Containers and R—the Rockerverse and beyond

by Daniel Nüst, Robrecht Cannoodt, Dirk Eddelbuettel, Mark Edmondson, Colin Fay, Karthik Ram, Noam Ross, Nan Xiao, Lori Shepherd, Nitesh Turaga, Jacqueline Nolis, Heather Nolis, Hong Ooi, Ellis Hughes, Sean Lopp, Ben Marwick

Abstract The Rocker project provides widely-used Docker images for R across different application scenarios. This articles surveys downstream projects building upon Rocker and presents the current state of R packages for managing Docker images and controlling containers. We also look beyond Rocker to other projects connecting containerisation with R, namely alternative suites of images. These use cases and the variety of applications demonstrate the power of Rocker specifically and container-isation in general. We identified common themes across this diversity: reproducible environments, scalability and effectivity, and portability across clouds.

Introduction

The R community keeps growing. This can be seen in the number of new packages on CRAN, which keeps on growing exponentially (Hornik et al., 2019), but also in the numbers of conferences, open educational resources, meetups, unconferences, and companies taking up, as exemplified by the useR! conference series[1], the global growth of the R and R-Ladies user groups[2], or the foundation and impact of the R Consortium[3]. All this cements the role of R as the *lingua franca* of statistics, data visualisation, and computational research. Coinciding with the rise of R was the advent of Docker as a general tool for distribution and deployment of server applications—in fact, Docker can be called the *lingua franca* of describing computing environments and packaging software. Combining both these topics, the *Rocker Project* (https://www.rocker-project.org/) provides images with R (see the next Section for more details). The considerable uptake and continued evolution of the Rocker Project has lead to numerous projects extending or building upon Rocker images, ranging from reproducible research\footnote{"Reproducible" in the sense of the *Claerbout/Donoho/Peng* terminology (Barba, 2018).} to production deployments. This article presents this *Rockerverse* of projects across all development stages: early demonstrations, working prototypes, and mature products. We also introduce related activities connecting the R language and environment with other containerisation solutions. The main contributions is a coherent picture of the current lay of the land of using containers in, with, and for R.

The article continues with a brief introduction of containerization basics and the Rocker Project, followed by descriptions of projects developing images with R installations. Then, we present applications, starting with the R packages specifically for interacting with Docker, second-level packages using containers mediately or only for specific features, up to complex use cases leveraging containers. We conclude with a reflection on the landscape of packages and applications and point out future directions of development.

Containerization and Rocker

Docker, an application and service provide by the eponymous company, has in just a few short years risen to prominence for development, testing, deployment and distribution of computer software (cf. Datadog, 2018; Muñoz, 2019). While there are related approaches such as LXC[4] or Singularity (Kurtzer et al., 2017), Docker has become synomymous with "containerization"—the method of taking software artefacts and bundling them in such a way that use becomes standardized and portable across operationg systems. In doing so, Docker had recognised and validated the importance of one very important thread that had been emerging, namely virtualization. By allowing (one or possibly) multiple applications or services to run concurrently on one host machine without any fear of interference between them, an important scalability opportunity is being provided. But Docker improved this compartmentalization by accessing the host system—generally Linux—through a much thinner and smaller shim than a full operating system emulation or virtualization. This containerization is also called operating-system-level virtualization **??**. Typically a container runs one

---

[1] https://www.r-project.org/conferences/
[2] https://www.r-consortium.org/blog/2019/09/09/r-community-explorer-r-user-groups, https://www.r-consortium.org/blog/2019/08/12/r-community-explorer
[3] https://www.r-consortium.org/news/announcements, https://www.r-consortium.org/blog/2019/11/14/data-driven-tracking-and-discovery-of-r-consortium-activities
[4] https://en.wikipedia.org/wiki/LXC

Figure 1: Rockerverse hex sticker

process, whereas virtualization may run whole operating systems at a larger footprint. This makes for more efficient use of system resources (Felter et al., 2015) and allowed another order of magnitude in terms of scalability of deployment (cf. Datadog, 2018). While Docker makes use of Linux kernel features, its importance was large enough so that some required aspects of running Docker have been added to other operating systems to support Docker there more efficiently too (Microsoft, 2019b). The success even lead to standardisation and industry collaboration (OCI, 2019).

The key accomplishment of Docker as an "application" is to make a "bundled" aggregation of software (the so-called "image") available to any system equipped to run Docker, without requiring much else from the host besides the actual Docker application installation. This is a rather attractive proposition and Docker's very easy to use user interface has lead to widespread adoption and use of Docker in a variety of domains, e.g., cloud computing infrastructure (e.g., Bernstein, 2014), data science (e.g., Boettiger, 2015), and edge computing (e.g., Alam et al., 2018). It provided to be a natural match for "cloud deployment" which runs, or at least appears to run, "seamlessly" without much explicit reference to the underlying machine, architecture or operating system: containers are portable and can be deployed with very little in terms of dependencies on the host system—only the container runtime is required. Images are normally build from plain text documents called `Dockerfile`. A `Dockerfile` has a specific set of instructions to create and document a well-defined environment, i.e., install specific software and expose specific ports.

For statistical computing and analysis centered around R, the Rocker Project has provided a variety of Docker containers since its start in 2014 (Boettiger and Eddelbuettel, 2017). The Rocker Project provides several lines of containers spanning to from building blocks with `R-release` or `R-devel`, via containers with RStudio Server and Shiny Server, to domain-specific containers such as `rocker/geospatial` (Boettiger et al., 2019). These containers form *image stacks*, building on top of each other for better maintainability (i.e. smaller `Dockerfiles`), composability, and to reduce build time. Also of note is a series of "versioned" containers which match the R release they contain with the *then-current* set of packages via the MRAN Snapshot views of CRAN (Microsoft, 2019a). The Rocker Project's impact and importance was acknowledged by the Chan Zuckerberg Initiative's *Essential Open Source Software for Science*, who provide findung for the projects's sustainable maintenance, community growth, and targeting new hardware platforms including GPUs (Chan Zuckerberg Initiative et al., 2019).

Docker is not the only containerisation software. An alternative stemming from the domain of high-perfomance computing is **Singularity** (Kurtzer et al., 2017). Singularity can run Docker images, and in the case of Rocker works out of the box if the main process is R, e.g., in `rocker/r-base`, but does not succeed in running images where there is an init script, e.g., in containers that by default run RStudio Server. In the latter case, a `Singularity` file, a recipe akin to a `Dockerfile`, needs to be used. To date, no comparable image stack to Rocker exists on Singularity Hub. A further tool for running containers is **podman**, which also can build `Dockerfiles` and run Docker images. Proof of concepts for using podman to build and run Rocker containers exist[5]. Yet the prevelance of Docker, especially in the broader user community beyond experts or niche systems, and the vast amount of blog posts and courses for Docker, currently caps specific development efforts for both Singularity and podman in the R community. This might quickly change when usability and spread increase, or security features such as rootless/unprivileged containers, which both these tools support out of the box, become more sought after.

---

[5]See https://github.com/nuest/rodman and https://github.com/rocker-org/rocker-versioned/issues/187

Container images

Images for alternative R distributions

As outlined above, R is a widely-used language with a large community. The large number of extension packages provides access to an unrivaled variety of established and upcoming features. Nevertheless, special use cases and experimental projects exist to test approaches or provide features different to what *"base R"* provides. These projects stem both form academia and industry. *Base R*, sometimes called `GNU-R`, is the R distribution maintained by the R Core Team and and provided via CRAN.

Microsoft R Open (MRO) is an R distribution formerly known as Revolution R Open (RRO) before Revolution Analytics was acquired by Microsoft. MRO is compatible with main R and its packages. "It includes additional capabilities for improved performance, reproducibility, and platform support." (Microsoft, a) Most notably these capabilities are the MRAN repository, which is enabled by default, and the (optional) integration with Intel® Math Kernel Library (MKL) for multi-threading in linear algebra operations (Microsoft, b). MRO does not provide official Docker images, but a set of community-maintained `Dockerfiles` and Docker images `nuest/mro` are available on Docker Hub at `https://github.com/nuest/mro-docker` and on GitHub at `https://github.com/nuest/mro-docker` respectively. The images are inspired by the Rocker images and can be used much in the same fashion, effectively a drop-in replacement allowing users to quickly evaluate if the benefits of MRO + Intel® MKL apply to their use case. Version-tagged images are provided for the latest bugfix release of recent R versions. Extended license information about MKL is printed at every startup.

Renjin is an interpreter for the R language running on top of the Java Virtual Machine (JVM) providing full two-way access between Java and R code (Wikipedia, 2018). It was developed to combine the benefits of R, such scripting and extension packages, with the JVM's advantages in the areas of security, cross-platform availability, and established position in enterprise settings. R extension packages need to be specially compiled and are distributed via the Java package manager Apache Maven, cf. `http://packages.renjin.org/packages` for available packages. Packages are loaded on demand, i.e., at the first call to `library()`. Not all R packages, especially one linking to binary libraries, are available, e.g., `rgdal` [6]. There are no offical Docker images for Renjin, but community-maintained images for selected releases only are available under `nuest/renjin` on Docker Hub and GitHub at `https://hub.docker.com/r/nuest/renjin` and `https://github.com/nuest/renjin-docker` respectively. These images expose the command line interface of Renjin in a similar fashion as Rocker images and allow an easy evaluation of Renjin's suitability, but are not intended for production use.

pqR is a "a pretty quick version of R". *pqR* attempts to improve R on some opinionated issues in the R language and is the basis for experimental features, e.g., automatic diffentiation [7]. The source code development on GitHub is a one man project and it does not provide any Docker images. But especially disruptive approaches may contribute to the development of the R ecosystem, so the `nuest/pqr` images on Docker Hub and GitHub at `https://hub.docker.com/r/nuest/pqr/` and `https://github.com/nuest/pqr-docker` respectively.

