Containers and R—the Rockerverse and beyond

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

Abstract The Rocker project provides widely-used Docker images for R across different application scenarios. This articles surveys describe downstream projects building upon Rocker. We also look beyond Rocker to other projects connecting containerisation with R. These use cases and the diversity of applications demonstrate the power of Rocker and containerisation for collaboration, effectivity, scalability, and transparency.

Introduction

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

Containerization and Rocker

Docker, an application and service provide by the eponymous company, has in just a few short years risen to prominence for development, testing, deployment and distribution of computer software (cf. Datadog, 2018; Muñoz, 2019). While there are related approaches such as LXC[4] or Singularity (Kurtzer et al., 2017), Docker has become synomymous with "containerization"—the method of taking software artefacts and bundling them in such a way that use becomes standardized and portable across operationg systems. In doing so, Docker had recognised and validated the importance of one very important thread that had been emerging, namely virtualization. By allowing (one or possibly) multiple applications or services to run concurrently on one host machine without any fear of interference between them, an important scalability opportunity is being provided. But Docker improved on virtualization by accessing the host system—generally Linux—through a much thinner and smaller shim than a full operating system emulation or virtualization. This makes for more efficient use of system resources (**?**) and allowed another order of magnitude in terms of scalability of deployment (cf. Datadog, 2018). While Docker makes use of Linux kernel features, its importance was large enough so that some required aspects of running Docker have been added to other operating systems to support Docker there more efficiently too (Microsoft, 2019b). The success even lead to standardisation and industry collaboration (OCI, 2019).

The key accomplishment of Docker (as an "application") is to make a "bundled" aggregation of software (the so-called "image") available to any system equipped to run Docker, without requiring much else from the host besides the actual Docker application installation. This is a rather attractive, and novel, proposition which has lead to widespread adoption and use of Docker in a variety of domains, *e.g.,* cloud computing infrastructure (Bernstein, 2014), data science (Boettiger, 2015), and edge computing (Alam et al., 2018) [more examples!]. It provided to be a natural match for "cloud

---

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

deployment" which runs (or at least appears to run) "seamlessly" without much explicit reference to the underlying machine, architecture or operating system: containers can be deployed with very little in terms of dependencies on the host system—only the container runtime is required.

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 `geospatial`. Also of note is a series of "versioned" containers which match the R release they contain with the *then-current* set of packages via the MRAN Snapshot views of CRAN (Microsoft, 2019a). The Rocker Project's impact and importance was acknowledged by the Chan Zuckerberg Initiative's *Essential Open Source Software for Science*, who provide findung for the projects's sustainable maintenance, community growth, and targeting new hardware platforms including GPUs (Chan Zuckerberg Initiative et al., 2019).

Container Images

Images for alternative R distributions

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

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

Renjin is an interpreter for the R language running on top of the Java Virtual Machine (JVM) providing full two-way access between Java and R code (Wikipedia, 2018). It was developed to combine the benefits of R, such scripting and extension packages, with the JVM's advantages in the areas of security, cross-platform availability, and established position in enterprise settings. R extension packages need to be specially compiled and are distributed via the Java package manager Apache Maven, cf. `http://packages.renjin.org/packages` for available packages. Packages are loaded on demand, i.e. at the first call to `library()`. Not all R packages, especially one linking to binary libraries, are available, e.g. `rgdal`[5]. There are no offical Docker images for Renjin, but community-maintained images for selected releases only are available under `nuest/renjin` on Docker Hub and GitHub at `https://hub.docker.com/r/nuest/renjin` and `https://github.com/nuest/renjin-docker` respectively. These images expose the command line interface of Renjin in a similar fashion as Rocker images and allow an easy evaluation of Renjin's suitability, but are not intended for production use.

pqR is a "a pretty quick version of R". *pqR* attempts to improve R on some opinionated issues in the R language and is the basis for experimental features, e.g. automatic diffentiation[6]. The source code development on GitHub is a one man project and it does not provide any Docker images. But especially disruptive approaches may contribute to the development of the R ecosystem, so the `nuest/pqr` images on Docker Hub and GitHub at `https://hub.docker.com/r/nuest/pqr/` and `https://github.com/nuest/pqr-docker` respectively.

