

The Rockerverse: Packages and Applications for Containerization with R

by Daniel Nüst, Dirk Eddelbuettel, Dom Bennett, Robrecht Cannoodt, Dav Clark, Gergely Daroczi, Mark Edmondson, Colin Fay, Ellis Hughes, Sean Lopp, Ben Marwick, Heather Nolis, Jacqueline Nolis, Hong Ooi, Karthik Ram, Noam Ross, Lori Shepherd, Nitesh Turaga, Craig Willis, Nan Xiao, Charlotte Van Petegem

Abstract The Rocker Project provides widely-used Docker images for R across different application scenarios. This article surveys downstream projects building upon Rocker and presents the current state of R packages for managing Docker images and controlling containers. These use cases and the variety of applications demonstrate the power of Rocker specifically and containerisation in general. We identified common themes across this diversity: reproducible environments, scalability and efficiency, 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 series¹, the global growth of the R and R-Ladies user groups², or the foundation and impact of the R Consortium³. 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 led to numerous projects extending or building upon Rocker images, ranging from reproducible research⁴ 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 contribution 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 use cases and applications, starting with the R packages specifically for interacting with Docker, second-level packages using containers indirectly 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 provided 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 LXC⁵ or Singularity (Kurtzer et al., 2017), Docker has become synonymous with “containerization”—the method of taking software artefacts and bundling them in such a way that use becomes standardized and portable across operating 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 (Wikipedia contributors, 2020). Typically a container runs one process, whereas virtualization may run whole operating systems at a

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“Reproducible” in the sense of the Claerbout/Donoho/Peng terminology (Barba, 2018).

⁵<https://en.wikipedia.org/wiki/LXC>

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 led 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 led 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 proved 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 built 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 funding 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-performance 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 exist⁶. Yet the prevalence 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.

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 an 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.

⁶See <https://github.com/nuest/rodman> and <https://github.com/rocker-org/rocker-versioned/issues/187>

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's basic functionality, while evolving quickly and with small concern for avoiding breaking changes, is unchanged for a long time so a core function such as `harbor::docker_run(image = "hello-world")` still works despite the stopped development.

stevedore (<https://cran.r-project.org/package=stevedore>) is currently the most powerful Docker client in R. It interfaces with the Docker daemon over the Docker HTTP API⁷ via a Unix socket on Linux or MacOS, over a named pipe on Windows, or over an HTTP/TCP connection. The package is the only one not using system calls to the docker CLI for managing images and containers and easily exposes connections to remote Docker daemons, which has to be configured on the Docker level otherwise. Using the API gives access to more information and is system independent and likely more reliable than parsing command line output. **stevedore**'s own interface is automatically generated based on the OpenAPI specification of the Docker daemon, but still similar to the Docker CLI. The interface is similar to R6 objects, in that a `stevedore_object` has a number of functions attached to it that can be called, and multiple specific versions of the Docker API can be supported thanks to the automatic generation⁸.

AzureContainers is an interface to a number of container-related services in Microsoft's **Azure Cloud** (Ooi, 2019). While it is mainly intended for working with Azure, as a convenience feature it includes lightweight, cross-platform shells to Docker and Kubernetes (tools `kubect1` 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 allows executing and interacting with containers, which can use either Docker or Singularity as a backend (Cannoodt and Saelens, 2019). 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, building Docker images, and deploying Docker containers. The package follows the increasingly used piping paradigm of the tidyverse style of programming for chaining R functions representing the instructions in a Dockerfile. An existing Dockerfile can be used as a template. **dockyard** also includes wrappers for common steps, such as installing an R package or copying files, and build-in functions for building an image running a container, to make using Docker even more approachable to R users with a single API.

dockermachine (<https://github.com/cboettig/dockermachine>) is an R package to provide a convenient interface to **Docker Machine** from R. The 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 transparently forward all commands issued on the local Docker CLI to a selected (remote) virtual host. Docker Machine was especially crucial for local use in early days of Docker, when no native 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 is functional with a current version of `docker-machine` (0.16.2). It potentially lowers the barriers for R users to run containers on various hosts, if using the Docker Machine CLI directly is perceived as a

⁷<https://docs.docker.com/engine/api/latest/>

⁸See <https://github.com/richfitz/stevedore/blob/master/development.md>.

barrier, or enables scripted workflows with remote processing.

Use cases and applications

Image stacks for communities of practice

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 1823 R software packages as of October 30th 2019, as well as packages containing annotation or experiment data. *Bioconductor* has a semi-annual release cycle, each release is associated with a particular version of R, and Docker images are provided for current and past versions of *Bioconductor* for convenience and reproducibility. All images, included are described on the *Bioconductor* web site (see <https://bioconductor.org/help/docker/>), created with Dockerfiles maintained on GitHub, and distributed through Docker Hub⁹. *Bioconductor*'s 'base' Docker images are built on top of the rocker/rstudio image. *Bioconductor* installs packages based on the R version in combination with the *Bioconductor* version, and therefore uses *Bioconductor* version tagging devel and RELEASE_X_Y, e.g., RELEASE_3_10. Past and current combinations of R and *Bioconductor* will therefore be accessible via a specific image tags. The *Bioconductor* Dockerfile selects the desired version of R from Rocker, adds required system dependencies, and uses the **BiocManager** package for installing appropriate versions of *Bioconductor* packages. 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. A recent audit of the *Bioconductor* image stack Dockerfile led to the deprecation of several community-maintained images, because the numerous specific images became too hard to understand, complex to maintain, and cumbersome to extent. As part of the simplification, a recent innovation is to produce a `bioconductor_docker:devel` image to emulate the *Bioconductor* nightly Linux build machine as closely as possible. This image contains the build system environment variables and the *system dependencies* needed to install and check almost all (1813 out of 1823) *Bioconductor* software packages and saves users and package developers from managing these themselves. Furthermore 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 can persist installed packages across Docker sessions or image updates. Many R users pursue flexible work flows tailored to particular analysis needs, rather than standardized work flows. The new `bioconductor_docker` image is well-suited for this pattern, while `bioconductor_docker:devel` provides developers with a test environment close to *Bioconductor*'s build system.

Data Science is a widely discussed topic among all academic disciplines (e.g., Donoho, 2017). The discussions shed a light on the tools and craftsmanship behind the analysis of data with computational methods. The practice of Data Science often involves a combination tools and software stacks and requires a cross-cutting skillset. This complexity and an inherent concern for openness and reproducibility in the Data Science community lead to Docker being used widely. The remainder of this section presents exemplary Docker images and image stacks featuring R intended for Data Science. 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 naturally also the IRKernel (<https://irkernel.github.io/>), an R kernel for Jupyter so that Jupyter Notebooks can contain R code cells. R is also included in the catchall `jupyter/datascience-notebook` image¹⁰. 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. R is installed using the Conda package manager, which can manage environments for various programming languages, pinning both the R version and the versions of R packages¹¹. 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 learning libraries such as Tensorflow and Keras (see also image `rocker/ml` in Section **Common or public work environments**), 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 out¹². 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

⁹See https://github.com/Bioconductor/bioconductor_docker and <https://hub.docker.com/u/bioconductor> respectively.