FastR is "A high-performance implementation of the R programming language, built on GraalVM" (`https://github.com/oracle/fastr`). It is developed by Oracle, connects R to the GraalVM ecosystem (wik, 2019), and also claims superior performance but also targets full compatibility with base R [8]. There are no offical Docker images provided, but `nuest/fastr` images and `Dockerfile` are independently maintained on Docker Hub and GitHub respectively.

While the images presented in this section are far from being as vetted, stable, and widely used as any of the Rocker images, they demonstrate an important advantage of containerisation technology, namely the abiliy to transparently build portable stacks of open source software and make them easily accessible to users. All different distributions are published under GPL licenses. Since all of the different R distributions claim better performance as a core motivation, a comparision based on Docker images, potentially leveraging the resource restriction mechanisms of Docker to level the playing field, seems useful future work.

Bioconductor

Bioconductor (`https://bioconductor.org/`) is an open source, open development project for the analysis and comprehension of genomic data (Gentleman et al., 2004). The project consists of 1741 R software packages as of August 15th 2019, as well as packages containing annotation or experiment data. _Bioconductor_has a semiannual release cycle, each release is associated with a particular

---

[6] http://packages.renjin.org/package/org.renjin.cran/rgdal/1.4-4/build/1
[7] https://riotworkshop.github.io/abstracts/riot-2019-pqr.txt
[8] https://github.com/oracle/fastr

version of R. Docker images allow availability of current and past versions of *Bioconductor* for convenience and reproducibility. *Bioconductor* 'base' docker images are built on top of `rocker/r-ver` and `rocker/rstudio`. *Bioconductor* installs packages based on the R version, and therefore uses `rocker/rstudio` and `rocker/r-ver` version tagging. *Bioconductor* selects the desired version of R from Rocker, adds the BiocManager CRAN package for installing appropriate versions of *Bioconductor* packages, and creates a *Bioconductor* docker image with an informative tag (R_version_Bioc_version). The images are summarized on the *Bioconductor* web site (https://bioconductor.org/help/docker/), maintained on GitHub (https://github.com/Bioconductor/bioc_docker), and available to the community through DockerHub. Past and current combinations of R and *Bioconductor* are therefore accessible via a specific docker tag.

Bioconductor has several images in addition to 'base', specific to various areas of research. The 'core' image installs the most commonly used *Bioconductor* packages. *Bioconductor* images for proteomics, metabolomics, and flow cytometry are community maintained. All community maintained images build on top of the *Bioconductor* base image and therefore indirectly the Rocker images. To simplify building and maintaining *Bioconductor* images, we use a Ruby templating engine. A recent audit of the *Bioconductor* Dockerfiles, following best practices from the Docker website, led to a reduction in the size and number of layers. The most important insights involve the 'union' file system used by Docker. In this file system, once a layer (e.g., `RUN` statement) writes to a file path, the file path is never altered. A subsequent layer that might appear to remove or overwrite the path actually masks, rather than alters, the original. It is therefore important to clean up (e.g., cache removal) within each layer, and to avoid re-installing existing dependencies.

A recent innovation is to produce a `bioconductor_full` image to emulate the *Bioconductor* nightly Linux build machine. The image contains the *system dependencies* needed to install and check almost all (1730 of 1741) *Bioconductor* software packages. Users no longer have to manage complciated system dependencies. The image is configured so that `.libPaths()` has `/usr/local/lib/R/host-site-library` as the first location. Users mounting a location on the host file system to this location then persist installed packages across docker sessions or updates. Many *R* users pursue flexible work flows tailored to particular analysis needs, rather than standardized work flows. The `bioconductor_full` image is well-suited to this pattern. `bioconductor_full` provides developers with a test environment like *Bioconductor*'s build system.

Use of images suggests several interesting possibilities for the *Bioconductor* project. Images may be valuable in teaching, where participants pull pre-built images to avoid complicated configuration of their own computing environemnts. An appeal of this over our current approach (providing Amazon Machine Instances for the duration of the course) is the utility of the image to participant after the course is over. `bioconductor_full` introduces a common system configuration, so it becomes increasingly sensible for *Bioconductor* to distribute convenient *binary* packages. Images also suggest approaches to more advanced computational models. For instance, we are exploring use of images for Helm-orchestrated Kubernetes clusters on the Google Cloud Platform. The user interacts with a manager image based on `bioconductor_full`, configured to perform map-reduce style computations via the BiocParallel package communicating with minimally-configured worker images. A strength of this approach is that the responsibility for complex software configuration (including customized development) is shifted from the user to the experienced *Bioconductor* core team.

Images for (historic) R versions

As with any other software, each new version of R comes with its share of changes. Some are breaking changes, some are not, but the fact is that running a piece of code in a given version might give different results from another version can be an issue when it comes to reproducibility of workflows and stability of applications. For example, think about R random seed: starting with `R 3.6.0`, running `set.seed(2811); sample(1:1000, 2)` will not give the same result as if it was run inside an older version of R. That might seem trivial, but all the code using the random number generator will not be exactly reproducible after this breaking change. Therefore, controlling the version of software is most crucial for reproducible research (e.g. Boettiger, 2015).

Containers are perfectly suited to capture a specific configuration of a computing environment to prohibit such problems. The Rocker images provides the *versioned stack* for different the version of R since the project inception, which use the *then-current* stable Debian base image (cf. Boettiger and Eddelbuettel, 2017). Based on custom build phase hooks[9], i.e., small shell scripts executed at different phases in the automated build of Rocker images, there are also semantic version tags for the most recent freezed (i.e., using MRAN) versions with only the major and minor version[10]. So, at the time of

---

[9]https://docs.docker.com/docker-hub/builds/advanced/
[10]See   https://github.com/rocker-org/rocker-versioned/blob/master/VERSIONS.md,   FIXME: https://github.com/rocker-org/rocker-versioned/issues/42.

writing this article, `rocker/r-ver:3` and `3.6` are aliases for `3.6.0`, because `3.6.1` is the latest release. With the release of `3.6.2` and pinning of the MRAN version in `3.6.1`, the tags `3` and `3.6` will deliver the same image as `3.6.1`. These tags allow users to update the base image to retrieve bugfixes while reducing the risk of also introducing breaking changes.

r-online is an app for helping users to detect breaking changes between different R versions, and for historic exploration of R. With a standalone NodeJS app or online, the user can compare a piece of code run in two separate versions of R. Internally, r-online opens one or two Docker instances with the given version of R based on Rocker images, executes a given piece of code, and returns the result to the user.

A container also provides an isolated sandbox environment suitable for testing and evaluation, without interfering with the "main" working environment. This enables cross version testing and bugfixing. To take this even further, the Rocker contributors are discussing the provision of R versions reaching back futher than the project's own inception, reaching back as far as R `2.x` and `1.x` [11]. The main challenges are finding a suitable base image with a matching OS version, or making the adjustments to compile R on more recent OS releases.

Windows Images

Docker containers on the Windows operating system were originally quite cumbersome to use. They required an extra tool, Docker Toolbox, which by now only exists as a legacy solution for older Windows systems. Docker toolbox leverages `docker-machine` (see also Section Interfaces for Docker in R for an R package interfacing with `docker-machine`) to handle the process of creating a local virtual host which could host Docker Engine, while exposing the regular Docker CLI. The Docker CLI commands are forwarded to the virtual host transparently for the user.

For current Windows Server (2016 and later) and Windows Desktop (Docker Desktop requires Windows 10) versions, Docker is supported natively[12] and different base images are offered by Microsoft[13]. The Docker CLI can be used in just the same way as on other operating systems but not every base image is supported on every Windows host[14]. On Docker Desktop for Windows, the user can run both Linux-based and Windows-based containers, but only one of the Docker daemons can be used at a time[15].

rocker-win (`https://github.com/nuest/rocker-win`) is a proof of concept for running R in Windows-based containers (Nüst, 2019). It provides selected R versions using three different base images using `microsoft/windowsservercore` and `mcr.microsoft.com/windows/servercore`. These base images match different Windows versions: Windows Server 2019, Windows Server 2016, and Windows Server, version 1803. The images are built using two CI services, Travis CI and Appveyor, which provide the different Windows versions and therefore support different base images, and published automatically on Docker Hub. The images are built from manually maintained seperate `Dockerfiles`, are tagged with both Windows-variant and R version, e.g., `nuest/rocker-win:ltsc2019-3.6.2` or `nuest/rocker-win:1803-latest`, and run the `R.exe` process by default. All these images can be run on Docker for Desktop on Windows 10. New R versions are only added on demand. The explorative rocker-win project demonstrates the variety and pontentially confusing complexity of Windows on Docker, e.g., because of the different base images, but also provides a starting point for any Windows-constrained user to leverage containerisation, including examples for apps using \pkg{plumber, **Shiny**, or R packages with system dependencies such as **sf**. The latter is quite comfortable and fast actually, compared to installation from source on Linux, as R ships pre-compiled binaries for Windows. However, compared to the various Linux images, the footprint of Windows images is quite large.

For Windows, just as originally for Linux, the cloud use cases drive the development and Docker and Microsoft collaborate closely[16]. In most cases, individual developers or researchers worrying about reproducibility, will prefer the more widely and host independent Linux-based containers. Naturally, a Windows license is required for the host machine, and the licensing impacts the potnetial to build images in clouds or redistribute exported images as archive files. But the developments in the `rocker-win` prototype show that containers can also be used for R workflows depending on Windows-only tools, for leveraging an existing Windows Server-based infrastructure where policies can otherwise not be met, or for streamlining interactions with system operating staff that runs Windows-based containers themselves.

