The Next Generation Sequencing Quality Initiative

The Next Generation Sequencing (NGS) Quality Initiative is a collaboration between the Centers for Disease Control and Prevention (CDC), the Association of Public Health Laboratories (APHL), and state and local public health laboratories (PHLs) to address the many challenges laboratories encounter when implementing NGS-based assays. The Initiative is developing an NGS-focused quality management system (QMS) to assure foundational quality during the development and implementation of sequencing-based tests by providing customizable, ready-to-implement tools and resources that laboratories can use to standardize and institute quality management practices and procedures. The NGS Quality Initiative has published additional tools and resources, including templates and procedures, that may be of assistance to laboratories throughout their NGS workflow. Please visit the following website to access these resources: <https://www.cdc.gov/labquality/qms-tools-and-resources.html>.

This document is intended to be used as a tool for implementing, improving, or maintaining an NGS QMS. Blue text provides examples for appropriate input and can be changed, deleted, or augmented as needed for the laboratory’s specific requirements.

These documents and tools are not controlled files; format and content **must** be modified as needed to meet the document control, QMS, or regulatory requirements within your laboratory. It is the responsibility of your laboratory to take any necessary actions to ensure the information within these documents remains applicable.

**Disclaimer:**

**Use of trade names and commercial sources is for identification only and does not imply endorsement by the U.S. Centers for Disease Control and Prevention or by the U.S. Department of Health and Human Services.**

|  |
| --- |
| ***Insert Laboratory-Specific Name Here*** |
| **Docker Container Requirements, Installation, and Execution**  *Standard Operating Procedure* |

# Purpose

* 1. The objective of this document is to provide guidance on Docker, which is a commonly used container program in research computing. This procedure functions to do the following:

1. Orient first-time users to Docker resources.
2. Define and explain key terms.
3. Summarize Docker features and functions.
4. Assess the program’s data privacy and security protections.
   1. Containerization has been addressed in the Containerization Purpose and Approaches document and should be consulted for further information on the purpose of containers and approaches to their creation/use.

# Scope

This document describes the requirements, installation, and execution of Docker and contains information about Docker resources. Broad topics within Docker’s functions and options are summarized, and important terms are defined.

# Related Documents

|  |  |
| --- | --- |
| **Title** | **Document Control Number** |
| Bioinformatics Pipeline Containerization Purpose and Approaches | *[insert laboratory-specific document control number here]* |

# Definitions

|  |  |
| --- | --- |
| **Term** | **Definition** |
| Bind mount | A filesystem on the host machine used by Docker to store container output |
| Container | An image that is currently running and editable changes will not be saved to the image |
| Dockerfile | A file that contains the commands needed by Docker to build an image |
| Image | The complete virtual reproduction of a computer storage device |
| Base image | An image that starts from the minimal image “scratch” (i.e., is built from a Dockerfile that starts with “FROM scratch”) |
| FROM | The FROM instruction specifies the parent image onto which further information will be built. FROM may only be preceded by one or more ARG instructions, which declare arguments that are used in FROM lines in the Dockerfile. |
| Parent image | An image called by FROM in the Dockerfile, which is modified to create the new image |
| Layer | One of a series of objects or commands in an image, as specified in the Dockerfile |
| Port | A process or service defined within a computer operating system out of which information flows |
| Registry | A storehouse of images, which can be either public (e.g., Docker Hub, Google, Amazon Web Services (AWS)) or private (cloud or on-site, using Docker Trusted Registry or home-built) |
| Volume | A filesystem managed by Docker and used to store container output, sometimes from multiple containers; volumes can be “named” (having a specified source container) or anonymous (without a specified source) |
| Tarball | A jargon term for a tar archive (.tar file), which is a group of files collected together as one |

