doi: 10.15496/publikation-29062

Unified Container Environments for Scientific Cluster Scenarios

Benjamin Schanzel Mark Leznik Simon Volpert Jörg Domaschka
Stefan Wesner

Institute of Information Resource Management, Ulm University, Germany

Providing runtime dependencies for computational workflows in shared environments, like HPC clusters, requires appropriate management efforts from users and administrators. Users of a cluster define the software stack required for a workflow to execute successfully, while administrators maintain the mechanisms to offer libraries and applications in different versions and combinations for the users to have maximum flexibility. The Environment Modules system is the tool of choice on bwForCluster BinAC for this purpose. In this paper, we present a solution to execute a workflow which relies on a software stack not available via Environment Modules on BinAC. The paper describes the usage of a containerized, user-defined software stack for this particular problem using the Singularity and Docker container platforms. Additionally, we present a solution for the reproducible provisioning of identical software stacks across HPC and non-HPC environments. The approach uses a Docker image as the basis for a Singularity container. This allows users to define arbitrary software stacks giving them the ability to execute their workflows across different environments, from local workstations to HPC clusters. This approach provides identical versions of software and libraries across all environments.

1 Introduction

The reproducibility of experimental findings and results is an essential requirement of many scientific processes. Papers and reports presenting results and conclusions

doi: 10.15496/publikation-29052

obtained from experiments therefore not only describe the final results, but also the experiment methodology and environment. This holds true for computational sciences as well (Sandve et al., 2013). Environmental influences on computational workflows – i. e. different operating systems (OS's) and various versions of software libraries and applications – interfere with the reproducibility of scientific results computed within such an environment. This implies that for a computational scientific workflow to be reproducible, the runtime environment, in which results are computed in, must be controlled, described, and provided alongside the results (Boettiger, 2015; Bartusch et al., 2018).

A number of solutions are available for the definition and reproducible provisioning of computational environments. In the industry of enterprise applications and particularly in cloud computing environments, hardware level virtualization is a widely adopted method for providing complete environments, called Virtual Machines (VM's) (Gartner Inc., 2016). The primary goals of this virtualization mechanism are the isolation of virtualized environments from the host system, as well as increasing the utilization of costly resources by hosting multiple VM's on one physical machine. Despite these objectives, VM's are also useful for providing environments in an automated and reproducible manner. VM images contain a predefined stack of software – from the OS, software libraries, applications, to user provided code. These images can be deployed and run on any compatible system. Hardware level virtualization thus is an effective approach for isolating a scientific computational workflow from external environmental influences. It might, however, restrict direct access to accelerated hardware (e. g. GPUs), and could therefore introduce significant performance regressions. Consequently, this contradicts the use in High-Performance Computing (HPC) environments, where users want their code to be executed as close as possible to real hardware for a maximum of performance.

Another, more recent approach to describing and providing complete software stacks for computational workflows is containerization. Here again, images of complete environments (i. e. the OS, libraries, etc.) are predefined and can be deployed and executed in a reproducible and platform-independent manner. Processes within a container are executed directly on the host system using lightweight virtualization features of the host OS. This does not involve any virtualization of hardware and is therefore more suitable for HPC use cases than hardware level virtualization.

One of the most widely adopted solutions for OS-level virtualization is Docker (Portworx Inc., 2017). Its main use cases are micro-service architectures and web-based cloud applications. In such environments, Docker containers are generally executed inside of VM's.

For HPC use cases, executing containerized software stacks would require running the containers directly on the host OS. Docker, however, is not perfectly suitable for this use case. The Docker host daemon is a root-owned process on the host OS, which means that containers are executed with the highest privileges available on the host system. It would therefore be theoretically possible for a user to run arbitrary code on the host OS with root privileges, effectively rendering the system compromised. This implies security concerns, hindering the adoption of Docker in HPC environments (Kurtzer et al., 2017). Other container technologies, such as Singularity, not only address these security downsides of Docker but also explicitly focus on the use of containers for scientific workflows in HPC environments (Kurtzer et al., 2017).