---

[11]https://github.com/rocker-org/rocker-versioned/issues/138
[12]https://www.docker.com/products/windows-containers
[13]https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/index
[14]https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/version-compatibility
[15]See "Switch between Windows and Linux containers" on https://docs.docker.com/docker-for-windows/.
[16]https://www.docker.com/partners/microsoft

Non-Debian Linux images

The **R-hub** project provides *"a collection of services to help R package development"*, with the package builder as the most prominent one(R-hub project, 2019). The builder allows R package developers to check their R package on different platforms and R versions using a web form or the package **rhub** (Csárdi and Salmon, 2019). The builder uses Docker containers to conduct these checks, taking advantages of their well-defined environments and sandboxing. The `Dockerfiles` and images are published on Docker Hub[17] and GitHub[18] respectively. The images comprise release and development builds of R running in the base images of the Debian, Ubuntu, Fedora, and CentOS linux distributions (Arch Linux is under development[19], see `rhub::local_check_linux_images()` for a full list), and are intended for debugging (see Section Development and debugging) not for reproducible workflows. *The rationale* of not re-using the Rocker image stack is the high number of operatings systems and configurations, which can be more uniformly maintained in a seperate suite of images. Also, a common structure across the respective image stacks of the operating systems, e.g., for `debian`, adding GCC in `debian-gcc` and then R's development version in `debian-gcc-devel`. The platform also has a specific user configuration differing from the setup covering typical applications of Rocker images, and the images include considerably more software, such as a fairly complete LaTeX installation, to provide an experience closer to CRAN.

RStudio also maintains a set of Docker images for multiple operating systems including SUSE, Ubuntu, CentOS, and Debian. The image stack includes an opinionated R installation and ensures the R installation and profile is consistent across the different Linux distributions. The base image also installs R in a versioned directory and uses a minimal set of build and runtime dependencies. Furthermore these images are responsible for creating pre-compiled binary R packages for Linux, which supplement the binaries built by CRAN. These pre-compiled packages dramatically decrease the installation time of R packages on Linux and subsequently can decrease the build time of Docker images (Lopp, 2019).

There are also independent projects installing R or RStudio on **Alpine Linux**, but not beyond the proof of concept stage[20]. In general, Alpine-based Docker images are often chosen for minimalistic and thereby small and secure containers, e.g., in hardware with limited storage. R-hub's proof of concept `r-minimal` takes this to the extreme with compressed image size of 20MB and bespoke install scripts, e.g., to cleverly remove compilers after package installation. For R though, this advantage quickly deteriorates if more features are needed, e.g. if adding compilers, packages, or especially if installing powerful data science and communication tools, like RStudio or LaTeX. Furthermore the distribution uses the musl C library instead of the `glibc` used in Debian/Ubuntu, for which more tooling and experiences exist[21]. In fact, using Debian's 'slim' variant and removing capabilities, one could probalby achieve similarly small image sizes on a more common stack.

altRnative (https://github.com/ismailsunni/altRnative/) is an experimental R package for running the same code across multiple containerised versions of R, intended for comparison across operating systems and different implementations of R. It comes with a collection of `Dockerfiles` and corresponding images in multiple combinations, currently including, e.g., MRO, FastR, Fedora, and TERR[22].

Data Science images

Data Science is a widely discussed topic among all academic disciplines (cf. e.g., Donoho, 2017). Using computers and complex software to make sense of (large amounts of) collected or simulated data is a cross-cutting phenomenon leading to *X Data Science* disciplines, seminars, and study majors. The practice to apply a combination tools and software stacks for analysing data and the cross-cutting skillsets connected with data science puts the management of computing environments and the challenges of reproducibility high up on the agenda. This awareness leads to a large number of bespoke Docker images but also some commonly used image stacks, ranging from base images with very common tools to 'catch all' images including *a lot* of software. Here we only mention a few image stacks as examples, which are not based on Rocker images or stem from the R community, but

---

[17]https://hub.docker.com/u/rhub
[18]https://github.com/r-hub/rhub-linux-builders
[19]https://github.com/r-hub/rhub-linux-builders/pull/41
[20]https://gitlab.com/artemklevtsov/r-alpine or https://www.github.com/velaco/alpine-r provide different use case images but are not recently maintained (development of the latter documented in Ratesic (2018)); https://github.com/cmplopes/alpine-r has only base R images, is relatively up-to-date but sparsely documented; https://github.com/CenterForStatistics-UGent/mountainr is a ghost project with only a single short 'Dockerfile', yet it shows the easy installation of the latest R from Alpine sources
[21]Cf. https://github.com/rocker-org/rocker/issues/231
[22]See 'Dockerfile's at https://github.com/ismailsunni/dockeRs.

prominently include installations of R. A general description of what to consider when creating a Docker image used for data science projects in R is provided on environments.rstudio.com.

The **Jupyter Docker Stacks** project are a set of ready-to-run Docker images containing Jupyter applications and interactive computing tools (Jupyter, 2018). The `jupyter/r-notebook` image includes R and "popular packages", and R is also included in the catchall `datascience-notebook` image[23]. For example, these images allow users to quickly start a Jupyter Notebook server locally or build their own specialised images on top of stable toolsets.

Kaggle provides the `gcr.io/kaggle-images/rstats` image (previously `kaggle/rstats`) and corresponding `Dockerfile` for usage in their Machine Learning competitions and easy access to the associated datasets. It includes machine learling libraries such as Tensorflow and Keras, and also configures the **reticulate** package. The image uses a base image with *all packages from CRAN*, `gcr.io/kaggle-images/rcran`, which requires a Google Cloud Build as Docker Hub would time out[24]. The final extracted image size is over 25GB, which makes it debatable if having everything available is actually convenient.

As a further example, **Radiant** project provides several images, e.g., `vnijs/rsm-msba-spark`, for their browser-based business analytics interface based on **Shiny** (`Dockerfile` on GitHub) and for use in education as part of an MSc course. As data science often applies a multitude of tools, this image favours inclusion over selection and features Python, Postgres, JupyterLab and Visual Studio Code besides R and RStudio, bringing the image size up to 9GB.

Gigantum also its own image stack (`https://github.com/gigantum/base-images`). . . .

Use cases and applications

Interfaces for Docker in R

Interfacing with the Docker daemon is typically done through the Docker Command Line Interface (Docker CLI). However, moving back and forth between an R console and the command line can create friction in workflows and reduce reproducibility. A number of first-order R packages provide a interface to the Docker CLI, allowing to automate interaction with the Docker CLI from an R console.

Each of these packages has particular advantages as they provide function wrappers for interacting with the Docker CLI at different stages of a container's life cycle. Examples of such interactions are installing the Docker software, creating `Dockerfiles` (**dockerfiler**), **containerit**), building images and launching a containers (**stevedore**, **docker**) on a local machine or on the cloud. As such, the choice of which package is most useful depends on the use-case at hand, but also the users level of expertise.

Functionality	AzureContainers	babelwhale	dockermachine	dockyard	harbor	stevedore
Generate a Dockerfile				✓		
Build an image	✓			✓		
Execute a container locally or remotely	✓	✓	✓	✓	✓	✓
Deploy or manage an instances in the cloud	✓		✓		✓	✓
Interact with an instance (e.g. file transfer)		✓	✓			✓
Manage storage of images					✓	✓
Supports Docker and Singularity		✓				
Direct access to Docker API instead of using the CLI						✓
Installing Docker software			✓			

harbor (`https://github.com/wch/harbor` is not actively maintained anymore, but should be honorably mentioned as the first R package for managing Docker images and containers. It uses the **sys** package to run system commands against the Docker CLI, both locally and through an ssh connection, and has convenience functions, e.g., for listing and removing containers/images and for accessing logs. The output of container executions are converted to appropriate R types. The Docker CLI, while evolving quickly and not with great concern for introducing breaking changes,

---

[23]https://jupyter-docker-stacks.readthedocs.io/en/latest/using/selecting.html

[24]Originally, a stacked collection of over 20 images with automated builds on Docker Hub was used, see https://web.archive.org/web/20190606043353/http://blog.kaggle.com/2016/02/05/how-to-get-started-with-data-science-in-containers/ and https://hub.docker.com/r/kaggle/rcran/dockerfile

basic functionality is unchanged for a long time so harbor::docker_run(image = "hello-world") works like a charm.

stevedore (https://cran.r-project.org/package=stevedore) currently the most powerful Docker client in R.

AzureContainers (https://cran.r-project.org/package=AzureContainers) is an interface to a number of container-related services in Microsoft's Azure Cloud, namely Container Instances, Container Registry, and Kubernetes Service. While it is mainly intended for working with Azure, as a convenience feature it includes lightweight, cross-platform shells to Docker and Kubernetes (tools kubectl and helm). These can be used to create and manage arbitrary Docker images and containers, as well as Kubernetes clusters on any platform or cloud service.

babelwhale (https://cran.r-project.org/package=babelwhale) allows executing and interacting with containers, which can use either Docker or Singularity as a backend. The package provides a unified interface to interact with Docker and Singularity containers. Users can, for example, execute a command inside a container, mount a volume or copy a file.

dockyard (https://github.com/thebioengineer/dockyard) has the goal of lowering barrier to creating Dockerfiles and Docker images, and deploying Docker containers. The way docker images are made is by reading and following a set of instructions found in Dockerfiles. This requires stepping out of the R ecosystem and into an infrastructure mindset where the user needs to specify the environment they require. Because Dockerfiles can be read as a series of steps, this lends itself well to the tidyverse style of programming where each step is piped into the next. dockyard has a Dockerfile object that either starts out as an empty Dockerfile, or can use existing Dockerfile to use as a template. This combined with a series of functions that are analogous to the steps commonly found in a Dockerfile, it makes building Docker images much more approachable to R users. There are also wrappers for commonly used steps for R programmers like adding libraries and copying files into the R Docker containers. Once the Dockerfile is created, dockyard has built-in functionality to build the image and run a container.

dockermachine (https://github.com/cboettig/dockermachine) is an R package to provide a convenient interface to Docker Machine from R. Docker Machine's CLI tool docker-machine allows users to create and manage virtual host on local computers, local data centers, or at cloud providers. A local Docker installation can be configured to forward all commands issues on the local Docker CLI to a selected (remote) virtual host fully transparent. For local Docker Machine was especially crucial in early days of Docker, when no native Docker support was available for Mac or Windows computers, but remains relevant for provisioning on remote systems. The package has not received any updates for two years, but was functional with a current version of docker-machine (0.16.2) and potentially lowers the barriers for R users to run containers on various hosts, if using the Docker Machine CLI is perceived as a barrier.

