study duration and after it concludes. Code Ocean [14], The Whole Tale [11], Renku[1], and GenePattern (for Bioinformatics)[51] are all services that create and manage computational environments for researchers, and there are many benefits for scientists who use this approach. These platforms facilitate collaboration, sharing, dependency management, and publication. These benefits also include the preservation of the computational environment, such as a Docker container. The fundamental problem with this approach is that it requires that scientists adopt an entire work environment. While this is sometimes possible at the beginning of a project, it is difficult to convince researchers to use an environment with

[1] renkulab.io

which they are unfamiliar, and once a project starts, it is even more challenging to convince them to switch to a new environment. Researchers have previously stated that one reason they do not share code and data is the "time it takes to clean up and document for release" [57]. Most importantly, these proactive approaches do not address reproducibility for the thousands of experiments already publicly released. While these platforms create new environments, other reproducibility tools typically rely on an existing computational environment.

6.2 Environment Preservation

Scientists who have finished a computational experiment and still have access to the original computational environment can use a proactive tool to preserve that environment. Containerit [43] is a tool for R that takes an R script or R session information and creates a Dockerfile, the set of instructions that tells the Docker service how to construct a container image. It also utilizes the sysreqs database as RaaS does to find system requirements. However, unlike RaaS, they only query the database with the top-level R packages they identify. Some R packages depend not only on system libraries but also on other R packages that could have separate system dependencies, as shown in Figure 3.4. The script they use to install the packages also continues regardless of whether all the packages install successfully. Since containerit never executes the computational experiments in the new environment, the user must execute each script themself manually. If a researcher is not aware of these issues, they could unknowingly distribute a misconfigured environment. In contrast, RaaS not only executes all of the necessary scripts in a dataset but collects provenance and can process entire datasets at a time as opposed to just a single script or session.

ReproZip [13] is another proactive tool facilitating post-experiment reproducibility. This tool packages an experiment by tracing system calls to identify all files needed for the experiment to run. It then provides methods to unpack the experiment into a Docker container or Vagrant virtual machine. CDE is a similar method [21]. CDE traces system calls to identify the computational environment's needed files, then places them in a package the user can distribute. Unlike ReproZip, CDE cannot export computational experiments to a virtualized environment and instead

relies on the environment being a 32- or 64-bit Linux system. Since these methods require executing the initial analysis, the researcher can use them only when they can still run the experiment. However, these tools present a limitation even within its intended use case. Studies can sometimes span months and even years. The researcher could write one half of the analysis using one software version and the second half using a different version. This difference means that the portion written earlier might not execute with the tools by the time they return to it. The most significant difference between these tools and RaaS is their entirely proactive approach. These tools will not work unless the user is already capable of correctly executing the computational experiment. Also, since RaaS is a service, it is not tied to a single operating system like CDE. Although it is important to remember that the images RaaS generates are typically Linux-based.

A lighter-weight alternative to these tools is a project management system such as `renv` [59] or `conda`[3]. While these tools might not be able to preserve an environment entirely like Docker, they can create a record of the state of the environment. Using that record with the same software can allow the user to recreate the original environment on different devices. For example, with `renv`, a user can create a `renv.lock` file that contains information about the state of the environment such as R version, names of packages, and versions of packages. Then, on another device, they can use the `renv::restore` function with the `renv.lock` file to recreate the environment. `Conda` has similar functionality with environment.yml files. Similar to CDE and ReproZip, these tools assume that the user is still in possession of the original computational environment, *and* that they created that environment using `renv` or `conda`. This approach is an excellent practice for researchers creating a new computational experiment, but unlike RaaS, it cannot help users reproduce an existing dataset that is missing a computational environment.

6.3 Retroactive Reproducibility

Existing reproducibility tools effectively provide reproducibility *if* researchers are proactive and take concrete steps before publishing their analyses. However, not all scientists are proactive, and many computational experiments are already archived. It is in this space that there is a lack of support for facilitating reproducibility.

Reproducing one of these experiments can require significant time and resources [40].

Only a few tools have recently started to take the steps necessary to address retroactive reproducibility. Repo2docker [48] is a service that creates a custom environment for a dataset in Python, R, or Julia using Docker. However, it creates the environment based on a `requirements.txt` file or some other similar configuration file. Pimentel et al. [47] discovered significant reproducibility problems with Python notebooks, even those with a `requirements.txt` file. If that file is incorrect or there are necessary system libraries to install, the environment might not execute the experiment. Additionally, problems with file path names cause execution errors, even if repo2docker correctly reproduced the computational environment. So while this tool does retroactively recreate computational environments, it relies on the researcher to put in the initial effort to ensure reproducibility through a proactive tool or their own documentation. RaaS, in contrast, currently assumes no configuration files are present and creates environments from scratch.