# Procedure

* 1. Docker is open source; local versions can be installed on Linux, Windows, or Mac operating systems. Requirements, instructions, and downloads can all be accessed [here](http://docs.docker.com/get-docker).
     1. On Linux, “Docker Engine” is installed directly. This is the program that runs containers and is bundled with other Docker programs in the Docker Desktop program for MacOS and Windows.
     2. Docker Engine packages are available for many common Linux distributions.
     3. Docker supports Docker Desktop for Windows and the installation can be found [here](https://docs.docker.com/desktop/install/windows-install/).
     4. Docker supports Docker Desktop for Mac on the most recent versions of macOS [here](https://docs.docker.com/desktop/install/mac-install/).
  2. The [home page](http://docs.docker.com/) of Docker Docs offers a comprehensive collection of resources, including program downloads, installation guides, FAQs, videos, community platforms, manuals, news, and developer guides.
  3. Docker Labs
     1. Docker Labs can be accessed by right-clicking on the Docker icon in the System Tray, then going to the “Learn” menu option.
     2. A tutorial then opens in a Docker console, with one-click functionality for downloading a Dockerfile, using a Dockerfile to build a container in which a tutorial program is run, and saving the built image to Docker Hub.
     3. The container is shown in the Docker console and can be opened in a browser.
     4. In this “Getting Started” container, steps are provided to create, share, and run another container, which holds a JavaScript program that generates an interactive To-Do list manager.
  4. The State Public Health Bioinformatics (StaPH-B) community has written a User Guide for Docker, which can be found [here](https://staph-b.github.io/docker-builds/).

# Dockerfiles

* 1. A Dockerfile is a simple text file (saved without an extension and usually named “Dockerfile”) that instructs Docker on the software, dependencies, and data needed to replicate a particular computing environment.
  2. The Dockerfile is executed to build another, much larger file, or “image,” which houses everything specified by the Dockerfile.
  3. The environment generated by the image can be used on a different machine or on the same machine that initially generated the environment.
  4. The Dockerfile instructs the creation of layers and temporary images, on top of which is a new, writeable layer, also known as the container layer.
  5. For example, the following Dockerfile creates a temporary image from Python 3, then a layer by adding the script (script\_1.py), then a layer from running PyBio (a Python library required by the script), and finally a temporary, interactive layer from the execution of the script:

FROM python:3

ADD script\_1.py /

RUN pip install pyBio

CMD [ "python", "./script\_1.py" ]

* 1. Dockerfiles use a limited number of instructions. Only the commands RUN, COPY, and ADD create layers that increase the size of a build. The most commonly used commands are as follows:

1. **FROM** – All Dockerfiles need to start with this command, which specifies a parent image that is subsequently modified to create the new image. In some cases, users may want to specify the complete contents of the starting image, in which case the Dockerfile begins with the minimal Docker image “scratch.” The Dockerfile will begin with “FROM scratch” and build what is called a “base image.”
2. **WORKDIR** – sets directory for command execution.
3. **VOLUME** – enables container access to a host directory.
4. **ENV** – sets environmental variables.
5. **COPY** – copies files and directories from host to container filesystem.
6. **ADD** – copies files, directories, url sources, and tar files from host to container filesystem.
7. **LABEL** – specifies metadata for Docker objects.
8. **EXPOSE** – sets a port for networking with container, which can be overridden at runtime with new instructions.
9. **USER** – sets the user ID that runs the container.
10. **ENTRYPOINT** – specifies the executable to be run when the container is started.
11. **RUN** – executes a command in a new layer.
12. **CMD** – executes command(s) within the container, which can be overwritten from the command line.
    1. A file named “.dockerignore” can be placed in the working directory alongside the Dockerfile to indicate which files and directories should be ignored when building the image. Wildcards help when asking Docker to ignore all files of a particular type. (e.g., .pyc).
    2. Best Practices for writing Dockerfiles (i.e., Docker Docs) – Docker focuses on speed, storage efficiency, and mobility in the use of containers, and there are several best practices that ensure these focuses are achieved when creating Dockerfiles:
       1. Create ephemeral containers – use Dockerfiles to create containers that can be “stopped and destroyed, then rebuilt and replaced,” as often as possible.
       2. Understand build context – “Build context” refers to the current working directory, which is usually the same as the Dockerfile location but can be specified elsewhere.
       3. Pipe Dockerfile through “stdin” – Standard input (stdin) refers to an input stream, and Docker can build a container without an actual Dockerfile. This can be achieved by specifying the path with a dash and specifying the Dockerfile instructions in the command line.
       4. Exclude undesired files and directories with .dockerignore.
       5. Use multi-stage builds – A relatively new feature in Docker, multi-stage builds enable an efficient workflow in a Dockerfile. The output from one application is used to continue the build using another application; previously, this required two Dockerfiles and a more elaborate build command.
       6. Do not install unnecessary packages.
       7. Decouple applications – Putting different applications into different containers allows the containers to be reused, which facilitates container management and modularity.
       8. Minimize the number of layers – By employing multi-stage builds and using RUN, COPY, and ADD only as necessary, the size of builds is minimized.
       9. Sort multi-line arguments – Putting several arguments on different lines and alphabetizing them helps avoid the mistake of duplicating packages and makes Dockerfiles easier to review.
       10. Leverage build cache – Unless instructed otherwise, Docker will look through images in the existing cache for reuse rather than creating new images. This is often inefficient, because validating a cache match can take time.

