Bulker: a multi-container environment manager

Nathan C. Sheffield^{1-4,✉}

¹Center for Public Health Genomics, University of Virginia

²Department of Public Health Sciences, University of Virginia

³Department of Biomedical Engineering, University of Virginia

⁴Department of Biochemistry and Molecular Genetics, University of Virginia

✉ Correspondence: nsheffield@virginia.edu

Reproducible computing environments are required for reproducible analysis and are also useful for interactive exploratory analysis. In the past, scientific computing environments have been managed with package managers or with virtual machines. More recently, modern workflow managers have incorporated linux container technology to improve reproducibility. These existing solutions solve some of the challenges of managing reproducible computing environments, but they remain limited: System-wide environments and native package managers lack the portability, efficiency, and robustness of modern containers, while container-aware workflows are specialized and incapable of interactive computing. Here, I introduce *bulker*, an approach that combines the advantages of virtual machines, native package managers, and container-aware workflow managers. Bulker creates and distributes complete environments, like virtual machines, but with re-usable modular components, like a native package manager. It uses individually containerized tools like a container-aware workflow manager, but also allows these environments to be activated interactively and distributed independently of a particular workflow. Bulker is thus a more general approach to portable and reproducible computing environments.

Availability: <https://bulker.io>

Background

Recent studies have raised concerns over poor reproducibility in scientific research, leading to dozens of articles outlining its causes and proposing solutions¹⁻¹⁰. One critical aspect of scientific reproducibility is *computational reproducibility*, which is becoming increasingly important as data availability increases and research becomes more computational. Achieving computational reproducibility has many facets; it starts with making sure both code and data are FAIR (findable, accessible, interoperable, reproducible)¹¹⁻¹⁷; but even a scripted analysis on available data will be affected by the software versions installed, so the *computing environment itself* must also be reproducible^{11,18-21}.

Creating reproducible computing environments is particularly challenging in computational biology, where the breadth of analysis requires many tools written in many languages and spread across many package sources. This has driven effort to develop a uniform solution to software environment control. One strategy is a self-contained, *system-wide environment* like a virtual machine²², Biolinux²³, or Debian Med²⁴, but

this approach lacks the flexibility necessary for widely variable research applications. A more flexible and popular approach is a *modular tool environment*²⁵⁻²⁷, epitomized by the venerable Environment Modules system²⁶ and the *conda* package manager²⁷, which can be tailored to biological research by *Bioconda*²⁸; but these approaches require installing software natively into a host system.

While virtual machines and native package managers have advantages, the landscape of scientific computing has been disrupted due to the recent debut of Docker containers²⁹. Containers are lightweight and offer the advantages of both virtualization and modularity. Containers have enormous potential and have quickly become a *de facto* standard for software development and deployment in the tech industry. This enthusiasm also extends to biology, with dozens of studies using containers or hailing their potential to improve reproducibility³⁰⁻⁴⁷. Most modern workflow languages and systems now allow developers to specify containers for individual commands, improving the reproducibility of these workflows^{25,48-53}.

Fig. 1: Bulker crates add a new organization layer to container management. A: Native workflows are run directly in the host operating system. B: A monolithic image contains many tools, and workflows can be run inside the container as if they are native. C: With modular images, workflow commands are executed from the host environment by specifying an individual image with, for example, `docker run...` D: Bulker adds a new organizational layer called a crate. Crates point to modular images, but allow the user to interact in the simpler interface of the monolithic image. E: The user manage individual tools in the native approach; one large image in the monolithic image approach; each individual image in the modular image approach; but only a single crate in the bulker approach.

However, existing modular container approaches restrict the environment to a particular use case; they cannot distribute and re-use a multi-tool environment for other purposes. Reproducible computing environments are not only useful for workflows; they are also convenient for exploratory interactive analysis, everyday computing, and collaborative software development. This type of environment-level portability is provided by system-wide environment approaches or by monolithic container images, but this strategy does not allow efficient re-use of individual components. There is no existing tool that provides reproducible, interactive environments built from individual, modular container images.

Here, I present *bulker*, a multi-container environment manager. Bulker environments provide access to multiple tools like a system-wide environment, but keep each command in an individual container. Bulker is compatible with any existing workflow management system, and adds the advantage of interactive and portable environments that are decoupled from the workflow. Bulker simplifies both interactive analysis and workflow development by building drop-in replacements to command-line tools that act like native tools, but run in containers.