FastR is "A high-performance implementation of the R programming language, built on GraalVM" (`https://github.com/oracle/fastr`). It is developed by Oracle, connects R to the GraalVM ecosystem (wik, 2019), and also claims superior performance but also targets full compatibility with base R[7]. There are no offical Docker images provided, but `nuest/fastr` images and `Dockerfile` are independently

---

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

maintained on Docker Hub and GitHub respectively.

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

Bioconductor

Bioconductor is an open source, open development project for the analysis and comprehension of genomic data (Gentleman et al., 2004). The project consists of 1741 R software packages as of August 15th 2019, as well as packages containing annotation or experiment data. _Bioconductor_has a semiannual release cycle, each release is associated with a particular version of R. Docker images allow availability of current and past versions of *Bioconductor* for convenience and reproducibility. *Bioconductor* 'base' docker images are built on top of `rocker/r-ver` and `rocker/rstudio`. *Bioconductor* installs packages based on the R version, and therefore uses `rocker/rstudio` and `rocker/r-ver` version tagging. *Bioconductor* selects the desired version of R from Rocker, adds the BiocManager CRAN package for installing appropriate versions of *Bioconductor* packages, and creates a *Bioconductor* docker image with an informative tag (R_version_Bioc_version). The images are summarized on the *Bioconductor* web site (`https://bioconductor.org/help/docker/`), maintained on GitHub (`https://github.com/Bioconductor/bioc_docker`), and available to the community through DockerHub. Past and current combinations of R and *Bioconductor* are therefore accessible via a specific docker tag.

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

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

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

Images for (historic) R versions

As with any other software, each new version of R comes with its share of changes. Some are breaking changes, some are not, but the fact is that running a piece of code in a given version might give different results from another version can be an issue when it comes to reproducibility of workflows

and stability of applications. For example, think about R random seed: starting with `R 3.6.0`, running `set.seed(2811); sample(1:1000, 2)` will not give the same result as if it was run inside an older version of R. That might seem trivial, but all the code using the random number generator will not be exactly reproducible after this breaking change. Therefore, controlling the version of software is most crucial for reproducible research (e.g. Boettiger, 2015).

Containers are perfectly suited to capture a specific configuration of a computing environment to prohibit such problems. The Rocker images provides the *versioned stack* for different the version of R since the project inception, which use the *then-current* stable Debian base image (cf. Boettiger and Eddelbuettel, 2017). Based on custom build phase hooks[8], i.e. small shell scripts executed at different phases in the automated build of Rocker images, there are also semantic version tags for the most recent freezed (i.e. using MRAN) versions with only the major and minor version[9]. So, at the time of writing this article, `rocker/r-ver:3` and `3.6` are aliases for `3.6.0`, because `3.6.1` is the latest release. With the release of `3.6.2` and pinning of the MRAN version in `3.6.1`, the tags 3 and 3.6 will deliver the same image as `3.6.1`. These tags allow users to update the base image to retrieve bugfixes while reducing the risk of also introducing breaking changes.

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

A container also provides an isolated sandbox environment suitable for testing and evaluation, without interfering with the "main" working environment. This enables cross version testing and bugfixing. To take this even further, the Rocker contributors are discussing the provision of R versions reaching back futher than the project's own inception, reaching back as far as R `2.x` and `1.x` [10]. The main challenges are finding a suitable base image with a matching OS version, or making the adjustments to compile R on more recent OS releases.

Windows Images (?)

- rocker-win
- is possible, only relevant in organisations with an existing Windows Server-based infrastructure, can meet policies then

Non-Debian Linux images [NN]

- Alpine images
- Images used by R-Hub (overlap with CI?)
- `https://github.com/jlisic/R-docker-centos`

R Packages

Interfaces for Docker in R

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

Each of these packages has particular advantages as they provide function wrappers for interacting with the Docker CLI at different stages of a container's life cycle. Examples of such interactions are installing the Docker software, creating Dockerfiles, building an image, and launching a container 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.

---

[8]https://docs.docker.com/docker-hub/builds/advanced/
[9]See   https://github.com/rocker-org/rocker-versioned/blob/master/VERSIONS.md,   FIXME:
https://github.com/rocker-org/rocker-versioned/issues/42.
[10]https://github.com/rocker-org/rocker-versioned/issues/138

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			✓			

stevedore (https://cran.r-project.org/package=stevedore)

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

dockermachine (https://github.com/cboettig/dockermachine)

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

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

harbor (https://github.com/wch/harbor)

Capture and create environments (?)

