Recommendations to package and containerize bioinformatics software

Authors: Bjorn Gruening (2), Olivier Sallou (3), Pablo Moreno (1), Felipe da Veiga Leprevost (4), Hervé Ménager (5), Dan Søndergaard (6), Hannes Röst (10), Fábio Madeira (1), Victoria Dominguez Del Angel (7), Michael R. Crusoe (8), Susheel Varma (1), Rafael C Jimenez (9)* and Yasset Perez-Riverol (1)*

Affiliation:

(1) EMBL-European Bioinformatics Institute (EMBL-EBI), Hinxton, Cambridge, UK.

(2) Bioinformatics Group, Department of Computer Science, University of Freiburg, 79110, Freiburg, Germany.

(3) Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA/INRIA) - GenOuest platform, Université de Rennes 1, Rennes, France.

(4) Department of Pathology, University of Michigan, Ann Arbor, Michigan, USA.

(5) Center of Bioinformatics, Biostatistics and Integrative Biology Institut Pasteur Paris, France.

(6) Aarhus University, Bioinformatics Research Centre, C.F. Møllers Allé 8, Aarhus DK-8000, Denmark.

(7) Institut Français de Bioinformatique (Elixir-FR), UMS3601-CNRS, Université Paris-Saclay, Orsay, 91403, France

(8) Microbiology and Molecular Genetics, Michigan State University, East Lansing, MI, USA.

(9) ELIXIR Hub, Cambridge, CB10 1SD, UK.

(10) The Donnelly Centre, University of Toronto, 160 College Street, Toronto, Ontario M5S 3E1, Canada.

Introduction

The ability to reproduce the original results of scientific discovery has been one of the significant challenges in modern science. Evidence from multiple authors suggest that reproducibility in biomedical research is lower than 85% (Macleod et al., 2014), with 90% of researchers seeing science in reproducibility crisis (Baker, 2016). One of the fundamental principles of reproducible science is accurate and complete reporting of all experimental and computational steps required to obtain the described results. The ability to accurately reproduce a bioinformatics analysis, including data handling and statistical downstream processing, is one of the significant challenges in many scientific projects (Sandve, Nekrutenko, Taylor, & Hovig, 2013), this can be non-trivial especially for the computational part of an analysis. Different authors have pointed out three central premises for reproducible bioinformatics software deployment: (i) documenting the version of all software, (ii) open source the source code and all custom software, (iii) adopting a license and comply with third-party dependency licenses (Jimenez et al., 2017; Sandve, Nekrutenko, Taylor, & Hovig, 2013). Most of this work has focused on the openness and availability of the tools, software, scripts and data to perform the bioinformatics analysis (Perez-Riverol et al., 2016; Sandve, Nekrutenko, Taylor, & Hovig, 2013).

However, even if source code and data are published in a public repository, alongside the paper as open source supplementary material, the source code may have non-obvious dependencies on other software, configuration options, operating system and other subtleties that hamper straightforward re-usability (Boettiger, 2015). The build, installation and deployment of scientific software often require knowledge of what is missing in the published manuscript as well as the accompanying documentation. Also, workflows and pipelines commonly combine software developed by different teams and groups, adding another layer of complexity and introducing challenges such as compatibility and management of dependencies, running serial and parallel processes and working with a broad variety of software types and user-defined parameters. Software containers have emerged as a powerful technology to distribute and deploy scientific software and workflows (da Veiga Leprevost et al., 2017). Containers constitute lightweight software components and libraries that can be quickly packaged, are designed to run anywhere (da Veiga Leprevost et al., 2017), and are a useful and essential tool to leverage bioinformatics software reproducibility. Conda packages and Docker/Singularity containers are well-known technologies that have already gained traction in the field of bioinformatics. By May 2018, the BioContainers (da Veiga Leprevost et al., 2017), and Bioconda ("Bioconda: A sustainable and comprehensive software distribution for the life sciences," 2017) communities have released more than 4000 public containers, facilitating the development of complex and reproducible workflows and pipelines (Afgan et al., 2016; Pfeuffer et al., 2017).

This manuscript describes a core set of recommendations and guidelines to improve the quality and sustainability of research software based on software packages and containers. It provides easy-to-implement recommendations that encourage adoption of packaging (e.g. Conda) or containers (e.g. Docker, Singularity) technologies in bioinformatics and software development for research. It provides recommendations about making research software and its source code more reproducible, deployable, reusable, transparent and more compatible with other tools and software. In this manuscript, software is broadly defined to include command line tools, graphical user interfaces, application program interfaces (APIs), infrastructure scripts and software packages (e.g. R packages).