Bulker addresses a growing need in scientific computing by building and distributing version-controlled, modular, containerized computing environments.

Results and discussion

Bulker crates unify modular containers

In the simplest case, workflows may be run natively within a host environment (Fig. 1A). However, to improve reproducibility and robustness, modern workflows are increasingly using containers instead of native host processing. There are two common approaches to using containers for scientific computing: 1) build a single, *monolithic image* with all required tools, and run an analysis *inside* that container (Fig. 1B); or 2) specify a *modular image* for each command, then run an analysis in the host environment, executing each command in its own container (Fig. 1C). Each approach has strengths and weaknesses.

In the ‘monolithic image’ approach, the workflow itself runs native commands, concentrating the “container awareness” overhead into the single mega-container, which makes it easier to write and maintain the workflow. This also allows the environment image to be distributed and used separately from the workflow. However, the image components are not modular and therefore cannot be easily applied to other tasks with related requirements, leading to inefficient disk use and increased effort to maintain the environment.

In the ‘modular image’ approach, the images can be easily re-used within and across workflows, which solves the efficiency problems. However, the workflow developer is now required to make each command individually container-aware, which takes effort and may make the workflow dependent on a particular container system, reducing its portability. More significantly, each command must individually specify host-specific and

Fig. 2: Bulker automatically creates containerized executables. A: A containerized executable has a simple user interface, but runs the command inside a container. B: An example containerized executable for the `samtools` command. This script includes 3 types of information: engine-specific, tool-specific, and host-specific. C: Bulker decouples tool-specific settings from environment-specific settings, making the manifest (mapping of tools to images) portable across computing environments. Bulker integrates these types of information to produce a set of auto-generated containerized executables, or a crate.

container-engine-specific settings, so the workflow may need to be tweaked to change compute hosts. Furthermore, because the environment configuration is tightly coupled to the specific workflow, it is nontrivial to use the *entire* environment interactively or for a different purpose.

Bulker takes a hybrid approach by adding an additional organizational layer, the *crate*, which includes a collection of individual tools (Fig. 1D). Like the monolithic image approach, a bulker crate is a single, unified environment that includes multiple commands that run like native commands. This environment is created, distributed, and activated as a unit, so complete environments can be easily shared, re-used, and deployed interactively. But like the modular image approach, bulker crates are not made from single image, but from individual images for each command. Together, bulker crates provide the benefits of using modular images, while maintaining the simplified user interface provided by a monolithic image (Fig. 1E). Bulker does this by creating environments built out of what I call *containerized executables*.

Containerized executables create an environment

A *containerized executable* is an executable file that runs a command in a container (Fig. 2A). Essentially, it wraps container and host settings to simplify the user interface to the tool. For example, a containerized executable for `samtools` wraps the `samtools` command in code that runs it in a container (Fig. 2B). If this executable file is named `samtools`, it can then be invoked by calling `samtools` on the command line as usual, thus behaving like a native tool.

Containerized executables allow users to interactively run individual commands as if they were installed natively, but they actually run in individual containers.

When a set of these containerized executables are added to a user's `PATH`, an interactive shell behaves as if a user is in the host environment, but commands are individually run in containers. From within such a shell, a workflow of native commands would in fact run each command in an individual container without any modification to the workflow. Thus, sets of containerized executables have multiple uses: they can be used to create interactive environments for everyday computing, and can also be used to run a native workflows in containers. This only requires that the host system be equipped with the necessary set of images and executables in the `PATH`.

Bulker makes containerized executables portable

Containerized executables are simple and powerful because they build on the universal interface of native command-line tools. Environments built from containerized executables are generalizable and efficient, and it would be convenient to share them. However, one critical challenge has prevented this: unlike images themselves, containerized executables are *not portable*. Some settings in a containerized executable depend on the host system; for example, vanilla containers are isolated from the host file system, so a container that requires input data from the host must explicitly mount any required file systems, which varies by host. In this `samtools` example, the executable mounts file systems at `/tmp` and `/ext` and loads a few environment variables (orange lines in Fig. 2B). This means that while an individual image can easily move from one host to another, a containerized executable cannot.