¹⁰<https://jupyter-docker-stacks.readthedocs.io/en/latest/using/selecting.html>

¹¹See `jupyter/datascience-notebook`'s Dockerfile at <https://github.com/jupyter/docker-stacks/blob/master/datascience-notebook/Dockerfile#L47>.

¹²Originally, 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>

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* (<http://gigantum.com/>) is a platform for open and decentralized data science with a focus on using automation and user-friendly tools for easy sharing of reproducible computational workflows. Gigantum builds on the *Gigantum Client* (running either locally or on a remote server) for development and execution of data-focused *Projects*, which can be stored and shared via the *Gigantum Hub* or via a zipfile export. The Client is a user friendly interface to a backend using Docker containers to package, build, and run Gigantum projects. It is configured to use a default set of Docker base images (<https://github.com/gigantum/base-images>), and users are able to define and configure their own custom images. The available images include two with R based on Ubuntu Linux have the [c2d4u CRAN PPA](#) pre-configured for installation of binary R packages¹³. The R images vary in the included authoring environment, i.e., Jupyter in *r-tidyverse* or both Jupyter & RStudio in *rstudio-server*. The independent image stack can be traced back to the Gigantum environment and its features. The R images are based on Gigantum's *python3-minimal* image, originally to keep the existing front-end configuration, but also to provide a consistent Python-to-R interoperability. The Dockerfiles also use build args to specify bases, for example for different version of NVIDIA CUDA for GPU processing¹⁴ so that appropriate GPU drivers can be enabled automatically when supported. Furthermore, Gigantum's focus lies on environment management via GUI and ensuring a smooth user interaction, e.g., with reliable and easy conflict detection and resolution. For this reason, project repositories store authoritative package information in a separate file per package, allowing Git to directly detect conflicts and changes. A Dockerfile is generated from this description that inherits from the specified base image, and additional custom Docker instructions may be appended by users, though Gigantum's default base images do not currently include the *littler* tool, which is used by *Rocker* to install packages within Dockerfiles. Because of these specifics, instructions from *rocker/r-ubuntu* could *not* be readily re-used in this image stack (see Section [Conclusions](#)). Both approaches enable *apt* as an installation method, and this is exposed via the GUI-based environment management¹⁵. The image build and publication process is scripted with Python and JSON template configuration files, unlike *Rocker* which relies on plain Dockerfiles. A minor reason in the inception of the images were also project constraints requiring a *Rocker*-incompatible licensing of the Dockerfiles, i.e., the MIT License.

Capture and create environments

Community maintained images provide a solid basis for extension as part of individual requirements. 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. While authoring and managing an environment with Docker by hand is possible and feasible for experts¹⁶, the following examples show the power of automation when environments become too cumbersome. Especially *version pinning*, with packages **remotes** and **versions** for R or by using MRAN, and with system package managers for different operating systems, can greatly increase the reproducibility of built images and are common approaches.

dockerfiler (<https://github.com/ColinFay/dockerfiler/>) is an R package designed for building Dockerfiles straight from R. A scripted creation of a Dockerfile enables iteration and automation, for example for packaging applications for deployment (see [Deployment and continuous delivery](#)). Being scriptable from R developers can leverage the tools available in R to parse a DESCRIPTION file, to get system requirements, to list dependencies, versions, etc. **containerit** (<https://github.com/o2r-project/containerit/>) attempts to take this one step further and includes these tools to automatically create a Dockerfile that can execute a given workflow ([Nüst and Hinz, 2019](#)). **containerit** accepts R sessionInfo objects as input and provides helper functions to derive these from workflows, e.g., an R script or R Markdown document, by analysing the session state at the end of the workflow. It relies on the **sysreqs** (<https://github.com/r-hub/sysreqs/>) package and it's mapping of package system dependencies to platform specific installation package names¹⁷. **containerit** uses [stevedore](#) to streamline the user interaction and improve the created Dockerfiles, e.g., by running a container for the desired base image to extract the already available R packages. **dockr**

¹³<https://docs.gigantum.com/docs/using-r>

¹⁴See https://github.com/gigantum/base-images/blob/master/_templates/python3-minimal-template/Dockerfile for the Dockerfile of *python3-minimal*.

¹⁵See <https://docs.gigantum.com/docs/environment-management>

¹⁶See, e.g., this tutorial by RStudio on how to manage environments and package versions and to ensure deterministic image builds with Docker: <https://environments.rstudio.com/docker>.

¹⁷See <https://sysreqs.r-hub.io/>.

(<https://github.com/smaakage85/docker>) is a very similar package attempting to mirror a given R session, including local non-CRAN packages (Kjeldgaard, 2019). Users can manually add statements for non-R dependencies to the Dockerfile. **liftr** aims to solve the problem of persistent reproducible reporting in statistical computing based on the R Markdown format (Xie et al., 2018) for dynamic documents (<https://nanx.me/liftr/>, Xiao, 2019). The irreproducibility of authoring environments can become an issue for collaborative documents and large-scale platforms for processing documents. **liftr** makes the document the main and sole workflow control file and the only file that needs to be shared between collaborators for consistent environments, e.g. demonstrated in the DockFlow project (<https://dockflow.org>). It introduces new fields to the R Markdown document header, allowing users to manually declare the dependencies, including versions, for rendering the document. The package then generates a Dockerfile from this metadata and provides a utility function to render the document inside a Docker container, i.e., `render_docker("foo.Rmd")`. An RStudio addin even allows compilation of documents with a single push of a button.

System dependencies are the domain of Docker, but for a full description of the computing environment one must also manage the R version and the R packages. R versions are available via the versioned Rocker image stack (Boettiger and Eddelbuettel, 2017). **r-online** leverages these images and provides 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. R package management can be achieved with MRAN, or with packages such as **checkpoint** and **renv**, which can naturally be applied within images and containers. For example, **renv** (<https://rstudio.github.io/renv/>) helps users to manage the state of the R library in a reproducible way, further providing isolation and portability (Ushey, 2019a). While **renv** does not cover system dependencies, the **renv**-based environment can be transferred into a container either by restoring the environment based on the main configuration file `renv.lock` or by storing the **renv**-cache on the host and not in the container (Ushey, 2019b). With both the system dependencies and R packages consciously managed in a Docker image, users can start using containers as the *only* environment for their workflows, which allows them to work independently of physical computers¹⁸ and to assert a specific degree of confidence in the stability of a developed software (cf. `README.Rmd` in Marwick, 2017).