ReproduceMeGit uses repo2docker to assess GitHub repositories' reproducibility by creating an environment and visualizing successful and failed executions [54]. However, they do not appear to have used this tool to performs a large-scale study such as Pimentel et al.'s evaluation. The Dataverse platform has taken a similar approach to repo2docker to retroactively increase the reproducibility of already published results [58], by integrating Dataverse with web platforms such as The Whole Tale to create environments for the archived analyses. These approaches provide the added benefit of having persistent identification for computational experiments. However, they have the same limitations as repo2docker; a researcher must create the original experiment with reproducibility in mind for these retroactive tools to be most effective. Unfortunately, as we saw during our evaluation, this is rarely the case.

With these tools in mind, we have reproducibility services for many steps of the publication process. Researchers who are just starting can use an online environment management service or create a local environment using `renv` or `conda`. Alternatively, if they just finished their computational experiment, they can preserve the environment using a tool such as ReproZip or containerit. Finally, we have tools that retroactively assess published experiments but will not address

the problems they have. RaaS automatically fixes specific issues that arise from retroactive reproducibility and notifies the user of others. Therefore, it can facilitate reproducibility for previously published datasets and help during the publication pipeline by facilitating reproducibility for researchers who lost their original computational environment prior to or during publication.

Chapter 7

Conclusion

Researchers are making progress towards reproducible computational experiments. They have released robust tools that can integrate with FAIR data management services. Unfortunately, we identified a class of yet-to-be-supported reproducibility issues that we categorize as *retroactive reproducibility*. Researchers publish thousands of computational experiments every year, written in various languages, that are non-trivial to reproduce. With our evaluation, we replicate prior work [12] and further extend it by evaluating our system we designed specifically to address retroactive reproducibility. Through our evaluation of Dataverse, we observed that the duration of time these experiments are online does not appear to affect their reproducibility rates. While published experiments are prone to "software rot" as is all code, this degradation is not the leading cause for reproducibility failures.

The most common issues with retroactive reproducibility are related to the computational environment. Researchers use a significantly diverse set of packages in their computational experiments. Out of the almost 2000 datasets we evaluated, most packages appeared in four or fewer datasets. Thus, it is not safe to assume that a computational experiment can run because future researchers will have access to the packages used. The fact that repositories that host R packages will drop support over time exacerbates this issue since researchers may have to relocate that package. Additionally, many packages require dependencies in the form of other R packages or system libraries that can all be susceptible to the same problem. If a researcher fails to preserve a computational experiment's environment, it is safe to

assume that it will not be reproducible.

We presented RaaS, to the best of our knowledge, the first service explicitly designed with retroactive reproducibility in mind. RaaS analyzes datasets to recreate a sufficient computational environment for them. It will preprocess and statically analyze source code to increase the chances of a successful reproduction while gathering information necessary to construct the computational environment. Then RaaS creates and preserves a new computational environment at the same time by creating a Docker image where it installs all of the identified dependencies. Since it collects provenance while executing the computational experiment as part of the build process, any future researcher who uses the image can examine what occurred during the experiment while it ran.

We demonstrated that RaaS fixes many of the most common types of errors found in R scripts on Harvard's Dataverse. While many of these errors were masking more complex errors, RaaS, in its current state, can reveal these reproducibility challenges automatically without the user having to fix common mistakes themselves. Currently, RaaS's ability to detect certain edge cases in lines of code is one of its most significant limitations. RaaS currently struggles to process scripts from researchers who load files or libraries in loops, provide a version-specific library location, or dynamically create file paths. These limitations are also in addition to any limitations inherent in Docker since it is the critical foundation for RaaS.

Our results demonstrate RaaS's value as a piece of the publication pipeline. Researchers might unintentionally neglect their computational environment as they prepare for publication. A service like RaaS will prevent them from distributing a dataset with errors since it will fix some and notify the researchers of the remaining errors. This method ensures that they will always distribute a reproducible computational experiment.

