Containerization for creating reusable model code

Manuela Vanegas Ferro[1]*, Allen Lee[1], Calvin Pritchard[1,2], C. Michael Barton[1,3], and Marco A. Janssen[1,4]

[1] School of Complex Adaptive Systems, Arizona State University, USA
[2] Inuvialuit Regional Corporation, Canada
[3] School of Human Evolution and Social Change, Arizona State University, USA
[4] School of Sustainability, Arizona State University, USA

Abstract

Will you be able to run your computational models in the future? Even with well-documented code, this can be difficult due to changes in the software frameworks and operating systems that your code was built on. In this paper we discuss the use of containers to preserve code and their software dependencies to reproduce simulation results in the future. Containers are standalone lightweight packages of the original model software and their dependencies that can be run independent of the platform. As such they are suitable for reuse and sharing results. However, the use of containers is rare in the field of modeling social-environmental systems. We provide an introduction to the basic principles of containerization, argue why it would be beneficial if this tool became common practice in the field, describe a conceptual walkthrough to the process of containerizing a model, and reflect on near future directions of containerization workflows.

Keywords

computational models; containers; research software; reproducibility; model portability

1. Introduction

Simulations of social-environmental systems help us explore the consequences of alternative assumptions of social and environmental processes in systematic and rigorous ways. Moreover, because computer simulations are based on explicit algorithms, their results should be more repeatable than natural language narratives about processes, even if non-deterministic or emergent chaotic behavior impose certain constraints (McPhillips et al., 2019). Advancing scientific understanding of societal and environmental phenomena depends on the ability to build on the existing body of work. Reusable models can aid in the development of new models as authors often borrow useful components from existing models (Bell et al., 2015; Grimm et al., 2020). In these cases, modelers would benefit from being able to run the component of interest in a controlled environment where the expected results were previously defined. The ability to easily reuse models also facilitates coupling models, combining their inputs and outputs to explore feedbacks in novel ways. As such, successful replications of previously reported simulation results and comparisons between algorithms to represent processes in different models are critical to the advancement of the field. Such knowledge scaffolding requires FAIR (Findable, Accessible, Interoperable, and Reusable) models (Wilkinson et al., 2016) that can be readily discovered, shared, and executed to produce the expected results.

Correspondence:
Contact M. Vanegas Ferro at Manuela.VanegasF@asu.edu

Cite this article as:
Vanegas Ferro, M., Lee, A., Pritchard, C., Barton, C.M., Janssen, M.A.
Containerization for creating reusable model code
Socio-Environmental Systems Modelling, vol. 3, 18074, 2022, doi:10.18174/sesmo.18074

Socio-Environmental Systems Modelling
An Open-Access Scholarly Journal
http://www.sesmo.org

However, FAIR models have not been the norm and reusability is rarely found in published models (Boettiger, 2015). This is at least in part because the field of modeling social-environmental systems emerged comparatively recently, with ad hoc contributions by scholars that often had no formal training in computer science or software engineering best practices. We have experienced these issues firsthand. One of the authors published an article on model-based research more than 20 years ago accompanied by the model software, taking advantage of the (then novel) potential of a web-based journal to disseminate both simulation results and the model code that produced them (Janssen & Jager, 1999). However, due to changes in operating systems and software versions, the provided executable (the source code was not included) no longer runs. The model-based research is not reproducible 20 years after publication. This example underscores why it is important that the field of social-environmental simulation adopts best practices for reproducibility from computer science and software engineering that have been successfully applied in other disciplines like bioinformatics (Jalili et al., 2020) and machine learning (Pineau et al., 2020).

Nevertheless, producing reusable models is not straightforward. Reliance on the algorithm's documentation presented in a publication has been proven to be insufficient to replicate the model's results due to the gap between an algorithm's description and the numerous ways in which it can be implemented (Meadows & Cliff, 2012; Polhill et al., 2005; Wilensky & Rand, 2007). This ambiguity has been found even when standardized documentation is used (Amouroux et al., 2010; Grimm et al., 2020). Noting that publishing model code facilitates the replication of its results, CoMSES Net developed a public repository where model source code and documentation can be archived and easily accessed (Rollins et al., 2014). The CoMSES Net Computational Model Library (https://www.comses.net/codebases/) is designed to provide long-term preservation of published models and assigns permanent URLs to uploaded models and DOIs to models that have passed peer review. These permanent identifiers should be cited in the corresponding article, so the specific version of the model used to generate published results is findable and accessible to the reader in accordance with the FAIR principles for scientific data management.

