An Introduction to Rocker: Docker Containers for R

by Carl Boettiger, Dirk Eddelbuettel

Abstract We describe the Rocker project, which provides a widely-used suite of Docker images with customized R environments for particular tasks. We discuss how this suite is organized, and how these tools can increase portability, scaling, reproducibility, and convenience of R users and developers.

Introduction

The Rocker project was launched in October 2014 as a collaboration between CB & DE to provide high-quality Docker images containing the R environment (Boettiger and Eddelbuettel, 2014). Since that time, the project has seen both considerable uptake in the community and substantial development and evolution. Here we seek to document the project's objectives and uses.

What is Docker?

Docker is a popular open source tool to create, distribute, deploy, and run software applications using *containers*. Containers provide a virtual environment (see Clark et al. (2014) for an overview of common virtual environments) requiring all operating-system components an application needs to run: code, runtime, system tools, system runtime. Docker containers are lightweight as they share the operating system kernel, starting instantly, using a layered filesystem which minimizes disk footprint and download time, and are built on open standards that run on all major platforms (Linux, Mac, Windows), and provide an added layer of security by running an application in an isolated environment (Docker, 2015).

Rocker organization and workflow

The Rocker project consists of a suite of images built automatically by and hosted on the Docker Hub, `https://hub.docker.com/r/rocker`. Source Dockerfiles, supporting scripts and documentation are hosted on GitHub under the organization `rocker-org`, `https://github.com/rocker-org`. The issues tracker and pull requests are used for community input, discussions and contributions to these images. The rocker project wiki provides a place to synthesize community-contributed documentation, use-cases, and other knowledge about using the Rocker images.

Images in the Rocker Project

The Rocker project aims to provide a small core of Docker images that serve as convenient 'base' images on which other users can build custom R environments by writing their own Dockerfiles, while also providing a 'batteries included' approach of images that can be used out of the box. The challenges of balancing diverse needs driven by very different use cases (discussed in more detail below) against the overarching goals of creating images that are still sufficiently light-weight, easy to use and easy to maintain is a difficult art. The implementation in both individual Rocker images and image stacks can never perfect that balance for everyone, but today reflects the considerable community input and testing over past few years.

All Rocker images are based on the Debian Linux distribution. The Debian platform provides a small base image, the well-known `apt` package management system and rich ecosystem of software libraries, making it the base image of choice for Docker images, including many of the "official" images maintained by Docker's own development team. The Debian platform is also perhaps the best supported Linux platform within the R community, including an active `r-sig-debian` listserve. The relatively long period between stable Debian releases (roughly two years recently) means that software in the `debian:stable` releases can lag significantly behind current releases of popular software, including R. More recent versions of packages can be found in the pre-release distribution, `debian:testing`, while the very latest binary builds can be found on `debian:unstable`. The Rocker project can be largely divided into two stacks which address different needs, defined by which Debian distribution they are based on. The first stack is based on `debian:testing`, the second, more recently introduced stack is based only on `debian` stable releases. These stacks have different aims and thus provide different images, as shown in Tables 1 & 2.

The debian:testing-based images

The debian:testing stack aims to makes the most efficient use of upstream builds – the precompiled .deb binaries provided by the Debian repositories. It is both quicker and easier to install software from binaries, since the package manager (apt) manages the necessary (binary) dependencies and bypasses the time-consuming process of compiling from source. Basing this stack on debian:testing means that much more recent versions of commonly-used libraries and compilers are available as binaries than would be found in a Debian stable release. In order to provide access to the most recent available binaries, this stack uses apt-pinning (Debian Project, 2017) to allow the apt package manager to also install binaries from debian:unstable, which represents the most recent, bleeding edge of packages built for Debian when necessary. For instance, the r-base image provided by Rocker installs the most recent version of R as a binary from Debian unstable. Similarly, recent versions of many popular R packages can also be installed through the package manager, e.g. apt-get install r-cran-xml. This can be particularly helpful for packages with external system dependencies (such as libxml2-dev in this example) which cannot be installed from the R console as they are system dependencies rather than R packages installed from within R. We should note, however, that only about 500 of the over 11000 CRAN packages are available as Debian packages.

