https://doi.org/10.1038/s43586-023-00236-9

Primer

Check for updates

# Containers for computational reproducibility

David Moreau[1], Kristina Wiebels[1] & Carl Boettiger[2]

## Abstract

The fast-paced development of computational tools has enabled tremendous scientific progress in recent years. However, this rapid surge of technological capability also comes at a cost, as it leads to an increase in the complexity of software environments and potential compatibility issues across systems. Advanced workflows in processing or analysis often require specific software versions and operating systems to run smoothly, and discrepancies across machines and researchers can impede reproducibility and efficient collaboration. As a result, scientific teams are increasingly relying on containers to implement robust, dependable research ecosystems. Originally popularized in software engineering, containers have become common in scientific projects, particularly in large collaborative efforts. In this Primer, we describe what containers are, how they work and the rationale for their use in scientific projects. We review state-of-the-art implementations in diverse contexts and fields, with examples in various scientific fields. Finally, we discuss the possibilities enabled by the widespread adoption of containerization, especially in the context of open and reproducible research, and propose recommendations to facilitate seamless implementation across platforms and domains, including within high-performance computing clusters such as those typically available at universities and research institutes.

## Sections

Introduction
Experimentation
Results
Applications
Reproducibility and data deposition
Limitations and optimizations
Outlook

[1]School of Psychology and Centre for Brain Research, University of Auckland, Auckland, New Zealand. [2]Department of Environmental Science, Policy and Management, University of California Berkeley, Berkeley, CA, USA. e-mail: d.moreau@auckland.ac.nz

## Introduction

In the past few decades, science has become increasingly collaborative, with modern scientific workflows typically involving multiple people, often spread across research teams and locations[1]. The distributed nature of modern scientific research has had a substantial impact on scientific discovery, enabling researchers to tackle complex problems that require a diverse range of expertise and resources, from genomic sequencing[2,3] to epidemiological modelling[4] and climate predictions[5]. This shift towards incorporating more data and techniques from various sources has led to science becoming more computational[6–9]. Scientists often build upon workflows of each other and share data and code publicly[10,11]. Given the tremendous amount of work and effort that often goes into collaborative projects, reusability is key to enable efficient, cumulative research and reproducibility has become an inherent part of modern scientific training[12–18].

In this context, computational reproducibility — the ability to obtain consistent and verifiable results from a computational experiment or analysis when the same input data, code and software environment are used — has become central to many research projects. Although the move towards more collaborative and open practices is undeniably beneficial to the scientific enterprise[19–22], the complexity afforded by shared and predominantly computational scientific workflows has also brought challenges[23]. With users distributed across machines, platforms and software versions, compatibility issues are bound to arise, with the potential to impede effective use and development — an issue colloquially referred to as dependency hell (Box 1). Collaborators attempting to reproduce or build upon existing work often face challenges that at best slow down scientific projects or in extreme cases can prevent reuse or collaboration altogether[24,25].

Containers provide an answer to these challenges[26]. Broadly speaking, containers encapsulate all information needed to run computer code in a fully configured environment. This includes specific software versions, as well as their dependencies and operating system configurations[27]. More specifically, containers solve five major problems associated with deploying and managing applications in scientific research. By allowing researchers to package their code, data and dependencies into a self-contained environment, containers solve issues of reproducibility. Containers also allow researchers to share their work with others more easily, enabling more efficient collaboration and faster progress, and run their code on different operating systems and hardware while circumventing compatibility issues. Containers can be seamlessly deployed to cloud environments[28], enabling seamless scalability. Finally, by enabling researchers to allocate resources more efficiently and avoid unnecessary consumption or conflicts with other projects, containers allow more efficient project management.

In this Primer, we provide the reader with a comprehensive overview of containerization in scientific research, including practical use and implementations, illustrated with examples. We focus on the Docker ecosystem[29] (docker.com), as it is the most common platform used to build and share containers[27], but also discuss alternatives such as Singularity[30,31] or Podman[32]. We consider challenges and limitations, in particular with respect to efficiency and compatibility with high-performance computing (HPC) environments, and provide guidance on implementation. We close with a discussion of the future of containerization and reproducibility in a rapidly evolving computational environment.

## Experimentation

This section introduces containers and provides the basics to run and personalize containerization from the perspective of a user. Examples are provided in the Docker ecosystem, for which a brief overview is provided.

### Introducing containers

A container is a self-contained and executable package that includes all of the necessary components for running a software application, such as system tools, libraries settings and the application itself, as well as any operating system components that are not provided by the host operating system. This means that containers are completely isolated from one another and the host operating system, and they can run anywhere, regardless of the environment. Applications can then be run consistently across different environments, including different operating systems and hardware configurations. Specifically, containers work by packaging an application and its dependencies into a single container image, which can then be run on any host that has a container runtime installed. The container runtime handles the execution of the application and manages the resources it requires, such as memory and central processing units (CPUs).

One of the main advantages of using containers is their portability. Because containers include all dependencies of an application, they can be moved between different environments. This allows developers to build and test applications on their own machines and then deploy them to other environments without worrying about compatibility issues. Containers take up less space and require fewer resources than traditional virtual machines (see Table 1 for a comparison of virtual machines and containers), making them well suited for use in cloud computing environments[33]. Containers also isolate applications from each other and the host system, which help prevent conflicts between applications. In addition, modern language-based package management tools, such as Python virtual environments, offer a solution to further reduce the chances of encountering dependency conflicts even within a container. Python virtual environments enable developers to create isolated environments with their own sets of dependencies, configuration and settings for each project. This means that even if multiple containers are running on the same host, each container can have its own Python virtual environment with its specific dependencies, avoiding conflicts and ensuring smooth operation.

The selection of virtual machines or containers depends on the specific needs and requirements of the application or process. Historically, virtual machines have been preferred when the research requires a highly isolated environment, for example, when the integrity of the research data and environment is critical, such as in medical or pharmacological research. Virtual machines offer fundamentally more isolation than containers, which can be an advantage in certain situations. However, developments in container technology such as namespaces, SELinux and AppArmor have improved container isolation and made them suitable for a wider range of research applications. Namespaces, for instance, have been a key enabling technology for containerization, and are now well established. In addition, technologies such as Singularity[30] and Shifter[34] have provided concrete solutions for accessing specific hardware resources, such as graphics processing units (GPUs) or HPC clusters. In general, the choice between virtual machines and containers will depend on the specific needs of the researcher and the degree of isolation required for the given application or process.

Containers are often preferable in most practical cases, especially in situations in which a researcher needs to run multiple experiments concurrently. In this case, using containers allows the researcher to run each experiment in its own container, which can be easily started, stopped and modified without affecting the other experiments. This is not possible with virtual machines, which require a separate operating system for each experiment. Virtual machines also tend to be much heavier and more resource-intensive than containers, which can be an important disadvantage in certain scenarios. Similarly, a key feature of containers over virtual machines is the ability to combine individual containers together, with each providing a different app. Containers are also advantageous when a researcher needs to scale their experiments or share their experiments with co-workers, as containers contain all the necessary dependencies and configuration settings in a lightweight package. Finally, containers cannot be matched when it comes to enabling reproducibility in a lightweight and portable manner: researchers can reproduce their experiments by creating an identical container with the same dependencies and configuration settings.

### The Docker platform

Containers are built on top of containerization platforms, which provide a standard format for packaging and distributing applications. These platforms include tools for creating, managing and deploying containers. One such platform is Docker, which was designed to make it easier to create, deploy and run applications by using containers.

We chose to focus on Docker for several reasons. First, Docker is relatively easy to use; it has a simple and intuitive interface that makes it straightforward to use and deploy containers. Second, with a vast and engaged community or users and developers, Docker has become the de facto standard for containerization, providing thousands of prebuilt container images that can be used as a starting point for building applications, as well as support for addressing challenges that might arise at any stage of container development. Third, Docker supports multiple operating systems including Windows, Linux and Mac, making it easier to deploy applications across different environments. Fourth, Docker allows scaling applications up or down on the basis of demand, making it an ideal choice for cloud-based deployment. Finally, with its built-in security features such as image signing and container scanning, Docker ensures the security of applications and prevents vulnerability, although its seamless integration with other tools enables the management and deployment of applications at scale.

By allowing developers to abstract away the complexities of the underlying infrastructure, Docker allows users to focus on writing code, enabling writing and testing of applications on their own machines and then deploying them to any environment running Docker. Although Linux runs within Docker containers, users can access the platform through a Windows or Mac computer. Docker consists of three components: the Docker software, Docker objects and online Docker registries, such as the Docker Hub (Fig. 1).

The Docker software itself consists of two parts: the Docker client and the Docker daemon. The Docker client is the primary way that users interact with Docker. It is a command-line interface that allows users to issue commands to the Docker daemon, such as building, running and distributing containers. The Docker daemon is the background service that manages the containers: it listens for commands from the Docker client and performs the necessary actions to create and run containers. The Docker daemon can run on the same host as the Docker client, or it can be remotely accessed through the network. In addition to the command-line interface, Docker also provides a user interface called Docker Desktop, which is available on both Mac and Windows operating systems. The Docker Desktop user interface provides a graphical way for users to manage containers, images and other Docker resources.

The main Docker objects are images and containers. A Docker image contains everything needed to run a piece of software, including the application code, libraries, dependencies and runtime. Docker images are built from Dockerfiles, which include details on which base image to use, commands to run and files and directories to copy. The main difference between Docker images and Docker containers is that images are static and cannot be changed or modified, whereas containers are dynamic and can be started, stopped and modified while they are running (Box 2). Images are used to create containers, but once a container is created, it can be modified and run independently of the image that was used to create it.

## Box 1

### Dependency hell in scientific research

Dependency hell describes a situation in which a software application or system becomes dependent on other software packages, libraries or frameworks, and the dependencies between these packages become complex and intertwined. This can make it difficult or impossible to update or maintain the system, as any changes to one package may have unintended consequences on others. Dependency hell can also occur when there are conflicts between different versions of the same dependency, or when one dependency requires another dependency that is incompatible with a different dependency in the system. This can lead to problems such as broken builds, runtime errors or instability in the system.

An example of dependency hell is described as follows: a researcher is trying to use a specific software (software *X*), but is unable to because they lack the appropriate version of a dependent library (library *Y*) required to install software *X*, and the correct version library *Y* requires software *Z*, which in turn requires the installation of library *W*, and so on, until all the dependencies are met for software *X*. Dependency hell can also arise when different software packages require different versions of the same library, creating conflicts that can be difficult to resolve. Altogether, this issue can be time-consuming and frustrating, and it can substantially delay progress. Dependencies can be either internal or external to each software, with the potential for issues to compound. Internal dependencies are those that are within the research software itself and can include different modules, libraries or classes that are part of the software and are used to perform specific tasks. Internal dependencies are generally easier to manage and control, as they are part of the research software and can be developed and maintained by the research team. External dependencies are dependencies that are not part of the research software but are required for its use. These can include external libraries, frameworks and other software packages that the research software depends on. They can be more challenging to manage as they are not under the control of the research team and may change or become unavailable over time.