1. One tool, one container.

Microservice and modular architectures (Balalaie, Heydarnoori, & Jamshidi, 2016) provide a way of breaking large software projects into smaller, independent, and loosely coupled modules. These software applications can be viewed as a suite of independently deployable, small, modular components in which each tool runs a **unique** process and communicates through a well-defined, lightweight mechanism to serve a business goal (Balalaie, Heydarnoori, & Jamshidi, 2016). Each of these independent modules is referred to as a *container*. A container is essentially an encapsulated and immutable version of an application, coupled with the bare-minimum operating system components (e.g. dependencies) required for execution (da Veiga Leprevost et al., 2017).

Containers should be defined to be as granular as possible, with the premise *one Tool, one Container*. Each container should encapsulate only one piece of software that performs a unique task with a well-defined goal (e.g. sequence aligner, mass spectra identification).

2. A package first

A software package is self-contained software including all the dependencies libraries and packages necessary to execute the software. When choosing a package manager, it is important to select one that is cross-platform (e.g. works on various Linux flavors and MacOS), allows multiple versions of each package, and, ideally, is not restricted to a single programming language. Being cross-platform enhances reusability, providing for multiple versions enables reproducibility, and allowing multiple development languages simplifies usability. Some of the most popular and well-known package distribution system are Homebrew (https://brew.sh) and Conda (https://conda.io) (todo: aren't apt and yum also very popular, if not more?). *Conda*, the most popular package manager in research software, quickly installs, runs and updates packages and their dependencies. It handles dependencies for many languages such as C, C++, R, Java and of course Python tools. ~~ Additionally, *Conda* has joined to other popular packages manager systems such as Gentoo, BSD Ports, MacPorts, and Homebrew which build packages from source instead of installing from a pre-built binary. ~~ The field of computational biology has developed an active community around (Bioconda)[https://bioconda.github.io)].

Bioconda is a channel for the Conda package manager specialised in bioinformatics software. You can create a *Conda* package by defining a *BioConda* recipe (**Box 1**). This recipe (https://github.com/bioconda/bioconda-recipes) includes enough information about the dependencies, the LICENSE and fundamental metadata to find, retrieve and use the package (see **Recomendation 10**). Each package added to Bioconda is also made available as a Docker container via Quay.io, as a Singularity container and displayed in the BioContainers registry (da Veiga Leprevost et al., 2017). The BioContainers registry presents Docker containers as well as containers from other technologies such as rkt and Singularity (Kurtzer, Sochat, & Bauer, 2017) making it easier to find and discover packages and related containers.

```
package:
  name: deeptools
  version: '3.0.2'

source:
  fn: deepTools-3.0.2.tar.gz
  url:
https://files.pythonhosted.org/packages/21/63/095615a9338c824dcc1496a302d04267c674175f
0081e1ee2f897f33539f/deepTools-3.0.2.tar.gz
  md5: 4553d9c828ba4b5b93ca387917649281

build:
  number: 0

requirements:
  build:
    - python
    - setuptools
    - gcc
  run:
    - python
    - pybigwig >=0.2.3
    - numpy >=1.9.0
    - scipy >=0.17.0
    - matplotlib >=2.1.1
    - pysam >=0.14.0
    - py2bit >=0.2.0
    - plotly >=1.9.0
    - pandas

test:
  imports:
    - deeptools
  commands:
    - bamCompare --version

about:
  home: https://github.com/fidelram/deepTools
  license: GPL3
  summary: A set of user-friendly tools for normalisation and visualisation of deep-
sequencing data

extra:
  identifiers:
    - biotools:deeptools
    - doi:10.1093/nar/gkw257
```

Box 1: Bioconda recipe for "deeptools", a set of user-friendly tools for normalisation and visualisation of deep-sequencing data.

3. Tool and container versions should be explicit

The tool or software wrapped inside the container should be fixed explicitly to a defined version through the mechanism available by the package manager used (**Box 2**). The version used for this main software should be included in both, the metadata of the container (for ease of identification) and the container tag. The tag and metadata of the container should also include a versioning number for the container itself, meaning that the tag could look like `<version-of-the-tool>_cv<version-of-the-container>`. The container version, which does not track the tool changes but the container, should follow semantic versioning to signal its backward compatibility.