# Images

1. 1. Dockerfiles are executed through the “build” command to produce an exact replica of a desired computing environment that can be saved in a single file. This file is similar to a virtual machine but without the kernel and OS; it is referred to as an “image.” Because image files contain applications, dependencies, and possibly data, they can be very large.
   2. The command “docker build” does not need to specify the Dockerfile if executed in the same directory, and the resulting image is named using “-t.” A variety of other options can be specified, including the removal of intermediate containers, setting image metadata, setting memory limits, and designating an output location.
   3. The docker image history command can be run to see the command used to create each layer within the image; the “--no-trunc” flag prevents truncation of longer lines. The final image file can then be run as a container or shared to be run elsewhere.

# Running Containers and Saving Output

* 1. A running image is known as a “container.” Containers can be interactive, new layers can be added, and new data can be generated in the writable layer. The changed image can be saved as a tarfile (also called a tarball). New layers will increase the size of the file; those layers can be rolled back when the image is run again.
  2. If the container is exported to a tarball (as opposed to saved), the resulting file lacks history and metadata, and rolling back added layers from the original image cannot be done. This functionality is demonstrated in the walk-through exercise in Appendix A.
  3. The preferred way to save data after a container is closed is in a “volume.” A volume is a filesystem completely managed by Docker that exists outside of a container.
  4. Historically, Docker used “bind mounts” (which are file systems on the host machine) to store data. Volumes, however, can be at any networked location (including the cloud), encrypted, managed using Docker utilities, and more safely shared between/among multiple containers.
  5. “Named” volumes have a specific source and are deleted when that container is deleted; meanwhile “anonymous” volumes can be mounted by any container and persist after the container is deleted.

# Networking

* 1. Docker will create networks through which containers can communicate with each other, with the host machine, and with the outside world. Networks can be seen using the command “docker network is,” and the details of any one network can be examined with “docker network inspect [network name or ID].”
  2. The default network through which containers can communicate to each other (using their IP addresses) is through Docker’s “bridge network,” and containers can be added to this as needed.
  3. Users can also define their own networks and associated containers using the Docker Compose tool. Networked containers can communicate with each other using their aliases (see Appendix B).
  4. Users should be aware of certain Docker settings. These can be seen in the Settings panel of the Docker Desktop Dashboard.
  5. The first setting to note is the “File Sharing” list, which allows users to add host machine locations that can be mounted by Docker containers. The second is the "Expose daemon on tcp://localhost:XXXX without TLS” option, which has risks and is unchecked by default.