Capture and create environments (?)

Several second order R packages attempt to make the process of creating Docker images and using containers for specific tasks, such as running tests or rendering reproducible reports, easier.

dockerfiler (https://github.com/ColinFay/dockerfiler/) . . .

containerit (https://github.com/o2r-project/containerit/) . . .

dockr (https://github.com/smaakage85/dockr) . . . see/cite http://smaakage85.netlify.com/2019/12/21/dockr-easy-containerization-for-r/

renv (https://rstudio.github.io/renv/) with Docker: https://rstudio.github.io/renv/articles/docker.html

version pinning

liftr (https://nanx.me/liftr/) (Xiao, 2019) aims to solve the problem of persistent reproducible reporting in statistical computing. Currently, the R Markdown format and its backend compilation engine *knitr* offer a *de facto* standard for creating dynamic documents (Xie et al., 2018). However, the reproducibility of such content authoring environments is often limited to individual machines — it is not easy to replicate the system environment (libraries, R versions, R packages) where the document was compiled. This issue becomes even more serious when it comes to collaborative document authoring and creating large-scale document building services. *liftr* solves this reproducibility problem by bringing Docker to the game. In essence, *liftr* helps R Markdown users create and manage Docker containers for rendering the documents, thus make the computations utterly reproducible across machines and systems.

On implementation, with no side effects, *liftr* extended and introduced new metadata fields to R Markdown, allowing users to declare the dependencies for rendering the document. *liftr* parses such

fields and generates a `Dockerfile` for creating Docker containers. *liftr* then helps render the document inside the created Docker container. This workflow is summarized in Figure 2.
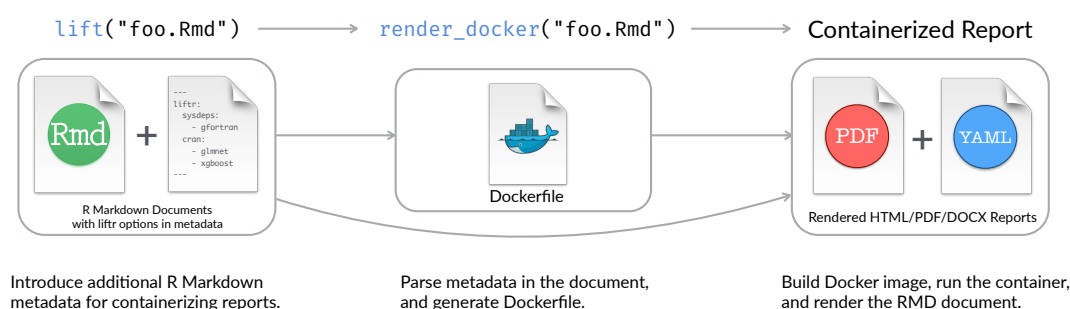

Figure 2: The liftr workflow for rendering containerized R Markdown documents.

knitr and R Markdown are used as the template engine to generate the `Dockerfile`. Features such as caching container layers for saving image build time, automatic housekeeping for fault-tolerant builds, and Docker status check are supported by *liftr*. Four RStudio addins are also offered by *liftr* to allow push-button compilation of documents and provide better IDE integrations.

Three basic principles are followed to design the *liftr* package since its inception.

1. Continuous reproducibility. Continuous integration, continuous delivery, and continuous deployment are well-accepted practices in software engineering. Similarly, it is believed by the authors of *liftr* that ensuring computational reproducibility means a continuous process instead of creating static data/code archives or a one-time deal. Specifically, the software packages used in data analysis should be upgraded regularly in a manageable way. Therefore, *liftr* supports specifying particular versions of package dependencies, while users are encouraged to always use the latest version of packages (without a version number) by default.

2. Document first. Many data analysis workflows could be wrapped as either R packages or dynamic documents. In *liftr*, the endpoint of dynamic report creation is the focus of containerization, because this offers more possibilities for organizing both computations and documentation. Users are encouraged to start thinking from the visible research output from the first day.

3. Minimal footprint. R Markdown and Docker are already complex software systems. Making them work together seamlessly can be complicated. Therefore, API designs such as function arguments are simplified while being kept as expressive and flexible as possible.

In summary, *liftr* tries to redefine the meaning of computational reproducibility by offering system-level reproducibility for data analysis. It provided a practical way for achieving it — a new perspective on how reproducible research could be done in reality. Further, sharing system environments for data analysis also becomes extremely easy, since users only need to share the R Markdown document (with a few extra metadata fields), and compile them with *liftr*. As an example, *liftr* demonstrated its advantage for R Markdown-based computational workflow orchestration, by effortlessly containerizing 18 complex *Bioconductor* workflows in the DockFlow project (https://dockflow.org) in 2017.

One of the key tool of Docker are `Dockerfiles`, which can be thought of as "recipes" used to build a specific image. But building these files can feel like a workflow break as it demands to open a separate file, and / or to write a small wrapper around R's `write()` function to add elements to the file. At the same time, doing it by hand prevents from using programming language for what they are good for: iteration, and automation. Iteration and automation for `Dockerfiles` is the very reason behind dockerfiler, an R package designed for building `Dockerfiles` straight from R.

For example, the **golem** package makes an heavy use of dockerfiler when it comes to creating the `Dockerfile` for building production-grade Shiny applications and deploying them. The reason behind this is that dockerfiler, on top of being scriptable from R, can leverage all the tools available in R to parse a `DESCRIPTION` file, to get system requirements, to list dependencies, versions, etc.

Using R to power enterprise software in production environments

R has been historically viewed as a tool for analysis and scientific research–not for creating software that corporations can rely on to continuously run in real time. However, thanks to advancements in R running as a web service, along with with the ability to deploy R in Docker containers, modern enterprises are now capable of having real-time machine learning powered by R.

The backbone of this work is the **plumber** (https://www.rplumber.io/) package. The plumber package allows R to be used to host web services through RESTful APIs. When R is running on a server,

a different server can ask R to make a computation through an HTTP request and get an immediate response. For instance, if a data scientist at a company creates a machine learning model to tell which customers are likely to make a repeat purchase, if they serve the model through a plumber API then someone else on the marketing team could write software that sends different emails depending on that real-time prediction. Plumber syntax is simple to read and understand: the API is created by commenting above function names in an R script.

```
#* Return the sum of two numbers
#* @get /random
function(){
  runif()
}
```

Example of a plumber API to generated a random number by visiting `http://127.0.0.1/random`

Plumber becomes far more powerful with the addition of Docker. Docker can be used to create images that include both R and plumber. Then when the image is deployed, that creates the web endpoints that other services can hit and call the R code. This pattern of deploying code mimics those used by software engineering services created in Java, Python, and other modern languages. By using Docker to deploy plumber APIs, R can be used alongside those langages as a first class member of a software engineering technical stack.

At T-Mobile, R is used to power real-time services that are called over a million times a day. T-Mobile created a set of neural network machine learning natural language processing models to help customer care agents manage text-based messages for customers (T-Mobile et al., 2018). The models are convolutional neural networks that use the RStudio Keras package and are built on top of a Rocker Docker image. Since the models power tools for agents and customers, they need to have extremely high uptime and reliability–and they found that R is able to perform. The AI @ T-Mobile team open sourced their Docker images on GitHub.

Deployment to the cloud

The cloud is the natural environment of containers, and is the natural mechanism to deploy R server applications.

- RSelenium
- googleComputeEngineR (function `gce_vm_template()`)
  - To enable quick deployments of key R services such as RStudio and Shiny onto cloud virtual machines (VMs), this package utilises Dockerfiles to move the labour of setting up those services from the user to a premade Docker image. For example, by specifying the template `template="rstudio"` in `gce_vm_template()`/`gce_vm()` an up to date RStudio Service image is launched. Specifying `template="rstudio-gpu"` will launch an RStudio Server image with a GPU attached, etc.

- analogsea (digital ocean R client)
- plumber
- production deployment of neural networks (**?, ?**)
- Kubernetes - A popular platform for managing Docker containers is Kubernetes, which is used for a wide variety of containerized applications. It may be your organisation has a Kubernetes cluster already for other applications. Docker containers are used within Kubernetes clusters to hold native code, for which Kubernetes creates a framework around network connections and scaling of resources up and down. An introduction on using Kubernetes with R is published at this blog post.
- AzureContainers: Umbrella package for working with containers in Microsoft's Azure cloud. Provides interfaces to three Azure services: Container Registry, Container Instances (for running individual containers), and Kubernetes Services (for orchestrated deployments)

While Plumber provides the low-level infrastructure for turning a statistical model into a predictive service, for production purposes it is usually necessary to take into account considerations such as scalability, reliability and ease of management. A single container is limited in the volume of requests it can service. Furthermore, if the container goes down, or is occupied with processing a computationally intensive task, the service becomes unavailable until it is restarted.

These issues can be mitigated by deploying not a single container, but a *cluster* of containers, orchestrated as a single deployment. The deployment system of choice at the time of writing is

Kubernetes. This allows you to deploy a single image to a pool of servers, which can then serve requests in parallel. Client-side requests are made to the cluster ingress endpoint, which then redirects them to individual servers. The orchestrator can identify cluster nodes that have failed, and restart them automatically; it can also detect when there is no capacity to service more requests, and start additional servers and containers as needed.

While these load-balancing and autoscaling properties of a Kubernetes cluster are invaluable, they are still limited by the hardware resources that it has access to. A cluster cannot scale out if it has run out of servers. This issue in turn can be addressed by deploying not to on-premise servers but to the cloud, where effectively unlimited computing resources are available (as long as one can pay for them). From a management perspective, cloud deployments have the additional advantage that hardware maintenance becomes the responsibility of the cloud provider, eliminating another pain point. Orchestrator services are available from all the major cloud service providers, including Amazon Web Services (AWS), Google Cloud Platform, and Microsoft Azure.