As the names testing and unstable imply, this approach is not without challenges. The particular version of any given package can change as packages move from unstable into testing. New versions are sent to unstable during the normal course of Debian development. This can occasionally break an previously working installation command in a Dockerfile until the maintainer redirects the package manager to install a the package from the unstable sources that could previously be installed from testing, or vice versa (using the -t option in apt, see Box 1 on apt-pinning.) That said, packages only migrate from unstable to testing after a period of several days—and if the migration and installation of the particular version is free of interactions with other packages in their dependency graph. That way, unstable serves as validation lab which leaves testing reasonably stable yet current.

Relative to stable, the testing stack thus offers the following advantages:

1. These Dockerfiles are easy to develop and extend because almost all software can be installed through the package manager
2. These Dockerfiles always install the most recent available software
3. These Dockerfiles can almost always build relatively quickly

and these down-sides:

1. These Dockerfiles require occasional minor maintenance to ensure successful builds. (e.g. changing an installation directive from unstable to testing or vice versa)
2. The resulting images are inherently dynamic: rebuilding the same Dockerfile months or years apart will generate images with significantly different versions of software installed.
3. The use of apt pinning may be unfamiliar to some users, where the user must ultimately be responsible to ensure compatibility (Debian Project, 2017).

Images overview

The debian:testing-based stack currently includes seven images actively maintained by the Rocker development team (Table 1). r-base builds on debian:testing, and the other six in the stack each build directly from r-base. The r-base image is unique in that it is designated as the official image for the R language by the Docker organization itself. This official image is reviewed and then built by employees of Docker Inc based on a Dockerfile maintained by the Rocker team. Consequently, users should refer to this image in Docker commands without an organization namespace, e.g. docker pull r-base to access the official image. All other images in the Rocker project are not individually reviewed and built by Docker Inc and must be referenced using the rocker namespace, e.g. docker pull rocker/r-devel.

Several of the images in this stack are oriented towards the R development community. r-devel, drd, r-devel-san & r-devel-ubsan-clang all add a copy of the development version of R side-by-side the current release of R provided by r-base. On these images, the development version is aliased to RD to distinguish from the current release, R. As the names suggest, each provide slightly different configurations. Of particular interest are the images providing development R built with support for C/C++ address and undefined-behavior sanitizers, which are somewhat more difficult to configure (Eddelbuettel, 2014).

As these images focus on developers and/or as base images for custom uses, this stack does include many specific R packages. Additional dependencies and packages can easily be installed from apt. R packages not available in the apt repositories can be installed directly from CRAN using the littler scripts, see *Extending Rocker images* section below.

This stack also include the images `shiny` and `rstudio:testing` that provide Shiny server and RStudio server IDE from RStudio Inc, built on the `r-base` image. RStudio and Shiny are registered trademarks of RStudio Inc, and their use and the distribution of their software in binary form on Docker Hub has been granted to the Rocker project by explicit permission from RStudio. Users should review RStudio's trademark use policy (http://www.rstudio.com/about/trademark/) and address inquiries about further distribution or other questions to permissions@rstudio.com. The Rocker project also provides images with RStudio server and Shiny server in the stable versioned stack.

Build schedule: The official `r-base` image is rebuilt by Docker following any updates to the official `debian` images (roughly every few weeks). The rest of the stack uses build triggers that rebuild the images either whenever `r-base` is updated, or the Dockerfile sources are updated on the corresponding GitHub repository. The only exception in this stack is the `drd` image, which is rebuilt each week by a `cron` trigger.