Development, debugging, and testing

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 for the purposes of developing R packages. 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 and quick evaluation**, because developers and users can readily start and later discard a container to investigate a bug report or try out a piece of software (cf. Ooms, 2017). The container does not affect their regular system. Using the Rocker images with RStudio, these disposable environments lack no development comfort (cf. Section **Packaging research reproducibly**). Ooms (2017) describes how `docker exec` can be used to get a root shell in a container for customization during software evaluation. Eddelbuettel (2019) describes an example how a Docker container was used to debug an issue with a package only occurring with a particular version of Fortran, and using tools which are not readily available on all platforms (e.g., not on macOS).

Second, the strong integration of **system libraries in core packages** in the **R-spatial community** makes containers essential for stable and proactive development of common classes for geospatial data modelling and analysis. For example, GDAL (GDAL/OGR contributors, 2019) is a crucial library in the geospatial domain. GDAL is a system dependency allowing R packages such as **sf**, which provides the core data model for geospatial vector data, or **rgdal**, to accommodate users to be able to read and write hundreds of different spatial raster and vector formats. **sf** and **rgdal** have hundreds of indirect reverse imports and dependencies and therefore the maintainers spend a lot of efforts not to break these. Purpose built Docker are used to prepare for upcoming releases of system libraries, individual bug reports, and for the lowest supported versions of system libraries¹⁹.

Third, there are special purpose 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

¹⁸Allowing them to be digital "nomads", cf. J. Bryan's <https://github.com/jennybc/docker-why>.

¹⁹Cf. <https://github.com/r-spatial/sf/tree/master/inst/docker>, https://github.com/Nowosad/rspatial_proj6, and <https://github.com/r-spatial/sf/issues/1231>

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.

Fourth, 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** project provides “a collection of services to help R package development”, with the package builder as the most prominent one ([R-hub project, 2019](#)). R-hub 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 compilers²⁰. The images go beyond the configurations, or *flavours*, used by CRAN for checking packages²¹, 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 tests²². 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 separates test environment, test code, and current working copy of the code. Another use case where a container helps to standardise tests across operating systems is detailed the vignettes of the package **RSelenium** ([Harrison, 2019](#)). The package recommends Docker for running the **Selenium** Server application needed to execute test suites on browser-based user interfaces and webpages, but requires users to manually manage the Docker containers.

Fifth, Docker images can be used on **continuous integration (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 \(2018\)](#) saved time, also on Travis CI, by re-using the testing image as the base for an image intended for publication on Docker Hub. `r-ci` is in turn used with **GitLab CI**, which itself is built on top of Docker images: the user specifies a base Docker image, and the whole tests are run inside this environment. The `r-ci` image stack combines rocker versioning and a series of tools specifically designed for testing in a fixed environment with a customized list of preinstalled packages. Especially for long running tests or complex system dependencies, these approaches to separate installation of build dependencies with code testing streamline the development process. Not due to a concern about time, but to control the environment used on a CI server, even this manuscript is rendered into a PDF and deployed to a GitHub-hosted website with every change (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.

Processing

The portability of containerized environments becomes particularly useful for improving expensive processing of data or shipping complex processing pipelines. First, it is possible to **offload complex processing to a server** or clouds, and also parallel executing of processes for speeding up or serving of many users. **batchtools** provides a parallel implementation of Map for various schedulers ([Lang et al., 2017](#)). The package can **schedule jobs with Docker Swarm** **googleComputeEngineR** has the function `gce_vm_cluster()` to 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 a Docker container with the same R environments. **googleComputeEngineR** integrates with the R parallelisation package **future** to enable this with only a few lines of R code²³. **Google Cloud Run** is a CaaS (Containers as a Service) platform. Users can launch containers using any Docker image without worrying about underlying infrastructure in a so called serverless configuration. The service takes care of network ingress, scaling machines up and down, 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 need be with little or no changes to existing code, as demonstrated by the proof of

²⁰See <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/>

²¹https://cran.r-project.org/web/checks/check_flavors.html

²²**dockertest** is not actively maintained, but mentioned still because of its interesting approach.

²³<https://cloudyr.github.io/googleComputeEngineR/articles/massive-parallel.html>

concept `cloudRunR`²⁴, which uses Cloud Run to create a scalable R-based API using **plumber**. **Google Cloud Build** and the Google Container Registry are a continuous integration service respectively image registry that offload building of images to the cloud, while serving the needs of commercial environments such as private Docker images or image stacks. As Google Cloud Build itself can run any container, the package **googleCloudRunner** demonstrates how R can be used as the control language for one time or batch processing jobs and scheduling of jobs²⁵. **drake** is a workflow manager for data science projects (Landau, 2018). It features implicit parallel computing and automated detection of the parts of the work that actually needs to be reexecuted. **drake** has been demonstrated to run inside containers for high reproducibility²⁶, but also how to use **future** package's function `makeClusterPSOCK()` to send parts of the workflow to a Docker image for execution²⁷ (see package's function documentation; Bengtsson, 2020). In the latter case, the container control code must be written by the user and the **future** package ensures the host and worker can connect for communicating over socket connections. **RStudio Server Pro** includes a functionality called **Launcher** (since version 1.2, released in 2019). It gives users the ability to spawn R sessions and background/batch jobs in a scalable way on external clusters, e.g., **Kubernetes based on Docker images** or **Slurm** clusters, and optionally, with Singularity containers. A benefit of the proprietary Launcher software is the ability for R and Python users to leverage containerisation's advantages in RStudio without writing specific deployment scripts or learning about Docker or managing clusters at all.

Second, containers are perfectly suited for **packaging and executing software pipelines** and required data. Containers allow building complex processing pipelines that are independent from the host programming language. Due to its original use case (see **Introduction**), Docker has no standard mechanisms for chaining containers together; it lacks definitions and protocols for environment variables, volume mounts and/or ports that could enable transfer of input (parameters and data) and output (results) to and from containers. Some packages, e.g., **containerit**, do provide Docker images that can be used similarly to CLIs, but their usage is cumbersome²⁸. **outsider** (<https://docs.ropensci.org/outsider/>) tackles the problem of integrating external programs into an R workflow (Bennett et al., 2020). Installation and usage of external programs can be difficult, convoluted and even impossible if the platform is incompatible. Therefore **outsider** uses the platform-independent Docker images to encapsulate processes in *outsider modules*. Each outsider module has a Dockerfile and an R package with functions for interacting with the encapsulated tool. Using only R functions, an end-user can install a module with the **outsider** package and then call module code to integrate a tool into their own R-based workflow seamlessly. The **outsider** package and module manage the containers and handle the transmission of arguments and the transfer of files to and from a container. These functionalities also allow a user to launch module code on a remote machine via SSH, expanding the potential computational scale. Outsider modules can be hosted code-sharing services, e.g., on GitHub, and **outsider** contains discovery functions for them.