In the future, we can supplement this process by adding more interactivity to RaaS, supporting proactive artifacts, and simplifying our language separation in our codebase. With added interactivity, RaaS as a piece of the publication pipeline can act less as a gatekeeper and more as a reproducibility assistant. It will notify users of problems mid-processing and allow them to fix these errors rather than restart. By supporting proactive artifacts such as requirements.txt or renv.lock files, we can encourage proactive techniques and increase the likelihood of a suc-

cessfully created Docker image. Finally, simplifying our language separation will make it easier for developers to add new languages and make it possible for us to support multi-language datasets easily. However, even without these features, RaaS can facilitate reproducibility for datasets that previously researchers would have to address manually.

Scientists can use RaaS to reproduce computational experiments that previously required significant time and resources to complete. This process also provides future reproducibility and allows them to examine and modify the computational experiment without fear of losing the original results. With this ability, it is simpler to verify their peers' results and strengthen our confidence in the scientific body of knowledge.

Bibliography

[1] C. Allen and D. M. Mehler. Open science challenges, benefits and tips in early career and beyond. *PLoS biology*, 17(5):e3000246, 2019. → page 1

[2] M. Altman, L. Andreev, M. Diggory, G. King, D. Kiskis, E. Kolster, and S. Verba. A digital library for the dissemination and replication of quantitative social science research. *Social Science Computer Review*, 19: 458–470, 2001. URL http://gking.harvard.edu/files/gking/files/vdcwhitepaper.pdf. → page 17

[3] Anaconda Inc. Anaconda software distribution, 2020. URL https://docs.anaconda.com/. → pages 2, 16, 77, 89

[4] M. Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533(7604): 452–454, 2016. doi:10.1038/533452a. → page 1

[5] M. Bayer. Sqlalchemy. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. aosabook.org, 2012. URL http://aosabook.org/en/sqlalchemy.html. → page 35

[6] K. Belhajjame, R. B'Far, J. Cheney, S. Coppens, S. Cresswell, Y. Gil, P. Groth, G. Klyne, T. Lebo, J. McCusker, et al. Prov-dm: The prov data model. *W3C Recommendation*, 2013. → page 23

[7] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi:10.1137/141000671. → page 29

[8] R. Bivand, T. Keitt, and B. Rowlingson. *rgdal: Bindings for the 'Geospatial' Data Abstraction Library*, 2020. URL https://CRAN.R-project.org/package=rgdal. R package version 1.5-8. → page 41

[9] E. Boose. *provSummarizeR: Summarizes Provenance Related to Inputs and Outputs of a Script or Console Commands*, 2020. URL https://github.com/End-to-end-provenance. R package version 1.4.2. → page 24

[10] O. Brand, E. Fong, R. Sheehan, J. Wonsil, and E. Boose. *provDebugR: A Time-Travelling Debugger*, 2021. URL https://CRAN.R-project.org/package=provDebugR. R package version 1.0. → pages 24, 83

[11] A. Brinckman, K. Chard, N. Gaffney, M. Hategan, M. B. Jones, K. Kowalik, S. Kulasekaran, B. Ludäscher, B. D. Mecum, J. Nabrzyski, V. Stodden, I. J. Taylor, M. J. Turk, and K. Turner. Computing environments for reproducibility: Capturing the "whole tale". *Future Generation Computer Systems*, 94:854 – 867, 2019. ISSN 0167-739X. doi:https://doi.org/10.1016/j.future.2017.12.029. URL http://www.sciencedirect.com/science/article/pii/S0167739X17310695. → page 87

[12] C. Chen. Coding betteR: Assessing and improving the reproducibility of R-based research with containR. Undergraduate thesis, Harvard University, 2018. → pages 2, 3, 7, 12, 25, 45, 47, 48, 50, 51, 53, 57, 59, 60, 61, 68, 69, 70, 92

[13] F. Chirigati, R. Rampin, D. Shasha, and J. Freire. Reprozip: Computational reproducibility with ease. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2085–2088. ACM, 2016. ISBN 978-1-4503-3531-7. doi:10.1145/2882903.2899401. → pages 21, 88

[14] A. Clyburne-Sherin, X. Fei, and S. A. Green. Computational reproducibility via containers in social psychology. *Meta-Psychology*, 3, 2019. → page 87

[15] M. J. Denny and A. Spirling. *preText: Simple, Consistent Wrappers for Common String Operations*, 2020. URL github.com/matthewjdenny/preText. R package version 0.7.1. → pages 14, 71

[16] K. Diethelm. The limits of reproducibility in numerical simulation. *Computing in Science & Engineering*, 14(1):64–72, 2012. doi:10.1109/mcse.2011.21. → page 9