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

- dockerfiler
- containerit
- dockertest

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

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

fields and generates a `Dockerfile` for creating Docker containers. *liftr* then helps render the document inside the created Docker container. This workflow is summarized in Figure 1.
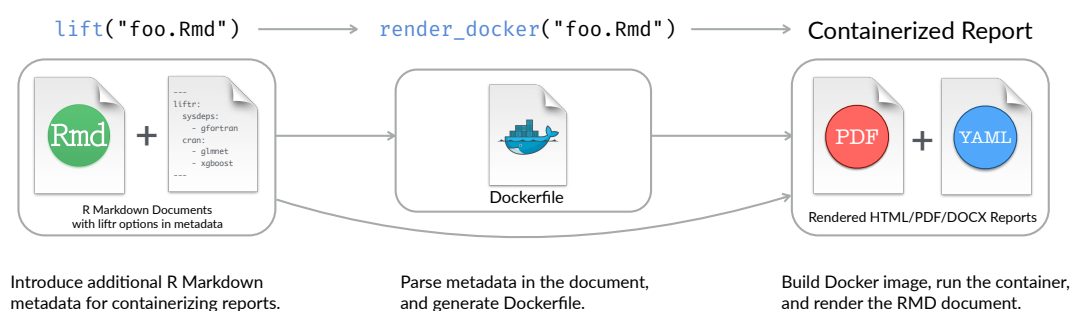

Figure 1: The liftr workflow for rendering containerized R Markdown documents.

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

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

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

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

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

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

- rize for Shiny

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

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

Use cases and applications

Deployment to cloud services [div]

The cloud is the natural environment of containers, and becomes the go-to mechanism to expose applications written in R to users.

- RSelenium
- googleComputeEngineR (function docker_run)
    - I don't think docker_run is main focus for containers for this package, its more how it uses Docker to launch custom environments that can be templated (e.g. rstudio, shiny etc.) and for parrallisation e.g. library(future) using GCE VMs as endpoints. (?)

- analogsea (digital ocean R client)
- AzureContainers: Umbrella package for working with containers in Microsoft's Azure cloud. Provides interfaces to three Azure services: Container Registry, Container Instances (for running individual containers), and Kubernetes Services (for orchestrated deployments)

Using R to power enterprise software in production environments (?, ?)

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

The backbone of this work is the *plumber* package, created by Jeff Allen. The plumber package allows R to be used to host web services through RESTful APIs. When R is running on a server, a different server can ask R to make a computation through an HTTP request and get an immediate response. For instance, if a data scientist at a company creates a machine learning model to tell which customers are likely to make a repeat purchase, if they serve the model through a plumber API then someone else on the marketing team could write software that sends different emails depending on that real-time prediction. Plumber syntax is simple to read and understand: the API is created by commenting above function names in an R script.

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

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

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

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

Deployment to the cloud (?)

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

These issues can be mitigated by deploying not a single container, but a *cluster* of containers, orchestrated as a single deployment. The deployment system of choice at the time of writing is Kubernetes. This allows you to deploy a single image to a pool of servers, which can then serve requests in parallel. Client-side requests are made to the cluster ingress endpoint, which then redirects them to individual servers. The orchestrator can identify cluster nodes that have failed, and restart them automatically; it can also detect when there is no capacity to service more requests, and start additional servers and containers as needed.