# Sharing

* 1. To fulfill their intended purpose of facilitating the replication of computational research projects, images need to be shared. This is best done by adding them to a “registry,” which is a user’s collection of images accessible by other users via the internet.
  2. There are several registry options. With an account on Docker Hub, users can easily “push” images to it and organize them into various “repositories.”
  3. However, there are other public registry services, such as Google and AWS, and in many cases Docker images need to be private and stored behind a firewall. In such cases, users can establish a protected cloud-based registry.
  4. The Docker Hub [site](http://hub.docker.com/u/staphb) is where images built by members of the StaPH-B community are shared.
  5. There may be hurdles to sharing full images, as they may contain proprietary software and/or sensitive data. For these reasons, it is common to simply share Dockerfiles (sometimes also called “builds”).
  6. The Github [site](http://github.com/StaPH-B/docker-builds) is where Dockerfiles written by members of the StaPH-B community are shared.
  7. Images built from the same Dockerfile at different times will differ with respect to versions of added software, so only full images or saved tarball files should be used to best replicate an original computing environment.
  8. Docker is not considered appropriate for use on high-performance computing (HPC) systems, as it has access to the system root, which makes it an exploitable pathway for malfeasance. However, Singularity addresses container related security issues. Docker images are easily ingested and converted by Singularity and other applications designed for HPC.

# General Docker Best Practices (Boettiger, 2015)

Boettiger (2015) listed best practices for Docker, which are summarized below. They emphasize reproducibility using Docker throughout workflow development and archiving containers regularly.

* 1. Use Docker containers during development:
     1. If a researcher begins the creation of a workflow from within a container, code will appear to run natively, but the computational environment and processes can be reproduced, or imaged and shared, with only a few commands.
  2. Write Dockerfiles instead of installing interactive sessions.
  3. Add tests or checks to the Dockerfile:
     1. Dockerfiles are usually used to describe the installation of software, but they can also contain commands for executing it once installed. This acts as a check that installations have been successful and that the software is ready to use.
  4. Use and provide appropriate base images:
     1. Docker is highly amenable to modular workflows, and when successful environments have been established and containerized, it is efficient to reuse them as needed for new project.
  5. Share Docker images and Dockerfiles.
  6. Archive tarball snapshots:
     1. Saved images can revert back to original layers through the preservation of historical information. However, layers cannot revert back to earlier versions of themselves that were used in previous builds of images from the same Dockerfile. Therefore, saving containers as .tar files (i.e., tarballs) in different runs of the same image is important to test whether software updates change results.

# Appendices

* 1. Appendix A – Running Mash 2.2 in Docker on Windows
  2. Appendix B – Running Three Networked Containers with a Shared Host Folder

# References

1. Boettiger, C., 2015. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1), pp.71-79.

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev #** | **DCR #** | **Change Summary** | **Date** |
| *[insert laboratory-specific revision number here]* | *[insert laboratory-specific document change request number here]* | *[insert change summary here]* | *[insert revision date here]* |

# Approval

Approved By:  Date: 

Author

Print Name and Title

Approved By:  Date: 

Team Lead/Supervisor



Print Name and Title

Approved By:  Date: 

Quality Manager



Print Name

**Appendix A – Running Mash 2.2 in Docker on Windows**

1. Below is a detailed description of the steps needed to build and run a Docker container on a local machine with Windows 10 or latest Windows 11.
2. It is assumed that Docker is already installed, a process that is described on the Docker [website](http://docs.docker.com/docker-for-windows/install/) and which may require permissions from an IT administrator.
3. In this walk-through exercise, users will do the following:

* Download a Dockerfile for the program Mash V2.2 from the StaPH-B GitHub site.
* Download data from the Mash website.
* Build an image from the downloaded Dockerfile.
* Test that the image build was successful.
* Run the image to create an interactive container.
* Copy downloaded data from the host machine to the container.
* Conduct a simple analysis in Mash inside the container.
* Copy this output to the host machine.
* Save the output in the container and save this to a new image in Docker.
* Save the new image with data as a tar file on the machine.
* Create an image from the tar file, run the image, and confirm that the data and output are there.