**Table 1 | Containers versus virtual machines**

| Feature | Containers | Virtual machines |
|---|---|---|
| Resource usage | Share the host operating system and the host kernel, making them lightweight and efficient | Require more resources than containers, as each virtual machine needs its own copy of the operating system and resources are divided among all the virtual machines running on the host machine |
| Deployment | Can be deployed and run quickly and smoothly, as they do not require a full operating system installation | Require a full operating system installation and can take longer to deploy and run |
| Portability | Highly portable and can run on any host system with the same architecture — ideal for moving applications between environments | Can also be portable but require virtualization software that must be installed on each host system, making them less flexible for moving between environments |
| Isolation | Less isolated; although they share the same operating system and kernel as the host machine, they are still isolated from one another and can run different applications and processes. More lightweight and efficient, as they do not require a separate copy of the operating system for each container | Completely isolated from one another and the host operating system. Each virtual machine has its own copy of the operating system and runs in its own self-contained environment. Useful for creating multiple environments that need to be separate from one another, such as for testing or development purposes |
| Scalability | Can be scaled up or down as needed, making them ideal for applications that require horizontal scaling | Can also be scaled, but it may be more challenging, as adding or removing virtual machines requires changes to the virtualization environment itself |

Finally, the third Docker component is Docker Hub, a cloud-based registry service for storing and distributing Docker images. It allows users to create and share Docker images with others, as well as to discover and download prebuilt images created by other users. Docker Hub also provides features such as automated builds, version control and collaboration tools. It is the default registry for Docker users and is used by many organizations to store and share their containerized applications. For readers familiar with the Git ecosystem, one can think of the relationship between Docker and Docker Hub as that of Git and GitHub[35,36]. Alternatives to Docker Hub abound; one very popular is GitHub Container Registry.

## Personalizing containers

Reusing existing containers saves time and effort, but researchers often need to create personalized containers. Here are some steps to personalize containers. Personalization of containers requires identifying the specific software and data to be used, including consideration of programming languages, libraries and packages, as well as data type, storage and access. The inclusion of a Dockerfile, which tells the Docker engine what to do when building the image, also must include the software, libraries and other dependencies needed. Researchers can also use the 'docker commit' command to personalize a container by creating a new image from a running container and modifying it with additional software or configuration changes. However, it is important to note that using docker commit reduces visibility and may make the container less reproducible and reliable than using a Dockerfile.

It is important to emphasize that using language-based package managers in containers, such as pip for Python or npm for Node.js, can facilitate the installation of software packages and dependencies within the container. This helps ensure that the container is reproducible and reliable, as it allows for fine-grained control over the versions of packages installed. The resulting Docker image will contain all the software and data specified in the Dockerfile, as well as any additional files or resources that were included in the image.

The container runtime can be further customized by setting environment variables, mounting volumes and specifying network configurations. The entry point and command can be customized to allow the specific actions that the container will take upon launch. After a container has been created, users may want to expose certain ports from the container to the host system, which allows accessing network services running inside the container from outside the container. This is typically done using port mapping, which involves mapping a port on the host system to a port in the container.

Once built, the container must be tested, and tools such as strace and gdb are used to debug any issues. By personalizing their containers, researchers can ensure that their research is reproducible, collaborative and portable, making it easier to share and build upon[37]. An overview of common Docker commands is provided in the Supplementary information.

## Complements and alternatives to Docker

Docker benefits from a rich ecosystem of interrelated components that are in constant development, owing to its growing popularity. The Docker ecosystem includes various open-source and commercial tools, services and technologies that facilitate the development, deployment and management of containerized applications. It extends the capabilities of the Docker platform and enables integration with other systems and technologies. These components include Docker Engine (the core container engine that allows building, running and managing containers), Docker Compose (a tool for defining and running multicontainer applications), Docker Swarm (a container orchestration platform for managing large clusters of Docker nodes), Docker Machine (a tool for provisioning and managing Docker hosts) and many others, each serving a specific purpose.

In addition to the core Docker ecosystem, several tools have been developed that rely on Docker to implement additional functionalities or features. For example, the Rocker project[38,39] provides containers with environments that can accommodate R users straightforwardly. It includes tools for building Docker images, creating and managing containers and automating tasks using shell scripts. Rocker is specifically designed for scientific research and includes several prebuilt images for common scientific computing tasks. Similarly, containerit[40] makes it easy to package research software and dependencies into containers and includes tools for building and managing Docker images and containers. It is intended to be used as a command-line tool and can help automate the creation of containers for research software. Both Rocker and containerit are tools that are designed to help scientists create and manage containers for scientific research; Rocker is geared towards building and managing container-based workflows for reproducible research, whereas containerit is focused on creating containers for research software.

Relatedly, the R package liftr[41], which uses Docker to containerize and render RMarkdown documents, can also be used for reproducible reporting. To make it easier from the perspective of a user, an RStudio add-in is available, which enables self-contained implementation from within R. Further guidelines exist on how to build reproducible data analysis workflows, including via combining tools such as R Markdown, Git, Make and Docker[42].

Although Docker is still the most popular containerization platform with the largest ecosystem and user community, there are other alternatives available. These include not only Singularity and Podman but also OpenShift, LXC, Rocket or Mesos. We discuss two popular alternatives, Singularity (a containerization platform specifically designed for HPC environments) and Podman (a command-line tool that is designed to be used in a similar way to Docker but with a few key differences), in Table 2, including by comparing their features and functionalities with those of Docker.

## Results

By using containers, researchers can address several issues that can arise over the course of a research project or research programme, including reproducibility, collaboration, compatibility, scalability and management. Here, we discuss these five problems in the context of compatibility across systems, reliability across versions, resource allocation and the facilitation of large-scale collaboration.

### Compatibility across systems

Containers are able to abstract the application from the underlying hardware and operating system[43]. This means that the same containerized application can be run on various different systems, thus drastically reducing compatibility issues, allowing researchers to share and compare data and results. If different systems are not compatible, it can be difficult to exchange data and collaborate on research projects[44,45].

Several advantages of containerization also stem indirectly from improved compatibility across systems. For example, containers establish a standard for data storage and analysis, simplifying comparison and validation of results. In neuroscience, containerization has helped implement a standardized functional MRI preprocessing pipeline known as fmriprep. Similarly, projects such as the Experiment Factory[46] in the behavioural sciences have facilitated the use of Docker containers to ensure that experiments can run smoothly across platforms.

### Reliability across versions

Another benefit of using containers is the ability to ensure reliability across different versions of the software[47,48]. This is particularly important when it comes to deploying and maintaining applications in a production environment, as it can be challenging to ensure that the application will run smoothly and consistently across different versions of the operating system or other dependencies[49].

Containers can help ensure reliability across versions by providing a consistent runtime environment for the application, regardless of the underlying operating system or hardware. Containers also provide isolation between different applications and their dependencies, as each container has its own dedicated resources and is unaware of the existence of other containers if they are not actively networked together[50]. Finally, containers can make it easier to manage and maintain applications in a production environment[29]. For example, containers can be used to automate the deployment and updates of applications[51], ensuring that the correct versions are deployed to the correct locations, thereby reducing the risk of errors or downtime caused by manual deployment processes, as well as making it easier to roll back to a previous version if necessary[52].

### Resource allocation

Containers streamline resource allocation, reduce resource consumption and minimize conflicts between research projects, facilitating project management and promoting smooth coordination among researchers. Because they encapsulate all the dependencies and configurations required for their research within a self-contained unit, containers allow researchers to create reproducible and isolated environments that do not interfere with other research projects. Containers also ensure that research projects do not consume unnecessary resources, as only the required dependencies are included in the container image, avoiding any unnecessary overhead.

### Implications for large-scale collaborative efforts

Containerization improves efficiency; teams can standardize their development environments, reducing conflicts and improving compatibility between different systems[53], making it easier to share code and resources and to ensure that code written on one system will work

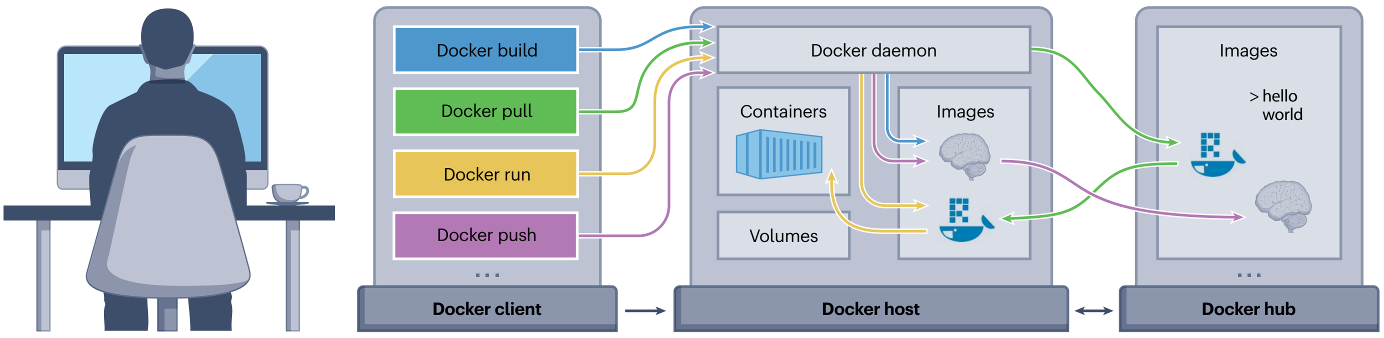

**Fig. 1 | Docker architecture.** Docker uses of a client-server architecture, whereby the Docker client talks with the docker daemon – a software component that runs on the Docker host and helps build, run and distribute containers. The Docker client and daemon can run either on the same system or remotely. The Docker daemon interacts with Docker Hub through the Docker command-line interface client, which provides an interface for users to interact with the Docker API. The Docker command-line interface client sends requests to the Docker daemon, which then communicates with Docker Hub to pull, push or manage Docker images and containers.

## Box 2

## Getting started with containers

Here we walk you through the steps to get started with containers, including how to install the necessary tools and run your first Docker container.

**Step 1: install a container runtime**

To install Docker, follow these steps:

- Go to the Docker website (docker.com).
- Follow the instructions to install Docker on your machine. This will typically involve downloading an installer and running it on your machine.

**Step 2: pull a container image**

Once you have installed Docker, you can start pulling container images from a public registry, such as Docker Hub, or you can also set up your own private registry. To pull a container image from Docker Hub, use the following command:

```
$ docker pull <image name>:<tag>
```

Images on Docker Hub are typically tagged with version information or other identifiers to help ensure reproducibility of research workflows. It is important to specify a specific tag when pulling an image to ensure that you get the same version every time. For example, to pull the latest version of the Ubuntu environment, you would run:

```
$ docker pull ubuntu:latest
```

**Step 3: run a container**

Now that you have pulled a container image, you can run it as a container using the following command:

```
$ docker run <image name>
```

For example, to run the Ubuntu container that you just pulled, you would run:

```
$ docker run ubuntu
```

This will start a new container on the basis of the Ubuntu image and give you a command prompt inside the container. From here, you can run any commands you would normally run on a Ubuntu machine.