Deployment and continuous delivery

The cloud is the natural environment of containers, and subsequently containers are the go-to mechanism to deploy R server applications. More and more continuous integration (CI) and continuous delivery (CD) services also use containers, opening up new options for use. The controlled nature of containers, i.e., the possibility to abstract internal software environment from a minimal dependency outside of the container is crucial, for example to match test or build environments with production environments or transfer runnable entities to as-a-service infrastructures.

First, different packages use containers for the **deployment of R and Shiny apps**. **Shiny** is a popular package for creating interactive online dashboards with R and it enables users with very diverse backgrounds to create stable and user friendly web applications. For example, **ShinyProxy** (<https://www.shinyproxy.io/>) is an open source tool to deploy Shiny apps in an enterprise context. They feature single sign-on, but also in scientific use cases (e.g., Savini et al., 2019; Glouzon et al., 2017). ShinyProxy uses Docker containers to isolate user sessions and to achieve scalability for multi-user scenarios with multiple apps. ShinyProxy itself is written in Java to accommodate corporate requirements and may itself run in a container for stability and availability. The tool is built on ContainerProxy (<https://www.containerproxy.io/>), which provides similar features for executing long-running R jobs or interactive R sessions. The started containers can run on a regular Docker host but also in clusters. Another example is the package **golem** (<https://github.com/ThinkR-open/golem>),

²⁴<https://github.com/MarkEdmondson1234/cloudRunR>

²⁵<https://code.markedmondson.me/googleCloudRunner/articles/cloudbuild.html>

²⁶See for example <https://github.com/joelnitta/pleurosoripsis> or <https://gitlab.com/ecohealthalliance/drake-gitlab-docker-example>, the latter even running in a continuous integration platform (cf. **Development, debugging, and testing**).

²⁷<https://docs.ropensci.org/drake/index.html?q=docker#with-docker>

²⁸<https://o2r.info/containerit/articles/container.html>

which makes an heavy use of dockerfiler when it comes to creating the Dockerfile for building production-grade Shiny applications and deploying them. **googleComputeEngineR** (<https://cloudyr.github.io/googleComputeEngineR/>) enables quick deployments of key R services, such as RStudio and Shiny, onto cloud virtual machines (VMs) with Google Cloud Compute Engine (Edmondson, 2019). The package utilises Dockerfiles to move the labour of setting up those services from the user to a premade Docker image, which is configured and run in the cloud VM. For example, by specifying the template `template="rstudio"` in functions `gce_vm_template()/gce_vm()` an up to date RStudio Service image is launched for development work, while specifying `template="rstudio-gpu"` will launch an RStudio Server image with a GPU attached, etc.

Second, containers can be used to create **platform installation packages** in a DevOps setting. The **OpenCPU** system provides an HTTP API for data analysis based on R. Ooms (2017) describes how various platform-specific installation files for OpenCPU are created using Docker Hub: the automated builds install the software stack from source on different operating systems; afterwards a script file downloads the images and extracts the OpenCPU binaries.

Third, containers can greatly facilitate the **deployment to existing infrastructures**. **Kubernetes** (<https://kubernetes.io/>) is a container-orchestration system for managing container-based application deployment and scaling. A *cluster* of containers, orchestrated as a single deployment, e.g., with Kubernetes, can mitigate limitations on request volumes or a container occupied with computationally intensive task. A cluster features load-balancing, autoscaling of containers across numerous servers (in the cloud or on premise), and restarting failed ones. It may be your organisation has a Kubernetes cluster already for other applications, or you may use one of the different provides for managed Kubernetes clusters. 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. R applications, big parallel tasks, or scheduled batch jobs can be deployed in a scalable way using Kubernetes, and the deployment can even be triggered by changes to code repositories (i.e., CD), see blog post "R on Kubernetes" (Edmondson, 2018). The package **googleKubernetesR** (<https://github.com/RhysJackson/googleKubernetesR>) is a proof of concept for wrapping the Google Kubernetes Engine API, Google's hosted Kubernetes solution, in an easy to use R package. The package **analogsea** provides a solution to programmatically create and destroy cloud VMs on the Digital Ocean platform (Chamberlain et al., 2019). It also includes R wrapper functions to install Docker in such a VM, manage images, and control containers straight from R functions. These functions are translated to Docker CLI commands and transferred transparently to the respective remove machine using SSH. **AzureContainers** is an umbrella package providing interfaces three commercial services of Microsoft's Azure Cloud, namely **Container Instances** for running individual containers, **Container Registry** for private image distribution, and **Kubernetes Service** for orchestrated deployments. While a package like **plumber** provides the infrastructure for turning an R workflow a service, for production purposes it is usually necessary to take into account scalability, reliability and ease of management. **AzureContainers** provides an R-based interface to these features and thereby simplifies complex infrastructure management to a number of R function calls, given an Azure account with sufficient credit²⁹. **Heroku** is a further cloud platform as a service provider, who supports container-based applications. **heroku-docker-r** (<https://github.com/virtualstaticvoid/heroku-docker-r>) is an independent project providing a template for deploying R applications based on Heroku's image stack, including multiple examples for interfacing R with other programming languages. Yet the approach requires a manual management of the computing environment.

The prevalence of Docker in industry naturally leads to usage of containers with R in such settings as well, as customers already manage platforms in Docker containers. These products often entail a high amount of open source software in combination with proprietary layers adding the relevant commercializable features. One such example is RStudio's data science platform **RStudio Team**. It allows teams of data scientists and their respective IT/DevOps groups to develop and deploy code in R and Python around the RStudio Open Source Server inside of Docker images, without requiring users to learn new tools or directly interact with containers. The best practices for **running RStudio with Docker containers** as well as **Docker images** for RStudio's commercial products are publicly available. **## Using R to power enterprise software in production environments**

R has been historically viewed as a tool for analysis and scientific research, but not for creating software that corporations can rely on for production services. However, thanks to advancements in R running as a web service, along with the ability to deploy R in Docker containers, modern enterprises are now capable of having real-time machine learning powered by R. A number of packages and projects enabled R to respond to client requests over TCP/IP and local socket servers, such as **Rserve**, **svSocket**, **rApache** and more recently **plumber** (<https://www.rplumber.io/>) and **RestRserve** (<http://restrserve.org>), which even processes incoming requests in parallel with forked processes using **Rserve**. The latter two also provide documentation for deployment with Docker or ready to

²⁹See "Deploying a prediction service with Plumber" vignette for details: https://cran.r-project.org/web/packages/AzureContainers/vignettes/vig01_plumber_deploy.html.

use automated builds of images³⁰. These software allow other (remote) processes and programming languages to interact with R and to expose R-based function in a service architecture with HTTP APIs. APIs based on these package can be deployed with scalability and high availability using containers. This pattern of deploying code matches those used by software engineering services created in more established languages in the enterprise domain, such as Java or Python, and R can be used alongside those languages as a first class member of a software engineering technical stack.