Table 1: The `debian:testing` image stack

image	description	size	downloads
r-base	official image with current version of R	254 MB	510,000
r-devel	R-devel added side-by-side r-base (using alias `RD`)	1 GB	4,000
drd	lightweight r-devel, built almost daily	571 MB	4,000
r-devel-san	as r-devel, but built with compiler sanitizers	1.1 GB	1,000
r-devel-ubsan-clang	Sanitizers, clang c compiler (instead of gcc)	1.1 GB	525
rstudio:testing	rstudio on debian:testing	1.1 GB	1,000
shiny	shiny-server on r-base	409 MB	63,000

The `debian:stable`-based stack

This stack emphasizes stability and reproducibility of the Docker build. This stack was introduced much more recently (November 2016) in response to considerable user input and requests. The key feature of this stack is the ability to run older versions of R along with the contemporaneous versions of R packages. A user specifies the version desired using an image tag, e.g. `rocker/r-ver:3.3.1` will refer to an image with R version 3.3.1 installed. Omitting the tag is equivalent to using the tag `latest`, which, as the name implies, will always point to an image using the current R release. Thus, users wanting to create downstream Dockerfiles which are based on the current release at time (but will continue to reconstruct the same environment in the future after newer R versions are released) should explicitly include the corresponding version tag, e.g. `rocker/r-ver:3.4.0` at the time of writing, and not the `latest` tag. Users can also run the current development version of R using the tag `devel`, which is built nightly from R-devel sources from `subversion`.

MRAN archives: To facilitate installation of only contemporaneous versions of R packages on these images, the default CRAN mirror from which to install R packages is fixed to a snapshot of CRAN corresponding to the last date for which that version of R was the latest release. These snapshots are provided by the MRAN archive created by Revolution Analytics (now part of Microsoft). It archives daily snapshots of all of CRAN from which a user can install packages with the usual `install.packages()` function (Revolution Analytics, 2017). Users can always override this default by passing any current CRAN repository explicitly. Unlike CRAN, Bioconductor only updates its repositories through bi-annual releases aligned to R's spring release schedule. Thus, Bioconductor packages can be installed in the usual way using `bioclite`, which automatically selects the Bioconductor release corresponding to the version of R in use.

Version tags: The version tags are propagated throughout this stack: *e.g.*, `rocker/tidyverse:devel` will provide the currently released versions of the R packages in the `tidyverse` (Wickham, 2017) installed on the nightly build of R-devel. Developers building packages on this stack are encouraged to tag their images accordingly as well. Table 3 indicates which versions of R are currently available in the stack, going back to `3.1.0`. While older versions may be added to the stack at a later date, we note that the MRAN snapshots began in starting 2014-09-17 and thus go back only to the R `3.1` era. Each tag must be built from a separate Dockerfile, enabling minor differences in the build instructions to accommodate changing dependencies. Dockerfiles for past versions (e.g. $< 3.4.0$ currently) are intended to remain static over the long term, while the tag for the current version, `latest`, and `devel` may be tweaked to accommodate new features or dependencies.

Installation: In this stack, the desired version of R is always built directly from source rather than the `apt` repositories. Compilers and dependencies are still installed from the stable `apt` repositories, and thus lag behind the more recent versions found in the `testing` stack. Version tags `3.3.3` and older are based on the Debian 8.0 release, code-named `jessie`, while `3.4.0`, `devel`, and `latest` are based on

Debian 9.0, `stretch`, released 2017-06-17, and thus have access to much newer versions of common system dependencies and compilers. Dependencies needed to compile R that are not required at runtime are removed once R is installed, keeping the base images light-weight for faster download times. While most system dependencies required by common R packages can still be installed from the `apt` repositories, occasionally a more recent version must be compiled from source (e.g. the Gibbs Sampling library, JAGS, and the geospatial toolkit GDAL, must both be compiled from source on `debian:jessie` images). In this stack, users should never install R packages, using `apt`, since this will install a second, older version of R from the Debian repositories and a dated version of the R package (since any `r-cran-pkgname` package in the Debian repositories will depend on `r-base` in `apt` as well).