[17] Docker Inc., 2021. URL https://hub.docker.com/. → page 17

[18] A. M. Ellison. Repeatability and transparency in ecological research. *Ecology*, 91(9):2536–2539, 2010. → pages 1, 9, 14, 45

[19] GDAL/OGR contributors. *GDAL/OGR Geospatial Data Abstraction software Library*. Open Source Geospatial Foundation, 2021. URL https://gdal.org. → pages 14, 19, 31, 41

[20] R. Gentleman, E. Whalen, W. Huber, and S. Falcon. *graph: graph: A package to handle graph data structures*, 2019. R package version 1.64.0. → page 48

[21] P. Guo. Cde: A tool for creating portable experimental software packages. *Computing in Science & Engineering*, 14(4):32–35, 2012. → pages 21, 88

[22] P. J. Guo. *Software tools to facilitate research programming*. Stanford University, 2012. → pages 10, 11

[23] HashiCorp. Vagrant, 2021. URL www.vagrantup.com. → page 2

[24] R. D. Hipp. SQLite, 2020. URL https://www.sqlite.org/index.html. → page 35

[25] K. Hornik. R FAQ, 2020. URL https://CRAN.R-project.org/doc/FAQ/R-FAQ.html. → pages 14, 42

[26] W. Huber, V. J. Carey, R. Gentleman, S. Anders, M. Carlson, B. S. Carvalho, H. C. Bravo, S. Davis, L. Gatto, T. Girke, R. Gottardo, F. Hahne, K. D. Hansen, R. A. Irizarry, M. Lawrence, M. I. Love, J. MacDonald, V. Obenchain, A. K. Ole's, H. Pag'es, A. Reyes, P. Shannon, G. K. Smyth, D. Tenenbaum, L. Waldron, and M. Morgan. Orchestrating high-throughput genomic analysis with Bioconductor. *Nature Methods*, 12(2):115–121, 2015. URL http://www.nature.com/nmeth/journal/v12/n2/full/nmeth.3252.html. → pages 14, 42

[27] M. B. Kery and B. A. Myers. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–29. IEEE, 2017. → pages 10, 11

[28] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. development team. Jupyter notebooks - a publishing format for reproducible computational workflows. In F. Loizides and B. Scmidt, editors, *Positioning and Power in Academic Publishing:*

Players, Agents and Agendas, pages 87–90, Netherlands, 2016. IOS Press. URL https://eprints.soton.ac.uk/403913/. → pages 1, 57

[29] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017. → page 76

[30] W. M. Landau. The targets r package: a dynamic make-like function-oriented pipeline toolkit for reproducibility and high-performance computing. *Journal of Open Source Software*, 6(57):2959, 2021. URL https://doi.org/10.21105/joss.02959. → pages 16, 78, 85

[31] D. T. Lang, R. Peng, D. Nolan, and G. Becker. *CodeDepends: Analysis of R Code for Reproducible Research and Code Comprehension*, 2018. URL https://CRAN.R-project.org/package=CodeDepends. R package version 0.6.5. → page 39

[32] B. Lerner and E. Boose. RDataTracker: Collecting provenance in an interactive scripting environment. In *USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, 2014. doi:10.1007/978-3-319-16462-5_36. → pages 21, 23, 83

[33] B. Lerner, E. Boose, and L. Perez. Using introspection to collect provenance in R. *Informatics*, 5(12), 2018. URL http://www.mdpi.com/2227-9709/5/1/12/htm. → page 42

[34] E. C. McKiernan, P. E. Bourne, C. T. Brown, S. Buck, A. Kenall, J. Lin, D. McDougall, B. A. Nosek, K. Ram, C. K. Soderberg, et al. Point of view: How open science helps researchers succeed. *elife*, 5:e16800, 2016. → page 1

[35] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014. → pages 2, 3, 18

[36] R. C. T. Microsoft. *Microsoft R Open*. Microsoft, Redmond, Washington, 2017. URL https://mran.microsoft.com/. → page 16

[37] J.-L. D. MORLHON. Docker hub image retention policy delayed, subscription updates, Oct 2020. URL https://www.docker.com/blog/docker-hub-image-retention-policy-delayed-and-subscription-updates/. → pages 17, 34

[38] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noWorkflow: Capturing and analyzing provenance of scripts. In B. Ludäscher and B. Plale, editors, *Provenance and Annotation of Data and Processes*, pages 71–83. Springer International Publishing, 2015. ISBN 978-3-319-16462-5. → pages 21, 24