**Step 4: stop and remove a container**

When you are finished with a container, you can stop it and remove it from your machine using the following command:

```
$ docker stop <container name>
```

To remove a container, use the following command:

```
$ docker rm <container name>
```

You can find the name of your container by running the docker ps command, which will list all running containers.

**Step 5: use preset or personalized environments**

For most research projects, we want to have a whole environment setup with all tools and dependencies needed for the entirety of the project. Many preconfigured Docker images exist for such purposes, but you can also create an environment from scratch to suit your personalized research needs and then save it into your own Docker image, which can then be used by collaborators. Detailed, step-by-step instructions on how to do this have been published elsewhere[102]. See the Docker documentation to learn more.

correctly on another. Because containers are platform-independent, researchers are also able to work on different platforms, with seamless deployment to HPC environments. In HPC environments, it is common to have a cluster of machines with different operating systems, different versions of libraries and other dependencies; containerization helps overcome this problem by ensuring that the application and its dependencies are all included in the container[54] and that the container is portable and can run on any system with a compatible container runtime[49].

Containers can be scaled up or down as needed and they are portable, allowing teams to respond quickly to changing demands and to work across multiple platforms or locations[55,56].The latter can be particularly useful for teams that need to move applications between different stages of the development or deployment process or between different environments such as test, staging and production.

Finally, containers can be easily integrated with collaboration tools that allow multiple developers to work on a project at the same time, such as Git[57]. For example, developers can use Git to track changes to the application code and its dependencies, and when they are ready to share their changes with the team, they can use Git to push the changes to a central repository. Other team members can then pull the changes and use the containerized application with all its dependencies and requirements.

## Applications

Containers are being used in a growing number of scientific fields, enabling efficient, collaborative forms of research. Here, we describe usage in several disciplines – neuroscience, ecology, genomics, astronomy, physics and environmental science – with concrete examples of container implementations in each case.

### Neuroscience

Containers have gained popularity in recent years as a means of packaging and distributing software and code in neuroscience and are currently used for a number of applications, such as neuroimaging data analysis. Neuroimaging data can be extremely large and complex[58,59]; using containers, researchers can share tools for neuroimaging analysis, such as MRI processing software or brain connectivity analysis tools[28,60,61]. These developments facilitate the analysis and visualization of brain imaging data, as well as sharing reproducible results. For example, tools for neuroimaging analysis, including those for processing and analysing MRI data, are provided as Docker images by the FSL project.

Containers are also used to provide neural simulation software, such as NEURON or NEST, to enable running simulations on different computer systems and sharing them. Similarly, the NeuroDebian project provides Docker images for tools for neural data analysis, including tools for processing and analysing electrophysiology data[62]. In the area of brain–computer interfaces, containers are being increasingly used to package software tools for EEG analysis software or BCI control software. For example, the BCI2000 project provides Docker images that facilitate development and testing for brain–computer interface systems[63].

More generally, containers are also used for data sharing and collaboration – as neuroscientists often work with sensitive or proprietary data, containers can provide a secure and controlled environment for sharing and accessing data, enabling more effective collaborations[64]. For example, containers can be configured to run with limited permissions and user accounts, making it possible to give access to specific data only to authorized users. This built-in flexibility enables easy collaboration and data processing, while simultaneously providing a secure and controlled environment in which data are isolated and protected when necessary.

### Ecology
In ecology, Docker is often used to run simulations of ecosystem dynamics[65,66]. Containers can be used to package and deploy the necessary code and data for running complex ecosystem simulations; for example, the Ecopath with Ecosim project provides Docker images for ecological simulation models that can be used to explore the impacts of different management scenarios. This allows ecologists to easily share and reproduce results[67], as well as scale their simulations to large compute resources.

Containers can be used to package and distribute software tools for data processing and analysis, such as Geographical Information System (GIS) software or machine learning libraries. This is often useful to analyse and visualize spatial data or to apply machine learning techniques to ecological data. The QGIS project provides Docker images for the QGIS GIS software that can be used to analyse and visualize spatial data. Containers are also used to share software tools for environmental monitoring[68], such as sensor networks or remote-sensing platforms, making it easier for ecologists to collect and analyse data from field sites or to integrate data from multiple sources. For example, the Environmental Data Commons project provides Docker images for environmental monitoring tools that can be used to collect and analyse data from field sites.

Finally, containers can also be used to distribute data management and analysis tools, such as databases or data visualization software. These help ecologists store, organize and analyse large data sets. For example, the EcoData Retriever project provides a Docker image for downloading and cleaning up ecological data from various sources[69]. Containers are also central to the packaging and distribution of ecological modelling software, such as population dynamics models or ecosystem models used to build and test models of ecological systems. The Ecological Niche Modeling on Docker project provides Docker images for ecological modelling in R, including tools for building and fitting models and visualizing results.

### Genomics
In the field of genomics, containers are routinely used to package and distribute software tools for analysing various types of genomic data[70–72]. For example, in the DNA sequencing data analysis, the BioContainers project[73] provides Docker images for tools, including tools for read alignment and variant calling. Docker images for tools in the gene expression analysis (such as RNA sequencing (RNA-seq) and microarray analyses) are provided by the Bioconductor project[74]. Similarly, Docker images for tools in population genetics and evolutionary analyses are provided by the EIGENSOFT project.

For genome annotation, the GFF3 Tools project[75] provides Docker images for tools such as gene prediction software and functional annotation tools, whereas Docker images for tools for structural variation analysis, such as copy number variation analysis software and translocation detection tools, are provided by the Breakdancer project. A number of projects provide Docker images for tools for functional genomics, such as gRNA design and validation to aid in CRISPR-related research[76].

**Table 2 | Docker, Singularity and Podman feature comparison**

| Feature | Docker | Singularity | Podman |
|---|---|---|---|
| Container runtime | Yes | Yes | Yes |
| Container image management | Yes | Yes | Yes |
| Container orchestration | Yes[a] | No | No |
| Support for multiple operating systems | Yes | Yes | Yes |
| Integration with scientific workflow tools | Yes[b] | Yes[b] | Yes[b] |
| Support for reproducible research | Yes[c] | Yes[c] | Yes[c] |
| Support for data management[d] | Yes[e] | Yes[e] | Yes[e] |
| Compatibility with Docker images | Yes | Yes | Yes |
| Support for legacy software[f] | Yes[g] | Yes[g] | No |
| Support for OS-level virtualization | No | Yes | No |
| Rootless mode | No | Yes | Yes |
| Build-time customization | Yes | No | Yes |
| Image format | Docker Image Format (DIF) | Singularity Image Format (SIF) | OCI Image Format |
| Security features | User namespaces, AppArmor profiles | Run containers as unprivileged users | Seccomp support |
| Networking and storage options | Docker networking | Custom networking options | Host system networking with custom options |
| Ecosystem and community | Large and mature | Limited but growing | Limited but growing |

OCI, Open Container Initiative. [a]Via tools such as Docker Swarm and Kubernetes. [b]Via tools such as Snakemake, Nextflow, WholeTale, Binder and CodeOcean. [c]Via tools such as WholeTale, Binder and CodeOcean. [d]Data management refers to the process of collecting, storing, organizing, preserving, maintaining and using data in a way that ensures its quality, security, accessibility and reliability over time. [e]Via tools such as DataLad and XNAT. [f]Legacy software refers to software that is no longer being actively developed or maintained or to software that was written for an older operating system or hardware architecture. [g]Via tools such as Shifter.

Containers can also be used to share tools for the RNA-seq analysis[71,77,78], such as read alignment tools or expression quantification software, facilitating the analysis and interpretation of RNA-seq data. Some examples of tools that can be packaged in containers for RNA-seq analysis include STAR[79,80] and Salmon[81]. Recently, containerized software has been developed to share tools for epigenomics analysis, such as DNA methylation analysis software or chromatin accessibility tools, including Bismark and ATAC-seq Pipeline[82].

Finally, containerization has a central role in the development of tools for analysing genetic variation data, such as single-nucleotide variant calling software or structural variation detection tools[83–85]. These include tools for genetic variation analysis such as GATK and SVTyper.

### Astronomy

Containers are increasingly being used in the field of astronomy, making it easier for astronomers to access and use specialized software and data and enabling reproducibility and collaboration[86–88]. One application of containers in astronomy is the packaging and distribution of software tools for analysing astronomical data. The Astropy project[89] provides a Docker image for the Astropy python library, which is a widely used toolkit for astronomy and astrophysics and includes tools for handling and manipulating astronomical data, such as reading and writing FITS files, performing coordinate transformations and fitting models to data. By packaging the Astropy library in a container, astronomers can seamlessly install and use the library on their own systems, while minimizing issues with dependencies or conflicts with other software[89,90].

Astronomy research often involves the use of specialized software and data that may be difficult to obtain or install. By creating a Docker image that includes all the necessary software and data, astronomers can share their research environment with others, enabling them to reproduce and verify results of each other. The Sloan Digital Sky Survey provides Docker images[91] to facilitate reuse by astronomers and astrophysicists, whereas the Marble Station project provides a Linux environment with spectroscopic and photometric tools that are commonly used in astronomy.

Finally, containers can be used in astronomy to facilitate collaboration and data sharing[92]. The SciServer project provides a scalable collaborative data-driven science platform for astronomers and other scientists, using a Docker-based architecture[87]. SciServer enables astronomers to access and work with data stored on the platform, regardless of their own computing environment.

### Physics

Containers have been used in the field of physics for some time, mainly to distribute software tools for analysing physics data[93,94].The CERN Container Registry provides container images for various tools and libraries that are commonly used in particle physics, including software for analysing particle collision data and simulations. Physics research often involves the use of specialized software and data that may be difficult to obtain or install. By creating a container image that includes all the necessary software and data, physicists can share their research environment with others, enabling them to reproduce and verify results of each other. GEANT4 (ref. 95), also developed by CERN, offers resources for simulating the passage of particles through matter, with practical uses in high-energy, nuclear and accelerator physics. The LIGO Open Science Center[96] provides a Docker image for the LIGO Data Grid, a cloud-based platform for storing and accessing data from the LIGO gravitational wave detectors, so as to allow physicists to work with data stored on the LIGO Data Grid.

### Environmental science

Containers are increasingly relevant to the field of environmental science; the Planet Research Data Commons for environmental and earth science research provides container images for various tools and libraries that are commonly used in environmental science, including software for analysing data on air quality, water quality and land use. These tools are designed to help address important environmental issues such as adapting to climate change[97], saving threatened species[98] and reversing ecosystem deterioration[99].