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

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

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

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

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

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

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


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

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


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

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

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

```
deployclus$get("deployment bos-rf")
#> Kubernetes operation: get deployment bos-rf  --kubeconfig=".../kubeconfigxxxx"
#> NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
#> bos-rf    1         1         1            1           5m

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

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

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

Continuous integration and continuous delivery (?, ?)

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

- DevOps

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

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

- dynwrap (?)

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

- Google Cloud Build (?)

    - https://cloud.google.com/cloud-build/
    - For me this is what makes Docker containers viable, as it builds the Dockerfiles on each GitHub commit. It couples with Google Container Registry to build private Docker images, for downstream applications.

Common or public work environments

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

holepunch is an R package that was designed to make sharing work environments accessible to novice R users based on *Binder*. The Binder project, maintained by the team behind Jupyter, makes it possible for users to create and share their computing environments with others (Jupyter et al., 2018). A *BinderHub* allows anyone with access to a web browser and an internet connection to launch a temporary instance of these custom environments and execute any workflows contained within. From a reproducibility standpoint, Binder makes it exceedingly easy to compile a paper, visualize data, and run small examples from papers or tutorials without the need for any local installation. To set up Binder for a project, a user typically starts at an instance of a BinderHub and passes the location of a repository with a workspace, e.g., a hosted Git repository, or a data repository like Zenodo. Binder's core internal tool is repo2docker. It deterministically builds a Docker image by parsing the contents of a repository, e.g. project dependency configurations or simple configuration files. In the most powerful case, repo2docker builds a given Dockerfile. While this approach works well for most run of the mill Python projects, it is not so seamless for R projects. For any R projects that use the Tidyverse suite [citation in review, will add by the time this gets submitted], the time and resources required to

build all dependencies from source can often time out before completion, making it frustrating for the average R user. holepunch removes some of these limitations by leveraging Rocker images that contain the Tidyverse along special Jupyter dependencies, and only installs additional packages from CRAN and Bioconductor that are not already part of these images. It short cicuits the configuration file parsing in repo2docker and starts with the Binder/Tidyverse base images, which eliminates a large part of the build time and in most cases results in a binder instance launching within a minute. holepunch as a side effect also creates a DESCRIPTION file which then turns any project into a research compendium (Marwick et al., 2018). The Dockerfile included with the project can also be used to launch a RStudio server locally independent of Binder which is especially useful when more or special computational resources are required.

holepunch is a R package that was designed to remove some of these limitations and make Binder more accessible to novice R users. holepunch achieves this by leveraging Rocker images that contains the Tidyverse along special Jupyter dependencies, and only installs additional packages from CRAN and Bioconductor that are not already part of these images. Starting with the Binder/Tidyverse base images 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 independent of binder which is especially useful when more computational resources are required.

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

Using **GPUs** (graphical processing units) as a specialised hardware from containerized common work environments is also possible and useful (**?**). GPUs are increasingly popular for compute-intensive machine learning tasks, e.g. deep artificial neural networks (**?**). Though in this case, containers are not completely portable between hardware environments. 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 (beta) images based on these that include GPU-enabled versions of machine-learning R packages: https://github.com/rocker-org/ml.

Teaching is a further example where sharing a prepared computing environment can greatly improve the process, especially for courses that require access to a relatively complex setup of software tools, e.g., as in the case of database systems. R is a fantastic tool when it comes to interfacing with databases: almost every open source and proprietary database system has an R package that allows users to connect and interact with the it This flexibility is even broader now that we have tools like DBI, that allows to create a common API for interfacing these databases, or like dbplyr, which are designed to run dplyr code straight against the database. But learning and teaching these tools comes with a cost: the cost of deploying or having access to an environment with the softwares and drivers installed. For people teaching R, it can become a barrier if they need to install local versions of the drivers, or to connect to remote instances which might or might not be made available by IT services. Giving access to a sandbox for the most common database environments is the idea behind r-db (**?**), a Docker image that contains everything needed to connect, from R, to a database. Notably, with r-db, the users don't have to install complex drivers or to configure their machine in a specific way. On top of that, r-db comes with a comprehensive online guide, explaining how to spin a Docker instance of a specific database, and how to use r-db to connect to it. Each package contained inside this image also has a series examples, so that the user can get started with the database right away. The rocker/tidyverse

base image ensures that users can also readily use packages for analysis, display, and reporting.

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

Processing [div]

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

- Docker images for cloud services (**?**)