An example of an R package for working with containers in the cloud is AzureContainers, which interacts with three Azure services: Container Instances (for running a single container), Container Registry (a private docker registry service), and Kubernetes Service (for Kubernetes on Azure). Azure-Containers provides a lightweight yet powerful interface for working with Resource Manager, the framework for deploying and managing arbitrary resources in Azure. On the client side, it provides simple shells to the Docker, Kubectl and Helm tools that are commonly used for managing containers and Kubernetes clusters.

Here is some simple code that pushes a Docker image to an Azure Container Registry, and then creates a deployment using Azure Kubernetes Service. This is a slightly modified version of the code from the "Deploying a prediction service with Plumber" AzureContainers vignette. The code makes use of a yaml file to define the deployment and predictive service, which is standard practice with Kubernetes; the yaml can be seen in the aforementioned vignette.

```
library(AzureContainers)
# create a resource group for our deployments
deployresgrp <- AzureRMR::get_azure_login()$
    get_subscription("subscription_id")$
    create_resource_group("deployresgrp", location="australiaeast")

### create a container registry
deployreg_svc <- deployresgrp$create_acr("deployreg")

# build our image (a random forest on the Boston housing data, available in the MASS package)
call_docker("build -t bos-rf .")

# upload the image to Azure
deployreg <- deployreg_svc$get_docker_registry()
deployreg$push("bos-rf")


### create a Kubernetes cluster with 2 nodes
deployclus_svc <- deployresgrp$create_aks("deployclus", agent_pools=aks_pools("pool1", 2))

# grant the cluster pull access to the registry
gr <- AzureGraph::get_graph_login()
aks_app_id <- deployclus$properties$servicePrincipalProfile$clientID
reg$add_role_assignment(gr$get_app(aks_app_id), "Acrpull")


# get the cluster endpoint
deployclus <- deployclus_svc$get_cluster()

# create and start the service
deployclus$create("bos-rf.yaml")
```

Once the service has been created, we can check on its status and obtain predictions:

```
deployclus$get("deployment bos-rf")
#> Kubernetes operation: get deployment bos-rf  --kubeconfig=".../kubeconfigxxxx"
#> NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
```

```
#> bos-rf       1          1          1          1           5m

deployclus$get("service bos-rf-svc")
#> Kubernetes operation: get service bos-rf-svc  --kubeconfig=".../kubeconfigxxxx"
#> NAME          TYPE           CLUSTER-IP    EXTERNAL-IP    PORT(S)        AGE
#> bos-rf-svc    LoadBalancer   10.0.8.189    xxx.xxx.xxx.xxx 8000:32276/TCP 5m


response <- httr::POST("http://xxx.xxx.xxx.xxx:8000/score",
    body=list(df=MASS::Boston[1:10,]), encode="json")
httr::content(response, simplifyVector=TRUE)
#> [1] 25.9269 22.0636 34.1876 33.7737 34.8081 27.6394 21.8007 22.3577 16.7812 18.9785
```

Note that Plumber, by itself, provides no authentication or security features; anybody who knows the cluster's IP address can access the predictive service. At minimum, a basic authentication layer should be added to any Kubernetes deployment. Standard Kubernetes procedure involves creating an ingress controller, for example with the nginx or traefik reverse proxy software, and one can then add application-specific authentication layers on top of that (for example, to authenticate with a user's Azure Active Directory or LDAP credentials). More details can be found from most Kubernetes learning resources, and are outside the scope of this paper.

Continuous integration and continuous delivery (?, ?)

The controlled nature of containers, i.e., it is possible to define the software environment very well, even on remote machines, make them also useful for continuous integration (CI) and continuous delivery of applications.

- DevOps

  - https://www.opencpu.org/posts/opencpu-with-docker/

When doing continuous integration and continuous delivery, it's crucial to test in an environment that matches the production environment. Tools like `Gitlab-CI` are built on top of Docker images: the user specifies a base Docker image, and the whole tests are run inside this environment. But, as we just said, this environment has to be fixed, but it also have to contain the necessary toolkit. In order to achieve that, `r-ci` combines `rocker` versioning and a series of tools specifically designed for testing. That way, package builders can use this image as a base image for there testing environment, without having to install the necessary packages every time they need to run a new test.

- dynwrap (?)

  - For this project, we use travis-ci to build rocker-derived containers, test them, and only push them to Docker Hub (from travis-ci.org) if the integration tests succeed.

- Google Cloud Build

  - https://cloud.google.com/cloud-build/
  - Cloud Build is another continuous intergation service that helps move the workload of building the Docker images to an online service. Cloud Build can be set up to build the Dockerfiles on each GitHub commit or release. This means you do not need to build the Docker images locally, which can tie up resources since Docker images can be several GBs and take a long time to compile. Google Cloud Build works alongside Google Container Registry to allow you to build private and public Docker images, which allows you to build up your own dependency graphs for downstream applications.
  - https://code.markedmondson.me/googleCloudRunner/articles/cloudbuild.html

Common or public work environments

The fact that Docker images are portable and well defined make them useful when more than one person needs access to the same computing environment. This is even more useful when some of the users do not have the expertise to create such an environment themselves, and when these environments can be run in public or shared infrastructure.

holepunch (https://github.com/karthik/holepunch) is an R package that was designed to make sharing work environments accessible to novice R users based on **Binder**. The Binder project, maintained by the team behind Jupyter, makes it possible for users to create and share their computing

environments with others (Jupyter et al., 2018). A *BinderHub* allows anyone with access to a web browser and an internet connection to launch a temporary instance of these custom environments and execute any workflows contained within. From a reproducibility standpoint, Binder makes it exceedingly easy to compile a paper, visualize data, and run small examples from papers or tutorials without the need for any local installation. To set up Binder for a project, a user typically starts at an instance of a BinderHub and passes the location of a repository with a workspace, e.g., a hosted Git repository, or a data repository like Zenodo. Binder's core internal tool is repo2docker. It deterministically builds a Docker image by parsing the contents of a repository, e.g., project dependency configurations or simple configuration files. In the most powerful case, repo2docker builds a given Dockerfile. While this approach works well for most run of the mill Python projects, it is not so seamless for R projects. For any R projects that use the Tidyverse suite (Wickham et al., 2019), the time and resources required to build all dependencies from source can often time out before completion, making it frustrating for the average R user. holepunch removes some of these limitations by leveraging Rocker images that contain the Tidyverse along special Jupyter dependencies, and only installs additional packages from CRAN and Bioconductor that are not already part of these images. It short cicuits the configuration file parsing in repo2docker and starts with the Binder/Tidyverse base images, which eliminates a large part of the build time and in most cases results in a binder instance launching within a minute. holepunch as a side effect also creates a DESCRIPTION file which then turns any project into a research compendium (Marwick et al., 2018). The Dockerfile included with the project can also be used to launch a RStudio server locally, i.e., independent of Binder, which is especially useful when more or special computational resources can be provided there. The local image usage reduces the number of seperately managed environments and thereby reduces work and increases portability and reproducibility.

In **high-performance computing**, one use for containers is to run workflows on shared local hardware where teams manage their own high-performance servers. This can follow one of several design patterns: users may deploy containers to hardware as a work environment for a specific project, conatiners may provide per-user persistent environments, or a single container can act as a common multi-user environment for a server. In all cases, though, the containerized approach provides several advantages: First, users may use the same image and thus work environment on desktop and laptop computers, as well. The former models provide modularity, while the latter approach is most similar to a simple shared server. Second, software updates can be achieved by updating and redeploying the container, rather than tracking local installs on each server. Third, the containerized environment can be quickly deployed to other hardware, cloud or local, if more resources are neccessary or in case of server destruction or failure. In any of these cases, users need a method to interact with the containers, be it and IDE, or command-like access and tools such as SSH, which is usually not part of standard container recipes and must be added. The Rocker project provides containers pre-installed with the RStudio IDE. In cases where users store nontrivial amounts of data for their projects, data needs to persist beyond the life of the container. This may be via in shared disks, attached network volumes, or in separate storage where it is uploaded between sessions. In the case of shared disks or network-attached volumes, care must be taken to persist user permissions, and of course backups are still neccessary. When working with multiple servers, an automation framework such as Ansible may be useful for managing users, permisions, and disks along with containers.

Using **GPUs** (graphical processing units) as a specialised hardware from containerized common work environments is also possible and useful (Haydel et al., 2015). GPUs are increasingly popular for compute-intensive machine learning (ML) tasks, e.g., deep artificial neural networks (Schmidhuber, 2015). Though in this case, containers are not completely portable between hardware environments, but the software stack for ML with GPUs is so complex to set up that a ready-to-use container is helpful. Containers running GPU software require drivers and libraries specific to GPU models and versions, and containers require a specialized runtime to connect to the underlying GPU hardware. For NVIDIA GPUs, the NVIDIA Container Toolkit includes a specialized runtime plugin for Docker and a set of base images with appropriate drivers and libraries. The Rocker project has a repository with (beta) images based on these that include GPU-enabled versions of machine-learning R packages, e.g., rocker/ml and rocker/tensorflow-gpu.

Teaching is a further example where sharing a prepared computing environment can greatly improve the process, especially for courses that require access to a relatively complex setup of software tools, e.g., as in the case of database systems. R is a fantastic tool when it comes to interfacing with databases: almost every open source and proprietary database system has an R package that allows users to connect and interact with the it This flexibility is even broader now that we have tools like DBI, that allows to create a common API for interfacing these databases, or like dbplyr, which are designed to run dplyr code straight against the database. But learning and teaching these tools comes with a cost: the cost of deploying or having access to an environment with the softwares and drivers installed. For people teaching R, it can become a barrier if they need to install local versions of the drivers, or to connect to remote instances which might or might not be made available by IT services. Giving

access to a sandbox for the most common database environments is the idea behind `r-db` (**?**), a Docker image that contains everything needed to connect, from R, to a database. Notably, with `r-db`, the users don't have to install complex drivers or to configure their machine in a specific way. On top of that, `r-db` comes with a comprehensive online guide, explaining how to spin a Docker instance of a specific database, and how to use `r-db` to connect to it. Each package contained inside this image also has a series examples, so that the user can get started with the database right away. The `rocker/tidyverse` base image ensures that users can also readily use packages for analysis, display, and reporting.