A related application is for the reproducibility of research environments. For example, the EarthData project provides container images for various tools and data that are commonly used in environmental science, including software for analysing data from satellites and remote-sensing instruments, and data sets such as the NASA Earth Observing System data[100]. GRASS GIS — a free and open-source GIS software — can be used for data analysis, visualization and spatial modelling. The GRASS GIS container image provides a preconfigured environment for running GRASS GIS, including all necessary dependencies, libraries and configurations. Similarly, GeoServer is a container that provides a preconfigured environment for running analyses on geospatial data and enables data processing, analysing and sharing. Together, these tools facilitate collaboration between environmental scientists worldwide[101].

## Reproducibility and data deposition

Providing reproducible content is a fundamental aspect of scientific research and has been discussed at length elsewhere, either generally[23] or in the context of containerization[102,103]. Here, we focus on best practices for sharing containers, commenting and documenting and elaborate on how these can help communicate and disseminate research findings to maximize their value.

### Sharing containers

Containers can help researchers share their work with co-workers, regardless of the underlying hardware or operating system, and help improve the reproducibility of research results[104]. By providing a consistent environment for running experiments, researchers can ensure that their results are not affected by differences in hardware or software configurations. This can be particularly important in fields such as machine learning or data analysis, in which small differences in environment can lead to significant differences in results[105,106]. Sharing both the Dockerfile and the Docker image when sharing containers is considered best practice because it allows others to reproduce the exact same environment and configuration of the containerized application. However, simply having a Dockerfile does not guarantee the same build every time. There are several factors that can affect the build, such as the version of the base image, the availability of packages or the version of the software used. A Dockerfile may be created that specifies the base image and includes instructions to install the required versions of dependencies, as well as instructions to copy the source code of the application into the container to configure the environment variables. If other researchers use different versions of the dependencies or if the dependencies are not available, the build can be different. Nonetheless, sharing both the Dockerfile and the container image is still essential for enabling reproducibility, as it provides a starting point for other researchers to build upon and modify for their own purposes.

To ensure that others can reproduce the exact same build, it is best practice to also share the image of the container in a Docker registry such as Docker Hub or Quay. Sharing the container in this way allows others to download and run it without having to build it themselves, which guarantees that they are running the same environment and configuration as the original application. In addition, for certain applications, sharing the data set used in the training process with the container could be both useful and facilitate reproduction of simulations or experiments. It is worth pointing out that it is possible to construct Dockerfiles to be reproducible, for example, by freezing older versions of container images using RStudio Package Manager, yet even so images that are not pulled by anyone from Docker Hub for extended periods of time get purged, and Dockerfiles are not guaranteed to build indefinitely. Alternatives to specialized registries exist for sharing containers; for example, researchers can also use a cloud platform such as Amazon Web Services or Google Cloud Platform to host and share containers. These platforms provide tools for building, storing and distributing containers and can be useful for sharing large or complex containers.

Finally, researchers can implement more advanced workflows to build and share their work, for example, by generating automated Docker builds. One common approach to do that is to configure automated builds in Docker Hub, which will re-build an image whenever changes are pushed to the source code repository. The advanced features of Docker Compose — a tool that allows defining and running multicontainer applications — can help implement automated builds when dealing with multiple containers. Similarly, Jenkins is a commonly used open-source automation server that facilitates automation of diverse tasks such as creating and launching Docker images. A Jenkins pipeline can be set up that will build an image and push it to a registry when certain conditions are met, like when changes are pushed to the source code repository. Alternatively, GitHub Actions is a Continuous Integration/Continuous Deployment (CI/CD) platform that allows developers to automate their workflow on GitHub such as building and testing code, deploying code to different environments and managing dependencies. GitHub Actions allows creating custom workflows that are triggered by events such as commits, pull requests and releases and has been implemented successfully in research workflows[107]. Other options include TravisCI, GitLabCI and CircleCI — all CI/CD tools specifically designed for automating the software development process and provide integration with various services, including Docker.

## Best practices in commenting and documenting

Containers are self-contained units of software that include all of the dependencies and resources needed to run an application and, as such, they can be complex and difficult to understand. For research development projects, effective commenting and documentation are essential. Commenting refers to explanations, descriptions and notes within the source code, which aids in understanding the purpose of the code, whereas documentation, which is typically external to the code and presented in the form of a README file or user manual, provides additional details[103]. Proper commenting and documentation can help make containers more readable and maintainable and facilitate the use and utility of containerized software[108]. Table 3 provides important best practices for commenting and documenting in the process of sharing containers. By following these best practices and using appropriate tools, researchers can share their containers with others seamlessly and effectively. In addition to these general best practices, it is also important to follow any required field-specific guidelines or conventions (F1000Research guidelines)[109–111].

## Communication and dissemination of research findings

Containers can facilitate the communication and dissemination of findings in a number of ways. First and foremost, they allow researchers to share their software applications with other researchers, regardless of the operating system or hardware being used, allowing for greater collaboration and the potential for faster progress in their respective fields. Containers also allow for the creation of fully reproducible research environments, ensuring that findings can be accurately replicated and verified. In addition, containers make it easier for researchers to publish their findings in an accessible format by creating a self-contained package that can be easily deployed and run by anyone with access to the container.

Containers have emerged as a promising approach for archiving research software alongside a publication. Many computing-focused archives, such as the ACM Digital Library, offer services for archiving research software. However, the adoption of such services has been low, and there is a need for better archiving practices that can ensure long-term preservation and reproducibility of research software. Containerization can offer several advantages for archiving research software. By encapsulating all dependencies and configurations of

**Table 3 | Best practices for commenting and documenting containers**

| Best practice | Example |
|---|---|
| Clearly document the purpose of the container | # This container provides a preconfigured environment for running the my-science-app application |
| Include detailed instructions for how to use the container | # To use this container:<br># 1. Pull the image from the Docker registry:<br>`$ docker pull myusername/my-web-app`<br># 2. Run the container: `docker run -p 8080:80 myusername/my-web-app`<br># 3. Access the app at http://localhost:8080 |
| Provide example usage or run commands | # Here is an example of how to run the container with a custom configuration file:<br>`$ docker run -p 8080:80 -v/path/to/config:/etc/my-web-app myusername/my-web-app'` |
| List any environment variables that the container expects to be set | # The container expects the following environment variables to be set:<br># - DB_USERNAME (username for the database)<br># - DB_PASSWORD (password for the database)<br># - DB_HOST (hostname of the database server) |
| Document any ports exposed by the container | # This container exposes port 80 for HTTP traffic |
| List any external dependencies | # This container requires access to a MongoDB server running on hostname mongodb.example.com |
| Document any data or volume mounts used by the container | # This container expects a volume to be mounted at /var/www/html for the application code |
| Reference any related projects, links or documentation | # This container is based on the official Node.js Docker image, see more details at https://hub.docker.com/_/node/ |
| Add any version information | # Current version: 1.0.0 |
| Add any licensing information | # Licensed under the CC BY 4.0 licence |

## Glossary

| | |
|---|---|
| Clusters | Groups of machines that work together to run containerized applications. |
| Compute resources | The resources required by a container to run, including central processing units, memory and storage. |
| Containerization platform | A complete system for building, deploying and managing containerized applications, typically including a container runtime, and additional tools and services for things such as container orchestration, networking, storage and security. |
| Container runtime | The software responsible for running and managing containers on a host machine, involving tasks such as starting and stopping containers, allocating resources to them and providing an isolated environment for them to run in. |
| Continuous Integration/ Continuous Deployment | (CI/CD). A software development practice that involves continuously integrating code changes into a shared repository and continuously deploying changes to a production environment. |
| Dependencies | Software components that a particular application relies on to run properly, including libraries, tools and frameworks. |
| Distributed-control model | A deployment model in which control is distributed among multiple independent nodes, rather than being centralized in a single control node. |
| Docker engine | The containerization technology that Docker uses, consisting of the Docker daemon running on the computer and the Docker client that communicates with the daemon to execute commands. |
| Dockerfiles | A script that contains instructions for building a Docker image. |
| Environment variables | A variable that is passed to a container at runtime, allowing the container to configure itself on the basis of the value of the variable. |
| High-performance computing | The use of supercomputers and parallel processing techniques to solve complex computational problems that require a large amount of processing power, memory and storage capacity. |
| Host operating system | Primary operating system running on the physical computer or server in which virtual machines or containers are created and managed. |
| Image | A preconfigured package that contains all the necessary files and dependencies for running a piece of software in a container. |
| Namespaces | Virtualization mechanisms for containers, which allow multiple containers to share the same system resources without interfering with each other. |
| Networking | The process of connecting multiple containers together and to external networks, allowing communication between containers and the outside world. |
| Orchestration | The process of automating the deployment, scaling and management of containerized applications in a cluster. |
| Orchestration platform | System for automating the deployment, scaling and management of containerized applications. |
| Port mapping | The process of exposing the network ports of a container to the host machine, allowing communication between the container and the host or other networked systems. |
| Production environment | Live, operational system in which software applications are deployed and used by end-users. |
| Runtime environment | Specific set of software and hardware configurations that are present and available for an application to run on, including the operating system, libraries, system tools and other dependencies. |
| Scaling | The process of increasing or decreasing the number of running instances of a containerized application to meet changing demand. |
| Shared-control model | Deployment model in which a single central entity has control over multiple resources or nodes. |
| Volumes | A storage mechanism for containers, which allows data to persist outside the file system of the container, including after a container has been deleted or replaced. |

the software into a single container image, containers provide a self-contained and portable environment that can be easily shared and preserved. Specifically, container images can be archived as part of the publication or deposited in a container registry for long-term preservation. Moreover, containers allow for the reproducibility of research findings by ensuring that the software environment remains consistent, even as the underlying infrastructure changes over time.

Researchers can ensure effective archiving with containers by adopting several recommended practices. This involves using well-defined and well-maintained container images, with clear documentation on the included software dependencies and configurations. Additionally, all software dependencies and configurations should be well documented, including versions of libraries, software and operating systems. Widely used and supported container formats, such as Docker, Singularity or Podman, should be chosen on the basis of target archive or repository requirements. Metadata and documentation, such as a README file providing instructions for using the container, research software information and licensing details should be provided with the container image. Following these best practices, archiving research software with containers can be a valuable approach for ensuring long-term preservation and reproducibility of scientific findings.

## Limitations and optimizations

Although containers are powerful and versatile, they also have important limitations. In this section, we discuss some of the pros and cons of containerization and explore key restrictions. We also discuss compatibility with HPC environments, which have become increasingly popular among computational research groups.