Efforts like public repositories contribute to the transparency of the field and to the reproducibility of archived models, but access to the source model code does not guarantee the reproduction of previously obtained results. In order to execute published model code, it is often necessary to have full knowledge of all of its software, system, and data dependencies and establish an appropriate computational compile-time and run-time environment accordingly. Even when it is possible to run the model, results may vary with the version of the simulation platform, operating system, other dependencies, or hardware used (Hacker et al., 2017; Seppelt & Richter, 2005). Over the longer-term (5, 10, or 20 years), the model code will become increasingly difficult to run as the modeling frameworks, programming languages, and software systems they depend on continue to evolve and the dependencies needed to build and execute the model become unsupported (Hinsen, 2019).

To improve model reusability, we provide an introduction to the use of containers. Containerizing models allows model developers to create reproducible packages that include model code, their simulation platform, and any additional software requirements, and which can be executed in any machine that supports containerization software. Researchers can leverage this tool to facilitate the model development process, since it allows them to operate and seamlessly collaborate within a clean and controlled environment where external software updates or modifications do not introduce unpredictable alterations to the model's execution. While we favor an early integration of containers into the model development cycle due to the advantages it offers for authors, it requires more experience with containerization software than this introductory paper aims to provide, and is not required to make models more reusable. Indeed, models can be containerized after they are deemed to be ready for publication.

By archiving a containerized version of their model, whether it was developed within a containerized environment or not, researchers can ensure that its results are reproducible and that they and other researchers will be able to run the code further into the future. Producing a containerized model requires attention to the software versions used, any other required dependencies, and following a series of simple steps. However, knowledge of how to build or use containerized environments is mostly limited to professional software engineers and developers and relatively rare within the social and ecological sciences modeling community. In this paper we provide an overview of containers, the concepts needed to understand the containerization of simulation models, and their relevance to the production of high-quality scientific software.

2. What is a container?

Containerization is the process of producing a standalone, executable package that includes all software and data needed to run a given software application - this package is called a *container image* (usually referred to as just "image"). For example, a NetLogo model's image can be pictured as a compressed archive (like a zip file) that includes the model source code files, a specific version of the NetLogo platform with its accompanying Java Development Kit runtime, and any NetLogo extensions used. These images are simply files and thus can be readily shared with other authors or users via dedicated image repositories (e.g., DockerHub for Docker, or Cloud Library for Singularity) or through private channels.

Container images can then be used to create a running instance (called a *container*) with an embedded virtual filesystem that only contains the files (e.g., model source code, software or data dependencies, system libraries) originally included in the image. This arrangement has been characterized as a *host-guest* relationship, where the host's hardware is utilized by the guest environment of the container. Host and guest environments are independent and do not have direct access to elements from each other, unless explicitly specified and under restricted uses. Irrespective of the original operating system or the versions of simulation platforms available in the host environment of the computer, the simulations in the container will be run using only the tools that were purposefully included in the underlying image. As a consequence, a container can be seen as an isolated and controlled runtime environment.

Before containers, these kinds of isolated environments were created through virtual machines, which required the installation of a full copy of an operating system and the pre-allocation of a set of computing resources per runtime environment (Figure 1). Having several copies of operating systems running on a single machine results in increased memory requirements and longer waiting times to start each environment. Instead, containers are intended as lightweight applications that can dynamically share computing resources with others, allowing for more time- and energy- efficient execution. Virtual machines provide a stronger level of isolation than containers because they minimize sharing of computing resources with other virtual machines. In contrast, containers share the operating system kernel, which handles the communication with the machine's hardware, and use its scheduler to assign computing resources to each running container. Isolation is achieved by controlling access and visibility of processes, so each container can only "see" what belongs to its runtime environment. This translates to a noticeable difference in computing resources needed from the point of view of prospective users: instead of having to dedicate an entire operating system for each isolated environment (which can quickly exhaust a machine's CPU and memory), it is sufficient to install the container runtime platform that executes various containers, using the original operating system as the host.