To make these environments portable will require a way to isolate the portable components from the non-portable ones. A containerized executable is really

made up of information of 3 types: 1) information pertaining to the container engine (in this case, the syntax for `docker run`); 2) information about the host, like volumes; and 3) information about the tool, like the image and command to run (Fig. 2B). A computing environment with containerized executables is therefore specific to both a host environment *and* a container engine, but the third component (tool-specific information) is portable. Given that the tool-specific information is in some sense the most important part of the environment, this opens the possibility of making portable containerized executables.

Bulker approaches this problem by decoupling the information sources, and then re-combining them on the fly to create executables (Fig. 2C). Tool-specific settings are stored in a file that describes commands and their images; these files, called *manifests*, are the portable representation of the bulker environment. Host-specific settings are stored in a host configuration file configured locally for each host. Finally, the engine-specific information is stored in a template shipped with bulker. This standardizes the computing environment across host systems and container engines, so the same manifest may be deployed on systems using either docker or singularity. Bulker unites these files to produce executables tailored to the host and its container engine. In a sense, bulker makes containerized executables portable.

Bulker distributes tools in sets

On a configured host, bulker requires only a mapping of commands to images specified by a *manifest* file. When the manifest is merged with the host settings, the resulting set of containerized executables is a *crate*, bulker's fundamental operating unit. This approach provides several benefits over existing approaches.

First, unlike existing container engines, bulker's atomic unit of distribution is a set of images rather than a single image. Thus, a single command like `load toolset` or `activate toolset` prepares a complete environment – but, critically, made from individual images.

Second, this simplifies creating environments for workflows because the host only needs to be configured once. After the host is configured with bulker, given an appropriate manifest, any workflow built with any workflow system is now containerized. In contrast, using workflow-based containerization, the host configuration must be specified for each workflow or workflow system.

Third, because crates are sets of containerized executables that simply *point* to images, they are very lightweight (Fig. 3). A crate doesn't actually contain any images, which are managed entirely by the host

Fig. 3: **Bulker crates are lightweight.** A: Monolithic images must duplicate tools. B: Crates contain only pointers to images, allowing individual images to be re-used among multiple crates.

Fig. 4: **Bulker operates on sets of images.** The docker workflow progresses from Dockerfile to image to container. Analogously but on collections of containers rather than individual containers, Bulker manifests produce crates which run environments.

container engine. This means individual tools can reside in multiple crates without increasing disk use.

The relationship between a bulker manifest and a bulker crate is most easily understood by analogy to docker (Fig. 4). In docker, a *Dockerfile* builds an *image*, which is run to instantiate a *container*. Analogously, a bulker *manifest* can be loaded into a *crate*, which can be activated to create an *environment*.

Manifests portably map commands to images

The bulker *manifest* is the portable representation of the environment. It contains a mapping of commands to images, and any other arguments required to run the command in a container that are specific to the tool. Because they are merely text configuration files, distributing manifests is quick and easy. It's also easy to create your own manifest. Once you've created one, you can load that manifest on any host that has bulker configured, automatically creating a computing environment with host-specific containerized executables.

Here's an example of a demo manifest with two commands: `cowsay` and `fortune`:


```
manifest:
  name: demo
  commands:
  - command: cowsay
    docker_image: nsheff/cowsay
    docker_command: cowsay
    docker_args: "-i"
  - command: fortune
    docker_image: nsheff/fortune
    docker_command: fortune
```

The `command` attribute is the string that will become executable in your host environment, once this manifest is activated. The `docker_image` specifies a registry path to an image on some docker registry, and the `docker_command` is the command that will be executed within the container, which is often but not necessarily identical to the `command` attribute. Finally, the `docker_args` specifies any specific arguments that must be passed to docker for this particular command. Most commonly, I use `docker_args` to add `-i` or `-t` for commands that require pipe interfaces. Notice that the manifest file contains *all* information relative to specific tools and their images, but *no* information relative to a specific host.

Bulker's manifests are managed via a command-line interface (CLI).