### Costs of containerization

Despite its numerous advantages, there are also some costs associated with using containers in scientific research[112]. One of the main drawbacks is the learning curve involved in using containerization tools and technologies. Researchers need to familiarize themselves with containerization concepts, such as images, containers and registries, as well as how to use tools such as Docker to manage and deploy containers. This can require a significant amount of time and resources, especially for researchers who are new to containerization. To address this challenge, it is often helpful to start with the basics, that is, focusing on understanding the concepts and fundamentals of containerization, such as what a container is, how it works and the benefits of using containers. Getting hands-on experience creating and running containers using tools such as Docker in practice scenarios helps users learn to work with containers. Online communities dedicated to containerization and related technologies, such as forums and social media groups, can provide resources, tips and best practices from experienced developers. In addition, there are a plenty of online tutorials, books and courses available that teach both the basics and more advanced concepts of containerization.

Building container images requires a certain level of expertise and specialized knowledge, which can be challenging and time-consuming to obtain. In addition, building, testing and deploying container images can require dedicated staff, infrastructure and resources — these can include servers, storage and networking, as well as the orchestration software needed to manage and deploy the containers. These resources come at a cost, which can increase when scaling and maintaining the infrastructure.

In addition to the cost of infrastructure and resources, there are also sustainability costs associated with the use and maintenance of containers. The energy consumption of containerized workloads can be substantial, as they require server and networking infrastructure to support their operation. Serverless computing — a cloud computing model that allows developers to deploy and run code without the need to manage infrastructure — has been proposed as a way to mitigate these costs via dynamic allocation of computing resources on the basis of current workload demands. However, it is important to note that the suitability of serverless computing for long-running computations in scientific research may depend on various factors, such as the specific requirements of the computation, the available budget and trade-offs among cost, performance and convenience. For certain types of computations or workloads, serverless computing may still be a viable option, especially when considering factors such as ease of deployment, automatic scaling and reduced operational overhead.

In most cases, the benefits of using containers in scientific research outweigh the costs of learning and managing containerization tools and technologies. Containers are often seen as a middle ground between the lightweight and easy-to-use nature of package managers and the comprehensive isolation and reproducibility of virtual machines. Containers provide a balance between these two extremes, offering a higher level of isolation and reproducibility compared with package managers, while being more resource-efficient and portable than virtual machines. This is one of the reasons containers have become widely adopted in various contexts such as continuous integration, industry and cloud use and increasingly in research, in which reproducibility, portability and resource efficiency are crucial factors for success. However, it is important for researchers to carefully consider their needs and resources when deciding whether to use containers in their research projects. One important aspect to consider is the current efficiency of the research workflow and how it can be influenced by containerization. We now turn to specific limitations and concrete solutions to build more efficient workflows in scientific research.

**Limitations of containerization**

Containerization may not be suitable for certain types of research that depend on the kernel level or on hardware. This can be a particular issue for machine learning workflows that rely on GPU acceleration, as containers may not be able to access the necessary hardware resources[30,32], because they are designed to be hardware-agnostic and rely on the kernel of the host system to interface with hardware. As a result, the host operating system kernel version and configuration can have an impact on the behaviour and performance of containers. For example, if a container requires a specific kernel feature that is not available in the host operating system kernel, it may not function correctly or may require modifications. Similarly, if the host operating system kernel has specific security settings or restrictions, they may apply to containers as well. This can be a challenge for machine learning workflows that require access to specialized hardware such as GPUs, as containers may not have direct access to these resources. It is therefore important to consider the compatibility and dependencies of the host operating system kernel when working with containers to ensure proper functionality and reproducibility.

There are several solutions that can be implemented to address these potential limitations. Containerization technologies exist that can facilitate better access to host resources, such as Podman or Singularity (Table 2). By allowing the containers to run as regular processes on the host operating system, without requiring a separate daemon or root privileges, these tools provide a more native experience and facilitate access to host resources more directly and efficiently. This can be especially important for low-level access to system resources, such as kernel-level features or hardware devices. By contrast, Docker relies on a daemon process to manage container execution, which can introduce additional layers of abstraction and potential performance overhead. In addition, Docker containers typically run as the root user by default, which can pose security risks and limit access to certain system resources. As a best practice, it is recommended to set the USER in Docker containers to a non-root user to mitigate potential security risks and restrict unnecessary access to system resources, following the principle of least privilege. This can help improve the security posture of Docker containers and reduce the risk of unauthorized access or exploits.

If specific access to the kernel or to hardware resources is required, it is also possible to use virtual machines, which can provide access to host resources through virtualization (Table 1). The choice between containerization and virtual machines does not have to be dichotomous; however, hybrid solutions exist, such as running a containerized application on top of a virtual machine, which allows the containerized application to have the portability and isolation benefits of containerization while also having access to the host resources through the virtual machine.

It may not always be possible to fully replicate the environment in which research was conducted, which can be particularly challenging when using containers to replicate research environments with complex dependencies or that rely on specific hardware configurations[113]. In these cases, it may be difficult to fully replicate the environment using containers, which can limit the reproducibility of the research. Specific resources can help make the environment more portable and reproducible; for example, Platform as a Service (PaaS) is a cloud computing service that allows developers to create, launch and manage applications without the need to handle the underlying infrastructure. Examples include Heroku, Google App Engine and Microsoft Azure. PaaS can provide resources, scaling and dependency management, which can help make the environment more portable and reproducible. Similarly, Infrastructure as a Service (IaaS), a cloud computing service, offers virtualized computing resources, such as storage, networking and virtual machines. Some examples of IaaS providers are Amazon Web Services, Microsoft Azure and Google Cloud Platform. IaaS gives users additional control over the underlying infrastructure, including hardware and operating system, and can configure it to match the desired environment for their research. Alternatively, hardware abstraction layer (HAL) is a layer of software or hardware that abstracts the underlying hardware and operating system, allowing applications to run in a more isolated and portable manner. HAL can help isolate the application and its dependencies from the underlying hardware and operating system, providing a consistent environment for reproducible research. Containerization technologies such as Docker can be considered as a form of HAL, as they abstract the underlying host system and provide a consistent environment for running applications.

Containers can introduce additional complexity to research workflows, as researchers may need to manage and maintain multiple containerized environments. This can be time-consuming and may require additional training and support for researchers. Container orchestration tools such as Docker Swarm or Kubernetes can help manage this complexity by providing an abstraction layer that simplifies the process of deploying and scaling containers. Docker Swarm and Kubernetes handle both Embarrassingly parallel (EP) and non-EP workflows. EP workflows refer to workflows that can be parallelized and run in isolation, in which each task can be executed independently of the others, making them well suited for container orchestration. Non-EP workflows, on the contrary, have interdependent tasks that require coordination and communication between containers, which can be more challenging to manage. The two orchestration tools may require different configurations and setups depending on the workflow requirements.

For deployment, Docker Swarm uses a shared-control model, whereas Kubernetes uses a distributed-control model. Docker Swarm is tightly integrated with the Docker ecosystem and is optimized for use with Docker containers. Kubernetes, on the contrary is more flexible and can work with any container runtime, not just Docker. Both Docker Swarm and Kubernetes can scale to thousands of nodes, but Kubernetes has better support for autoscaling and can scale applications more quickly. Kubernetes has a wider range of features and capabilities, including support for rolling updates, resource quotas and pod security policies. Docker Swarm has fewer features but is generally easier to use and set up. Overall, Kubernetes is generally considered to be a more powerful and feature-rich platform, but it can be more complex to use, whereas Docker Swarm is a good choice for users who want a simpler, more streamlined solution for container orchestration.

Researchers are starting to incorporate containers into larger workflow management systems, which provide a framework for orchestrating and executing complex scientific workflows. Workflow management systems such as Nextflow, CWL and Snakemake, among others, have gained traction in the scientific community owing to their support for containerization[114]. Automation tools such as Ansible, Puppet and Chef can be used to automate the process of building, deploying and managing containers. Researchers can also rely on container management platforms such as Google Kubernetes Engine or Amazon Elastic Container Service to access a user-friendly interface for managing containers. For optimal results, it is important to consider the specific requirements of the research and the size and structure of the organization when selecting and implementing a solution.

## Adapting containers to HPC environments

Challenges can also arise when attempting to deploy containers over HPC environments[115,116]. These environments typically consist of clusters of computers with powerful processors, large amounts of memory and high-speed interconnects that allow the computers to work together in a coordinated way. They are often used to solve problems that require large amounts of data processing or simulations that would be too time-consuming or impossible to perform on a single computer. Yet, the complexity HPC often produces also adds hurdles to containerization. For example, containers can introduce overhead when compared with traditional virtualization technologies, as they run isolated processes and require additional system resources to manage the containers. Although containers are generally considered lightweight compared with virtual machines, this can result in higher overhead when running performance-critical workloads on HPC systems. However, the lightweight nature of containers can also improve resource utilization, as multiple containers can be run on the same host without significant performance degradation. This trade-off between overhead and resource utilization should be carefully considered when deciding whether to use containers for HPC workloads. Furthermore, the degree of isolation containers provide between different applications and their dependencies may not be sufficient for all HPC workloads that require tight control over system resources such as CPU, memory and input/output (I/O).

HPC environments can also lead to limitations in terms of compatibility, as these systems typically have complex and specialized infrastructure — such as parallel file systems — that may not be easily integrated with container technologies. This architecture can lead to difficulties when trying to use containers in HPC environments. Scalability can also be an issue, as HPC environments may in some instances not be well suited for running large numbers of simultaneously executing containers on a shared infrastructure. Finally, in university and research settings, HPC environments may require additional security measures to protect against unauthorized access, especially when dealing with sensitive patient-level data or patent information. In this context, containers can pose security risks, as they may not provide the same level of isolation and control as traditional virtualization technologies.

Despite these challenges, the use of containerization in HPC environments provides very attractive features and opportunities for researchers. One of the main advantages of using containers in HPC environments is their portability[117], which allows HPC workloads to be deployed on a wide range of hardware and operating systems, without the need to worry about compatibility issues or manual configuration. This can greatly simplify the process of deploying and managing HPC applications, especially in large-scale environments in which there may be many different hardware configurations and operating systems in use. Containers also improve resource utilization in HPC environments[118]; because they are lightweight and only contain the resources that are necessary for the application to run, they can be more efficient at utilizing hardware resources such as CPU, memory and storage. With containers, HPC applications can be more efficiently scheduled and run on available resources, potentially improving overall performance and minimizing resource contention. Containers can also be used to improve the security and isolation of HPC workloads, as dependencies can be isolated from the rest of the system, reducing the risk of interference or conflicts with other applications. Finally, although HPC resources have traditionally been accessed using specialized software and protocols, the use of containers can allow researchers to access HPC resources in a more cloud-native way[119], that is, in a way that is similar to how one would access cloud computing resources. This increases flexibility and scalability in a user-friendly way, in contrast to strict reliance on specialized software and protocols. For example, by using containerized workflows and tools such as Singularity, researchers can access HPC resources using familiar container orchestration tools and APIs, such as those provided by Kubernetes[120], making it easier for researchers to access and manage HPC resources, allowing a seamless integration with other tools and services. Additionally, the use of containerized workflows can enable researchers to scale their workloads more easily across HPC resources, as containers can be seamlessly transferred and executed on different HPC systems. This can be particularly useful for researchers who need to run large-scale simulations or data analyses that require significant computing resources.