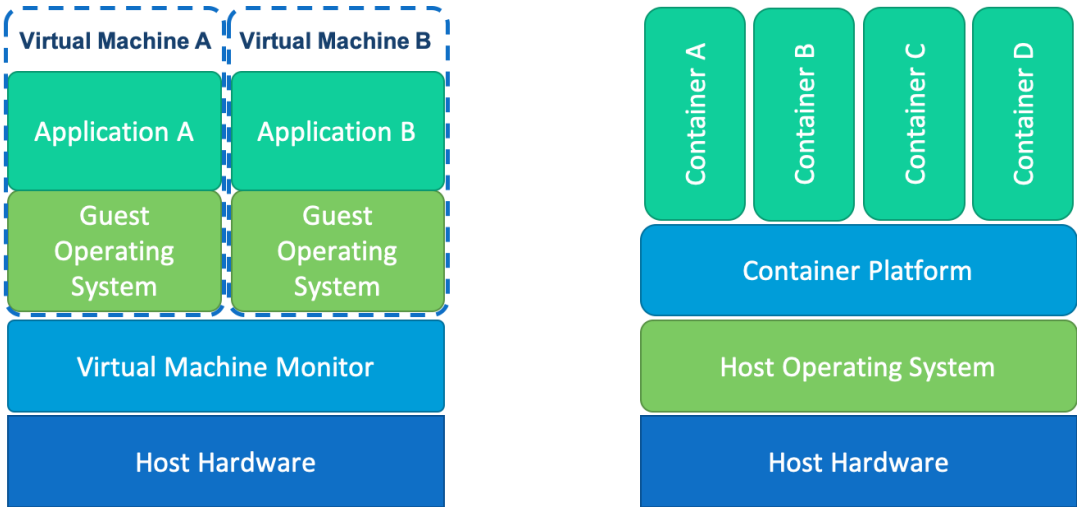

Figure 1: Comparison between virtual machines (left) and containers (right).

Container development was made possible by features present in Linux kernels, thus most of the available tools are based on this operating system. In host machines originally running this operating system, containers will simply use its kernel as discussed above. Notably, the same kernel can be shared by containers using different Linux distributions or versions, which allows to use environments with different Linux distributions on the same host machine. When containers are run in machines using a different operating system, it is first necessary to start a virtual machine containing the Linux features that are minimally needed to run the containers. It is worth noting that Docker containers can also be Windows-based, a functionality supported by Microsoft in recent years, which eliminates the need to start a virtual machine if run by a Windows host operating system. For the rest of cases where the host operating system does not match the container's, a lightweight virtual machine is used to support the execution of the container.

3. Advantages of containers

Being able to produce a controlled running environment that is lightweight and easily shared facilitates the production of consistent results from its corresponding model, irrespective of current versions of any software dependencies or the operating system of the host machine used to run it. Model development within containers is the answer for researchers that feel hesitant about updating their modeling software or any of the extensions or packages it uses, in fear of breaking or changing the behavior of their model. Compared to native (i.e., non-containerized) models, generating the controlled running environments of containerized models requires the execution of some additional processes. However, containers have been found to produce a negligible impact on CPU, memory, and execution time performance under most circumstances (Di Tommaso et al., 2015; Felter et al., 2014), when using a Linux-based host operating system. Still, this efficient performance is expected to be lost if it is necessary to start a virtual machine to solve operating system incompatibilities between the host and the container platform, as would be the case with MacOS and Windows machines when running Linux-based containers.

Containers provide a ready-to-use environment where models can be edited, manipulated and run, eliminating the need to ensure that a prospective host machine has all the right versions of the model's dependencies installed. This tool allows models to be reusable among researchers and portable across different operating systems. Models that are reusable can aid in model-to-model comparisons (Hales et al., 2003), expedite replication studies (Wilensky & Rand, 2007), or can be coupled with other reusable models to facilitate the study of feedbacks between the newly connected processes (Robinson et al., 2018). In addition, using the portable environment provided by containers allows model users and developers to focus on the code as a probable source of inconsistency when unexpected results are obtained, instead of exploring the different ways in which the host machine's operating system or installed software versions could have influenced such results. Accordingly, containers can accelerate the collaborative development of a model by allowing team members to work on a ready-to-use development environment that can be easily ported to any machine. Portability is also important in an education context: if the instructor uses containerized example models, students only need to follow a handful of straightforward instructions (discussed in the section "Running a container") to run the model. This would allow them to independently run and interact with the educational material, while having the confidence that their version of the model is producing consistent results (Hacker et al., 2017). In the longer term, it would also be beneficial to include containerization practices in the curriculum, so future researchers have more familiarity with building and running containers.