The bulker CLI loads crates

The primary workhorse of bulker is the command-line interface (CLI), which is what users will use to interact with their computing environments. Its implementation as a command-line tool makes it useful not only for discovery and interactive management of environments, but also allows it to be integrated into existing workflows that require setting up a reproducible computing environment. The bulker CLI can be installed with `pip install bulker` and invoked by calling `bulker` on the command line (Fig.5). The bulker CLI currently provides five functions for interacting with computing environments:

- `bulker init` initializes an empty bulker host configuration file
- `bulker load` adds a new crate from a file
- `bulker list` summarizes the available local crates
- `bulker activate` loads a new shell, now with commands from the activated crate
- `bulker run` runs a single command in an environment activated with specified crate

The `init` and `list` functions essentially provide a basic CLI to conveniently create or explore the host configuration. The `load` function is the critical task of

```
version: 0.2.0
usage: bulker [-h] [-V] [--silent] [--verbosity V]
           {run,load,init,activate,list} ...

bulker - manage containerized executables

positional arguments:
  {run,load,init,activate,list}
    run                Run a command in a crate
    load               Load a crate from manifest
    init               Initialize a config file
    activate            Activate a crate
    list               List available crates
```

Fig. 5: **Bulker command-line usage.** Bulker is implemented as a command-line tool that allows users load and interact with bulker crates.

bulker, which merges the manifest tool settings with the host settings to create a folder of containerized executables. To load a manifest, a user must provide the manifest file as well as an identifier for the crate. Crate identifiers follow the same pattern as a docker image and are referred to with a namespace, name, and tag: `namespace/name:tag`. For example, a user could load a manifest with this command:

```
bulker load nsheff/mycrate:0.2.4 \
  -f mycrate_0.2.4.yaml
```

The bulker registry shares remote manifests

In addition to using a local file, the CLI `load` command can also load a manifest from a server. A remote server that hosts bulker manifests is known as a bulker *registry*. Analogous to pulling a docker image, users can load a manifest from a registry by specifying the namespace, manifest name, and tag: `bulker load bulker/demo:default`. (Tags are optional, and will default to the 'default' tag if not provided). This will automatically download the specified manifest from the bulker registry and load it locally. The bulker registry URL is configured in the bulker configuration file, which is by default set to <http://hub.bulker.io>. The registry itself requires only a basic file server, and the default registry is backed by a GitHub repository and hosted via GitHub Pages. Users can easily contribute a new manifest via a pull request. Alternatively, users may easily host their own registries either using GitHub or any other file server.

Activating crates adds commands to your PATH

Once loaded, a crate shows up in the bulker list menu and is now available for activation. The `activate` and `run` functions are what instantiates the environment, making the containerized executables available for use. The `activate` command returns an interactive shell, while the `run` command creates a

shell, runs a command, and then exits. These functions use the same identifiers as the load command, so we could activate the example crate with `bulker activate nsheff/mycrate:0.2.4`. The activate and run commands are quite simple: to construct the computing environment, they simply prepend the appropriate folder to the user's PATH variable, providing a shell execution environment in which all the manifest commands are available for direct execution. By prepending to the PATH variable, these commands have the effect of *overlaying* the commands on the current environment. In other words, any commands included in the crate will mask existing commands, but any unmasked host commands will be also be available. This feature is desirable in some cases, such as interactive analysis, but may be problematic if strict reproducibility is required. In this case, bulker also provides a `--strict` flag that provides an environment in which no host commands are directly accessible, so that only commands included in the crate will be available. This is accomplished by overwriting rather than prepending the PATH with a crate path.

Bulker can activate multiple crates simultaneously

The way bulker activates crates by overlaying commands using the PATH variable makes it very flexible. One benefit is merging multiple crates. By simply activating crates in a priority order, it is simple and efficient to create computing environments that are made up of multiple crates. Bulker's CLI accomplishes this by allowing multiple comma-separated crate names in the activate and run commands. Thus, `bulker activate namespace/crate1,namespace/crate2` will first activate `crate1`, prepending its commands to the PATH, and then activate `crate2`. `Crate2` thus has a higher priority and will mask any commands provided by `crate1`, which in turns masks host commands. This creates a flexible, modular system that allows easy combining of existing environments, which will encourage re-use of collections of related tools.