Build schedule: All images are built automatically from their corresponding Dockerfiles (found in the GitHub repositories `rocker-org/rocker-versioned` and `rocker-org/geospatial`). A `cron` job sends nightly build triggers to Docker Hub to rebuild the `latest` and `devel` tagged images throughout the stack. To decrease load on the hub, build triggers for the numeric version tags are sent monthly. Although the Dockerfiles for older R versions installs an almost-identical software environment every time, the monthly rebuilding of these images on Docker Hub ensures they continue to receive Debian security updates from upstream, and proves the build recipe still executes successfully. Note that rebuilding images with software from external repositories never produces a bit-wise identical image, and thus the image identifier hash will change at each build.

Images overview

In this stack, each image builds on the previous image, rather than all other images building directly on the base image, as in the `testing` stack. Table 2 lists the names and descriptions of the five images in this stack, along with image size and approximate download counts from Docker Hub. Sizes reflect (compressed) cumulative size: a user who has already downloaded the most recent version of `r-ver` and then pulls a copy of `rstudio` image will only need to download the additional 115 MB in the `rstudio` layers and not the full 334 MB listed. This linear design limits flexibility (no option for `tidyverse` without `rstudio`) but simplifies use and maintenance. While no single environment will be optimal for everyone, both the packages selected in this stack and the stack ordering reflect considerable community input and tuning.

The `rstudio` image includes a lightweight, easy-to-use and docker-friendly `init` system, s6 (Bercot, 2017) for running persistent services, including the RStudio server. This system provides a convenient way for downstream Dockerfile developers to add additional persistent services (such as an `ssh` server) to a single container, or additional start-up or shutdown scripts that should be run when an instance starts up or shuts down. The `rstudio` image uses such a start-up script to configure user settings such as login password and permissions through environmental variables at run time.

The `tidyverse` image contains all required and suggested dependencies of the commonly-used `tidyverse` and `devtools` R packages, including external database libraries (e.g. MariaDB and PostgreSQL). Users should consult the package Dockerfiles or `installed.packages()` list directly for a complete list of installed packages. The `verse` library adds commonly-used dependencies, notably a large but not comprehensive LaTeX environment and Java development libraries. Previously the Rocker project provided the image `hadleyverse` which was since divided into `tidyverse` and `verse` through community input.

Table 2: The `rocker-versioned` stack of images

image	description	size	downloads
r-ver	Version-stable base R & src build tools	219 MB	2,000
rstudio	Adds rstudio	334 MB	285,000
tidyverse	Adds tidyverse & devtools	656 MB	14,000 (+46,000 as `hadleyverse`)
verse	Adds java, tex & publishing-related packages	947 MB	4,000
geospatial	Adds geospatial libraries	1.3 GB	1,000

Table 3: Available tags in the `rocker-versioned` stack.

tag	apt repos	MRAN date	Build frequency	images with tag
devel	`stretch`	current date	nightly	`r-ver`, `rstudio`, `tidyverse`, `verse`, `geospatial`
latest	`stretch`	current date	nightly	`r-ver`, `rstudio`, `tidyverse`, `verse`, `geospatial`

tag	apt repos	MRAN date	Build frequency	images with tag
3.4.1	stretch	current date	monthly	r-ver, rstudio, tidyverse, verse, geospatial
3.4.0	stretch	current date	monthly	r-ver, rstudio, tidyverse, verse, geospatial
3.3.3	jessie	2017-04-21	monthly	r-ver, rstudio, tidyverse, verse, geospatial
3.3.2	jessie	2017-03-06	monthly	r-ver, rstudio, tidyverse, verse, geospatial
3.3.1	jessie	2016-10-31	monthly	r-ver, rstudio, tidyverse, verse, geospatial
3.3.0	jessie	2016-06-21	monthly	r-ver
3.2.0	jessie	2015-06-18	monthly	r-ver
3.1.0	jessie	2014-09-17	monthly	r-ver

Several images in the `rocker-versioned` stack can be customized on build when built locally (rather than pulling prebuilt images from Docker Hub) by using the `--build-arg` option of `docker build`. In the `r-ver` image, users can set `R_VERSION`, `BUILD_DATE` (MRAN default snapshot). In the `rstudio` image users can set `RSTUDIO_VERSION` (otherwise defaults to the most recent version), and the `PANDOC_TEMPLATES_VERSION` .