CARD.com implemented a web application for the optimization of the acquisition flow and the real-time analysis of debit card transactions. The software used **Rserve** and rApache and was deployed in Docker containers. The R session behind **Rserve** acted as a read-only in-memory database, which was extremely fast and scalable, for the many concurrent rApache processes responding to the live-scoring requests of various divisions of the company. Similarly dockerized R scripts were responsible for the ETL processes and even the client-facing email, text message and push notification alerts sent in real-time based on card transactions. The related Docker images were made available at <https://github.com/cardcorp/card-rocker>. The images extend rocker/r-base and additionally entailed an SSH client and a workaround for being able to mount SSH keys from the host, Pandoc, the Amazon Web Services (AWS) SDK, and Java, which is required by the AWS SDK. The AWS SDK allowed to run R consumers reading from real-time data processing streams of **AWS Kinesis**³¹. The applications were deployed on Amazon Elastic Container Service (ECS). The main learnings from using R in Docker was the importance of not only pinning the R package versions via MRAN, but also moving away from Debian testing to a distribution with long-term support. For the use case at hand, this switch served the priority to have more control over upstream updates, and to minimize the risk of breaking the automated builds of the Docker images and production jobs.

The AI @ T-Mobile team 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). For example, one model quickly identifies if a message is from a customer or not (see **Shiny**-based demo, Nolis and Werdell, 2019), others tell which customers are likely to make a repeat purchase. If a data scientist creates a machine learning model and exposes it through a **plumber** API, then someone else on the marketing team could write software that sends different emails depending on that real-time prediction. The models are convolutional neural networks that use the **keras** package and run in a Rocker Docker image. The corresponding Dockerfiles are published on **GitHub**. Since the models power tools for agents and customers, they need to have extremely high uptime and reliability. The AI @ T-Mobile team found that the models performed well and today these models power real-time services that are called over a million times a day.

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.

The **Binder** project, maintained by the team behind Jupyter, makes it possible for users to **create and share 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 files³². 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. This is partly because **repo2docker** does not support arbitrary base images due to the complex auto-generation of the Dockerfile instructions. Two approaches make using Binder easier. for R users. First, **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. 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

³⁰See <https://www.rplumber.io/docs/hosting.html#docker>, <https://hub.docker.com/r/trestletech/plumber/> and <https://hub.docker.com/r/rexyai/restrserve/>.

³¹See useR!2017 talk "Stream processing with R in AWS".

³²See supported file types at https://repo2docker.readthedocs.io/en/latest/config_files.html.

installs additional packages from CRAN and Bioconductor that are not already part of these images. It short circuits 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 separately managed environments and thereby reduces work and increases portability and reproducibility. Second, the **Whole Tale** project (<https://wholetale.org>) combines the strengths of the Rocker Project's curated Docker images with `repo2docker`. Whole Tale is a National Science Foundation (NSF) funded project developing a scalable, open-source, multi-user platform for reproducible research (Brinckman et al., 2019; Chard et al., 2019b). A central goal of the platform is to enable researchers to easily create and publish executable research objects³³ associated with published research (Chard et al., 2019a). Using Whole Tale, researchers can create and publish Rocker-based reproducible research objects to a growing number of repositories including DataONE member nodes, Zenodo and soon Dataverse. Additionally, Whole Tale supports automatic data citation and is working on capabilities for image preservation and provenance capture to improve the transparency of published computational research artifacts (Mecum et al., 2018; McPhillips et al., 2019). For R users, Whole Tale extends the Jupyter Project's `repo2docker` component to simplify the customization of R-based environments for researchers with limited experience with either Docker or Git. Multiple options have been discussed to allow users to change the base image used in `repo2docker` from the default Ubuntu LTS (long-term support) required to support the Rocker Project images. Whole Tale implemented a custom `RockerBuildPack`³⁴ to support customization of the `rocker/geospatial` image through `repo2docker` composability³⁵. This works in part because Rocker images are based on a Debian distribution, so the instructions created by `repo2docker` for Ubuntu work because of compatible shell and package manager.

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, containers 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 necessary or in case of server destruction or failure. In any of these cases, users need a method to interact with the containers, be it an 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 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 necessary. When working with multiple servers, an automation framework such as **Ansible** may be useful for managing users, permissions, 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 shared computing environments and sandboxing can greatly improve the process. First, **prepared environments for teaching** are especially helpful for (a) intro-

³³In Whole Tale a *tale* is a research object that contains metadata, data (by copy or reference), code, narrative, documentation, provenance, and information about the computational environment to support computational reproducibility.

³⁴See https://github.com/whole-tale/repo2docker_wholetale

³⁵Composability refers to the ability to combine multiple package managers – such as R, 'pip', and 'conda'

ductory courses where the first step of installation and configuration can be hurdles for students (Çetinkaya Rundel and Rundel, 2018), and (b) courses that require access to a relatively complex setup of software tools, e.g., database systems. Çetinkaya Rundel and Rundel (2018) describe how a Docker-based deployment of RStudio (i) avoided problems with troubleshooting individual students computers and greatly increase engagement through very quickly showing tangible outcomes, e.g., a visualisation, and (ii) reduced demand on teaching and IT staff. Each student get's acces to a personal RStudio instance running in a container after authentication with the university login, which gives the benefits of sandboxing and possibility of limiting resources. Çetinkaya Rundel and Rundel (2018) found that for the courses at hand, actual usage of the UI is intermittent so a single cloud based VM with 4 cores and 28GB RAM sufficed for over 100 containers. An example for mitigating *complex setups* is teaching databases. R is very useful tool for interfacing with databases, because 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 broadened by **DBI**, which allows to create a common API for interfacing these databases, or the **dbplyr** package, which runs **dplyr** code straight against the database as queries. But learning and teaching these tools comes with the cost of deploying or having access to an environment with the software and drivers installed. For people teaching R, it can become a barrier if they need to install local versions of database 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 to a database from R. Notably, with **r-db**, the users don't have to install complex drivers or to configure their machine in a specific way. The rocker/tidyverse base image ensures that users can also readily use packages for analysis, display, and reporting. Second, the idea of a common environment and partitioning allow using **containers in teaching for secure execution and automated testing** of submissions by students. **Dodona** is a web platform developed at Ghent University and is used to teach students basic programming skills, and it uses Docker containers to test submissions by students. This means that both the code testing the students' submissions and the submission itself are executed in a predictable environment, avoiding compatibility issues between the wide variety of configurations used by students. The containerization is also used to shield the Dodona servers from bad or even malicious code: memory, time and I/O limits are used to make sure students can't overload the system. The web application managing the containers communicates with them by sending configuration information as a JSON document over standard input. Every Dodona Docker image shares a `main.sh` file that passes through this information to the actual testing framework, while setting up some error handling. The testing process in the Docker containers sends back the test results by writing a JSON document to its standard output channel. In June 2019, R support was added to Dodona using an image derived from the rocker/r-base image that sets up the runner user and and `main.sh` file expected by Dodona³⁶. It also installs the packages required for the testing framework and the exercises so that this doesn't have to happen every time a student's submission is evaluated. The actual testing of R exercises is done using a custom framework loosely based on **testthat**. During the development of the testing framework it was found that the **testthat** framework did not provide enough information to its reporter system to send back all the fields required by Dodona to render its feedback. Right now, multiple statistics courses are developing exercises to automate the feedback for their lab classes.