Bulker is an environment manager

Bulker is a new type of tool, a *multi-container environment manager*. It shares some features with native workflow managers, package managers, and container engines, but it provides novel features that are not currently provided by any of these existing systems:

Workflow managers

Bulker is not a workflow manager, but it does overlap with them in some features (Fig. 6). Like bulker, modern workflow managers accommodate individual containers for each command. The difference is that workflow management systems integrate container management with the workflow; in contrast, bulker decouples

	Multi-tool environment distributed as atomic unit	Run individual commands in individual containers	Capable of interactive analysis	Multi-tool environment portable across workflows	Efficient re-use of tools as modular components	Processes are isolated from host environment	Handles distribution of environments	Installs tools individually	Manages workflows
Native package manager	✗	✗	✓	✓	✓	✗	✗	✓	✗
Container-aware workflow manager	✗	✓	✗	✗	✓	✓	✗	✗	✓
Virtual machine	✓	✗	✓	✓	✗	✓	✗	✗	✗
Container engine monolithic image	✓	✗	✓	✓	✗	✓	✗	✗	✗
Container engine modular images	✗	✓	✗	✗	✓	✓	✗	✗	✗
🔧 Bulker	✓	✓	✓	✓	✓	✓	✓	✗	✗

Fig. 6: **Bulker offers a unique set of features.** Bulker's feature set partly overlaps with package managers (e.g. `conda`), workflow systems (e.g. `Nextflow`), and vanilla container engines (e.g. `singularity`), but the total feature set cannot be accomplished by any one system, or even by a combination of existing systems.

the two: bulker only manages environments. The decoupled approach taken by bulker offers several advantages:

First, with bulker, the workflow needs not be container-aware. In contrast, a container-aware workflow must include some container configuration, such as the mapping of commands to images, the host file systems, etc. With bulker, these settings are configured globally, bulker handles the containerization, and workflows simply run each command natively. This simplifies workflow development. Second, for a workflow manager, the container information is tied to a particular workflow, which means it cannot be re-used in a different setting. In contrast, bulker modular multi-tool environments can be re-used across workflows and even across workflow systems. Furthermore, a bulker environment can be activated independent of a workflow entirely for interactive analysis, making bulker useful not just for workflows, but for everyday computing tasks and exploratory analysis. Finally, because bulker sits atop the container engine, it unifies the user interface regardless of the container engine used (e.g. `docker` or `singularity`).

In short, bulker transcends an individual workflow or even workflow manager. With bulker, there is no longer any need for workflow managers to accommodate containers individually, which will allow workflow systems to focus effort on the task of workflow development. By decoupling the environment management from the workflow management, bulker solves the environment problem for all possible workflow managers.

Fig. 7: *Bulker wraps existing container engines. Bulker is not a container engine; it sits on top of existing container engines, providing a unified interface and distributing sets of individual images as a unit.*

Package managers

Bulker features also partly overlap those of traditional package managers. For example, instead of installing software natively using pip or conda, you could load it using bulker. There are two key distinctions:

First, package managers install tools natively, but bulker downloads container images and produces containerized executables. Containers have substantial advantages over native installation⁴⁰: *portability*, as tools are built only once, but can run anywhere; *uniformity*, because a single engine can manage tools from any language, as well as arbitrary data; *robustness*, because processes are isolated in containers; *provenance*, because individual containers can be version-controlled and checksummed; and *standardized distribution* through the container registry protocol.

Second, bulker is ‘collection first’. Most package managers operate on tools (e.g. pip install tool, conda install tool, or apt install tool). Bulker operates on collections: bulker load toolset. This makes bulker suited to tasks that require multiple tools, such as a scientific workflow, or even just a user’s everyday computing environment.

With these two features, bulker makes it easier than a traditional package manager to distribute and use a standard, version-controlled computing environment. Nevertheless, bulker does not supplant native package manager installers entirely, but lives side-by-side.

Container engines

Bulker is also not a container engine. In fact, bulker relies extensively on existing container engines like docker or singularity. Bulker *crates* are a simple layer on top of container engines to allow distributing

multi-container sets and easily wrapping them into native-like tools. One of the benefits of bulker is that it abstracts away the specifics of the container engine to provide a unified user interface that spans container engines (Fig. 7). Bulker users need not be particularly aware of the specifics of the underlying container engine to utilize the power of containers.