In this paper we present an exemplary use case of a machine learning (ML) workflow on the bwForCluster BinAC (Krüger et al., 2017) which is executed within a Singularity container. We argue why providing a user-defined software stack in the form of a container is a well-suited method for this use case in particular and for computational workflows in general. Furthermore, we present a two-layered solution with a combination of Docker and Singularity, which enables reproducible and platform independent execution of workflows. Section 2 describes the technical requirements of a specific use case leading to the solution based on a containerized software stack. Section 3 explains how the required software stack has been implemented and how it can be executed, either as a Docker or Singularity container. Section 4 concludes how the use of containers not only solves the specific use case described in this paper, but also helps to improve the reproducibility of computational workflows across different HPC and non-HPC environments in general.

2 Status Quo and Problem Statement

This section describes the use case of an ML workflow, intended to run on bwForCluster BinAC. We explain how the runtime environments for workflows are usually provided on the cluster and why this approach is not sufficient for this exemplary use case.

Providing software for computational jobs submitted by users on bwHPC clusters is commonly accomplished via the Environment Modules system. By this means, it is possible to dynamically modify the applications and software library environment available to the user. Figure 1 visualizes the process of providing software libraries on BinAC. It additionally shows a corresponding process in a non-HPC environment, like a local workstation.

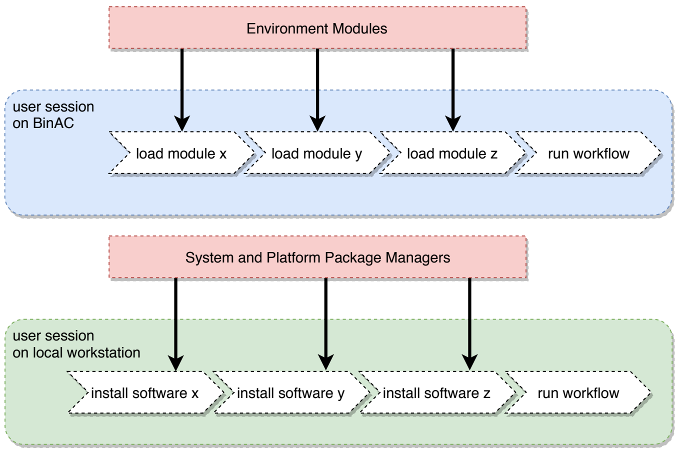

Figure 1: The process of providing software modules required for a computational workflow. On BinAC these modules are loaded via the Environment Modules system. Identical modules might not be available on other HPC clusters. On a local workstation the software modules are installed using the system package manager or platform specific platform managers, e. g. pip for Python libraries. It cannot be guaranteed that the exact same libraries as on the HPC cluster are available via these package managers.

On HPC clusters like BinAC, users integrate module load operations in their job scripts submitted to the scheduling system of the computing cluster. A command to load Keras, TensorFlow and Python in specific versions on the BinAC cluster would look like the following:

```
module load cs/keras/2.1.0-tensorflow-1.4-python-3.5
```

Loading another module, which would provide any of these applications in a different version, is then prevented by the Environment Modules system. This is to prevent conflicts with already loaded modules. Each of the available modules is described in a so-called `modulefile`, which is executed upon loading and contains every step needed to provide an application or software library for the current user. Modules are available system-wide, and thus loadable by all users of this shared environment.