    - the most popular functionality for `googleComputeEngineR` is its use of Docker to enable parallel processing across VMS - some demos here
- Google Cloud Run - CaaS (Containers as a Service) that lets you launch a Docker container without worrying about underlying infrastructure. An R implementation is shown here at cloudRunR which uses it to create a scalable R plumber API.
- `batchtools` (Lang et al., 2017) can schedule jobs with Docker Swarm
- scalable deployments, e.g. start with numerous Shiny talks mentioning Rocker at useR!2017
- dynmethods (**?**): In order to evaluate ±50 computational methods which all used different environments (R, Python, C++, . . .), we wrapped each of them in a docker container and can execute these methods from R. Again, all of these containers are being built on travis-ci, and will only be pushed to docker hub if the integration test succeeds.

Research Compendia (?)

. . .

Development and debugging

Containers can also serve as useful playgrounds to create environments ad-hoc or to provide very specific environments that are not often needed. Developers can readily start a local container based on a Rocker image to investigate a bug report, which might require them to install a specific combination of package versions or use a specific R version (see also versioned images above). Unlike recreating the reporter's environment on their regular machine, a container can simply be discarded after use, and there is no need to tediously roll back explorative changes to a previous state. Using the Rocker images with RStudio, the disposable environments lack no development comfort (see also the usage in research compendia).

r-debug is a purpose built Docker image for debugging R memory problems. Eddelbuettel (2019) describes how a Docker container was used to debug an issue with a package only occuring with a particular version of Fortran, and using tools which are not readily available on all platforms (e.g., not on macOS).

- **R-Hub**

Other containerisation platforms

Docker is not the only containerisation software. An alternative stemming from the domain of high-perfomance computing is Singularity (Kurtzer et al., 2017). Singularity can run Docker images, and in the case of Rocker works out of the box if the main process is R, e.g., in `rocker/r-base`, but does not succeed in running images where there is an init script, e.g. in containers that by default run RStudio. In the latter case, a `Singularity` file, a set of instructions akin to a `Dockerfile` needs to be used.

Conclusions

- next steps
    - consolidation?
- support more complex set-ups with ready-to-use stacks, e.g. GPU, AI, ML
- Missing pieces?
- consolidation (e.g. via packages using `dockerfiler` and `stevedore`)

- Common themes
    - reproducibility

- will knowledge about containers continue to spread?
- what is needed for even more containers with R?
- the ability to move processing between services easily (e.g locally, one cloud providers VM, another cloud provider's Container-as-a-Service)
- running RStudio in a container is a common approach for reproducibility - *is there a need to run/control containers from a UI integrated in RStudio?*
- moving away from Docker focus towards containers?
    - is there a need for a wrapper for podman's Varlink-API (similar to stevedore) and buildah to manage containers and images respectively with alternative software stacks?
    - will there be control packages for Singularity support?

- …

Author contributions

DN conceived of the presented idea and initialised the formation of the writing team, wrote the section about images for R distributions, wrote the conclusions section, and revised all sections. CB .. RC and EH wrote the section on interfaces for Docker in R. DE wrote the introduction and section about Rocker. ME .. CF wrote the paragraphs about `r-online`, `dockerfiler`, `r-ci` and `r-db`. BW .. KR .. NR .. NX wrote the section on liftr. LS & NT wrote the section on Bioconductor. All authors contributed minor edits, gave feedback on other sections, and 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.

Bibliography

GraalVM, Aug. 2019. URL https://en.wikipedia.org/w/index.php?title=GraalVM&oldid=912552499. Page Version ID: 912552499. [p2]

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. [p1]

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. [p1]

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. [p1, 4]

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

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]

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

D. Eddelbuettel. Debugging with Docker and Rocker – A Concrete Example helping on macOS, Aug. 2019. URL http://dirk.eddelbuettel.com/blog/2019/08/05/. [p11]

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

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. URL https://conference.scipy.org/proceedings/scipy2018/project_jupyter.html. [p9]

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, 11]

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. [p11]

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, 1537-2731. doi: 10.1080/00031305.2017.1375986. [p10]

Microsoft. About Microsoft R Open: The Enhanced R Distribution . MRAN, a. URL https://mran.revolutionanalytics.com/rro. [p2]

Microsoft. The Benefits of Multithreaded Performance with Microsoft R Open . MRAN, b. URL https://mran.revolutionanalytics.com/documents/rro/multithread. [p2]

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. [p1]

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]

OCI. Open Containers Initiative - About, 2019. URL https://www.opencontainers.org/about. [p1]

T-Mobile, J. Nolis, and H. Nolis. Enterprise Web Services with Neural Networks Using R and Tensor-Flow, Nov. 2018. URL https://opensource.t-mobile.com/blog/posts/r-tensorflow-api/. [p7]

Wikipedia. Renjin, Aug. 2018. URL https://en.wikipedia.org/w/index.php?title=Renjin&oldid=857160986. Page Version ID: 857160986. [p2]

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. [p5]

Y. Xie, J. J. Allaire, and G. Grolemund. *R Markdown: The Definitive Guide*. Chapman and Hall/CRC, 2018. [p5]

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

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

Dirk Eddelbuettel
University of Illinois at Urbana-Champaign
Department of Statistics
Illini Hall, 725 S Wright St

Champaign, IL 61820, USA
0000-0001-6419-907X
dirkd@eddelbuettel.com

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

mark@markedmondson.me

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

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

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

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

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

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

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

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

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

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

Ben Marwick