(todo: could we use an example that uses an stock image, at an explicit version, and where the container has been versioned using semantic versioning? I have many few examples.)

```
FROM biocontainers/biocontainers:latest ## should this not as well be versioned?

LABEL base_image="biocontainers:latest"

LABEL version="3"

LABEL software="Comet"

LABEL software.version="2016012"

LABEL about.summary="an open source tandem mass spectrometry sequence database search
tool"

LABEL about.home="http://comet-ms.sourceforge.net"

LABEL about.documentation="http://comet-
ms.sourceforge.net/parameters/parameters_2016010"

LABEL about.license_file="http://comet-ms.sourceforge.net"

LABEL about.license="SPDX:Apache-2.0"

LABEL extra.identifiers.biotools="comet"

LABEL about.tags="Proteomics"

LABEL maintainer="Felipe da Veiga Leprevost <felipe@leprevost.com.br>"

USER biodocker

RUN ZIP=comet_binaries_2016012.zip && \
  wget https://github.com/BioDocker/software-archive/releases/download/Comet/$ZIP -O
/tmp/$ZIP && \
  unzip /tmp/$ZIP -d /home/biodocker/bin/Comet/ && \
  chmod -R 755 /home/biodocker/bin/Comet/* && \
  rm /tmp/$ZIP

RUN mv /home/biodocker/bin/Comet/comet_binaries_2016012/comet.2016012.linux.exe
/home/biodocker/bin/Comet/comet

ENV PATH /home/biodocker/bin/Comet:$PATH

WORKDIR /data/
```

Box 2: BioContainers recipe for comet software. The metadata contains the license of the software.

If a copy is done via `git clone` or equivalent, a specific commit or a tagged version should be specified, never a branch only. Cloning a branch (master, develop, etc.) will always use the latest source code in that branch making impossible to reproduce the build process since the different

source code will be built as soon as the branch is updated by the software authors. Upstream authors should be asked to create a stable version of their software with reasonable guarantees that the specified version works as advertise including passing all automated tests — this will often be a *release* version. Any patches added on top of the officially released source code should be highlighted. For projects that practice agile software development (including continuous integration) where each version is stable, tested and works as advertised, the SVN or git identifier should be used as the tool version for the container — possibly with the addition of a date in YYYYMMDD format to easily identify newer versions from older versions.

4. Avoid using ENTRYPOINT

It is a well-known feature of Docker that the entry-point of the container can be over-written by definition (e.g., `ENTRYPOINT ["/bin/ping"]`). The **`ENTRYPOINT`** specifies a command that will always be executed when the container starts. Even when the `ENTRYPOINT` helps the user to get a *default* behaviour for a tool, it is generally not recommended because of reproducibility concerns of the implicit hidden execution point. By explicitly executing the tool by its executable inside the container (using the container as an environment and not as a fat binary merely through its `ENTRYPOINT`) the user (e.g. workflow) can recognise and trace the tool that is used within the container.

5. Relevant tools and software should be executable and in the PATH

If for some reason the container needs to expose more than a single executable or script (for instance, EMBOSS or other packages with many executables), these should always be executable and be available in the container's default PATH. This will, almost always, be the case by default for everything installed via package managers (dpkg, yum, pip, etc.), but if you are adding tailored made scripts or installing by source, take care of adding the executables to the PATH. This allows the container to be used as an environment (rule 4) or to specify alternative commands to the main `ENTRYPOINT` easily.

6. Reduce the size of your container as much as possible

As containers are frequently pushed and pulled (uploaded and downloaded) to/from container registries over the internet, their size matters. There are many tips to reduce the size of your container during builds:

- Avoid installing "recommended" packages in apt based systems.
- Do not keep build tools in the image: this includes compilers and development libraries that will seldom, if at all, be used at runtime when others are using your container. For instance, packages like GCC can consume several hundred megabytes. This also applies to tools like git, wget or curl, which you might have used to retrieve software during container build time, but are not needed for runtime.
- Make sure you clean caches, unneeded downloads and temporary files.
- In Dockerfiles, combine multiple `RUN`s so that the initial packages installations and the final deletions (of compilers, development libraries and caches/temporary files) are left within the same layer.
- If installing or cloning from git repositories, use shallow clones, which for large repositories will save a lot of space. (todo: the git repo is deleted in the same step, right? So why shallow clones? - there are many tools that don’t provide an installation process (Galaxy for instance), so for some of them the installation might be just a git clone. Also, using shallow clones will reduce download times during build time for large projects.)