This stack also makes use of Docker metadata labels defined by `http://schema-label.org`, indicating image `license` (GPL-2.0), `vcs-url` (GitHub repository), and `vendor` (Rocker Project). These metadata can be altered or extended in downstream images.

Design principles & use cases

Portability: From laptop to cloud

One common use case for Rocker containers is to provide a fast and reliable mechanism to deploy a custom R environment to a remote server, such as Amazon Web Services Elastic Compute (AWS EC2), DigitalOcean, NSF's Jetstream servers (Stewart et al., 2015), or private or institutional server hardware. Lawrence Berkeley National Labs (LBNL) has made `Singularity`: software which can import and run Docker containers without root privileges, as most cluster administrators require (Lawrence Berkeley National Laboratories, 2017b).
Another LBNL project, *Shifter*, allows Docker containers to be deployed on leadership-scale high performance computing platforms (Lawrence Berkeley National Laboratories, 2017a).

Importantly, Rocker containers are just as easy to run on locally on most modern laptops by first installing the appropriate Docker distribution for Windows, MacOS, and Linux-based operating systems. By sharing volumes with the local host, users can still manipulate files with familiar, native tools while performing computation through a reproducible, containerized environment (Boettiger, 2015). Being able to test code in a predictable, pre-configured R environment on a local machine and to then run the same code in an identical environment on a remote server (e.g. for access to greater RAM, more processors, or merely to free up the local machine from a long-running computation) is essential for low-friction scaling of analysis. Without such containerization, getting code to run appropriately in a remote environment can be a major undertaking, requiring both time and knowledge many would-be users may not have.

For instance, the following code installs Docker on almost any Linux-based server and launches a Rocker container providing the RStudio-server environment over a web interface.

```
wget -qO- https://get.docker.com/ | sh
sudo docker run -d -p 80:8787 -e PASSWORD=<PICK-A-PASSWORD> rocker/rstudio
```

Many academic and commercial cloud providers make it possible to execute such code snippets when an instance is launched, without ever needing to `ssh` into the machine. The user may log into the server merely by pasting its IP address or DNS name into a browser and entering the appropriate password. This provides the user with a familiar, interactive environment running on a remote machine while requiring a minimum of expertise.

This portability is also valuable in an instructional context. Requiring students to install all necessary software on personal laptops can be particularly challenging for short workshops, where download and installation time and troubleshooting across heterogeneous machines can prove time consuming and frustrating for students and instructors alike. By deploying a Rocker image or Rocker-derived image (see *Extensibility*) on a cloud machine, an instructor can easily provide all students

access to the pre-configured software environment using only the browser on their laptops. This strategy has proven effective in our own experience in both workshops and semester-length courses. Similar Docker-based cloud deployments have been scaled to courses of 100s of students, e.g. at Duke (Cetinkaya-Rundel and Rundel, 2017) and UC Berkeley (UC Berkeley, 2017).

Interfaces

An important aspect of the Rocker project design is the ability for users to interact with the software on the container through either an interactive shell session (such as the R shell or a bash shell), or through a web browser accessing the RStudio® Server integrated development environment (IDE). Traditional remote and high-performance computing workflows for R users have usually required the use of `ssh` and a terminal only interface, posing a challenge for interactive graphics and a barrier to users unfamiliar with these tools and environments. Accessing an RStudio instance through the browser removes these barriers. Rocker images include the RStudio-server software pre-installed and configured with the explicit permission of RStudio Inc.

Users of the `rstudio` instances can access a root `bash` shell in an instance running the container using

```
docker exec -ti <container-id> bash
```

which can be useful for administrative tasks such as installing system dependencies. All Rocker images can also be run as an interactive R, RScript or bash shell without running RStudio, which can be useful for batch jobs or for anyone who prefers that environment.