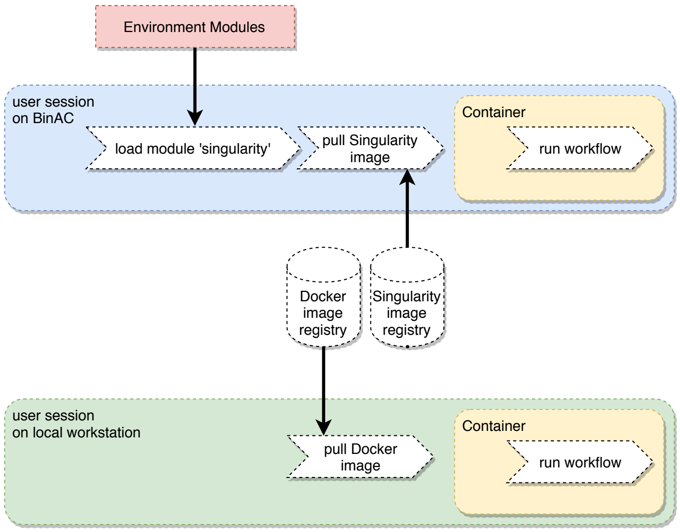

Figure 2: The process of providing a containerized runtime environment for a computational workflow. For HPC environments a Singularity image is pulled from a Singularity image registry. The Singularity system must still be provided on the HPC cluster in any way, e. g. via Environment Modules. In a non-HPC environment Docker might be used as a container platform. Here a Docker image is pulled and executed as a container. The containers on both sides provide an identical application stack, as they both are based on the same `Dockerfile`.

In this paper, we assume an exemplary use case with the goal of training an Artificial Neural Network (ANN) using the Keras library on top of the TensorFlow framework, coupled with Python in their most recent versions. There is no `modulefile` defined on BinAC, which provides this exact combination of the libraries required. These libraries are also not loadable as independent single modules. Under these conditions, we cannot execute our model training code on BinAC in its current setup. There are several solutions to this problem. As a first option, the system administrator could provide a new module with the required software versions. A second solution is to rewrite the Python code for it to be compatible with the software stack that is available now. Also, a local build of these modules in the user's home directory can be performed. These solutions come with manual effort, with the particular downside of an ever-growing number of modules which need to be maintained by an administrator or user. The approach of providing the required software stack via Environment Modules is therefore not well-suited for the described use case. This is particularly true considering the high frequency of TensorFlow releases (TensorFlow, 2018).

Another option for this problem is to let the user define the required software stack in the form of a container and run the computational workflow within it. This would not only reduce manual effort for the system administrators, but, as previously explained, also improve the reproducibility of the computational results across different environments outside the HPC cluster. The process of providing containerized software stacks in both, HPC and non-HPC environments is displayed in Figure 2.

The following sections describe the setup of a container environment we used to circumvent this specific problem faced on the BinAC cluster. Additionally, we describe how the setup allows the execution of such computational workflows not only on HPC clusters, but also in many other container-enabled environments, in a reproducible manner.

3 Operating Singularity Containers on BinAC

This section highlights how we provide the necessary libraries for the previously described use case as a user-defined software stack and how to execute the resulting container in the BinAC HPC cluster environment. We also demonstrate how the approach of using a Docker image as a starting point, rather than Singularity, allows to provide the software stack in non-HPC environments, such as Windows or Mac workstations, where Singularity is not easily available. While Docker does require a virtualization layer on such non-Linux host systems, it is arguably more comfortable to use Docker on these systems. The virtualization layer is directly provided by Docker for Windows and macOS host systems and does not require the user to install any virtualization software separately.

Implementing a scientific computational workflow for HPC environments does not only involve remote interaction with HPC cluster machines, e. g. via SSH, but also writing code on the users local workstation. Users feel certainly more comfortable developing locally on their workstation, having their own editors, development environments, and other tools of choice at hand. This is especially true for debugging purposes. A user's common process might therefore involve local development, debugging, and testing of a computational workflow, eventually submitting it as a job to an HPC cluster. This scenario involves two entirely different software and hardware environments, expected to deliver identical results upon execution. Taking this scenario as a premise, we must extend our previous goal of providing a reproducible software stack for the execution on BinAC by the requirement of serving an identical stack for arbitrary user workstations. Singularity is, however, only available for Linux OS's, requiring the installation of a VM hypervisor on the majority of desktop workstations, as they mostly run on Windows or macOS. Singularity offers the capability of automatically migrating Docker images to its own image format. By this means, a promising solution is to provide a Docker image for workflow execution in non-HPC environments, as it is available for a range of operating systems including Windows and macOS. In a subsequent step, this Docker image can be used as the basis for a Singularity image, which can be executed on BinAC. It is then possible to execute a computational workflow within a defined and reproducible environment, having an identical software stack available across different environments, be it HPC clusters or local workstations.