RCloud (<https://rcloud.social>) is a cloud-based platform for data analysis, visualisation and collaboration using R. It provides a rocker/drd base image for easy evaluation of the platform³⁷.

Packaging research reproducibly

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 difficult for others to reuse existing pipelines. These breakages can make the original the results irreproducible and not , 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

³⁶<https://github.com/dodona-edu/docker-images/blob/master/dodona-r.dockerfile>

³⁷<https://github.com/att/rcloud/tree/master/docker>

has been taken up and extended³⁸, 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 to minimize problems with breaking changes and make workflows easier to share (cf. [Boettiger, 2015](#); [Marwick et al., 2018](#)). Research workflows in containers are safe from contamination from other activities occurring 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 typical 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 are tied to a specific date, rather than the most recent version. A more extreme case is the *dynverse* project (<https://dynverse.org/>), which packages over 50 computational methods with different environments (R, Python, C++, etc.) in Docker images, which can be executed from R. *dynverse* uses a CI platform (see [Development, debugging, and testing](#)) to build Rocker-derived images, test them, and publish them on Docker Hub if the tests succeed.

Future researchers can download the compendium from the repository and run the included Dockerfile to 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 the starting point for rebuilding the environment. When combined with CI (see [Development, debugging, and testing](#)), a research compendium set-up can enable *continuous analysis* with easier verification of reproducibility and audits trails ([Beaulieu-Jones and Greene, 2017](#)).

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](#)). A recommendation of the recent National Academies' report on *Reproducibility and Replicability in Science* is that journals "consider ways to ensure computational reproducibility for publications that make claims based on computations" ([Committee on Reproducibility and Replicability in Science, 2019](#)). In fields such as Political Science and Economics, journals are increasingly adopting policies requiring authors to publish the code and data required to reproduce computational findings reported in published manuscripts, subject to independent verification ([Jacoby et al., 2017](#); [Vilhuber, 2019](#); [Alvarez et al., 2018](#); [Christian et al., 2018](#); [Eubank, 2016](#); [King, 1995](#)). Problems with the computational environment, installation and availability of software dependencies are common. R is gaining popularity in these communities, such as creating a research compendium. In a sample of 105 replication packages published by the *American Journal of Political Science* (AJPS) over 65% use R. The NSF-funded Whole Tale project uses the Rocker Project community images with the goal of improving the reproducibility of published research artifacts and simplifying the publication and verification process for both authors and reviewers by reducing errors and time spent specifying the environment.

Conclusions

This article is a snapshot of the R-corner in a universe of applications built with a 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). Reproducible environments, scalability and efficiency, and portability across platforms/infrastructures are the common themes behind R packages, use cases, and applications in this work.

³⁸See full literature list at <https://research-compendium.science/>.

The R packages and use cases presented show the growing number of users, developers, and real-world applications in the community and the resulting innovations. But the applications also point to 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 with Docker, such as an independent Docker image stack, to serve one's specific needs leads to parallel developments. This ease-of-DIY in combination with the difficulty to compose environments based on Dockerfiles alone is a further reason for **fragmentation**. Instructions can be outsourced into distributable scripts and then copied into the image during build, but that make Dockerfile hard to read and adds a layer of complexity. Despite the different image stacks presented here, the pervasiveness of Rocker can be traced back to its maintainers and the user community valuing collaboration and shared starting points over impulses to create individual solutions. Aside from that, 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.

Nonetheless, 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 R objects representing a Dockerfile and its instructions. In this consolidation, the Rocker Project could play the role of coordinating entity. 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 since currently containerisation happens more in the background for UI-based development (e.g., a **rocker/rstudio** image in the cloud). Furthermore, a maturing of the Rockerverse packages for managing containers may lead to their adoption where currently manual coding is required, e.g. in the case of **RSelenium** or **drake** (see Sections [Development, debugging, and testing](#) and [Processing](#) respectively). In specific cases, e.g., for **analogsea**, the interaction with the Docker daemon may remain to specific to reuse first order packages to control Docker.

New features, which make complex workflows accessible and reproducible, and the variety in packages connected with containerisation, 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 files. Such efforts can 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

The ordering of authors following DN and DE is alphabetical. DN conceived the article idea, [initialised the formation of the writing team](#), wrote sections not mentioned below, and revised all sections. DB wrote the section on **outsider**. GD contributed the CARD.com use case. DE wrote the introduction and the section about Containerization and Rocker and reviewed all sections. RC & EH contributed to the section on interfaces for Docker in R (*dynverse* and *dynwrap*). DC contributed content on Gigantum. ME contributed to the section on processing and deployment to cloud services. CF wrote paragraphs about **r-online**, **dockerfiler**, **r-ci** and **r-db**. SL contributed content on RStudio's usage of Docker. BM

wrote the section on research compendia and made the project Binder-ready. HN & JN co-wrote the section on the T-Mobile use case. KR wrote the section about **holepunch**. NR wrote paragraphs about shared work environments and GPUs. LS & NT wrote the section on Bioconductor. CW wrote the sections on Whole Tale and contributed the publication reproducibility audit use case. NX contributed content on **liftr**. 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

DN is supported by the project Opening Reproducible Research (**o2r**) funded by the German Research Foundation (DFG) under project number [PE 1632/17-1](#). The funders had no role in data collection and analysis, decision to publish, or preparation of the manuscript. CW is supported by the Whole Tale projects (<https://wholetale.org>) funded by the US National Science Foundation (NSF) under award [OAC-1541450](#).