As with any interactive docker instance, users should specify the interactive (`-i`) and terminal (`-t`, here combined with interactive as `-ti`) flags, and specify the desired executable environment (e.g. `R`, though other common options may be `Rscript` or `bash`):

```
docker run --rm -ti rocker/tidyverse R
```

This example shows the use of the `--rm` flag to indicate that the container should be removed when the interactive session is finished.

Details on sharing volumes, managing user permissions, and more can be found on the Rocker website, `https://rocker-project.org`.

Sandboxed

Another feature of Rocker instances is the ability to provide a sandboxed environment, isolated from software and potentially from other data on the machine. Many users are reluctant to upgrade their suite of installed packages, which may break their existing code or even their R environment if the installation goes poorly. However, upgrading packages and/or the R environment is often necessary to run analysis of a colleague or access more recent methods. Rocker instances offer an easy solution. For instance, a user can run R code requiring the most recent versions of R and related packages inside a Rocker container without having to upgrade their local installations first. Conversely, one could use Rocker to run code on an older R release with prior versions of R packages, again without having to make any alteration to one's local R install. Another common use case is to access an instance of R with support for particular options such as using gcc or clang compiler sanitizers (Eddelbuettel, 2014). These requires R itself be built with specialized settings that may not be not available or familiar to many R users on their native system, but can be easily deployed by pulling the Rocker images `rocker/r-devel-san` or `rocker/r-devel-ubsan-clang`.

This sandboxing feature of Docker containers is also valuable in the remote computing context, where it can allow system administrators to provide users with administrative (root) privileges to alter and install software within a container without altering the computational environment of other users on the remote machine. Unlike traditional virtual machines, these containers do not impose a large footprint of reserved resources – a typical host can easily support 100s of containers Docker (2015).

Transparent

Users can easily determine the software stack installed on any Rocker image by examining the associated Dockerfile recipe, which provides a concise, human-readable record of the installation. All Rocker images use automated builds through DockerHub, which also acts as the central, default repository distributing the images. Using automated builds rather than uploading pre-built image

binaries to the Docker Hub avoids the potential for the build not to match the recipe. The corresponding Dockerfile is visible both on the Docker Hub and in the linked GitHub repository of the Rocker project (https://github.com/rocker-org), which provides a transparent versioned history of all changes made to these recipes, as well as documentation, a community wiki, and issues trackers for discussing proposed changes, bugs, improvements to the Dockerfiles and troubleshoot any issues users may encounter.

Community Optimized

Having a shared, transparent computational environment created by a publicly hosted, reproducible recipe facilitates community input into configuration details. R and many of its packages and related software can be configured with a wide range of options, compilers, different linear-algebra libraries and so forth. While this flexibility reflects varying needs, many users rely on default settings which may be optimized more for simplicity of installation than performance. The Rocker recipes reflect significant community input on these choices, as well as the considerable experience and expertise of the R Debian maintainer of 20 years in what configuration options to use. This helps create a more finely tuned, optimized reference implementation of the R environment as well as a platform for comparing and discussing these issues which are often overlooked elsewhere. Issues and Pull Requests on the Rocker repositories on GitHub attest to these discussions and improvements. Widespread use of the Rocker image helps promote both testing of these choices and contributions further tweaking the configuration from many members of the R community.

Versioned

The versioned stack, (`r-ver`, `rstudio`, `tidyverse`, `verse`, and `geospatial`), provides images which are intended to build an identical software stack every time, regardless of the release of new libraries and packages. This can be important for users who need computational reproducibility more than having the latest release of any piece of software, since subsequent releases can alter the behavior of code, introducing errors or altering previous results. Users should specify an R version tag in the Docker image name to request a version stable image, e.g. `rocker/verse:3.4.0`. If no tag is explicitly requested, Docker will provide the image with the tag `:latest`, which will always have the latest available versions of the software (built nightly).