3.1 Defining and Distributing a Docker Image

As mentioned before, we use Docker images as the basis for building equivalent Singularity images. Docker is used in version 18.03.1 and Singularity in version 2.4.1. The contents of Docker images are described by a so-called `Dockerfile`. This is a text file containing the name of the base image to start from, steps to install additional software on top of that image, and an entry point for the container to start upon execution. Dockerfiles can contain more information to define an image, e. g. directories to mount from the host system or network ports to expose to the host during execution. A comprehensive guide can be found in the online documentation provided by the Docker project (Docker Inc., 2018).

As described in Section 2, we provide a container with certain applications and libraries in specific versions. We start with a publicly available Docker base image, provided by the TensorFlow project. This already includes TensorFlow and Python in the required versions. As this image is based on an Ubuntu Linux base image, we can use Ubuntu's `apt-get` package manager to obtain additional system software. Python libraries, like Keras, can be installed via its own package manager `pip`. We use these two commands to install Keras and additional dependencies on top of the base image, eventually making all of the required software available in a running container. Listing 1 shows a `Dockerfile` providing the software stack described in Section 2. It can be built into an image on any machine with Docker installed by issuing the `build` command. The resulting image can then be executed as a container with the `run` command.

```
FROM tensorflow/tensorflow:1.8.0-gpu-py3
RUN pip install keras==2.1.6
ENTRYPOINT ['/usr/bin/python', 'main.py']
```

Listing 1: `Dockerfile` for the provisioning of Python 3.5, TensorFlow 1.8, and Keras 2.1.

After building the Docker image, we are able to run the computational workflow within the container on a local workstation, under the only premise of having Docker installed. To distribute the image, i. e., making it available to other users, it is necessary to publish it to a container registry. This can either be the official public registry, Docker Hub, or any privately hosted instance. Pre-built images can then be pulled and executed as containers from any remote machine with access to the repository. Listing 2 provides an example of how a `Dockerfile` can be built and executed. To push an image to a registry, it is necessary to authenticate to that registry, i. e. Docker Hub or any privately hosted registry.

```
docker build . -t tensorflow-keras-py3:latest-gpu
docker push schanzel/tensorflow-keras-py3:latest-gpu
docker run schanzel/tensorflow-keras-py3:latest-gpu
```

Listing 2: Shell commands to build and publish a docker image, finally executing the image as a container. The publishing `push` command can be omitted if the image is only needed locally. The period in the `build` command denotes for Docker to search for a `Dockerfile` in the current working directory.

3.2 Building a Singularity Image for HPC Environments

In a following step, we want to run the same container on BinAC, where Docker is not available, but Singularity is. Therefore, it is necessary to build a Singularity image with the identical software stack. Singularity has, as mentioned earlier in this section, the ability to migrate Docker images into its own image format. This conversion is usually done by providing the location and name of a Docker image to Singularity's `build` command. Building an image yields a `.img` file, representing the executable Singularity image in its own format. This file can then be distributed by copying it to a desired location on the file system of the target machine (i. e. a shared file system on BinAC in our case), or by pushing it to a Singularity repository, making it more accessible to other users. Comprehensive documentation about the usage of the Singularity container platform is available in the original publications (Kurtzer et al., 2017; V. V. Sochat et al., 2017; V. Sochat, 2017) as well as in the online user guide (Sylabs Inc., 2018b). Listing 3 provides an example of how to build a Singularity image from the previously published Docker image and executes the resulting image directly from the local file system.

```
singularity build tensorflow-keras-py3.img \
    docker://schanzel/tensorflow-keras-py3:latest-gpu
singularity run --nv tensorflow-keras-py3.img
```

Listing 3: Shell commands to build and run a Singularity container based on the previously built Docker image. The `docker://` prefix on the Docker image URL denotes for Singularity to pull the Docker image from Docker Hub. By invoking the `run` command with the `--nv` flag, Singularity's NVIDIA GPU support is enabled, which is required to leverage the accelerated hardware of BinAC.