Bibliography


- 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]
- R. M. Alvarez, E. M. Key, and L. Núñez. Research replication: Practical considerations. *PS: Political Science & Politics*, 51(2):422–426, Apr 2018. ISSN 1049-0965, 1537-5935. doi: 10.1017/S1049096517002566. URL <https://doi.org/10.1017/S1049096517002566>. [p13]
- L. A. Barba. Terminologies for Reproducible Research. *arXiv:1802.03311 [cs]*, Feb. 2018. URL <http://arxiv.org/abs/1802.03311>. arXiv: 1802.03311. [p1]
- B. K. Beaulieu-Jones and C. S. Greene. Reproducibility of computational workflows is automated using continuous analysis. *Nature Biotechnology*, advance online publication, Mar. 2017. ISSN 1087-0156. doi: 10.1038/nbt.3780. URL <https://doi.org/10.1038/nbt.3780>. [p13]
- H. Bengtsson. *future: Unified Parallel and Distributed Processing in R for Everyone*, 2020. URL <https://CRAN.R-project.org/package=future>. R package version 1.16.0. [p8]
- D. Bennett, H. Hettling, D. Silvestro, R. Vos, and A. Antonelli. outsider: Install and run programs, outside of r, inside of r (under review). *Journal of Open Source Software*, 5(45):2038, 2020. doi: 10.21105/joss.02038. URL <https://doi.org/10.21105/joss.02038>. [p8]
- 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, 13]
- 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, 6, 13]
- C. Boettiger, R. Lovelace, M. Howe, and J. Lamb. rocker-org/geospatial, Dec. 2019. [p2]
- A. Brinckman, K. Chard, N. Gaffney, M. Hategan, M. B. Jones, K. Kowalik, S. Kulasekaran, B. Ludäscher, B. D. Mecum, J. Nabrzyski, et al. Computing environments for reproducibility: Capturing the “Whole Tale”. *Future Generation Computer Systems*, 94:854–867, 2019. doi: 10.1016/j.future.2017.12.029. URL <https://doi.org/10.1016/j.future.2017.12.029>. [p11]
- R. Cannoodt and W. Saelens. *babelwhale: Talking to ‘Docker’ and ‘Singularity’ Containers*, 2019. URL <https://CRAN.R-project.org/package=babelwhale>. R package version 1.0.1. [p3]
- L. Cardozo. Faster Docker builds in Travis CI for R packages, 2018. URL <https://lecardozo.github.io/2018/03/01/automated-docker-build.html>. [p7]
- S. Chamberlain, H. Wickham, and W. Chang. *analogsea: Interface to ‘Digital Ocean’*, 2019. URL <https://CRAN.R-project.org/package=analogsea>. R package version 0.7.2. [p9]

- 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]
- K. Chard, N. Gaffney, M. B. Jones, K. Kowalik, B. Ludäscher, T. McPhillips, J. Nabrzyski, V. Stodden, I. Taylor, T. Thelen, M. J. Turk, and C. Willis. Application of BagIt-Serialized Research Object Bundles for Packaging and Re-execution of Computational Analyses. 2019a. doi: 10.5281/zenodo.3381754. URL <https://doi.org/10.5281/zenodo.3381754>. To appear in 2019 IEEE 15th International Conference on e-Science (e-Science). [p11]
- K. Chard, N. Gaffney, M. B. Jones, K. Kowalik, B. Ludäscher, J. Nabrzyski, V. Stodden, I. Taylor, M. J. Turk, and C. Willis. Implementing computational reproducibility in the whole tale environment. In *Proceedings of the 2nd International Workshop on Practical Reproducible Evaluation of Computer Systems*, P-RECS '19, pages 17–22, 2019b. doi: 10.1145/3322790.3330594. URL <https://doi.org/10.1145/3322790.3330594>. [p11]
- T.-M. Christian, W. G. Jacoby, S. Lafferty-Hess, and T. Carsey. Operationalizing the replication standard. *International Journal of Digital Curation*, 13(1), 2018. doi: 10.2218/ijdc.v13i1.555. URL <https://doi.org/10.2218/ijdc.v13i1.555>. [p13]
- Committee on Reproducibility and Replicability in Science. *Reproducibility and Replicability in Science*. National Academies Press, 2019. ISBN 978-0-309-48616-3. doi: 10.17226/25303. URL <https://doi.org/10.17226/25303>. [p13]
- 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>. [p4]
- 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.org/10.1093/biostatistics/kxq028>. [p12]
- 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/>. [p7]
- 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. [p7]
- D. Eddelbuettel. Debugging with Docker and Rocker – A Concrete Example helping on macOS, Aug. 2019. URL <http://dirk.eddelbuettel.com/blog/2019/08/05/>. [p6]
- M. Edmondson. R on Kubernetes - serverless Shiny, R APIs and scheduled scripts, May 2018. URL <https://code.markedmondson.me/r-on-kubernetes-serverless-shiny-r-apis-and-scheduled-scripts/>. [p9]
- M. Edmondson. *googleComputeEngineR: R Interface with Google Compute Engine*, 2019. URL <https://CRAN.R-project.org/package=googleComputeEngineR>. R package version 0.3.0. [p9]
- 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>. [p13]
- N. Eubank. Lessons from a decade of replications at the Quarterly Journal of Political Science. *PS: Political Science & Politics*, 49(2):273–276, Apr 2016. ISSN 1049-0965, 1537-5935. doi: 10.1017/S1049096516000196. URL <https://doi.org/10.1017/S1049096516000196>. [p13]
- 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]
- GDAL/OGR contributors. *GDAL/OGR Geospatial Data Abstraction software Library*. Open Source Geospatial Foundation, 2019. URL <https://gdal.org>. [p6]
- 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>. [p12]