There are several tools and platforms available that can be used to support the use of containers in HPC environments[34]. For example, the Open Container Initiative is a standard for building and running containerized applications and is supported by a range of container engines and orchestration tools such as Docker, Kubernetes and Mesos. These tools can be used to manage and deploy containerized HPC applications at scale, allowing organizations to take advantage of the benefits of containerization in their HPC environments.

## Outlook

Containers offer many benefits for scientific research, including the ability to package and distribute software and data in a consistent and portable manner, enabling reproducibility and collaboration and facilitating the use of cloud computing[121]. As the use of containers becomes more widespread, it is likely that they will become an increasingly important tool in scientific computing. Containers can make it easier for scientists to access and use specialized software and data and can facilitate the sharing and reproducibility of research environments[122]. This may lead to the development of new container-based tools and platforms specifically designed for scientific computing.

Containers will also become more and more useful in data-intensive research, in which large amounts of data are generated and analysed, and uptake in this space is expected to increase[123–125]. By using containers to package and distribute data analysis tools, scientists can easily share and reproduce their results and can also take advantage of the scalability and flexibility of cloud computing[126]. As scientists rely more and more on automated and reproducible research workflows, it is also likely that they will increasingly turn to containers to package and distribute these workflows. We have discussed a few of the available platforms and repositories in this article, but options will undoubtedly grow quickly in the future, as containers continue to have an important role in scientific research.

Container orchestration tools such as Kubernetes and Docker Swarm will further enable scientists to deploy and manage complex research workflows across multiple machines, improving the efficiency and scalability of their research[127]. These platforms allow researchers to deploy and manage their scientific applications and tools and enable the creation of scalable and fault-tolerant environments for running experiments and simulations, thus allowing researchers to prioritize and allocate resources to their most important tasks. Tools such as containerd and Docker Composed are helping to change the landscape of possibilities in containerization, providing convenience and enhancing capabilities for users. Containerd is an open-source container runtime that is designed to be lightweight and modular, which is becoming increasingly popular for managing containers in cloud environments, particularly in conjunction with Kubernetes. Docker Compose is a tool that allows developers to define and run multicontainer applications using a simple YAML configuration file, simplifying the definition and management of complex containerized environments.

Other recent developments such as Dev Containers are likely to gain prominence in the research space. Dev Containers allow specifying the container environment to use in conjunction with GitHub Codespaces, a feature that facilitates creating new development environments in the cloud – directly within GitHub – with the specific versions of languages, frameworks and tools that are required for a project. Dev Containers are defined using a configuration file called a 'devcontainer.json' file, which specifies the container image that should be used, along with any additional configuration options such as environment variables, volumes and ports. These files automatically launch the container environment in the codespace, allowing researchers to switch between different container environments.

Finally, recent developments in cloud are changing the way containers are being used and shared. Specifically, the trend towards building cloud-native applications, which are designed to be scalable and resilient, has led to the adoption of containerization to package and deploy these applications. Cloud-native applications often use microservices architecture, which relies on containers to manage individual components and services. Relatedly, serverless computing is often used in conjunction with containerization to package and deploy code in a more efficient and scalable manner. Together, these features can in turn allow researchers to effortlessly scale their computations across multiple machines, potentially improving the efficiency and speed of their research.

The implications of containerization are vast and far-ranging and could impact the whole ecosystem of scientific research. Containers have the potential to heavily influence scientific publishing, via tools such as WholeTale[128], Binder and CodeOcean, which are designed to facilitate the integration between published research and containerized research[129]. These tools enable researchers to create and share reproducible research environments using containers and provide platforms for publishing and sharing research that is based on containers, with additional features and functionality specifically designed for reproducible research above and beyond those available with Docker. It is also possible that funding agencies will recognize the value of containerization to ensure quality and reproducibility of scientific research[130,131] and thus require containerization for funded projects in the future. This may necessitate the development of new infrastructure, training and support for researchers – factors that funders will need to consider to successfully implement a requirement for containerization in scientific research.

In our view, the use of containerization in scientific research is a natural evolution that is likely to become standard practice[132]. Containerization is booming, with constant innovation and development, and has become the norm in fields such as software development and engineering[133]. There is no reason scientists should not leverage this tool to improve scientific practices, as well as the quality and impact of their research. Many scientists already share data and materials with their publications[12,134–136] – containerization is the next natural step in this direction[102,137], with the potential to revolutionize scientific research and discovery.

Published online: 13 July 2023

## References

1. Hsiehchen, D., Espinoza, M. & Hsieh, A. Multinational teams and diseconomies of scale in collaborative research. *Sci. Adv.* **1**, e1500211 (2015).
2. International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature* **409**, 860–921 (2001).
3. Kandoth, C. et al. Mutational landscape and significance across 12 major cancer types. *Nature* **502**, 333–339 (2013).
4. DeGrace, M. M. et al. Defining the risk of SARS-CoV-2 variants on immune protection. *Nature* **605**, 640–652 (2022).
5. Berrang-Ford, L. et al. A systematic global stocktake of evidence on human adaptation to climate change. *Nat. Clim. Change* **11**, 989–1000 (2021).
6. Donoho, D. L. An invitation to reproducible computational research. *Biostatistics* **11**, 385–388 (2010).
7. Prabhu, P. et al. in *State of the Practice Reports* 1–12 (Association for Computing Machinery, 2011).
8. Humphreys, P. in *Science in the Context of Application* (eds Carrier, M. & Nordmann, A.) 131–142 (Springer Netherlands, 2011).
9. Cioffi-Revilla, C. in *Introduction to Computational Social Science: Principles and Applications* (ed. Cioffi-Revilla, C.) 35–102 (Springer International Publishing, 2017).
10. Levenstein, M. C. & Lyle, J. A. Data: sharing is caring. *Adv. Methods Pract. Psychol. Sci.* **1**, 95–103 (2018).
11. Kidwell, M. C. et al. Badges to acknowledge open practices: a simple, low-cost, effective method for increasing transparency. *PLoS Biol.* **14**, e1002456 (2016).
12. Auer, S. et al. Science forum: a community-led initiative for training in reproducible research. *eLife* https://doi.org/10.7554/eLife.64719 (2021).
13. Epskamp, S. Reproducibility and replicability in a fast-paced methodological world. *Adv. Methods Pract. Psychol. Sci.* **2**, 145–155 (2019).
14. Pittard, W. S. & Li, S. in *Computational Methods and Data Analysis for Metabolomics* (ed. Li, S.) 265–311 (Springer US, 2020).
15. Baker, M. 1,500 Scientists lift the lid on reproducibility. *Nature* https://doi.org/10.1038/533452a (2016).
16. Baker, M. Reproducibility: seek out stronger science. *Nature* **537**, 703–704 (2016).
17. Button, K. S., Chambers, C. D., Lawrence, N. & Munafò, M. R. Grassroots training for reproducible science: a consortium-based approach to the empirical dissertation. *Psychol. Learn. Teach.* **19**, 77–90 (2020).
18. Wilson, G. et al. Good enough practices in scientific computing. *PLoS Comput. Biol.* **13**, e1005510 (2017).
**This article outlines a set of good computing practices that every researcher can adopt, regardless of their current level of computational skill. These practices encompass data management, programming, collaborating with colleagues, organizing projects, tracking work and writing manuscripts.**
19. Vicente-Saez, R. & Martinez-Fuentes, C. Open science now: a systematic literature review for an integrated definition. *J. Bus. Res.* **88**, 428–436 (2018).
20. McKiernan, E. C. et al. How open science helps researchers succeed. *eLife* **5**, e16800 (2016).
21. Woelfle, M., Olliaro, P. & Todd, M. H. Open science is a research accelerator. *Nat. Chem.* **3**, 745–748 (2011).
22. Evans, J. A. & Reimer, J. Open access and global participation in science. *Science* **323**, 1025 (2009).
23. Sandve, G. K., Nekrutenko, A., Taylor, J. & Hovig, E. Ten simple rules for reproducible computational research. *PLoS Comput. Biol.* **9**, e1003285 (2013).
24. Fan, G. et al. in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* 463–474 (Association for Computing Machinery, 2020).
25. Liu, K. & Aida, K. in *2016 International Conference on Cloud Computing Research and Innovations* (*ICCCRI*) 56–63 (IEEE, 2016).
26. Hale, J. S., Li, L., Richardson, C. N. & Wells, G. N. Containers for portable, productive, and performant scientific computing. *Comput. Sci. Eng.* **19**, 40–50 (2017).
27. Boettiger, C., Center for Stock Assessment Research. An introduction to Docker for reproducible research. *Oper. Syst. Rev.* https://doi.org/10.1145/2723872.2723882 (2015).
**This article explores how Docker can help address challenges in computational reproducibility in scientific research, examining how Docker combines several areas from systems research to facilitate reproducibility, portability and extensibility of computational work.**
28. Kiar, G. et al. Science in the cloud (SIC): a use case in MRI connectomics. *Gigascience* **6**, gix013 (2017).
29. Merkel, D. Docker: lightweight Linux containers for consistent development and deployment. *Seltzer* https://www.seltzer.com/margo/teaching/CS508.19/papers/merkel14.pdf (2013).
**This article describes how Docker can package applications and their dependencies into lightweight containers that move easily between different distros, start up quickly and are isolated from each other.**
30. Kurtzer, G. M., Sochat, V. & Bauer, M. W. Singularity: scientific containers for mobility of compute. *PLoS ONE* **12**, e0177459 (2017).
31. Sochat, V. V., Prybol, C. J. & Kurtzer, G. M. Enhancing reproducibility in scientific computing: metrics and registry for Singularity containers. *PLoS ONE* **12**, e0188511 (2017).
**This article presents Singularity Hub, a framework to build and deploy Singularity containers for mobility of compute. The article also introduces Singularity Python software with novel metrics for assessing reproducibility of such containers.**
32. Walsh, D. & Podman team. Podman: A Tool for Managing OCI Containers and Pods. *Github* https://github.com/containers/podman (2023).
33. Potdar, A. M., Narayan, D. G., Kengond, S. & Mulla, M. M. Performance evaluation of Docker container and virtual machine. *Procedia Comput. Sci.* **171**, 1419–1428 (2020).
34. Gerhardt, L. et al. Shifter: containers for HPC. *J. Phys. Conf. Ser.* **898**, 082021 (2017).
35. Ram, K. Git can facilitate greater reproducibility and increased transparency in science. *Source Code Biol. Med.* **8**, 7 (2013).
36. Vuorre, M. & Curley, J. P. Curating research assets: a tutorial on the git version control system. *Adv. Methods Pract. Psychol. Sci.* **1**, 219–236 (2018).
37. Clyburne-Sherin, A., Fei, X. & Green, S. A. Computational reproducibility via containers in psychology. *Meta Psychol.* **3**, 892 (2019).
38. Boettiger, C. & Eddelbuettel, D. An introduction to rocker: Docker containers for R. *R J.* **9**, 527 (2017).
39. Nüst, D. et al. The Rockerverse: packages and applications for containerization with R. Preprint at https://doi.org/10.48550/arXiv.2001.10641 (2020).
40. Nüst, D. & Hinz, M. containerit: generating Dockerfiles for reproducible research with R. *J. Open Source Softw.* **4**, 1603 (2019).
41. Xiao, N. Liftr: Containerize R markdown documents for continuous reproducibility (CRAN, 2019).
42. Peikert, A. & Brandmaier, A. M. A reproducible data analysis workflow with R Markdown, Git, Make, and Docker. Preprint at *PsyArXiv* https://doi.org/10.31234/osf.io/8xzqy (2019).
43. Younge, A. J., Pedretti, K., Grant, R. E. & Brightwell, R. in *2017 IEEE International Conference on Cloud Computing Technology and Science* (*CloudCom*) 74–81 (2017).
44. Freire, J., Bonnet, P. & Shasha, D. in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* 593–596 (Association for Computing Machinery, 2012).
45. Papin, J. A., Mac Gabhann, F., Sauro, H. M., Nickerson, D. & Rampadarath, A. Improving reproducibility in computational biology research. *PLoS Comput. Biol.* **16**, e1007881 (2020).
46. Sochat, V. V. et al. The experiment factory: standardizing behavioral experiments. *Front. Psychol.* **7**, 610 (2016).
47. Khan, F. Z. et al. Sharing interoperable workflow provenance: a review of best practices and their practical application in CWLProv. *Gigascience* **8**, giz095 (2019).
48. Kane, S. P. & Matthias, K. *Docker: Up & Running: Shipping Reliable Containers in Production* ('O'Reilly Media, Inc., 2018).
49. Khan, A. Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Comput.* **4**, 42–48 (2017).
50. Singh, S. & Singh, N. in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology* (*iCATccT*) 804–807 (2016).
51. Singh, V. & Peddoju, S. K. in *2017 International Conference on Computing, Communication and Automation* (*ICCCA*) 847–852 (IEEE, 2017).
52. Kang, H., Le, M. & Tao, S. in *2016 IEEE International Conference on Cloud Engineering* (*IC2E*) 202–211 (IEEE, 2016).
53. Sultan, S., Ahmad, I. & Dimitriou, T. Container security: issues, challenges, and the road ahead. *IEEE Access.* **7**, 52976–52996 (2019).
54. Ruiz, C., Jeanvoine, E. & Nussbaum, L. in *Euro-Par 2015: Parallel Processing Workshops* 813–824 (Springer International Publishing, 2015).
55. Nadgowda, S., Suneja, S. & Kanso, A. in *2017 IEEE International Conference on Cloud Engineering* (*IC2E*) 266–272 (IEEE, 2017).
56. Srirama, S. N., Adhikari, M. & Paul, S. Application deployment using containers with auto-scaling for microservices in cloud environment. *J. Netw. Computer Appl.* **160**, 102629 (2020).
57. Cito, J. et al. in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories* (*MSR*) 323–333 (IEEE, 2017).
58. Poldrack, R. A. & Gorgolewski, K. J. Making Big Data open: data sharing in neuroimaging. *Nat. Neurosci.* **17**, 1510–1517 (2014).
59. Smith, S. M. & Nichols, T. E. Statistical challenges in 'Big Data' human neuroimaging. *Neuron* **97**, 263–268 (2018).
60. Tourbier, S. et al. Connectome Mapper 3: a flexible and open-source pipeline software for multiscale multimodal human connectome mapping. *J. Open Source Softw.* **7**, 4248 (2022).
61. Nichols, T. E. et al. Best practices in data analysis and sharing in neuroimaging using MRI. *Nat. Neurosci.* **20**, 299–303 (2017).
62. Halchenko, Y. O. & Hanke, M. Open is not enough. Let's take the next step: an integrated, community-driven computing platform for neuroscience. *Front. Neuroinform.* **6**, 22 (2012).
63. Schalk, G. & Mellinger, J. *A Practical Guide to Brain–Computer Interfacing with BCI2000: General-Purpose Software for Brain–Computer Interface Research, Data Acquisition, Stimulus Presentation, and Brain Monitoring* (Springer Science & Business Media, 2010).
64. Kaur, B., Dugré, M., Hanna, A. & Glatard, T. An analysis of security vulnerabilities in container images for scientific data analysis. *Gigascience* **10**, giab025 (2021).
65. Huang, Y. et al. Realized ecological forecast through an interactive Ecological Platform for Assimilating Data (EcoPAD, v1.0) into models. *Geosci. Model. Dev.* **12**, 1119–1137 (2019).
66. White, E. P. et al. Developing an automated iterative near-term forecasting system for an ecological study. *Methods Ecol. Evol.* **10**, 332–344 (2019).
67. Powers, S. M. & Hampton, S. E. Open science, reproducibility, and transparency in ecology. *Ecol. Appl.* **29**, e01822 (2019).
68. Ali, A. S., Coté, C., Heidarinejad, M. & Stephens, B. Elemental: an open-source wireless hardware and software platform for building energy and indoor environmental monitoring and control. *Sensors* **19**, 4017 (2019).