Running this container on BinAC requires the Singularity executable to be available on the cluster machines. This is the only prerequisite, and since Singularity is tailored for HPC environments, can be regarded as a given. On the BinAC cluster, Singularity is provided as a loadable module via the Environment Modules system. After the execution of `module load devel/singularity/2.4.1` BinAC is able to pull and run Singularity containers.

It is then possible execute the ML use case, as described in Section 2, in a user-defined software stack on any HPC cluster where Singularity is available. Additionally, this setup ensures that a workflow is executed in an identical software stack, independently from the host environment. This ultimately minimizes environmental influences on the computational results and allows for a better reproducibility as containers are easily distributable via registries or even in a textual form as a `Dockerfile`. It should be noted here, that while Singularity is currently able to directly pull and run Docker containers, this practice should be avoided in this case, due to unpredictable changes on upstream Dockers images, as stated by the Singularity documentation (Sylabs Inc., 2018a).

3.3 A Continuous Delivery Pipeline

To not only improve the reproducibility of computational workflows by the use of containers, we would also like to maintain the reproducible provisioning of containers themselves. The reproducibility of build and deployment steps of containers, as for any other software, can be improved by maximizing the degree of automation in this process and hereby minimizing undocumented manual steps. The use of a Continuous Integration and Continuous Delivery (CI\CD) pipeline is used to automate such build and deploy processes (Humble et al., 2010). This is an aspect which often receives very little attention during research work (Volpert et al., 2018). This section describes how we automate the process of building and providing Docker and Singularity images with such a pipeline.

To continuously build and publish container images, we define a common process which executes every step needed for this purpose. As common in CI\CD, this process is automatically triggered when a relevant change is detected by the pipeline system. This could be a change to a `Dockerfile` being pushed to a version control system (VCS), e. g. Git.

There are two possible starting points for the process as we either want to

(i) build an own Docker image based on a `Dockerfile` or

(ii) use an existing Docker image already provided in a registry.

Case (i) involves building a Docker image from a `Dockerfile` and publishing it to a Docker registry first. In case (ii) there is a Docker image already provided in a registry. Here the process would just use an existing image, e. g. from Docker Hub, and convert this to a Singularity image, omitting the Docker-related build and publish steps. Either case involves building and publishing a Singularity image. The use case described previously matches with case (i) of the process. Case (ii) is suitable for applications that do not need additional libraries on top of an existing Docker base image.

The triggers for this process to be executed continuously also differ between the two cases. In case (i) there is a `Dockerfile` managed by a VCS which can easily trigger the CI\CD process upon change detection. For case (ii), things are slightly more difficult as it requires the detection of changes on an image in a Docker repository which might be owned by some other entity. Docker Hub allows to send push notifications, so-called Webhooks, to arbitrary HTTP endpoints after a new image version was published. This mechanism could be used to trigger the process of building and publishing a Singularity image based on the Docker image that has just received the update. However, such a Webhook must be configured by the owner of the Docker Hub repository. An alternative approach, not relying on the source repositories owner, is to poll the Docker repository for changes on a regular basis. Figure 3 illustrates this process with the described differences between the identified cases.

Having such a pipeline in place, it is possible to automatically and reproducibly build and publish containers for both Docker and Singularity. This further improves the interoperable provisioning of software stacks for computational workflows. A continuous delivery pipeline not only manages containers for both, HPC and non-HPC environments, but also makes sure that the two images are always published in the same version.

4 Conclusions and Future Work

In this paper, we have shown several approaches of deploying containers on local environments as well as clusters for an exemplary machine learning use case. Our approach is easily transferable or extendable and requires little to no knowledge about the underlying software. This solves the problem of running arbitrary software on clusters without administrator privileges, with Singularity being the sole prerequisite provided by the cluster administration.