7. Choose a base image wisely.

One of the decisions that will most likely impact on your final container image size will be your base image. If possible, start with a lightweight base image such as Alpine or similar, always using a fixed version and not the `⋯:latest` tag. If installing your software on top of such a minimal operating system doesn't work out well, only then use a more mainstream, yet minimal operating system base image where installation of the software tool might be more straightforward (such as Ubuntu, CentOS). Preferring mainstream base images means that others will be using the same base images and that your container will be pulled faster, as shared filesystem layers are more likely. Always aim to have predefined base images (for example, always use the same Alpine version as the first choice and always the same Ubuntu version as the second choice), so that most of your containers share the same base image.

8. Add functional testing logic

If others want to build your container locally, want to rebuild it later on with an updated base image, want to integrate it to a continuous integration system or for many other reasons, users might want to test that the built container still serves the function for which it was initially intended. For this, it is useful to add some functional testing logic to the container (in the form of a bash script for instance) in a standard location (here we propose a file called `runTest.sh`, executable and in the path) which includes all the logic for:

- Installing any packages that might be needed for testing, such as wget for instance to retrieve example files for the run.
- Obtain sample files for testing, which might be for instance an example data set from a reference archive.
- Run the software that the container wraps with that data to produce an output inside the container.
- Compare the generated output and exit with an error code if the comparison is not successful.

The file containing testing logic is not meant to be executed during container build time, so the retrieved data and/or packages do not increase the size of the container when it is built. However, because the testing file is inside the container, any user who has built the container or downloaded the container image can check that the container is working as intended by the author by executing `runTest.sh` inside the container.

9. Check the license of the software

When adding software or data in a container, always check the license of the resource being added. A free to use license is not always free to distribute or copy. License *must* always be explicitly defined in your Docker labels and depending on the license. You must also include a copy of the license with the software. The same care must be applied to included data. If a license is not specified, you should ask the upstream author to provide a license.

10. Make your package or container discoverable

Biomedical research and bioinformatics demands more efforts to make bioinformatics software and data more Findable (discoverable), Accessible, Interoperable, and Reusable (FAIR Principles) (Wilkinson et al., 2016). Leveraging those principles, we recommend to the bioinformatics community and software developers to make their containers and packages more findable. To make your package available, we recommend the following steps:

- Annotate packages and containers with metadata that allows users (e.g. biologists and bioinformaticians) to find them.
- Make packages and containers available. We recommend developers make the recipe of how to build a container available for others, including i) the source code or binaries of the original tools; ii) the configuration settings and test data.
- Register packages and container in existing bioinformatics registries helping users and services to find them. Registries such as BioContainers (da Veiga Leprevost et al., 2017), bio.tools (Ison et al., 2016), and Bioconda ("Bioconda: A sustainable and comprehensive software distribution for the life sciences," 2017) collaborate with each other by exchanging metadata and information using different APIs and a common identifier system.
- Deposit the built container image in a public container registry, such as Docker Hub, Quay.io or a publicly available and well supported institutional registry for container images.

11. Provide reproducible builds

While docker containers strive to make research reproducible and transparent, it is equally essential that the process of creating and building the docker containers themselves is transparent and reproducible. Many docker containers do not provide an associated Dockerfile, which would allow an independent party to reproduce and verify the container build independently. Other build procedures rely on the presence of specific web resources, download binary files from the internet or can only be built with in-house resources that are not available to the public. Furthermore, a poorly documented build process makes it harder to provide updated versions of the tool itself, leading people to rely on outdated versions of a tool or (in the worst case) the possibility of undetected tampering of the source code. Our recommendation is to provide clear documented steps on how to generate all the binaries directly from the source code. This documentation step not only relates to the distributed docker image but also the base image used and the procedure to generate any binary file that gets added to the container (preferably these files will be generated through a multi-stage build or in a different container whose Dockerfile is also available).

12. Document the build files

Adding documentation to Dockerfiles will allow the author as well as users to understand the build process and modify it their needs. This means describing the rationale for each RUN step and advising the user where additional information can be obtained. If a particular resource may not be readily available or consists of a binary file, provide further instructions on how to re-create this resource (e.g. a link to a second Dockerfile that creates the resource).