69. Morris, B. D. & White, E. P. The EcoData retriever: improving access to existing ecological data. *PLoS ONE* **8**, e65848 (2013).
70. Schulz, W. L., Durant, T. J. S., Siddon, A. J. & Torres, R. Use of application containers and workflows for genomic data analysis. *J. Pathol. Inform.* **7**, 53 (2016).
71. Di Tommaso, P. et al. The impact of Docker containers on the performance of genomic pipelines. *PeerJ* **3**, e1273 (2015).
72. O'Connor, B. D. et al. The Dockstore: enabling modular, community-focused sharing of Docker-based genomics tools and workflows. *F1000Res.* **6**, 52 (2017).
73. Bai, J. et al. BioContainers registry: searching bioinformatics and proteomics tools, packages, and containers. *J. Proteome Res.* **20**, 2056–2061 (2021).
74. Gentleman, R. C. et al. Bioconductor: open software development for computational biology and bioinformatics. *Genome Biol.* **5**, R80 (2004).
75. Zhu, T., Liang, C., Meng, Z., Guo, S. & Zhang, R. GFF3sort: a novel tool to sort GFF3 files for tabix indexing. *BMC Bioinformatics* **18**, 482 (2017).
76. Müller Paul, H., Istanto, D. D., Heldenbrand, J. & Hudson, M. E. CROPSR: an automated platform for complex genome-wide CRISPR gRNA design and validation. *BMC Bioinformatics* **23**, 74 (2022).
77. Torre, D., Lachmann, A. & Ma'ayan, A. BioJupies: automated generation of interactive notebooks for RNA-Seq data analysis in the cloud. *Cell Syst.* **7**, 556–561.e3 (2018).
78. Mahi, N. A., Najafabadi, M. F., Pilarczyk, M., Kouril, M. & Medvedovic, M. GREIN: an interactive web platform for re-analyzing GEO RNA-seq data. *Sci. Rep.* **9**, 7580 (2019).
79. Dobin, A. & Gingeras, T. R. Mapping RNA-seq reads with STAR. *Curr. Protoc. Bioinform.* **51**, 11.14.1–11.14.19 (2015).
80. Dobin, A. et al. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics* **29**, 15–21 (2013).
81. Patro, R., Duggal, G., Love, M. I., Irizarry, R. A. & Kingsford, C. Salmon provides fast and bias-aware quantification of transcript expression. *Nat. Methods* **14**, 417–419 (2017).
82. Yan, F., Powell, D. R., Curtis, D. J. & Wong, N. C. From reads to insight: a Hitchhiker's guide to ATAC-seq data analysis. *Genome Biol.* **21**, 22 (2020).
83. Garcia, M. et al. Sarek: a portable workflow for whole-genome sequencing analysis of germline and somatic variants. Preprint at *bioRxiv* https://doi.org/10.1101/316976 (2018).
84. Sirén, J. et al. Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. *Science* **374**, abg8871 (2021).
85. Zarate, S. et al. Parliament2: accurate structural variant calling at scale. *Gigascience* **9**, giaa145 (2020).
86. Morris, D., Voutsinas, S., Hambly, N. C. & Mann, R. G. Use of Docker for deployment and testing of astronomy software. *Astron. Comput.* **20**, 105–119 (2017).
87. Taghizadeh-Popp, M. et al. SciServer: a science platform for astronomy and beyond. *Astron. Comput.* **33**, 100412 (2020).
88. Herwig, F. et al. Cyberhubs: virtual research environments for astronomy. *Astrophys. J. Suppl. Ser.* **236**, 2 (2018).
89. The Astropy Collaboration. et al. The Astropy Project: building an open-science project and status of the v2.0 Core Package*. *Astron. J.* **156**, 123 (2018).
90. Robitaille, T. P. et al. Astropy: a community Python package for astronomy. *Astron. Astrophys. Suppl. Ser.* **558**, A33 (2013).
91. Abolfathi, B. et al. The fourteenth data release of the sloan digital sky survey: first spectroscopic data from the extended Baryon Oscillation Spectroscopic Survey and from the Second Phase of the Apache Point Observatory Galactic Evolution Experiment. *Astrophys. J. Suppl. Ser.* **235**, 42 (2018).
92. Nigro, C. et al. Towards open and reproducible multi-instrument analysis in gamma-ray astronomy. *Astron. Astrophys. Suppl. Ser.* **625**, A10 (2019).
93. Liu, Q., Zheng, W., Zhang, M., Wang, Y. & Yu, K. Docker-based automatic deployment for nuclear fusion experimental data archive cluster. *IEEE Trans. Plasma Sci. IEEE Nucl. Plasma Sci. Soc.* **46**, 1281–1284 (2018).
94. Meng, H. et al. An invariant framework for conducting reproducible computational science. *J. Comput. Sci.* **9**, 137–142 (2015).
95. Agostinelli, S. et al. Geant4 — a simulation toolkit. *Nucl. Instrum. Methods Phys. Res. A* **506**, 250–303 (2003).
96. Vallisneri, M., Kanner, J., Williams, R., Weinstein, A. & Stephens, B. The LIGO open science center. *J. Phys. Conf. Ser.* **610**, 012021 (2015).
97. Scott, D. & Becken, S. Adapting to climate change and climate policy: progress, problems and potentials. *J. Sustain. Tour.* **18**, 283–295 (2010).
98. Ebenhard, T. Conservation breeding as a tool for saving animal species from extinction. *Trends Ecol. Evol.* **10**, 438–443 (1995).
99. Warlenius, R., Pierce, G. & Ramasar, V. Reversing the arrow of arrears: the concept of 'ecological debt' and its value for environmental justice. *Glob. Environ. Change* **30**, 21–30 (2015).
100. Acker, J. G. & Leptoukh, G. Online analysis enhances use of NASA Earth science data. *Eos Trans. Am. Geophys. Union* **88**, 14–17 (2007).
101. Yang, C. et al. Big earth data analytics: a survey. *Big Earth Data* **3**, 83–107 (2019).
102. Wiebels, K. & Moreau, D. Leveraging containers for reproducible psychological research. *Adv. Methods Pract. Psychol. Sci.* **4**, 25152459211017853 (2021).
**This article describes the logic behind containers and the practical problems they can solve. The tutorial section walks the reader through the implementation of containerization within a research workflow, with examples using Docker and R. The article provides a worked example that includes all steps required to set up a container for a research project, which can be easily adapted and extended.**
103. Nüst, D. et al. Ten simple rules for writing Dockerfiles for reproducible data science. *PLoS Comput. Biol.* **16**, e1008316 (2020).
**This article presents a set of rules to help researchers write understandable Dockerfiles for typical data science workflows. By following these rules, researchers can create containers suitable for sharing with fellow scientists, for including in scholarly communication and for effective and sustainable personal workflows.**
104. Elmenreich, W., Moll, P., Theuermann, S. & Lux, M. Making simulation results reproducible — survey, guidelines, and examples based on Gradle and Docker. *PeerJ Comput. Sci.* **5**, e240 (2019).
105. Van Moffaert, K. & Nowé, A. Multi-objective reinforcement learning using sets of pareto dominating policies. *J. Mach. Learn. Res.* **15**, 3663–3692 (2014).
106. Gama, J., Sebastião, R. & Rodrigues, P. P. On evaluating stream learning algorithms. *Mach. Learn.* **90**, 317–346 (2013).
107. Kim, A. Y. et al. Implementing GitHub Actions continuous integration to reduce error rates in ecological data collection. *Methods Ecol. Evol.* **13**, 2572–2585 (2022).
108. Wilson, G. et al. Best practices for scientific computing. *PLoS Biol.* **12**, e1001745 (2014).
109. Eglen, S. J. et al. Toward standard practices for sharing computer code and programs in neuroscience. *Nat. Neurosci.* **20**, 770–773 (2017).
110. No authors listed. Rebooting review. *Nat. Biotechnol.* **33**, 319 (2015).
111. Kenall, A. et al. Better reporting for better research: a checklist for reproducibility. *BMC Neurosci.* **16**, 44 (2015).
112. Poldrack, R. A. The costs of reproducibility. *Neuron* **101**, 11–14 (2019).
113. Nagarajan, P., Warnell, G. & Stone, P. Deterministic implementations for reproducibility in deep reinforcement learning. Preprint at *arXiv* https://doi.org/10.48550/arXiv.1809.05676 (2018).
114. Piccolo, S. R., Ence, Z. E., Anderson, E. C., Chang, J. T. & Bild, A. H. Simplifying the development of portable, scalable, and reproducible workflows. *eLife* **10**, e71069 (2021).
115. Higgins, J., Holmes, V. & Venters, C. in *High Performance Computing* 506–513 (Springer International Publishing, 2015).
116. de Bayser, M. & Cerqueira, R. in *2017 IEEE International Conference on Cloud Engineering* (*IC2E*) 259–265 (IEEE, 2017).
117. Netto, M. A. S., Calheiros, R. N., Rodrigues, E. R., Cunha, R. L. F. & Buyya, R. HPC cloud for scientific and business applications: taxonomy, vision, and research challenges. *ACM Comput. Surv.* **51**, 1–29 (2018).
118. Azab, A. in *2017 IEEE International Conference on Cloud Engineering* (*IC2E*) 279–285 (IEEE, 2017).
119. Qasha, R., Cała, J. & Watson, P. in *2016 IEEE 12th International Conference on e-Science* (*e-Science*) 81–90 (IEEE, 2016).
120. Saha, P., Beltre, A., Uminski, P. & Govindaraju, M. in *Proceedings of the Practice and Experience on Advanced Research Computing* 1–8 (Association for Computing Machinery, 2018).
121. Abdelbaky, M., Diaz-Montes, J., Parashar, M., Unuvar, M. & Steinder, M. in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing* (*UCC*) 368–371 (IEEE, 2015).
122. Hung, L.-H., Kristiyanto, D., Lee, S. B. & Yeung, K. Y. GUIdock: using Docker containers with a common graphics user interface to address the reproducibility of research. *PLoS ONE* **11**, e0152686 (2016).
123. Salza, P. & Ferrucci, F. Speed up genetic algorithms in the cloud using software containers. *Future Gener. Comput. Syst.* **92**, 276–289 (2019).
124. Pahl, C., Brogi, A., Soldani, J. & Jamshidi, P. Cloud container technologies: a state-of-the-art review. *IEEE Trans. Cloud Comput.* **7**, 677–692 (2019).
125. Dessalk, Y. D., Nikolov, N., Matskin, M., Soylu, A. & Roman, D. in *Proceedings of the 12th International Conference on Management of Digital EcoSystems* 76–83 (Association for Computing Machinery, 2020).
126. Martín-Santana, S., Pérez-González, C. J., Colebrook, M., Roda-García, J. L. & González-Yanes, P. in *Data Science and Digital Business* (eds García Márquez, F. P. & Lev, B.) 121–146 (Springer International Publishing, 2019).
127. Jansen, C., Witt, M. & Krefting, D. in *Computational Science and Its Applications — ICCSA 2016* 303–318 (Springer International Publishing, 2016).
128. Brinckman, A. et al. Computing environments for reproducibility: capturing the 'Whole Tale'. *Future Gener. Comput. Syst.* **94**, 854–867 (2019).
129. Perkel, J. M. Make code accessible with these cloud services. *Nature* **575**, 247–248 (2019).
130. Poldrack, R. A., Gorgolewski, K. J. & Varoquaux, G. Computational and informatic advances for reproducible data analysis in neuroimaging. *Annu. Rev. Biomed. Data Sci.* **2**, 119–138 (2019).
131. Vaillancourt, P. Z., Coulter, J. E., Knepper, R. & Barker, B. in *2020 IEEE High Performance Extreme Computing Conference* (*HPEC*) 1–8 (IEEE, 2020).
132. Adufu, T., Choi, J. & Kim, Y. in *17th Asia-Pacific Network Operations and Management Symposium* (*APNOMS*) 507–510 (IEEE, 2015).
133. Cito, J., Ferme, V. & Gall, H. C. in *Web Engineering* 609–612 (Springer International Publishing, 2016).
134. Tedersoo, L. et al. Data sharing practices and data availability upon request differ across scientific disciplines. *Sci. Data* **8**, 192 (2021).
135. Tenopir, C. et al. Data sharing by scientists: practices and perceptions. *PLoS ONE* **6**, e21101 (2011).
136. Gomes, D. G. E. et al. Why don't we share data and code? Perceived barriers and benefits to public archiving practices. *Proc. Biol. Sci.* **289**, 20221113 (2022).
137. Weston, S. J., Ritchie, S. J., Rohrer, J. M. & Przybylski, A. K. Recommendations for increasing the transparency of analysis of preexisting data sets. *Adv. Methods Pract. Psychol. Sci.* **2**, 214–227 (2019).