Users building on the version-tagged images will by default use the MRAN snapshot mirror (Revolution Analytics, 2017) associated with the most recent date for which that image was current. This ensures that a Dockerfile building `FROM rocker/verse:3.4.0` will only install R package versions that were available on CRAN on 2017-06-30, that is, the day R v3.4.1 was released. This default can of course be overwritten in the standard R manner, e.g. by specifying a different CRAN mirror explicitly in any command to `install.packages()`, or by adjusting the default CRAN mirror in `options(repo=<CRAN-MIRROR>)` in an `.Rprofile`. Note that the MRAN date associated with the current release (e.g. `3.4.1` at the time of writing) will continue to advance on the Docker-hub image until the next R release. Software installed from `apt-get` in these images will come from the the stable Debian release (`stretch` or `jessie`) and thus not change versions (though it will receive security patches). Packages installed from BioConductor using `bioclite()` utility will also install the version appropriate to the version of R found on the system (the Bioconductor semi-annual release model avoids the need for an MRAN mirror). Users installing packages from GitHub or other sources can request a specific git release tag or hash for a more reproducible build, or adopt a more comprehensive solution such as `packrat` (Ushey et al., 2016); however, `packrat` should not be necessary for just maintaining reproducibility with respect to CRAN and Bioconductor stacks installed on versioned images. A more general discussion of the use of Docker for computational reproducibility can be found in (Boettiger, 2015).

Extensible

Any portable computational environment faces an inevitable tension between the "kitchen sink problem" at one extreme, and the "discovery problem" on the other. A kitchen sink image seeks to accommodate too many use cases in a single image. Such images are inevitably very large and thus slow or difficult to deploy, maintain and optimize. At the other extreme, providing too many specialized images makes it more difficult for a user to discover the one they need. The Rocker project seeks to avoid both of these problems by providing a carefully curated suite of images that an be easily extended by individuals and communities.

To make extensions transparent and persistent, Rocker images can be extended by any user by writing their own Dockerfiles based on an appropriate Rocker image.

The Dockerfiles in the Rocker stack should themselves provide a simple of example of this. A user begins by selecting an appropriate base image for their needs: if the RStudio interface is desired, a user might start with `FROM rocker/rstudio`; an image for testing a particular C code in an R package might use `FROM rocker/r-devel-san`, an image for reproducing a data analysis will probably select a stable version tag in addition to an appropriate base library, e.g.: `FROM rocker/tidyverse:3.4.1` Users can easily add additional software to any running Rocker image using the standard R and Debian mechanisms. Details on how to extend Rocker images can be found at https://rocker-project.org.

Sharing these Dockerfiles can also facilitate the emergence of extensions tuned to particular communities. For instance, the `rocker/geospatial` image emerged from the input of a number of Rocker users all adding common geospatial libraries and packages on top of the existing Rocker images. This coalescence helped create a more fine-tuned image with broad support for a wide range of commonly used data formats and libraries. Other community images are developed and maintained independently of the Rocker project, such as the `popgen` image of population-genetics-oriented software developed by the National Evolutionary Synthesis Center (NESCent). Rocker images are also being used as base Docker images in the NSF sponsored Whole Tale project for reproducible computing (Ludaescher et al., 2017).

Conclusions

Over the past several years, Docker has seen immense adoption across industry and academia. The Open Container initiative (The Linux Foundation: Projects, 2017) now provides an open standard that has further extended this container approach to research environments through projects such as Singularity (Lawrence Berkeley National Laboratories, 2017b), used on clusters without root access, and Shifter (Lawrence Berkeley National Laboratories, 2017a) for use on leadership high performance computing facilities. Containerization promises to solve numerous challenges such as portability and replicability in research computing, which often relies on complex and heterogeneous software stacks (Boettiger, 2015). Yet implementing such environments in containers is not a trivial task, and not all implementations provide the same usability, portability or reproducibility. Here we have detailed the approach taken by the Rocker project in creating and maintaining these environments through an open and community-driven process. This structure of the Rocker project has evolved over three years of operation while drawing in an ever widening base of academic researchers, university instructors and industry users. We believe this overview will be instructive not only to users and developers interested in the Rocker project, but as a model for similar efforts around other environments or domains.