Containerizing models also reduces the chance that the code will become obsolete and unrunnable in the near future. Despite efforts of simulation platforms and computer languages to maintain backwards compatibility, software updates regularly add and remove features that can potentially change the model's results or even impede its execution (Boettiger, 2015; Hinsen, 2019). As a result, many older models may need to be modified and updated to be capable of being executed today, as is the case for the model described by Janssen and Jager (1999). Because a model's runtime environment is provided by a container, it can run simulations as long as the container itself can be executed. Even if the support for current images is discontinued at some future time as containerization software itself is updated, it is reasonable to predict a longer life expectancy for the images, which enclose both the code and its matching version of the simulation platform, than for the code alone. Conversely, containers also offer the opportunity to configure an environment where older versions of an operating system and a simulation platform are used, which would allow to run models that could not be run otherwise. Rerunning these "legacy" models requires conditions that cannot always be met, like having access

to the appropriate software versions and ensuring that the operating system used can share the host machine's kernel. The process of backward engineering the dependencies of a model published several years ago may not be straightforward, but containers offer the ability to easily test different environments until the appropriate one is found.

Despite their increasing popularity in other areas (Almugbel et al., 2017; Clyburne-Sherin et al., 2019; Morris et al., 2017), containers are rare in the field of social-environmental systems and ecological modeling, in part because of the learning curve it entails. In many cases, using container platforms requires a basic familiarity with command line instructions and, although their integration with graphic user interfaces (GUIs) is possible, it sometimes requires experienced manipulation of the container environment to enable it. Nevertheless, we argue that the transparency, portability, reusability, and long-term preservation enabled by containerization easily outweighs the effort needed to learn this new tool.

The following section provides an introduction to containerization concepts and a simple workflow to containerize a model with the widely used containerization platform, Docker. Because specific commands can change according to the simulation platform used and as new Docker versions are released, we don't provide detailed instructions for containerization in this paper. Instead, our aim is to familiarize the readers with the conceptual organization of a container and the terminology used with Docker. This introductory material can help readers better understand more detailed instructions available elsewhere, like in our series of step-by-step tutorials for current Docker versions and specific simulation platforms, which can be found at https://forum.comses.net/t/what-is-a-container-and-why-should-you-containerize-your-model/8139.

4. How to containerize a model: a conceptual guide based on Docker

Docker is a container platform, available for multiple operating systems, that started as an open-source effort and continues to offer open access to its core functionality. It has found widespread use in different fields and an active user community (Morris et al. 2017). Although it can be considered as the container industry leader, Docker is only one of the many tools available in the market. In fact, in 2015 it helped launch the Open Container Initiative (Open Container Initiative, n.d.), with the purpose of creating open standards to share container format and runtime specifications with other actors in the industry, like Red Hat (buildah, podman), Canonical (LXD), and Google (Jib, Kubernetes). Among the other container options, Singularity is a platform of relevance for researchers (Kurtzer et al., 2017). It is not yet as widely used as Docker, but has the advantage of not requiring high-level permissions to run. This characteristic is desirable for users who do not have administrator's access in their machine, or for many users of High-Performance Computing (HPC) or High-Throughput Computing (HTC) environments. Singularity was developed with HPC use in mind, which makes it more compatible with the complicated architectures used in these environments than Docker. Although the concepts described below focus on Docker containers, the underlying logic is similar for Singularity. In fact, Singularity's documentation (https://sylabs.io/docs/) emphasizes the importance of its interoperability with Docker and provides a detailed guide to the different ways Docker images can be ported to Singularity.

Containerization of a model refers to the process of creating or building a package that includes the model code, the simulation platform, other dependencies, and, optionally, any data needed and the instructions to follow when the container is *run*. Accordingly, interaction with the Docker software revolves around the "build" (i.e., create the container image) and "run" (i.e., execute the container) commands (Figure 2), and the same is true for Singularity. For a given model, the "build" command would be executed once by the author to create the corresponding image, while the "run" command would be executed any number of times a user chooses to interact with the model or to run it from the image.

It is important to note that, whenever a container is shared, all of its contents and functionalities are made available to the prospective users. This adds an important point to consider when working with commercial software or sensitive data, since it may be necessary to remove software licenses and aggregate or produce some representative replacement data before starting the containerization process. Due to the introductory purpose of this paper, the following section provides a general overview of the steps needed to containerize a model and it does not mention any of the additional steps that are to be followed to comply with the different regulations. Before releasing or publishing a container, it is vital to confirm that it complies with the appropriate licenses and permissions.