RCloud uses a `rocker/drd` base image for creation of collaborative data analysis and visualisation environments.

Using Docker in a commercial Data Science Platform

In addition to embedding RStudio Open Source Server inside of Docker images, such as Rocker, RStudio offers the moduler data science platform RStudio Team, a commercial product allowing teams of data scientists and their respective IT/DevOps groups to develop and deploy code in R and Python. RStudio team enables these groups to use the benefits of containers, without requiring users to learn new tools or directly interact with containers. Notable patterns include:

1. *To deploy and manage RStudio Professional Products:* Many RStudio customers are managing professional products in Docker containers as an alternative to bare-metals servers or virtual machines. Best practices for running RStudio with Docker containers as well as Docker images for RStudio's commerical products are available.
2. *To run interactive RStudio sessions on a cluster:* In the RStudio Server Pro 1.2 release in 2019, RStudio added new functionality called Launcher. It gives users the ability to spawn R sessions and background jobs in a scalable way on external clusters, e.g., Kubernetes based on Docker images or Slurm clusters, and optionally, with Singularity containers. A key benefit of Launcher is the ability for R and Python users to interactively work in RStudio without learning about Docker at all - while still leveraging containers and Kubernetes. Users can submit batch jobs to Kubernetes clusters without writing specific deployment scripts.

RStudio Connect is a commercial publishing platform to deploy dashboards, REST APIs, and other R and Python content into production. RStudio Connect uses an approach to isolate deployed R applications that is similar to Docker's filesystem and process namespacing and leverages the RStudio Package Manager to manage R environments including system dependencies. RStudio Package manager may also be used to manage their R environments and package versions with Docker and to ensure deterministic image builds.

Processing [div]

The portability of containers becomes particularly useful when complex processing tasks shall be offloaded to a server.

- Docker images for cloud services
    - A popular application of using Docker is to create the environments that can run in parallel for speeding up R code. For example, `googleComputeEngineR`'s `gce_vm_cluster()` function can create clusters of 2 or more virtual machines, running multi-CPU architectures. Instead of running a local R script with the local CPU and RAM restrictions, the same code can be processed on all CPU threads of the cluster of machines in the cloud, all running in a Docker container with the same R environments. This is achieved through `googleComputeEngineR`'s integration with the R parralisation library `library(future)` by Henrik Bengtsson. Local R computation can be thrown up to a multi-CPU and VM environment to achieve parrell computation in a few lines of R code in your local session. - some demonstrations are available here.
- Google Cloud Run - This is a CaaS (Containers as a Service) that lets you launch a Docker container without worrying about underlying infrastructure. This dispenses with the developer creating a cloud server to run the Docker image on, by abstracting away those servers to a more serverless configuration. Cloud Run lets you run your code on top of a managed or your own Kubernetes cluster running Knative, and can accept any Docker image. The service takes care of network ingress, scaling machines up and down to zero, authentication and authorisation, all features which are non-trivial for a developer to create on their own. This can be used to scale up R code to millions of instances if they need to, with little or no changes to existing code. An R implementation is shown here at cloudRunR which uses Cloud Run to create a scalable R plumber API.

- – How does this relate to https://code.markedmondson.me/googleCloudRunner/index.html ¿
- batchtools (Lang et al., 2017) can schedule jobs with Docker Swarm
- scalable deployments, e.g., start with numerous Shiny talks mentioning Rocker at useR!2017
- dynmethods (**?**): In order to evaluate ±50 computational methods which all used different environments (R, Python, C++, . . .), we wrapped each of them in a docker container and can execute these methods from R. Again, all of these containers are being built on travis-ci, and will only be pushed to Docker Hub if the integration test succeeds.
- **drake**, https://docs.ropensci.org/drake/index.html?q=docker#with-docker

Packaging programs for processing pipelines

Docker is perfectly suited to package and execute any software and data, and is a good candidate to build complex processing pipelines independent of the used programming language. Because of the original use case (see Introduction), Docker does not have a standard mechanism to chain a number of containers together, i.e., to define how environment variables, volume mounts, or ports can be used to pass input parameters and data into a container and how to get results out. Some packages, e.g. **containerit**, provide a Docker image that can be used similar as a CLI, but the usage is cumbersome[25].

outsider (https://antonellilab.github.io/outsider/) tackles the problem of integrating external programmes into an R workflow (Bennett et al., 2019). Installation and usage of external programmes can be cumbersome and even prohibited if platform dependent. Therefore **outsider** uses the platform-independent Docker images to encapsulate processes in *outsider modules*. Each outsider modules has a Dockerfile and an R package with functions for interacting with the encapsulated tool. A user only needs to use the given functions to install a module and can then integrate a tool into their own workflow seaminglessly using only R functions. The module takes care of transmitting arguments and sending and receiving files to and from a container. Outsider modules are hosted, e.g., on GitHub, and **outsider** contains discovery functions for them.

Research Compendia

Containers provide a high degree of isolation that is often desirable when attempting to capture a specific computational environment so that others can reproduce and extend a research result. Many computationally intensive research projects depend on specific versions of original and third-party software packages in diverse languages, joined together to form a pipeline through which data flows. New releases of even just a single piece of software in this pipeline can break the entire workflow, making it difficult to find the error and for others to reuse existing pipelines, and making the original the results irreproducible and not extandable. The chance of a substantial disruption like this is high in a multi-year research project where key pieces of third-party software may have several major updates over the duration of the project. The classical "paper" article is insufficient to adequately communicate the knowledge behind such research projects (cf. Donoho, 2010; Marwick, 2015). Gentleman and Lang (2007) coined the term *Research Compendium* for a dynamic document together with supporting data and code. They use the R package system R Core Team (1999) as for the functional prototype to structuring, validation, and distribution of research compendia. This concept has been taken up and extended[26], not the least by applying containerisation and other methods for managing computing environments—see Section Capture and create environments (**?**). Containers give the researcher an isolated environment to assemble these research pipelines with specific versions of software. Research workflows in containers are safe from contamination from other activities occuring on the researcher's computer, for example the installation of newest version of packages for teaching demonstrations. Given the users in this scenario, i.e., often academics with limited formal software development training, templates and assistance with containers around research compendia is essential.

In many fields we see that a typical unit of research for a container is a research report or journal article, where the container holds the compendium, or self-contained set of data (or connections to data elsewhere) and code files needed to fully reproduce the article (Marwick et al., 2018). The package **rrtools** (https://github.com/benmarwick/rrtools) provides a template and convenience functions to apply good practices for research compendia, including a starter Dockerfile. Images of compendium containers can be hosted on services such as Docker Hub for convenient sharing among collaborators and others. Similarly, packages such as **containerit** and **dockerfiler** can be used to manage the Dockerfile to be archived with a compendium on a data repository (e.g. Zenodo, Dataverse, Figshare, OSF). A typial compendium's Dockerfile will pull a rocker image fixed to a specific version of R, and install R packages from the MRAN repository to ensure the package versions

---

[25]https://o2r.info/containerit/articles/container.html
[26]See full literature list at https://research-compendium.science/.

are tied to a specific date, rather than the most recent version. A future researchers can download then compendium from the repository and run the included `Dockerfile` build a new image that recreates the computational environment used to produce the original research results. If building the image fails, the human-readable instructions in a `Dockerfile` are a the starting point for rebuilding the environment. Further safeguarding practices are currently under development or not part of common practice yet, such as preservation of images (Emsley and De Roure, 2018) and storing them alongside `Dockerfiles` (cf. Nüst et al., 2017), or pinning of system libraries beyond the tagged base images, which may be seen as stable or dynamic depending on the applied time scale (see discussion on 'debian:testing' base image in Boettiger and Eddelbuettel, 2017).

Development and debugging

Containers can also serve as useful playgrounds to create environments ad-hoc or to provide very specific environments that are not needed or not easily available in day-to-day development. These environments may have specific versions of R, of R extension packages, and of system libraries used by R extension packages, and all of the above in a specific combination.

First, such containers can greatly facilitate **fixing bugs**, because developers can readily start and later discard a container to investigate a bug report without affecting their regular system, and potentially having to revert changes back. Using the Rocker images with RStudio, these disposable environments lack no development comfort (cf. Section Research Compendia). Eddelbuettel (2019) describes an example how a Docker container was used to debug an issue with a package only occuring with a particular version of Fortran, and using tools which are not readily available on all platforms (e.g., not on macOS). For the R-spatial community, using containers is particularly useful, because of the strong integration of system libraries in core packages for geospatial data modelling and analysis. Purpose built Docker are used to prepare for upcoming releases of system libraries, individual bug reports, and for the lowest supported versions of system libraries \footnote{https://github.com/r-spatial/sf/tree/master/inst/docker, \href{https://github.com/Nowosad/rspatial_proj6}{https://github.com/Nowosad/rspatial_proj6}, https://github.com/r-spatial/sf/issues/1231#issuecomment-570076440}.

There are also special images for identifying problems beyond the mere R code, such as **debugging R memory problems**. The images significantly reduce the barrier to follow complex steps for fixing memory allocation bugs (cf. Section 4.3 in R Core Team, 1999). These problems are hard to debug and critical, both because when they occur they lead to fatal crashing processes. `rocker/r-devel-san` and `rocker/r-devel-ubsan-clang` are Docker images have a particularly configured version of R to trace such problems with gcc and clang compilers, respectively (cf. **sanitizers** for examples, Eddelbuettel, 2014). The image `wch/r-debug` is a purpose built Docker image with *multiple* instrumented builds of R, each with a different diagnostic utility activated.