[39] K. Müller. *here: A Simpler Way to Find Your Files*, 2020. URL https://CRAN.R-project.org/package=here. R package version 1.0.1. → pages 16, 85

[40] National Academies of Sciences, Engineering, and Medicine. *Reproducibility and Replicability in Science*. The National Academies Press, Washington, DC, 2019. ISBN 978-0-309-48616-3. doi:10.17226/25303. URL https://www.nap.edu/catalog/25303/reproducibility-and-replicability-in-science. → pages 8, 90

[41] Netherlands' EU, Presidency. Amsterdam call for action on open science. *Open Science, from Vision to Action*, 2016. → page 1

[42] K. Ngo. *provExplainR: Compare two provenance collections to explain different script outputs*, 2019. URL https://github.com/End-to-end-provenance. R package version 0.1.0. → page 24

[43] D. Nüst and M. Hinz. containerit: Generating Dockerfiles for reproducible research with R. *Journal of Open Source Software*, 4(40):1603, 8 2019. doi:10.21105/joss.01603. URL https://doi.org/10.21105/joss.01603. → page 88

[44] E. Pebesma. Simple Features for R: Standardized Support for Spatial Vector Data. *The R Journal*, 10(1):439–446, 2018. doi:10.32614/RJ-2018-009. URL https://doi.org/10.32614/RJ-2018-009. → pages 14, 31, 41

[45] E. Pebesma, T. Mailund, and J. Hiebert. Measurement units in R. *R Journal*, 8(2):486–494, 2016. doi:10.32614/RJ-2016-061. → page 32

[46] E. J. Pebesma and R. S. Bivand. Classes and methods for spatial data in R. *R News*, 5(2):9–13, November 2005. URL https://CRAN.R-project.org/doc/Rnews/. → page 41

[47] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 IEEE/ACM*

16th International Conference on Mining Software Repositories (MSR), pages 507–517. IEEE, 2019. → pages 2, 7, 12, 71, 90

[48] Project Jupyter. repo2docker, 2020. URL https://repo2docker.readthedocs.io/en/latest/. → page 90

[49] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2020. URL https://www.R-project.org/. → pages 2, 26, 29

[50] Redis Labs. Redis, 2020. URL redis.io. → page 35

[51] M. Reich, T. Liefeld, J. Gould, J. Lerner, P. Tamayo, and J. P. Mesirov. Genepattern 2.0. *Nature genetics*, 38(5):500–501, 2006. → page 87

[52] P. D. S. Richard M. Stallman, Roland McGrath, 2020. URL https://www.gnu.org/software/make/manual/. → pages 16, 51, 77, 78, 85

[53] A. Ronacher. Flask, 2020. URL flask.palletsprojects.com. → page 34

[54] S. Samuel and B. König-Ries. Reproducemegit: A visualization tool for analyzing reproducibility of jupyter notebooks, 2020. → page 90

[55] D. Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer, New York, 2008. URL http://lmdvr.r-forge.r-project.org. ISBN 978-0-387-75968-5. → page 74

[56] A. Solem. Celery, 2020. URL docs.celeryproject.org. → page 35

[57] V. Stodden. The scientific method in practice: Reproducibility in the computational sciences. *SSRN Electronic Journal*, 2010. doi:10.2139/ssrn.1550193. → pages 7, 88

[58] A. Trisovic, P. Durbin, T. Schlatter, G. Durand, S. Barbosa, D. Brooke, and M. Crosas. Advancing computational reproducibility in the dataverse data repository platform. In *Proceedings of the 3rd International Workshop on Practical Reproducible Evaluation of Computer Systems*, P-RECS '20, page 15–20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379779. doi:10.1145/3391800.3398173. URL https://doi.org/10.1145/3391800.3398173. → pages 2, 17, 90

[59] K. Ushey. *renv: Project Environments*, 2021. URL https://CRAN.R-project.org/package=renv. R package version 0.13.2. → pages 2, 16, 42, 51, 77, 89

[60] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697. → page 29

[61] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi:10.25080/Majora-92bf1922-00a. → page 31

[62] H. Wickham. *stringr: Simple, Consistent Wrappers for Common String Operations*, 2019. URL https://CRAN.R-project.org/package=stringr. R package version 1.4.0. → page 71

[63] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, et al. The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3(1):1–9, 2016. → pages 2, 12