Conclusion

Bulker is a new approach to environment management. By decoupling the sources of information in containerized executables, bulker builds portable re-usable environments from individual container image components. Bulker’s utility is broad: it solves the reproducibility problem generally for any type of workflow or workflow system, from a container-blind bash script to a state-of-the-art CWL pipeline. Bulker environments also transcend workflow reproducibility, adding utility outside the workflows for which they are developed by making environments independently distributable and interactive. Interactivity enables users to debug workflows, or use environments for other non-workflow tasks, such as writing manuscripts with pandoc or using git for version control. Software typically installed via package managers can instead be managed by bulker, increasing portability and robustness. As users create bulker environments, the community will build up a resource of environments that can also be re-used, improving the reproducibility and portability of scientific computing.

License & availability

Bulker is a BSD2-licensed Python package. Source code and documentation can be found at bulker.io.

References

- Ioannidis, J. P. A. Why most published research findings are false. *PLoS Medicine* **2**, e124 (2005).
- Pashler, H. & Wagenmakers, E. Editors’ introduction to the special section on replicability in psychological science. *Perspectives on Psychological Science* **7**, 528–530 (2012).
- Schooler, J. W. Metascience could rescue the ‘replication crisis’. *Nature* **515**, 9–9 (2014).
- Baker, M. Reproducibility crisis: Blame it on the antibodies. *Nature* **521**, 274–276 (2015).
- Baker, M. 1,500 scientists lift the lid on reproducibility. *Nature* **533**, 452–454 (2016).
- Moonesinghe, R., Khoury, M. J. & Janssens, A. C. J. W. Most published research findings are false But a little replication goes a long way. *PLoS Medicine* **4**, e28 (2007).
- Ioannidis, J. P. A. Why most clinical research is not useful. *PLOS Medicine* **13**, e1002049 (2016).
- Munafo, M. R. & Smith, G. D. Robust research needs many lines of evidence. *Nature* **553**, 399–401 (2018).