Second, containers are useful for **testing** R code during development. To submit a package to CRAN, an R package must work with the development version of R, which must be compiled locally. That can be a challenge for some users. The R-hub service (see Section Non-Debian Linux images) makes it easy to ensure that no errors occur, but to fix errors a local setup is still often warranted, e.g. using the image `rocker/r-devel`, and to test packages with native code, which can make the process more complex (cf. Eckert, 2018). The R-hub Docker images can also be used to debug problems locally using various combinations of Linux platforms, R versions, and compilers[27]. The images go beyond the configurations, or *flavours*, used by CRAN for checking packages[28], e.g., with CentOS-based images, but lack a container for checking on Windows or OS X. The images greatly support package developers to provide support on operating systems they are not familiar. The package **dockertest** (https://github.com/traitecoevo/dockertest/) is a proof of concept for automatically generating `Dockerfiles` and building images specifically to run tests[29]. These images are accompanied with a special launch script so the tested source code is not stored in the image but the currently checked in version from a local Git repository is cloned into the container at runtime. This approach clearly seperates test environment, test code, and current working copy of the code.

Third, Docker images can be used **on CI platforms** to streamline the testing of packages. Ye (2019) describes how they speed up the process of testing by running tasks on Travis CI within a container using `docker exec`, e.g., the package check or rendering of documentation. Cardozo (2018), also on Travis CI saved time by re-using the testing image as the base iamge for an image intended for publication on Docker Hub. Especially for long running tests or complex system dependencies, these approaches seems to improve the development. Not due to a concern about time, but to control the

---

[27]See https://r-hub.github.io/rhub/articles/local-debugging.html and https://blog.r-hub.io/2019/04/25/r-devel-linux-x86-64-debian-clang/

[28]https://cran.r-project.org/web/checks/check_flavors.html

[29]**dockertest** is not actively maintained, but mentioned still because of its interesting approach.

environment used on a CI server, even this manuscript was rendered into a PDF and deployed to a GitHub-hosted website (see `.travis.yml` and `Dockerfile` in the manuscript repository). This gives on the one hand easy access after every update of the R Markdown source code, and on the other hand a second controlled environment making sure that the article renders successfully and correctly.

Conclusions

This article is a snapshot of the R-corner in a universe of applications built many-faced piece of software, Docker. `Dockerfiles` and Docker images are the go-to methods for collaboration between roles in an organisation, such as development and IT operations teams, and between parties in the communication of knowledge, such as research workflows or education. Docker became synonymous with applying the concept of containerisation to solve challenges of reproducible environments, e.g., in research and in development & production, and of scalable deployments with the ability to move processing between machines easily (e.g., locally, one cloud providers VM, another cloud provider's Container-as-a-Service). Collaboration, reproducibility, and portability are the common themes behind R packages, use cases, and applications in this work.

The R packages and use cases presented show the positive drive in the community to innovate, but also the challenge of keeping up with a continuously evolving landscape. The use cases contributed by co-authors also have a degree of overlap, which can be expected as a common language and understanding of good practices is still taking shape. Also, the ease with which one can create a complex software systems, such as an independent Docker image stack, to serve one's specific needs leads to some fragmentation. Furthermore fragmentation may not be a bad sign but instead a reflection of a growing market, which is able to sustain multiple related efforts. With the maturing of core building blocks, such as the Rocker suite of images, more systems will be built successfully but will also be behind the curtains. Docker alone, as a flexible core technology, is not a feasible level of collaboration and abstraction. Instead, the use cases and applications observed in this work provide a more useful division.

Still, at least *on the level of R packages* some consolidation seems in order, e.g., to reduce the number of packages creating `Dockerfiles` from R code or controlling the Docker daemon with R code. It remains to be seen which approach to control Docker, via the Docker API as `stevedore` or via system calls as `dockyard`/`docker`/`dockr`, is more sustainable, or if the question will be answered by the endurance of maintainers and sufficient funding. Similarly, the capturing of environments and their serialization in form of a `Dockerfile` currently happens at different levels of abstraction and re-use of functionality seems reasonable, e.g. `liftr` could generate the environment with `containerit`, which in turn may use `dockerfiler` for low level objects representing a `Dockerfile`. In this consolidation, the Rocker Project could play the role of a unifying and coordinating entity in the future. Though for the moment, the sign of the times points to more experimentation and feature growth, e.g. images for GPU-based computing and artificial intelligence. Even with coding being more and more accepted as a required, and achievable skill, an easier access, for example by exposing containerisation benefits via simple user interfaces in the users' IDE, could be an important next step. Currently containerisation happens more in the background at the system level.

New features, which make complex workflows accessible and reproducible, and variety in packages, even when they have overlapping features, are a signal and a support for a growing user base. This growth is possibly the most important goal for the foreseeable future in the *Rockerverse*, and just like the Rocker images have matured over years of use and millions of runs the new ideas and prototypes will have to proof themselves. It should be noted that the dominant position of Docker is a blessing and a curse for these goals. It could be wise to start experimenting with non-Docker containerisation tools now, e.g. R packages interfacing with other container engines such as podman/buildah or rkt, or an R package for creating Singularity image recipe files. Such efforts help to avoid lock-in and to design sustainable workflows based on concepts of *containerisation*, not on their implementation in Docker. If adoption of containerisation and R continue to grow, the missing pieces for a success predominantly lie in (a) coordination and documentation of activities to reduce repeated work in favour of open collaboration, (b) the sharing of lessons learned from use cases to build common knowledge and language, and (c) a sustainable continuation and funding for all of development, community support, and education. A concrete effort to work towards these pieces is to sustain the structure and captured status quo from this work in form of a *CRAN Task View on containerization*.

Author contributions

DN conceived of the presented idea and initialised the formation of the writing team, wrote sections not mentioned below, and revised the full first draft. CB .. RC and EH wrote the section on interfaces for Docker in R. DE wrote the introduction and section about Rocker. ME .. CF wrote the paragraphs about `r-online`, `dockerfiler`, `r-ci` and `r-db`. BW .. KR .. NR .. NX wrote the section on liftr. LS & NT wrote the section on Bioconductor. All authors approved the final version.

This articles was collaboratively written at https://github.com/nuest/rockerverse-paper/. The contributors page, commit history, and discussion issues provide a detailed view on the respective contributions.

Acknowledgements

Bibliography

GraalVM, Aug. 2019. URL https://en.wikipedia.org/w/index.php?title=GraalVM&oldid=912552499. Page Version ID: 912552499. [p3]

M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen. Orchestration of Microservices for IoT Using Docker and Edge Computing. *IEEE Communications Magazine*, 56(9):118–123, Sept. 2018. ISSN 0163-6804, 1558-1896. doi: 10.1109/MCOM.2018.1701233. [p2]

L. A. Barba. Terminologies for Reproducible Research. *arXiv:1802.03311 [cs]*, Feb. 2018. URL http://arxiv.org/abs/1802.03311. arXiv: 1802.03311. [p1]

Bennett et al. outsider: Install and run programs, outside of r, inside of r (under review). *Journal of Open Source Software*, 2019. URL https://github.com/ropensci/software-review/issues/282. [p15]

D. Bernstein. Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Computing*, 1(3): 81–84, Sept. 2014. doi: 10.1109/mcc.2014.51. [p2]

C. Boettiger. An introduction to Docker for reproducible research, with examples from the R environment. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, Jan. 2015. ISSN 01635980. doi: 10.1145/2723872.2723882. [p2, 4]

C. Boettiger and D. Eddelbuettel. An Introduction to Rocker: Docker Containers for R. *The R Journal*, 9 (2):527–536, 2017. doi: 10.32614/RJ-2017-065. [p2, 4, 16]

C. Boettiger, R. Lovelace, M. Howe, and J. Lamb. rocker-org/geospatial, Dec. 2019. [p2]

L. Cardozo. Faster Docker builds in Travis CI for R packages, 2018. URL https://lecardozo.github.io/2018/03/01/automated-docker-build.html. [p16]

Chan Zuckerberg Initiative, C. Boettiger, N. Ross, and D. Eddelbuettel. Maintaining Rocker: Sustainability for Containerized Reproducible Analyses, 2019. URL https://chanzuckerberg.com/eoss/proposals/maintaining-rocker-sustainability-for-containerized-reproducible-analyses/. [p2]

G. Csárdi and M. Salmon. *rhub: Connect to 'R-hub'*, 2019. URL https://CRAN.R-project.org/package=rhub. R package version 1.1.1. [p6]

Datadog. 8 surprising facts about real Docker adoption, June 2018. URL https://www.datadoghq.com/docker-adoption/. [p1, 2]

D. Donoho. 50 Years of Data Science. *Journal of Computational and Graphical Statistics*, 26(4):745–766, Oct. 2017. ISSN 1061-8600. doi: 10.1080/10618600.2017.1384734. URL https://doi.org/10.1080/10618600.2017.1384734. [p6]

D. L. Donoho. An invitation to reproducible computational research. *Biostatistics*, 11(3):385–388, July 2010. ISSN 1465-4644. doi: 10.1093/biostatistics/kxq028. URL https://doi.org10.1093/biostatistics/kxq028. [p15]

A. Eckert. Building and testing R packages with latest R-Devel, Feb. 2018. URL https://alexandereckert.com/post/testing-r-packages-with-latest-r-devel/. [p16]

D. Eddelbuettel. *sanitizers: C/C++ source code to trigger Address and Undefined Behaviour Sanitizers*, 2014. URL https://CRAN.R-project.org/package=sanitizers. R package version 0.1.0. [p16]

D. Eddelbuettel. Debugging with Docker and Rocker – A Concrete Example helping on macOS, Aug. 2019. URL http://dirk.eddelbuettel.com/blog/2019/08/05/. [p16]

