Title: Recomendations to packaged and contenarized bioinformatics software

Authors: Yasset Perez-Riverol (1), Felipe da Veiga Leprevost (2), Pablo Moreno (1), Hervé Ménager (3), Olivier Sallou (4), Dan Søndergaard (5), Michael R. Crusoe (6), Bjorn Grunning (7), Rafael C Jimenez (8)

Affiliation:

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

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

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

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

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

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

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

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

Introduction

The ability to reproduce the original results of a scientific discovery has been one of the mayor challenges in modern science. Evidences from multiple authors suggest that reproducibility in biomedical research is lower than 85% [PMID: 24411643], while 90% of researchers agree in a reproducibility crisis [PMID: 27225100]. One of the major drawbacks is to be able to reproduce the bioinformatics analysis, including the data processing and statistical downstream analysis [PMID: 24204232]. Different authors has pointed three main premises for reproducible bioinformatics software deployment: (i) the use of exact versions of software and tools, (ii) open source of the code and all custom software, (iii) adopt a licence and comply with the licence of third-party dependencies [PMID: 24204232, 28751965]. Most of these works has been focussed in the openness and availability of the tools, software, scripts and data to perform the bioinformatics analysis [PMID: 24204232, 27415786].

However, even if source code and data are published in a public repository (e.g Github) alongside the paper as open source artifacts, they come with many dependencies, configurations, versions that make their use hard to achieve [DOI:10.1145/2723872.2723882]. The build, installation and deployment of the bioinformatics solution (e.g. software or workflow) often requires internal knowledge that is missing from the published manuscript. In addition, most of the bioinformatics software is developed by different teams but used in combination with workflows, scripts or pipelines. This adds an additional layer of complexity introducing challenges such as compatibility and management of dependencies; running serial and parallel processes; and working with a broad variety of software data types and user-defined parameters.

Software containers has emerged as a powerful technology to document, distribute, and deploy bioinformatics software and workflows [PMID: 28379341]. Containers are lightweight software components and libraries easily packaged designed to run anywhere [PMID: 28379341], thus useful to leverage bioinformatics software reproducibility. Conda packages, Docker and Singularity are containers technologies applicable to the field of bioinformatics. The BioContainers and BioConda communities have released more than 3000 public containers [PMID: 28379341] facilitating the development of complex and reproducible workflows and pipelines [PMID: 28559010, PMID: 27137889].

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 recomendations 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 [DOI: 10.1109/MS.2016.64] is a way of breaking large software projects into smaller, independent, and loosely coupled modules. This software applications can be seen 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 [DOI: 10.4103/2153-3539.194835]. Each of these independent modules are 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 [PMID: 28379341].

Containers should be defined as most granular as possible with the premise *one Tool, one Container*. Each container should encapsulate only one piece of software, tool that perform a unique task with a well-defined goal (e.g. sequence aligner, mass spectra identification). In order to design the level of *granularity*, different metrics should be considered:

- *Simplicity*: the encapsulated software should not be a complex environment of dependencies, tools and scripts.
- *Maintainability*: the more software is included to the container the harder it is to maintain specially when the software comes from difference sources.
- *Sustainability*: the developers of the software should be engaged or made aware to support 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 as big as necessary. Smaller containers are much quicker to download and therefore they can be distributed to different machines much quicker. `RAFA: I do not know what we mean by attack surface. Please correct or remove the next sentence->`Less code/less programs in the container means less attack surface.

2. A package first

A software package is self-contained software including all the dependencies libraries and packages necesary to excecute the software. Some of the most popular and well-known packaging distribution system are homebrew (https://brew.sh/) or conda (https://conda.io/). *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. In addition, *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 built an strong community (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 X**). Each package added to Bioconda is also made available as a Docker container via Quay.io and displayed in the BioContainers registry [PMID: 28379341]. The BioContainers registry displays Docker containers as well as containers from other technologies such as rkt and Singularity [PMID: 28494014] making easier to find a discover packages and related containers.

3. Versions should be explicit, and consider both the tool version and the container version

The tool or software wrapped inside the container should be fixed explicitly to a defined version through the mechanism available by the package manager or install method used. The version used for this main software should be included in both, the metadata of the container (for findability reasons) 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 be versioned through semantic versioning to signal its backward compatibility.

If a copy is done via a git clone or equivalent, a specific tagged version should be copied, never the *latest* branch. Cloning the latest branch will copy the latest code making difficult to reproduce an operation since the latest branch might suffer constant changes. Upstream authors should be asked to create a release if not available. In the worst case, the HEAD commit id of the clone should be used as the tool version for the container.

4. Eschew ENTRYPOINT

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 be mostly always the case by default for everything that is installed via a package manager (apt-get, pip, R install.package, etc.) and you won't need to worry if that is the case, but if you are adding tailored made scripts or others that are not installed through a manager, then you need to take care of this. This will facilitate the use of the container as an environment or to specify alternative commands to the main entrypoint easily.

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

Since containers are being constantly being pushed and pulled over the internet, their size matters. There are many tips to reduce the size of your container in build time: - Avoid installing "recommended" packages in apt based systems. - Do not keep build tools in the image: this includes

compilers and development libraries that will seldomly, if not at all, used in runtime when your container is being used by others. For instance, packages like gcc can use several hundred megabytes. This also applies to tools like git, wget or curl, which you might have used to retrieve software during container buildtime, but are not needed for runtime. - Make sure you clean caches, unneeded downloads and temporary files. - In Dockerfiles, combine multiple RUNs 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 a git repo, use shallow clones, which for large repos will save a lot of space.

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 you can, start with a lightweight base image such as Alpine or similar, always at a fixed version. If installing your software on top of such a minimal operating system doesn't work out well, only then move to a larger, stock-image where installation of your tool software might be simpler (such as Ubuntu). Preferring stock images means that many other people will be using them and that your container will be pulled faster as shared layers are more likely. Always aim to have predefined images from where you choose (always the same Alpine version as first choice and always the same Ubuntu version as second choice), so that most of your containers share that base image.

8. Add some testing logic

If others want to build locally your container, want to rebuild it later on with an updated base image, want to integrate it to a continuous integration system for building it or for many other reasons, users might want to test that the built container still serves the function for which it was originally designed. For this is useful to add to the container some testing logic inside it (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 and output inside the container. - Compare the produced output and exit with an error code if the comparison is not successful. This file containing the testing logic is not meant to be executed during container buildtime, so the retrieved data/packages don't increase the size of the container when this is executed once the container is built. This means that, because the file is inside the container, any user who has built the container or downloaded the container image can check that the container is working adequately by executing `runTest.sh` through the container.

9. Check the license of the software

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

10. Make you package or container findable