The proposed solution does not take into account performance implications of containers, yet. Currently, the exact performance drop introduced by the use of containers is being further investigated but initial results indicate low impact for the tested single node jobs (Sweeney et al., 2018). Additionally, the intercommunication of containers with each other was not considered in this paper. A more visible performance impact is expected in this case. ML frameworks like TensorFlow offer

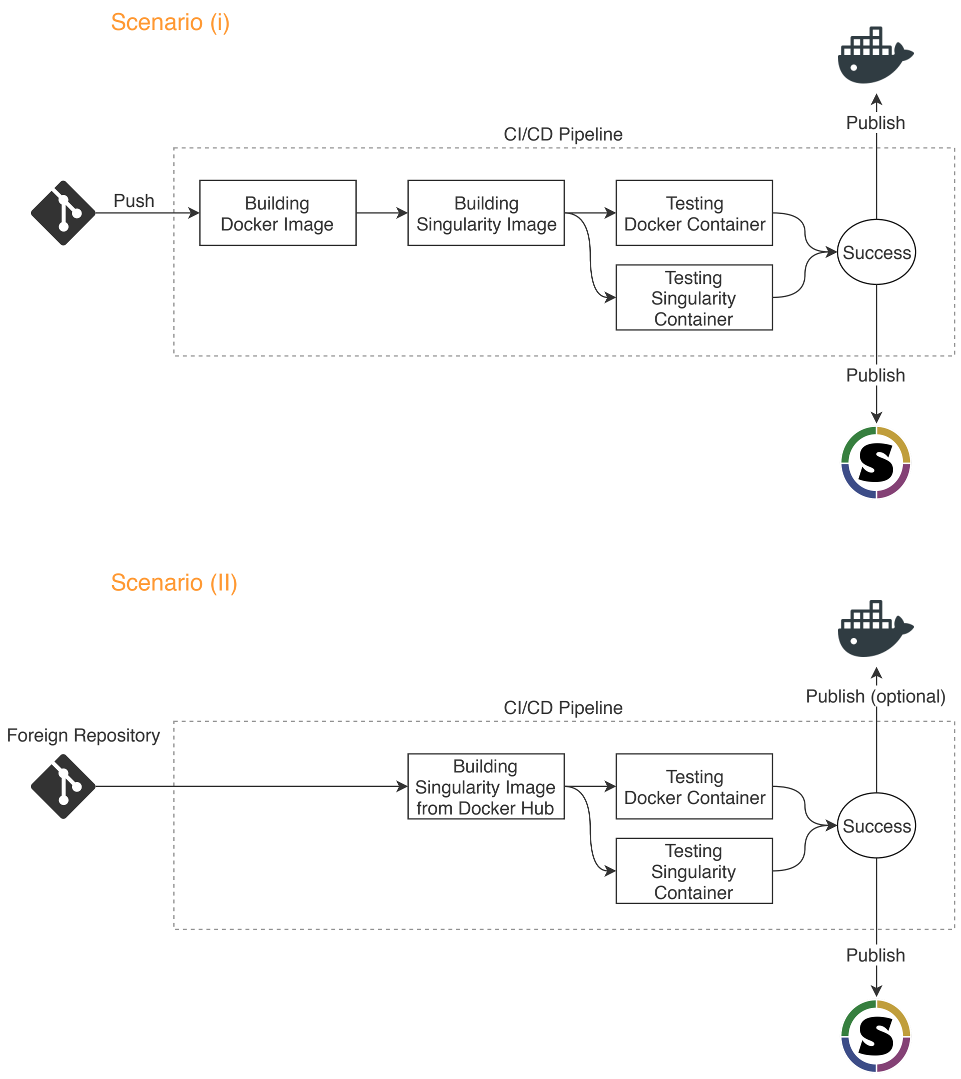

Figure 3: The proposed CI\CD architecture to automate the provisioning of both, Docker and Singularity images. Scenario (i) considers building an own Docker image, while in Scenario (ii) an existing image is pulled from Docker Hub. In either scenario, the Docker image is converted to a Singularity image and published to a registry. Since the image is already being pulled from Docker Hub, the Docker image can be optionally pushed to a private repository.

the possibility of distributed learning. Hereby, a coordinator node subdivides the data onto several nodes, decreasing the required training time. However, there is no clear indication whether the communication overhead introduced by such a setup can be efficiently supported by container solutions. For multi-node jobs, additional tests are needed.

Acknowledgements

The authors acknowledge the support by the High Performance and Cloud Computing Group at the Zentrum für Datenverarbeitung of the University of Tübingen, the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant no INST 37/935-1 FUGG.