9. Nosek, B. A. *et al.* Promoting an open research culture. *Science* **348**, 1422–1425 (2015).
10. Ball, P. High-profile journals put to reproducibility test. *Nature* (2018). doi:[10.1038/d41586-018-06075-z](https://doi.org/10.1038/d41586-018-06075-z)
11. Wilkinson, M. D. *et al.* The FAIR guiding principles for scientific data management and stewardship. *Sci. Data* **3**, 160018 (2016).
12. Reich, M. *et al.* GenePattern 2.0. *Nat. Genet.* **38**, 500–501 (2006).
13. Giardine, B. Galaxy: A platform for interactive large-scale genome analysis. *Genome Research* **15**, 1451–1455 (2005).
14. Leisch, F. in *Compstat* 575–580 (Physica-Verlag HD, 2002). doi:[10.1007/978-3-642-57489-4_89](https://doi.org/10.1007/978-3-642-57489-4_89)
15. Baumer, B., Cetinkaya-Rundel, M., Bray, A., Loi, L. & Horton, N. J. R markdown: Integrating a reproducible analysis tool into introductory statistics. *arXiv preprint arXiv:1402.1894* (2014).
16. Huber, W. *et al.* Orchestrating high-throughput genomic analysis with bioconductor. *Nature Methods* **12**, 115–121 (2015).
17. Kluyver, T. *et al.* Jupyter notebooks—a publishing format for reproducible computational workflows. in *ELPUB* 87–90 (2016).
18. Krumholz, H. M. & Waldstreicher, J. The yale open data access (YODA) project—a mechanism for data sharing. *The New England journal of medicine* **375**, 403–405 (2016).
19. Devereaux, P. J., Guyatt, G., Gerstein, H., Connolly, S. & Yusuf, S. Toward fairness in data sharing. *The New England journal of medicine* **375**, 405–407 (2016).
20. Warren, E. Strengthening research through data sharing. *The New England journal of medicine* **375**, 401–403 (2016).
21. Bui, A. A. T., Van Horn, J. D. & Consortium, N. B. C. Envisioning the future of 'big data' biomedicine. *J. Biomed. Inform.* **69**, 115–117 (2017).
22. Maciejewski, M. W. *et al.* NMRbox: A resource for biomolecular NMR computation. *Biophysical Journal* **112**, 1529–1534 (2017).
23. Krampis, K. *et al.* Cloud BioLinux: Pre-configured and on-demand bioinformatics computing for the genomics community. *BMC Bioinformatics* **13**, (2012).
24. Möller, S. *et al.* Community-driven computational biology with debian linux. *BMC Bioinformatics* **11**, (2010).
25. Courtès, L. Functional package management with guix. *CoRR abs/1305.4584*, (2013).
26. Furlani, J. L. Modules: Providing a flexible user environment. in *Proceedings of the fifth large installation systems administration conference (lisa v)* 141–152 (1991).
27. Anaconda software distribution. Computer software. Vers. 2-2.4.0. (2016).
28. Grüning, B. *et al.* Bioconda: Sustainable and comprehensive software distribution for the life sciences. *Nature Methods* **15**, 475–476 (2018).
29. Merkel, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal* **2014**, 2 (2014).
30. Belmann, P. *et al.* Bioboxes: Standardised containers for interchangeable bioinformatics software. *GigaScience* **4**, (2015).
31. Almugbel, R. *et al.* Reproducible bioconductor workflows using browser-based interactive notebooks and containers. *Journal of the American Medical Informatics Association* **25**, 4–12 (2017).
32. Veiga Leprevost, F. da *et al.* BioContainers: An open-source and community-driven framework for software standardization. *Bioinformatics* **33**, 2580–2582 (2017).
33. Boettiger, C. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review* **49**, 71–79 (2015).
34. Silver, A. Software simplified. *Nature* **546**, 173–174 (2017).
35. 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).
36. Moreews, F. *et al.* BioShaDock: A community driven bioinformatics shared docker-based tools registry. *F1000Research* (2015). doi:[10.12688/f1000research.7536.1](https://doi.org/10.12688/f1000research.7536.1)
37. Beaulieu-Jones, B. K. & Greene, C. S. Reproducibility of computational workflows is automated using continuous analysis. *Nature Biotechnology* **35**, 342–346 (2017).
38. Matelsky, J. *et al.* Container-based clinical solutions for portable and reproducible image analysis. *Journal of Digital Imaging* **31**, 315–320 (2018).
39. 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).
40. Strozzi, F. *et al.* in *Methods in molecular biology* 723–745 (Springer New York, 2019). doi:[10.1007/978-1-4939-9074-0_24](https://doi.org/10.1007/978-1-4939-9074-0_24)
41. Kulkarni, N. *et al.* Reproducible bioinformatics project: A community for reproducible bioinformatics analysis pipelines. *BMC Bioinformatics* **19**, (2018).
42. Novella, J. A. *et al.* Container-based bioinformatics with pachyderm. *Bioinformatics* **35**, 839–846 (2018).
43. Kim, Y.-M., Poline, J.-B. & Dumas, G. Experimenting with reproducibility: A case study of robustness in bioinformatics. *GigaScience* **7**, (2018).
44. Menegidio, F. B., Jabes, D. L., Oliveira, R. C. de & Nunes, L. R. Dugong: A docker image, based on ubuntu linux, focused on reproducibility and replicability for bioinformatics analyses. *Bioinformatics* **34**, 514–515 (2017).
45. Kim, B., Ali, T., Lijeron, C., Afgan, E. & Krampis, K. Bio-docklets: Virtualization containers for single-step execution of NGS pipelines. *GigaScience* **6**, (2017).
46. Hosny, A., Vera-Licona, P., Laubenbacher, R. & Favre, T. AlgoRun: A docker-based packaging system for platform-agnostic implemented algorithms. *Bioinformatics* **32**, 2396–2398 (2016).
47. Wurmus, R. *et al.* PiGx: Reproducible genomics analysis pipelines with GNU guix. *GigaScience* **7**, (2018).
48. Chapman, B. Bcbio. (2017).
49. Goodstadt, L. Ruffus: A lightweight python library for computational pipelines. *Bioinformatics* **26**, 2778–2779 (2010).
50. Ovaska, K. *et al.* Large-scale data integration framework provides a comprehensive view on glioblastoma multiforme. *Genome Med.* **2**, 65 (2010).
51. Amstutz, P. *et al.* Common workflow language, v1.0. (2016). doi:[10.6084/m9.figshare.3115156.v2](https://doi.org/10.6084/m9.figshare.3115156.v2)
52. Di Tommaso, P. *et al.* Nextflow enables reproducible computational workflows. *Nat. Biotechnol.* **35**, 316–319 (2017).
53. Köster, J. & Rahmann, S. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics* **28**, 2520–2522 (2012).