I. Emsley and D. De Roure. A Framework for the Preservation of a Docker Container | International Journal of Digital Curation. *International Journal of Digital Curation*, 12(2), Apr. 2018. doi: 10.2218/ijdc.v12i2.509. URL https://doi.org/10.2218/ijdc.v12i2.509. [p16]

W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, Mar. 2015. doi: 10.1109/ISPASS.2015.7095802. [p2]

R. Gentleman and D. T. Lang. Statistical Analyses and Reproducible Research. *Journal of Computational and Graphical Statistics*, 16(1):1–23, Mar. 2007. ISSN 1061-8600. doi: 10.1198/106186007X178663. URL https://doi.org/10.1198/106186007X178663. [p15]

R. C. Gentleman, V. J. Carey, D. M. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, J. Gentry, K. Hornik, T. Hothorn, W. Huber, S. Iacus, R. Irizarry, F. Leisch, C. Li, M. Maechler, A. J. Rossini, G. Sawitzki, C. Smith, G. Smyth, L. Tierney, J. Y. Yang, and J. Zhang. Bioconductor: open software development for computational biology and bioinformatics. *Genome Biology*, 5(10):R80, Sept. 2004. ISSN 1474-760X. doi: 10.1186/gb-2004-5-10-r80. [p3]

N. Haydel, G. Madey, S. Gesing, A. Dakkak, S. G. de Gonzalo, I. Taylor, and W.-m. W. Hwu. Enhancing the Usability and Utilization of Accelerated Architectures via Docker. In *Proceedings of the 8th International Conference on Utility and Cloud Computing*, UCC '15, pages 361–367. IEEE Press, 2015. ISBN 978-0-7695-5697-0. URL http://dl.acm.org/citation.cfm?id=3233397.3233456. [p13]

K. Hornik, U. Ligges, and A. Zeileis. Changes on cran. *The R Journal*, 11(1):438–441, June 2019. URL http://journal.r-project.org/archive/2019-1/cran.pdf. [p1]

P. Jupyter. Jupyter Docker Stacks — docker-stacks latest documentation, 2018. URL https://jupyter-docker-stacks.readthedocs.io/en/latest/. [p7]

P. Jupyter, M. Bussonnier, J. Forde, J. Freeman, B. Granger, T. Head, C. Holdgraf, K. Kelley, G. Nalvarte, A. Osheroff, M. Pacer, Y. Panda, F. Perez, B. Ragan-Kelley, and C. Willing. Binder 2.0 - Reproducible, interactive, sharable environments for science at scale. *Proceedings of the 17th Python in Science Conference*, pages 113–120, 2018. doi: 10.25080/Majora-4af1f417-011. [p13]

G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):e0177459, May 2017. ISSN 1932-6203. doi: 10.1371/journal.pone.0177459. [p1, 2]

M. Lang, B. Bischl, and D. Surmann. batchtools: Tools for r to work on batch systems. *The Journal of Open Source Software*, 2(10):135, 2 2017. ISSN 2475-9066. doi: 10.21105/joss.00135. [p15]

S. Lopp. Package Manager 1.1.0 - No Interruptions, Jan. 2019. URL https://blog.rstudio.com/2019/11/07/package-manager-v1-1-no-interruptions/. [p6]

B. Marwick. How computers broke science – and what we can do to fix it, Nov. 2015. URL http://theconversation.com/how-computers-broke-science-and-what-we-can-do-to-fix-it-49938. [p15]

B. Marwick, C. Boettiger, and L. Mullen. Packaging Data Analytical Work Reproducibly Using R (and Friends). *The American Statistician*, 72(1):80–88, Jan. 2018. ISSN 0003-1305. doi: 10.1080/00031305.2017.1375986. URL https://doi.org/10.1080/00031305.2017.1375986. [p13, 15]

Microsoft. About Microsoft R Open: The Enhanced R Distribution . MRAN, a. URL https://mran.revolutionanalytics.com/rro. [p3]

Microsoft. The Benefits of Multithreaded Performance with Microsoft R Open . MRAN, b. URL https://mran.revolutionanalytics.com/documents/rro/multithread. [p3]

Microsoft. CRAN Time Machine - MRAN, 2019a. URL https://mran.microsoft.com/timemachine. [p2]

Microsoft. Linux Containers on Windows, Sept. 2019b. URL https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/linux-containers. [p2]

S. Muñoz. The history of Docker's climb in the container management market, June 2019. URL https://searchservervirtualization.techtarget.com/feature/The-history-of-Dockers-climb-in-the-container-management-market. [p1]

D. Nüst. nuest/rocker-win: Proof on concept for R in Windows-based Docker Containers, Dec. 2019. URL https://doi.org/10.5281/zenodo.3584107. [p5]

D. Nüst, M. Konkol, E. Pebesma, C. Kray, M. Schutzeichel, H. Przibytzin, and J. Lorenz. Opening the Publication Process with Executable Research Compendia. *D-Lib Magazine*, 23(1/2), Jan. 2017. ISSN 1082-9873. doi: 10.1045/january2017-nuest. URL https://doi.org/10.1045/january2017-nuest. [p16]

OCI. Open Containers Initiative - About, 2019. URL https://www.opencontainers.org/about. [p2]

R Core Team. *Writing R extensions*, 1999. URL https://cran.r-project.org/doc/manuals/r-devel/R-exts.html. [p15, 16]

R-hub project. R-hub Docs, 2019. URL https://docs.r-hub.io/. [p6]

A. Ratesic. Building a Repository of Alpine-based Docker Images for R, Part II – Curiosity Killed the Cat – (... but satisfaction brought it back.), Nov. 2018. URL https://velaco.github.io/my-dockerfile-for-r-shiny-based-on-alpine-linux-II/. [p6]

J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan. 2015. ISSN 0893-6080. doi: 10.1016/j.neunet.2014.09.003. [p13]

T-Mobile, J. Nolis, and H. Nolis. Enterprise Web Services with Neural Networks Using R and TensorFlow, Nov. 2018. URL https://opensource.t-mobile.com/blog/posts/r-tensorflow-api/. [p10]

H. Wickham, M. Averick, J. Bryan, W. Chang, L. McGowan, R. François, G. Grolemund, A. Hayes, L. Henry, J. Hester, M. Kuhn, T. Pedersen, E. Miller, S. Bache, K. Müller, J. Ooms, D. Robinson, D. Seidel, V. Spinu, K. Takahashi, D. Vaughan, C. Wilke, K. Woo, and H. Yutani. Welcome to the Tidyverse. *Journal of Open Source Software*, 4(43):1686, Nov. 2019. ISSN 2475-9066. doi: 10.21105/joss.01686. URL https://joss.theoj.org/papers/10.21105/joss.01686. [p13]

Wikipedia. Renjin, Aug. 2018. URL https://en.wikipedia.org/w/index.php?title=Renjin&oldid=857160986. Page Version ID: 857160986. [p3]

N. Xiao. *liftr: Containerize R Markdown Documents for Continuous Reproducibility*, 2019. URL https://CRAN.R-project.org/package=liftr. R package version 0.9.2. [p8]

Y. Xie, J. J. Allaire, and G. Grolemund. *R Markdown: The Definitive Guide*. Chapman and Hall/CRC, 2018. [p8]

H. Ye. Docker Setup for R package Development, 2019. URL https://haoye.us/post/2019-10-10-docker-for-r-package-development/. [p16]

Daniel Nüst
University of Münster
Institute for Geoinformatics
Heisenbergstr. 2
48149 Münster, Germany
0000-0002-0024-5046
daniel.nuest@uni-muenster.de

Robrecht Cannoodt
Ghent University
Data Mining and Modelling for Biomedicine group
VIB Center for Inflammation Research
Technologiepark 71
9052 Ghent, Belgium
0000-0003-3641-729X
robrecht@cannoodt.dev

Dirk Eddelbuettel
University of Illinois at Urbana-Champaign
Department of Statistics
Illini Hall, 725 S Wright St
Champaign, IL 61820, USA
0000-0001-6419-907X
dirkd@eddelbuettel.com

Mark Edmondson
IIH Nordic A/S, Google Developer Expert for GCP

mark@markedmondson.me

Colin Fay
ThinkR
5O rue Arthur Rimbaud
93300 Aubervilliers, France
0000-0001-7343-1846
contact@colinfay.me

Karthik Ram
Berkeley Institute for Data Science
University of California
Berkeley, CA 94720, USA
0000-0002-0233-1757
karthik.ram@berkeley.edu

Noam Ross
EcoHealth Alliance
460 W 34th St., Ste. 1701
New York, NY 10001, USA
0000-0002-0233-1757
ross@ecohealthalliance.org

Nan Xiao
Seven Bridges Genomics
529 Main St, Suite 6610
Charlestown, MA 02129, USA
0000-0002-0250-5673
me@nanx.me

Lori Shepherd
Roswell Park Comprehensive Cancer Center
Elm & Carlton Streets
Buffalo, New York, 14263, USA
0000-0002-5910-4010
lori.shepherd@roswellpark.org

Nitesh Turaga
Roswell Park Comprehensive Cancer Center
Elm & Carlton Streets
Buffalo, New York, 14263, USA
0000-0002-0224-9817
nitesh.turaga@roswellpark.org

Jacqueline Nolis
Nolis, LLC
Seattle, WA, USA
0000-0001-9354-6501
jacqueline@nolisllc.com

Heather Nolis

T-Mobile
12920 Se 38th St.
Bellevue, WA, 98006
heather.wensler1@t-mobile.com

Hong Ooi
Microsoft
Level 5, 4 Freshwater Place
Southbank, VIC 3006, Australia
hongooi@microsoft.com

Ellis Hughes
Fred Hutchinson Cancer Research Center
Vaccine and Infectious Disease
1100 Fairview Ave. N., P.O. Box 19024
Seattle, WA 98109-1024, USA
ehhughes@fredhutch.org

Sean Lopp
RStudio, Inc
RStudio
250 Northern Ave
Boston, MA 02210
sean@rstudio.com

Ben Marwick