## Acknowledgements
D.M. and K.W. are supported by a Marsden grant from the Royal Society of New Zealand and a University of Auckland Early Career Research Excellence Award awarded to D.M.

## Author contributions
Introduction (D.M., K.W. and C.B.); Experimentation (D.M., K.W. and C.B.); Results (D.M., K.W. and C.B.); Applications (D.M., K.W. and C.B.); Reproducibility and data deposition (D.M., K.W. and C.B.); Limitations and optimizations (D.M., K.W. and C.B.); Outlook (D.M., K.W. and C.B.).

## Competing interests
The authors declare no competing interests.

## Additional information
**Supplementary information** The online version contains supplementary material available at https://doi.org/10.1038/s43586-023-00236-9.

**Peer review information** *Nature Reviews Methods Primers* thanks Beth Ciimini, Stephen Piccolo and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

## Related links
**ACM Digital Library:** https://dl.acm.org
**Amazon Web Services:** https://aws.amazon.com
**Ansible:** https://ansible.com
**Astropy:** https://astropy.org
**ATAC-seq Pipeline:** https://github.com/ENCODE-DCC/atac-seq-pipeline
**BCI2000 project:** https://bci2000.org/
**Binder:** https://mybinder.org/
**Bioconductor:** https://bioconductor.org
**BioContainers:** https://biocontainers.pro
**Bismark:** https://www.bioinformatics.babraham.ac.uk/projects/bismark/
**Breakdancer:** https://github.com/genome/breakdancer
**CERN Container Registry:** https://hub.docker.com/u/cern
**Chef:** https://chef.io
**CodeOcean:** http://codeocean.com
**Containerd:** https://containerd.io/
**Docker Hub:** https://hub.docker.com
**EarthData:** https://earthdata.nasa.gov
**EcoData Retriever:** https://ecodataretriever.org
**Ecological Niche Modelling on Docker:** https://github.com/ghuertaramos/ENMOD
**Ecopath:** https://ecopath.org/
**EIGENSOFT:** https://hsph.harvard.edu/alkes-price/software/eigensoft
**Environmental Data Commons:** https://edc.occ-data.org
**Experiment Factory:** https://expfactory.github.io
**F1000Research guidelines:** https://f1000research.com/for-authors/article-guidelines/software-tool-articles
**fmriprep:** https://fmriprep.org
**FSL project:** https://fsl.fmrib.ox.ac.uk/fsl/fslwiki
**GATK:** https://gatk.broadinstitute.org
**gdb:** https://github.com/haggaie/docker-gdb
**GEANT4:** https://geant4.web.cern.ch
**GeoServer:** https://geoserver.org
**GitHub Actions:** https://github.com/features/actions
**GitHub Container Registry:** https://github.com/features/packages
**Google Cloud Platform:** https://cloud.google.com
**GRASS GIS:** https://grass.osgeo.org
**Jenkins:** https://jenkins.io
**liftr:** https://liftr.me/
**LIGO Open Science Centre:** https://losc.ligo.org
**LXC:** https://linuxcontainers.org
**Marble Station:** https://github.com/marblestation/docker-astro
**Mesos:** https://mesos.apache.org
**NEST:** https://nest-simulator.org
**NeuroDebian:** https://neuro.debian.net
**NEURON:** https://neuron.yale.edu/neuron
**OpenShift:** https://openshift.com/
**Planet Research Data Commons:** https://ardc.edu.au/program/planet-research-data-commons
**Podman:** https://podman.io/
**Puppet:** https://puppet.com
**QGIS:** https://qgis.org
**Quay:** https://quay.io
**Rocker project:** https://rocker-project.org/
**Rocket:** https://github.com/rkt/rkt
**Salmon:** https://combine-lab.github.io/salmon
**SciServer:** https://sciserver.org
**Singularity:** https://sylabs.io/
**STAR:** https://github.com/alexdobin/STAR
**strace:** https://github.com/amrabed/strace-docker
**SVTyper:** https://github.com/hall-lab/svtyper