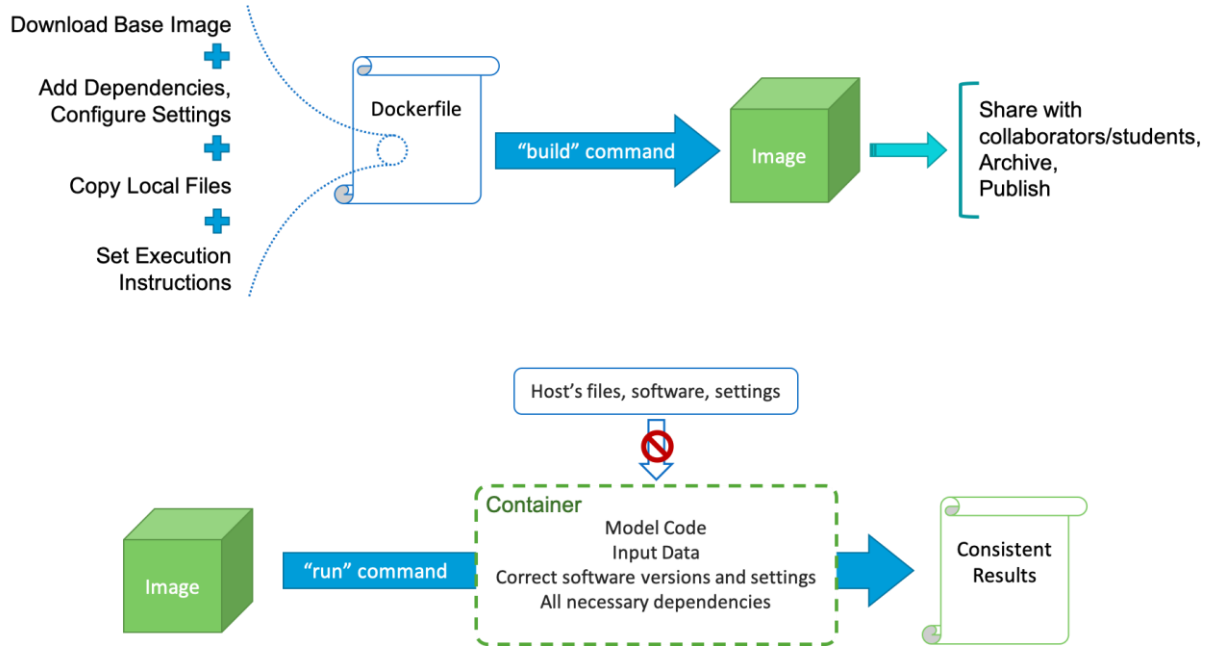

Figure 2: Schematic representation of the creation of an image through the "build" command (top) and of the execution of a container through the "run" command (bottom).

4.1 Building a container image

The first step when building an image is to find an appropriate *base image*, which includes a lightweight operating system and has the specified version of the simulation software, like R, NetLogo or Python, already installed. This enables a simulation to run in an environment that is isolated from the host's operating system. One of the advantages of Docker is the availability of a diverse array of base images to build upon, provided by both the Docker team and the community, and downloadable from the DockerHub repository. Base images especially relevant for socio-environmental models are the "Rocker" collection, comprising different versions of R and RStudio images (Boettiger & Eddelbuettel, 2017), and the official "OpenJDK" images, which provide Java Development Kit base images, needed to generate NetLogo containers.

After the base image is downloaded, the rest of the image building process can be pictured as a series of incremental modifications where additional software is downloaded, the model code and data files are copied, and settings may be customized. In a sense, the new image will be the product of adding a number of *layers* to a base image. Accordingly, "layer" is the term used by Docker to refer to these modifications. For example, when creating a container for a NetLogo model, the first instruction would direct Docker to download a base image from OpenJDK to provide the Java environment on which the simulation platform runs. The second instruction would be to download and install a specific version of NetLogo. In the case of R and RStudio, the base images provided by Rocker already include the specified version of the software, but a common modification is the addition of packages that are not present in the base distribution of R (e.g., adding "tidyverse" to the base image). To reproduce the correct environment, it is vital to specify all the necessary settings and the required packages, extensions, or any other dependencies that are not included in the base image, as well as the right version of each of these additions. Because discovering all the software requirements of a finished modeling project can be cumbersome, we recommend making dependencies and requirements explicit as the model is developed (Wilson et al., 2017). This task can be facilitated by keeping a modelling notebook, analogous to a laboratory notebook, that follows the TRACE structure presented by Ayllón et al. (2021).

Once the necessary software layers are added, the next step is to include the author's model source code, input data, and scripts. Both software and model-related files can be downloaded from user-specified URLs or copied from the author's file system. It is important to remember that the steps outlined above are only executed when building the image and not when the container is subsequently run. Items are downloaded and copied once and

are stored as a snapshot of the model and its simulation environment at the time the image was built. This is a vital feature to ensure the reusability and reproducibility of the package: any subsequent modifications made to the model or any of its dependencies will not be reflected in the image (although a new version of the image can always be built if the intent is indeed to include these changes).