Bibliography

L. Bercot. *s6: skarnet.org's small and secure supervision software suite*, 2017. URL https://skarnet.org/software/s6/. [p4]

C. Boettiger. An introduction to Docker for reproducible research, with examples from the R environment. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015. doi: 10.1145/2723872.2723882. URL https://dl.acm.org/citation.cfm?id=2723882http://arxiv.org/abs/1410.0846. [p5, 7, 8]

C. Boettiger and D. Eddelbuettel. Introducing Rocker: Docker for R, 2014. URL https://ropensci.org/blog/blog/2014/10/23/introducing-rocker. [p1]

M. Cetinkaya-Rundel and C. W. Rundel. Infrastructure and tools for teaching computing throughout the statistical curriculum. *PeerJ Preprints*, 5:e3181v1, Aug. 2017. ISSN 2167-9843. doi: 10.7287/peerj.preprints.3181v1. URL https://doi.org/10.7287/peerj.preprints.3181v1. [p6]

D. Clark, A. Culich, B. Hamlin, and R. Lovett. BCE: Berkeley's Common Scientific Compute Environment for Research and Education. *Proceedings of the 13th Python in Science Conference (SciPy 2014)*, pages 1–8, 2014. [p1]

Debian Project. Apt-preferences overview, 2017. URL https://wiki.debian.org/AptPreferences. [p2]

Docker. What is Docker?, 2015. URL https://www.docker.com/what-docker. [p1, 6]

D. Eddelbuettel. sanitizers, 2014. URL http://dirk.eddelbuettel.com/code/sanitizers.html. [p2, 6]

Lawrence Berkeley National Laboratories. Shifter, 2017a. URL https://github.com/NERSC/shifter. [p5, 8]

Lawrence Berkeley National Laboratories. Singularity, 2017b. URL http://singularity.lbl.gov/. [p5, 8]

B. Ludaescher, K. Chard, M. Turk, V. Stodden, and N. Gaffney. The whole tale: Merging science and cyberinfrastructure pathways, 2017. URL https://wholetale.org/. [p8]

Revolution Analytics. Microsoft R Application Network, 2017. URL https://mran.microsoft.com. [p3, 7]

C. A. Stewart, G. Turner, M. Vaughn, N. I. Gaffney, T. M. Cockerill, I. Foster, D. Hancock, N. Merchant, E. Skidmore, D. Stanzione, J. Taylor, and S. Tuecke. Jetstream: A self-provisioned, scalable science and engineering cloud environment. In *Proceedings of the 2015 XSEDE Conference on Scientific Advancements Enabled by Enhanced Cyberinfrastructure - XSEDE '15*, pages 1–8, New York, New York, USA, 2015. ACM Press. ISBN 9781450337205. doi: 10.1145/2792745.2792774. URL http://dl.acm.org/citation.cfm?doid=2792745.2792774. [p5]

The Linux Foundation: Projects. The Open Container Initiative, 2017. URL https://www.opencontainers.org/. [p8]

UC Berkeley. Curriculum Overview | Division of Data Sciences, 2017. URL http://data.berkeley.edu/education/curriculum-overview. [p6]

K. Ushey, J. McPherson, J. Cheng, A. Atkins, and J. Allaire. *packrat: A Dependency Management System for Projects and their R Package Dependencies*, 2016. URL https://CRAN.R-project.org/package=packrat. R package version 0.4.8-1. [p7]

H. Wickham. *tidyverse: Easily Install and Load 'Tidyverse' Packages*, 2017. URL https://CRAN.R-project.org/package=tidyverse. R package version 1.1.1. [p3]

Carl Boettiger
UC Berkeley
ESPM Department, University of California,
130 Mulford Hall Berkeley, CA 94720-3114, USA
cboettig@berkeley.edu

Dirk Eddelbuettel
Debian and R Projects; Ketchum Trading
Chicago, IL, USA
edd@debian.org