13. Provide helpful usage message

Usability and discoverability are crucial for packaged containers. If your tool provides a help `-h`, `--help` or `?` message, consider providing this as the default command `CMD` in the Dockerfile. If your tool does not provide a default usage message, consider providing this information in an ancillary `README.md` message. Your tool's help or usage message is a useful place to provide a list of commands in logical groups, along with each command, give a brief description, defaults, required arguments, and options. See example `git help` message

```
usage: git [--version] [--help] [-C <path>] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status

grow, mark and tweak your common history
   branch     List, create, or delete branches
   checkout   Switch branches or restore working tree files
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   merge      Join two or more development histories together
   rebase     Reapply commits on top of another base tip
   tag        Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
   fetch      Download objects and refs from another repository
   pull       Fetch from and integrate with another repository or a local branch
   push       Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

Conclusions

This manuscript promotes and encourages adoption of package/container technologies to improve the quality and reusability of research software. The recommendations share a set of core views that are summarised below:

- *Simplicity*: the encapsulated software should not be a complex environment of dependencies, tools and scripts.
- *Maintainability*: the more software included in the container, the harder it is to maintain it, especially when the software comes from different sources.
- *Sustainability*: the developers of the software should be engaged or made aware of supporting the sustainability of the container.
- *Reusability*: a tool container should be safe to reuse by any other workflow component or task through its access interface.
- *Interoperability*: different tools should be easy to connect and exchange information.
- *User's acceptability*: tool container should encapsulate domain business process units, so it can be easier to check and use.
- *Size*: containers should be as small as possible. Smaller containers are much quicker to download and therefore they can be distributed to different machines much quicker.
- *Transparency*: containers should be transparent in how they are built, which tasks they are designed to perform and how the build process can be reproduced.

For users involved in scientific research and bioinformatics interested in this topic without experience working with software packages or containers, we recommend to explore and engage with the BioContainers initiative (da Veiga Leprevost et al., 2017). As with many tools, a learning curve lays ahead, but several basic yet powerful features are accessible even to the beginner and may be applied to many different use-cases. To conclude, we would like to recommend some examples of bioinformatics containers in BioContainers (Table 1) and some useful training materials, including workshops, online courses, and manuscripts (Table 2).

References

Afgan, E., Baker, D., van den Beek, M., Blankenberg, D., Bouvier, D., ?ech, M., … Goecks, J. (2016). The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update. *Nucleic Acids Res.*, *44*(W1), W3–W10.

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. *Nature*, *533*(7604), 452–454.

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, *33*(3), 42–52. https://doi.org/10.1109/MS.2016.64

Bioconda: A sustainable and comprehensive software distribution for the life sciences. (2017). *BioRxiv*. https://doi.org/10.1101/207092

Boettiger, C. (2015). An Introduction to Docker for Reproducible Research. *SIGOPS Oper. Syst. Rev.*, *49*(1), 71–79. https://doi.org/10.1145/2723872.2723882

da Veiga Leprevost, F., Gruning, B. A., Alves Aflitos, S., Rost, H. L., Uszkoreit, J., Barsnes, H., … Perez-Riverol, Y. (2017). BioContainers: an open-source and community-driven framework for software standardization. *Bioinformatics*, *33*(16), 2580–2582.

Ison, J., Rapacki, K., Menager, H., Kala?, M., Rydza, E., Chmura, P., … Brunak, S. (2016). Tools and data services registry: a community effort to document bioinformatics resources. *Nucleic Acids Res.*, *44*(D1), 38–47.

Jimenez, R. C., Kuzak, M., Alhamdoosh, M., Barker, M., Batut, B., Borg, M., … Crouch, S. (2017). Four simple recommendations to encourage best practices in research software. *F1000Res*, *6*.

Kurtzer, G. M., Sochat, V., & Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PLoS ONE*, *12*(5), e0177459.

Macleod, M. R., Michie, S., Roberts, I., Dirnagl, U., Chalmers, I., Ioannidis, J. P., … Glasziou, P. (2014). Biomedical research: increasing value, reducing waste. *Lancet*, *383*(9912), 101–104.

Perez-Riverol, Y., Gatto, L., Wang, R., Sachsenberg, T., Uszkoreit, J., Leprevost, F. d. a. V., … Vizcaino, J. A. (2016). Ten Simple Rules for Taking Advantage of Git and GitHub. *PLoS Comput. Biol.*, *12*(7), e1004947.

Pfeuffer, J., Sachsenberg, T., Alka, O., Walzer, M., Fillbrunn, A., Nilse, L., … Kohlbacher, O. (2017). OpenMS - A platform for reproducible analysis of mass spectrometry data. *J. Biotechnol.*, *261*, 142–148.

Sandve, G. K., Nekrutenko, A., Taylor, J., & Hovig, E. (2013). Ten simple rules for reproducible computational research. *PLoS Comput. Biol.*, *9*(10), e1003285.

Wilkinson, M. D., Dumontier, M., Aalbersberg, I. J., Appleton, G., Axton, M., Baak, A., … Mons, B. (2016). The FAIR Guiding Principles for scientific data management and stewardship. *Sci Data*, *3*, 160018.