Depending on the purpose of the containerization, it may be necessary to provide execution instructions which are run every time the image is executed. Common uses of these commands are to instruct the container to open the simulation platform so the user can interact with the model, or to execute a script that produces output files. It is important to note that any files generated in the container will not be accessible from the host environment unless the container is correctly started by using the appropriate "run" command options.

The series of instructions for Docker to build an image are usually collected in a dedicated file called *Dockerfile* (the equivalent for Singularity users is called *Definition File*). A successful containerization depends on a careful adjustment of the Dockerfile to the software and settings requirements of the model (see our tutorials for examples of a Dockerfile using agent-based models implemented in R and Netlogo at https://forum.comses.net/t/what-is-a-container-and-why-should-you-containerize-your-model/8139). However, the additional effort devoted to this activity is repaid in the enhanced transparency and reproducibility of the published work. Once a Dockerfile is customized to the needs of a model, the author can simply execute the "build" command to create the image (see Fig. 2, top panel). Provided with the Dockerfile instructions, Docker will follow them sequentially: it will download the base image, install the specified additional software, modify any necessary settings and copy the model's input data and code files. This bundle of ready-to-run software plus any execution instructions will be stored in an efficient manner that can be easily shared using the dedicated DockerHub, which is a public repository that allows users to easily download any images available, or compressed and shared directly with collaborators.

4.2 Running a container

A *container* is a running instance of an image that was built previously. After downloading and installing Docker in their machine, any user with access to the image can run a container instance by executing a "run" command (the same is true with Singularity and Singularity images). A typical "run" command will include the name of the image to be run and the paths of the folders (in the host machine and in the container) that are designated to share the model's output files (Fig. 2, bottom panel). Other information like variable values or connection ports can be also specified in the "run" command. Docker Desktop, the application for MacOS and Windows, provides in its dashboard a simple graphic interface that allows to start any subsequent interactions with existing containers.

According to the instructions given in the Dockerfile or Definition File during the image building step, containers might allow the user's interaction through command-based or graphic interfaces, or be programmed to run certain execution instructions (e.g., a BehaviorSpace experiment in NetLogo, or an R script) and produce a determined set of results. Irrespective of how output files are generated, it is important to note that they will exist within the isolated environment that constitutes a container. Result files or any changes made inside the container will not be accessible on the host filesystem, and thus will be lost as soon as the container is stopped and removed, unless the "run" command includes an explicit instruction to "mount" a directory from the host filesystem into a container. "Mounting" a directory establishes a connection between a specific directory in the container's filesystem and the host directory of interest (hence the typical inclusion of the paths to host and guest folders in the "run" command) that allows them to share their contents. For this reason, this instruction is typically used to be able to obtain output files of interest from the container.

When containers are used during the model development stages, directories are typically "mounted" from the host filesystem into the container so changes to the model or analysis code made in the host system are immediately visible within the container. Alternatively, if no "mount" points were established when the container was "run", a "commit" command can be executed to build a new image that includes the latest changes. In any case, since a development environment can easily bloat with unused files, extensions, packages, or even variables, it is advisable to carefully build a dedicated publication-ready image for archival using a new Dockerfile or Definition File. Although we favor the use of containers during the development process, it is important to note that the goal of improving model reproducibility and reusability largely depends on the correct

containerization of the model that is deemed to be ready for publication, whether the model was already containerized for development purposes or not.

5. Future perspectives

Containerizing simulation models makes them more portable across different compute environments, facilitating team-based development, and also enhances their interoperability, reusability and reproducibility after they have been published. However, use of containerization software like Docker requires a learning phase that is not negligible. Projects like Code Ocean (https://codeocean.com/) and Whole Tale (https://wholetale.org/index.html) propose to ease the containerization process by providing an interactive GUI for containerizing and running code, and adding new functionality. Code Ocean provides a simple browser-based interface that allows developers to configure the software environment and upload code and data, and enables users to run the model online. When the project is ready for publication, the platform will produce a *compute capsule* containing the code, data, results, metadata and the computational environment and a corresponding DOI. Subsequent users are then able to visit the Code Ocean link embedded in the article and easily run the model (Clyburne-Sherin et al., 2019). Currently, Code Ocean offers commercial services to companies and publishers, and a limited monthly compute time is available for academics that sign up to the service. Along similar lines, Whole Tale is a web-based platform that facilitates the creation of executable research objects that include code, data, and software environment, as well as the scientific narrative that frames the project and its findings (Brinckman et al., 2019).