The research leading to these results has received funding from the EC's Framework Programme HORIZON 2020 under grant agreement number 732667 (RECAP).

Corresponding Author

Mark Leznik: `mark.leznik@uni-ulm.de`
Institute of Information Resource Management,
Ulm University, Germany

References

Bartusch, F., M. Hanussek and J. Krüger (2018). »Containerization of Galaxy Workflows Increases Reproducibility«. In: *Proceedings of the 4th bwHPC Symposium.* DOI: `10.15496/publikation-25200`.

Boettiger, C. (2015). »An Introduction to Docker for Reproducible Research«. In: *SIGOPS Oper. Syst. Rev.* 49.1, pp. 71–79. ISSN: 0163-5980. DOI: `10.1145/2723872.2723882`.

Docker Inc. (2018). *Docker Documentation.* URL: `https://docs.docker.com/` (visited on 13. 06. 2018).

Gartner Inc. (2016). *Gartner Says Worldwide Server Virtualization Market Is Reaching Its Peak.* URL: `https://www.gartner.com/newsroom/id/3315817` (visited on 29. 08. 2018).

Humble, J. and D. Farley (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Upper Saddle River, NJ: Addison-Wesley. ISBN: 978-0-321-60191-9.

Krüger, J. et al. (2017). »Bioinformatics and Astrophysics Cluster (BinAC)«. In: *Proceedings of the 3rd bwHPC-Symposium*. DOI: `10.11588/heibooks.308.418`.

Kurtzer, G. M., V. Sochat and M. W. Bauer (2017). »Singularity: Scientific Containers for Mobility of Compute«. In: *PLOS ONE* 12.5, e0177459. ISSN: 1932-6203. DOI: `10.1371/journal.pone.0177459`.

Portworx Inc. (2017). *2017 Annual Container Adoption Survey: Huge Growth in Containers*. URL: `https://portworx.com/2017-container-adoption-survey/` (visited on 29. 08. 2018).

Sandve, G. K., A. Nekrutenko, J. Taylor and E. Hovig (2013). »Ten Simple Rules for Reproducible Computational Research«. In: *PLoS Computational Biology* 9.10. Ed. by P. E. Bourne, e1003285. ISSN: 1553-7358. DOI: `10.1371/journal.pcbi.1003285`.

Sochat, V. (2017). »Singularity Registry: Open Source Registry for Singularity Images«. In: *The Journal of Open Source Software* 2.18, p. 426. ISSN: 2475-9066. DOI: `10.21105/joss.00426`.

Sochat, V. V., C. J. Prybol and G. M. Kurtzer (2017). »Enhancing Reproducibility in Scientific Computing: Metrics and Registry for Singularity Containers«. In: *PLOS ONE* 12.11, e0188511. ISSN: 1932-6203. DOI: `10.1371/journal.pone.0188511`.

Sweeney, K. M. D. and D. Thain (2018). »Efficient Integration of Containers into Scientific Workflows«. In: *Proceedings of the 9th Workshop on Scientific Cloud Computing*. ScienceCloud'18. New York, NY, USA: ACM, 7:1–7:6. ISBN: 978-1-4503-5863-7. DOI: `10.1145/3217880.3217887`.

Sylabs Inc. (2018a). *Singularity and Docker — Singularity Container 2.6 Documentation*. URL: `https://www.sylabs.io/guides/2.6/user-guide/singularity_and_docker.html#the-build-specification-file-singularity` (visited on 30. 08. 2018).

— (2018b). *User Guide — Singularity Container 2.6 Documentation*. URL: `https://www.sylabs.io/guides/2.6/user-guide/` (visited on 29. 08. 2018).

TensorFlow (2018). *Computation Using Data Flow Graphs for Scalable Machine Learning: Tensorflow/Tensorflow*. URL: `https://github.com/tensorflow/tensorflow` (visited on 30. 08. 2018).

Volpert, S., F. Griesinger and J. Domaschka (2018). »Continuous Anything for Distributed Research Projects«. In: *Dependability Engineering*. InTech. DOI: `10.5772/intechopen.72045`.