- 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. [p4]
- J.-P. S. Glouzon, J.-P. Perreault, and S. Wang. Structurexplor: a platform for the exploration of structural features of RNA secondary structures. *Bioinformatics*, 33(19):3117–3120, Oct. 2017. ISSN 1367-4803. doi: 10.1093/bioinformatics/btx323. URL <https://doi.org/10.1093/bioinformatics/btx323>. [p8]
- J. Harrison. *RSelenium: R Bindings for 'Selenium WebDriver'*, 2019. URL <https://CRAN.R-project.org/package=RSelenium>. R package version 1.7.5. [p7]
- 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>. [p11]
- 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]
- W. G. Jacoby, S. Lafferty-Hess, and T.-M. Christian. Should journals be responsible for reproducibility? *Inside Higher Ed*, Jul 2017. URL <https://www.insidehighered.com/blogs/rethinking-research/should-journals-be-responsible-reproducibility>. [p13]
- P. Jupyter. Jupyter Docker Stacks — docker-stacks latest documentation, 2018. URL <https://jupyter-docker-stacks.readthedocs.io/en/latest/>. [p4]
- 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. [p10]
- G. King. Replication, replication. *PS: Political Science & Politics*, 28(3):444–452, Sep 1995. doi: 10.2307/420301. URL <https://doi.org/10.2307/420301>. [p13]
- L. Kjeldgaard. 'dockr': easy containerization for R - pRopaganda by smaakagen, Dec. 2019. URL <http://smaakage85.netlify.com/2019/12/21/dockr-easy-containerization-for-r/>. [p6]
- 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]
- W. M. Landau. The drake r package: a pipeline toolkit for reproducibility and high-performance computing. *Journal of Open Source Software*, 3(21), 2018. URL <https://doi.org/10.21105/joss.00550>. [p8]
- 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. [p7]
- 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>. [p12]
- B. Marwick. Research compendium for the 1989 excavations at Madjedbebe rockshelter, NT, Australia, July 2017. URL <https://doi.org/10.6084/m9.figshare.1297059.v4>. [p6]
- 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>. [p11, 13]
- T. McPhillips, C. Willis, M. Gryk, S. Nunez-Corrales, and B. Ludäscher. Reproducibility by Other Means: Transparent Research Objects. 2019. doi: 10.5281/zenodo.3381754. URL <https://doi.org/10.5281/zenodo.3381754>. To appear in 2019 IEEE 15th International Conference on e-Science (e-Science). [p11]
- B. Mecum, M. B. Jones, D. Vieglais, and C. Willis. Preserving reproducibility: Provenance and executable containers in dataone data packages. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 45–49. IEEE, 2018. doi: 10.1109/eScience.2018.00019. URL <https://doi.org/10.1109/eScience.2018.00019>. [p11]


- 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]
- J. Nolis and J. Werdell. Small data, big value, Dec. 2019. URL <https://medium.com/tmobile-tech/small-data-big-value-f783ceca4fdb>. [p10]
- D. Nüst and M. Hinz. containerit: Generating Dockerfiles for reproducible research with R. *Journal of Open Source Software*, 4(40):1603, Aug. 2019. ISSN 2475-9066. doi: 10.21105/joss.01603. URL <https://joss.theoj.org/papers/10.21105/joss.01603>. [p5]
- D. Nüst, M. Konkol, E. Pebesma, C. Kray, M. Schutzzeichel, 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>. [p13]
- OCI. Open Containers Initiative - About, 2019. URL <https://www.opencontainers.org/about>. [p2]
- H. Ooi. *AzureContainers: Interface to 'Container Instances', 'Docker Registry' and 'Kubernetes' in 'Azure'*, 2019. URL <https://CRAN.R-project.org/package=AzureContainers>. R package version 1.2.0. [p3]
- J. Ooms. OpenCPU - Why Use Docker with R? A DevOps Perspective, Oct. 2017. URL <https://www.opencpu.org/posts/opencpu-with-docker/>. [p6, 9]
- R Core Team. *Writing R extensions*, 1999. URL <https://cran.r-project.org/doc/manuals/r-devel/R-exts.html>. [p6, 12]
- R-hub project. R-hub Docs, 2019. URL <https://docs.r-hub.io/>. [p7]
- L. Savini, L. Candeloro, S. Perticara, and A. Conte. EpiExploreR: A Shiny Web Application for the Analysis of Animal Disease Data. *Microorganisms*, 7(12):680, Dec. 2019. doi: 10.3390/microorganisms7120680. URL <https://doi.org/10.3390/microorganisms7120680>. [p8]
- 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. [p11]
- 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]
- K. Ushey. *renv: Project Environments*, 2019a. URL <https://CRAN.R-project.org/package=renv>. R package version 0.9.2. [p6]
- K. Ushey. Using renv with Docker, 2019b. URL <https://rstudio.github.io/renv/articles/docker.html>. [p6]
- L. Vilhuber. Report by the AEA Data Editor. *AEA Papers and Proceedings*, 109:718–729, May 2019. ISSN 2574-0768. doi: 10.1257/pandp.109.718. [p13]
- H. Wickham, M. Averick, J. Bryan, W. Chang, L. McGowan, R. François, G. Grolemond, 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>. [p10]
- Wikipedia contributors. OS-level virtualization, Jan. 2020. URL https://en.wikipedia.org/w/index.php?title=OS-level_virtualization&oldid=935110975. Page Version ID: 935110975. [p1]
- 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. [p6]
- Y. Xie, J. J. Allaire, and G. Grolemond. *R Markdown: The Definitive Guide*. Chapman and Hall/CRC, 2018. [p6]
- H. Ye. Docker Setup for R package Development, 2019. URL <https://haoye.us/post/2019-10-10-docker-for-r-package-development/>. [p7]

M. Çetinkaya Rundel and C. Rundel. Infrastructure and Tools for Teaching Computing Throughout the Statistical Curriculum. *The American Statistician*, 72(1):58–65, Jan. 2018. ISSN 0003-1305. doi: 10.1080/00031305.2017.1397549. URL <https://doi.org/10.1080/00031305.2017.1397549>. [p12]

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

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

Dom Bennett
Gothenburg Global Biodiversity Centre, Sweden
Carl Skottsbergs gata 22B
413 19 Göteborg, Sweden
 0000-0003-2722-1359
dominic.john.bennett@gmail.com

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

Dav Clark
Gigantum, Inc.
1140 3rd Street NE
Washington, D.C. 20002, USA
 0000-0002-3982-4416
dav@gigantum.com

Gergely Daroczi
 0000-0003-3149-8537
daroczig@rapporter.net


Mark Edmondson
IIH Nordic A/S, Google Developer Expert for GCP
mark@markedmondson.me

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

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
250 Northern Ave
Boston, MA 02210, USA
sean@rstudio.com*

*Ben Marwick
University of Washington
Department of Anthropology
Denny Hall 230, Spokane Ln
Seattle, WA 98105, USA
 0000-0001-7879-4531
bmarwick@uw.edu*

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

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

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


*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*

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

*Nitesh Turaga
Roswell Park Comprehensive Cancer Center
Elm & Carlton Streets*

Buffalo, NY, 14263, USA

 0000-0002-0224-9817

nitesh.turaga@roswellpark.org

Craig Willis

University of Illinois at Urbana-Champaign

501 E. Daniel St.

Champaign, IL 61820, USA

 0000-0002-6148-7196

willis8@illinois.edu

Nan Xiao

Seven Bridges Genomics

529 Main St, Suite 6610

Charlestown, MA 02129, USA

 0000-0002-0250-5673

me@nanx.me

Charlotte Van Petegem

Ghent University

Department WE02

Krijgslaan 281, S9

9000 Gent, Belgium

 0000-0003-0779-4897

charlotte.vanpetegem@ugent.be