Code Ocean and Whole Tale are exciting developments that support the adoption of good computing practices. However, these applications only are able to run a limited list of simulation platforms so far, and do not yet include NetLogo. As these applications develop and gain functionality, we encourage the use of more mature tools like Docker for most users. Irrespective of the specific tool used, adopting the practice of containerizing simulation models is a path for the production of scientific software of high quality, easing the processes of code review, validation, evaluation, result reproduction and model reuse.

As the use of scientific software continues to find expanded applications in the community, attention to practices that favor its transparency, reproducibility and reusability becomes essential. Therefore, we propose the consolidation of containerization as a future standard for the publication of high-quality scientific software.

References

Almugbel, R., Hung, L.-H., Hu, J., Almutairy, A., Ortogero, N., Tamta, Y., & Yeung, K. Y. (2017). Reproducible Bioconductor workflows using browser-based interactive notebooks and containers. Journal of the American Medical Informatics Association, 25(1), 4–12. https://doi.org/10.1093/jamia/ocx120

Amouroux, E., Gaudou, B., Desvaux, S., & Drogoul, A. (2010). O.D.D.: A Promising but Incomplete Formalism for Individual-Based Model Specification. 2010 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), 1–4. https://doi.org/10.1109/RIVF.2010.5633421

Ayllón, D., Railsback, S. F., Gallagher, C., Augusiak, J., Baveco, H., Berger, U., Charles, S., Martin, R., Focks, A., Galic, N., Liu, C., van Loon, E. E., Nabe-Nielsen, J., Piou, C., Polhill, J. G., Preuss, T. G., Radchuk, V., Schmolke, A., Stadnicka-Michalak, J., Thorbeck, T., Grimm, V. (2021). Keeping modelling notebooks with TRACE: Good for you and good for environmental research and management support. Environmental Modelling & Software, 136, 104932. https://doi.org/10.1016/j.envsoft.2020.104932

Bell, A. R., Robinson, D. T., Malik, A., & Dewal, S. (2015). Modular ABM development for improved dissemination and training. Environmental Modelling & Software, 73, 189–200. https://doi.org/10.1016/j.envsoft.2015.07.016

Boettiger, C. (2015). An introduction to Docker for reproducible research. ACM SIGOPS Operating Systems Review, 49(1), 71–79. https://doi.org/10.1145/2723872.2723882

Boettiger, C., & Eddelbuettel, D. (2017). An Introduction to Rocker: Docker Containers for R. The R Journal, 9(2), 527. https://doi.org/10.32614/RJ-2017-065

Brinckman, A., Chard, K., Gaffney, N., Hategan, M., Jones, M. B., Kowalik, K., Kulasekaran, S., Ludäscher, B., Mecum, B. D., Nabrzyski, J., Stodden, V., Taylor, I. J., Turk, M. J., & Turner, K. (2019). Computing environments for reproducibility: Capturing the "Whole Tale." Future Generation Computer Systems, 94, 854–867.

https://doi.org/10.1016/j.future.2017.12.029

Clyburne-Sherin, A., Fei, X., & Green, S. A. (2019). Computational Reproducibility via Containers in Psychology. Meta-Psychology, 3, MP.2018.892. https://doi.org/10.15626/MP.2018.892

Di Tommaso, P., Palumbo, E., Chatzou, M., Prieto, P., Heuer, M. L., & Notredame, C. (2015). The impact of Docker containers on the performance of genomic pipelines. PeerJ, 3, e1273. https://doi.org/10.7717/peerj.1273

Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2014). An updated performance comparison of virtual machines and linux containers (IBM Research Report). IBM Research Division. https://dominoweb.draco.res.ibm.com/reports/rc25482.pdf

Grimm, V., Railsback, S. F., Vincenot, C. E., Berger, U., Gallagher, C., DeAngelis, D. L., Edmonds, B., Ge, J., Giske, J., Groeneveld, J., Johnston, A. S. A., Milles, A., Nabe-Nielsen, J., Polhill, J. G., Radchuk, V., Rohwäder, M.-S., Stillman, R. A., Thiele, J. C., & Ayllón, D. (2020). The ODD Protocol for Describing Agent-Based and Other Simulation Models: A Second Update to Improve Clarity, Replication, and Structural Realism. Journal of Artificial Societies and Social Simulation, 23(2), 7. https://doi.org/10.18564/jasss.4259

Hacker, J. P., Exby, J., Gill, D., Jimenez, I., Maltzahn, C., See, T., Mullendore, G., & Fossell, K. (2017). A Containerized Mesoscale Model and Analysis Toolkit to Accelerate Classroom Learning, Collaborative Research, and Uncertainty Quantification. Bulletin of the American Meteorological Society, 98(6), 1129–1138. https://doi.org/10.1175/BAMS-D-15-00255.1

Hales, D., Rouchier, J., & Edmonds, B. (2003). Model-to-Model Analysis. Journal of Artificial Societies and Social Simulation, 6(4), 5. http://jasss.soc.surrey.ac.uk/6/4/5.html

Hinsen, K. (2019). Dealing With Software Collapse. Computing in Science & Engineering, 21(3), 104–108. https://doi.org/10.1109/MCSE.2019.2900945

Jalili, V., Afgan, E., Gu, Q., Clements, D., Blankenberg, D., Goecks, J., Taylor, J., & Nekrutenko, A. (2020). The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2020 update. Nucleic Acids Research, 48(W1), W395–W402. https://doi.org/10.1093/nar/gkaa434

Janssen, M., & Jager, W. (1999). An integrated approach to simulating behavioural processes: A case study of the lock-in of consumption patterns. Journal of Artificial Societies and Social Simulation, 2(2), 21–35.

Kurtzer, G. M., Sochat, V., & Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. PLOS ONE, 12(5), e0177459. https://doi.org/10.1371/journal.pone.0177459

McPhillips, T., Willis, C., Gryk, M. R., Nunez-Corrales, S., & Ludascher, B. (2019). Reproducibility by Other Means: Transparent Research Objects. 2019 15th International Conference on EScience (EScience), 502–509. https://doi.org/10.1109/eScience.2019.00066

Meadows, M., & Cliff, D. (2012). Reexamining the Relative Agreement Model of Opinion Dynamics. Journal of Artificial Societies and Social Simulation, 15(4), 4. https://doi.org/10.18564/jasss.2083

Morris, D., Voutsinas, S., Hambly, N. C., & Mann, R. G. (2017). Use of Docker for deployment and testing of astronomy software. Astronomy and Computing, 20, 105–119. https://doi.org/10.1016/j.ascom.2017.07.004

Open Container Initiative. (n.d.). About the Open Container Initiative. Retrieved October 21, 2021, from https://opencontainers.org/about/overview/

Pineau, J., Vincent-Lamarre, P., Sinha, K., Larivière, V., Beygelzimer, A., d'Alché-Buc, F., Fox, E., & Larochelle, H. (2020). Improving Reproducibility in Machine Learning Research (A Report from the NeurIPS 2019 Reproducibility Program). ArXiv:2003.12206 [Cs, Stat]. http://arxiv.org/abs/2003.12206

Polhill, J. G., Izquierdo, L. R., & Gotts, N. M. (2005). The Ghost in the Model (and Other Effects of Floating Point Arithmetic). Journal of Artificial Societies and Social Simulation, 8(1), 5.

Robinson, D. T., Di Vittorio, A., Alexander, P., Arneth, A., Barton, C. M., Brown, D. G., Kettner, A., Lemmen, C., O'Neill, B. C., Janssen, M., Pugh, T. A. M., Rabin, S. S., Rounsevell, M., Syvitski, J. P., Ullah, I., & Verburg, P. H. (2018). Modelling feedbacks between human and natural processes in the land system. Earth System Dynamics, 9(2), 895–914. https://doi.org/10.5194/esd-9-895-2018

Rollins, N. D., Barton, C. M., Bergin, S., Janssen, M. A., & Lee, A. (2014). A Computational Model Library for publishing model documentation and code. Environmental Modelling & Software, 61, 59–64. https://doi.org/10.1016/j.envsoft.2014.06.022

Seppelt, R., & Richter, O. (2005). "It was an artefact not the result": A note on systems dynamic model development tools. Environmental Modelling & Software, 20(12), 1543–1548. https://doi.org/10.1016/j.envsoft.2004.12.004

Wilensky, U., & Rand, W. (2007). Making Models Match: Replicating an Agent-Based Model. Journal of Artificial Societies and

Social Simulation, 10(4), 2.

Wilkinson, M. D., Dumontier, M., Aalbersberg, Ij. J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.-W., da Silva Santos, L. B., Bourne, P. E., Bouwman, J., Brookes, A. J., Clark, T., Crosas, M., Dillo, I., Dumon, O., Edmunds, S., Evelo, C. T., Finkers, R., … Mons, B. (2016). The FAIR Guiding Principles for scientific data management and stewardship. Scientific Data, 3(1), 160018. https://doi.org/10.1038/sdata.2016.18

Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., & Teal, T. K. (2017). Good enough practices in scientific computing. PLOS Computational Biology, 13(6), e1005510. https://doi.org/10.1371/journal.pcbi.1005510