1. Mash is a bioinformatics program that calculates the distance, or degree of difference, between two genomes.
2. It is aimed for use in metagenomics and the massive sequence collections now available, and it allows efficient searching and clustering of sequences. It does this in part by creating “sketches” of genomes first, which drastically reduces the sizes of genome files and speeds genome comparisons.
3. **Download the Mash 2.2 Dockerfile from the StaPH-B community site on GitHub:**
   1. [<http://github.com/StaPH-B/docker-builds/tree/master/mash/2.2/Dockerfile>](https://github.com/StaPH-B/docker-builds/blob/master/mash/2.2/Dockerfile)
   2. The code here can be copied into a text file named “Dockerfile” (without an extension), or one can right-click on the “Raw” button and save the file without an extension.
   3. The contents of the Dockerfile are below (with the maintainer information redacted). Comments are indicated by the pound sign (#), and it starts with a Linux (Ubuntu) base image.

# base image

FROM ubuntu:xenial

# metadata

LABEL base.image="ubuntu:xenial"

LABEL container.version="1"

LABEL software="Mash"

LABEL software.version="2.2"

LABEL description="Fast genome and metagenome distance estimation using MinHash"

LABEL website="https://mash.readthedocs.io/en/latest/index.html"

LABEL license="https://github.com/marbl/Mash/blob/master/LICENSE.txt"

LABEL maintainer="Maintainer Name"

LABEL maintainer.email="username@cdc.gov"

# install dependencies

RUN apt-get update && \

apt-get -y install wget && \

apt-get clean

RUN wget https://github.com/marbl/Mash/releases/download/v2.2/mash-Linux64-v2.2.tar && \

tar -xvf mash-Linux64-v2.2.tar && \

rm -rf mash-Linux64-v2.2.tar && \

mkdir /data

# add mash to path, and set perl locale settings

ENV PATH="${PATH}:/mash-Linux64-v2.2" \

LC\_ALL=C

WORKDIR /data

# make db dir. Store db there. Better to have db's added in the last layers

RUN mkdir /db && \

cd /db && \

wget https://gembox.cbcb.umd.edu/mash/RefSeqSketchesDefaults.msh.gz && \

gunzip RefSeqSketchesDefaults.msh.gz

1. **Download data from the Mash website:**
2. <https://gembox.cbcb.umd.edu/mash/genome1.fna>
3. <https://gembox.cbcb.umd.edu/mash/genome2.fna>
4. **Open Docker:**
5. Search for Docker Desktop among applications and then Open.
6. Find the Docker icon in the System Tray.
7. Right-click on the Docker icon and open the Dashboard.
8. **Open Command Prompt (a.k.a., Terminal or Console).**
9. **Build an image from the Dockerfile:**
10. In Terminal navigate to the same folder as the Dockerfile.
11. Enter docker build -t mash2.2.
12. The period at the end of the command tells Docker that one will start in the current directory, so it will use the file named “Dockerfile” there. If the Dockerfile has a different name, the option -f can be used to specify it (e.g., docker build -t mash2.2 -f Dockerfile-2).
13. This builds an image named “mash2.2” and puts in the default directory for images.
14. The image, or virtual hard drive, for Docker itself can be found in the settings in the Docker Dashboard, and it is usually in a hidden folder in the root (e.g., C:\ProgramData\DockerDesktop\vm-data).
15. Docker images are put inside Docker’s virtual hard drive.
16. **See the new Docker image by entering docker images:**
17. The newly built image lists “mash2.2” under the “Repository,” although it will be treated like a name. Alternatively, it is possible to add a tag to the image by putting it after a colon in the tag function when building the image (e.g., docker build -t mash2.2:new .).
18. The image has an ID number, which can also be used to run or save the image.
19. Adding -a after the images command makes it possible to see intermediate images downloaded while building the layers of the final images (e.g., docker images -a).
20. Images can be removed using the command rmi and the image name or ID. For example, to remove the image just built, enter docker rmi mash2.2.
21. **Run the image by entering docker run -it mash2.2. If no tag is specified, Docker will default to “mash2.2:latest,” but if it was assigned the tag “new” as per step 6a above, then the following command should be entered: docker run -it mash2.2: new.**
22. This creates a container in which Mash can be used.
23. All containers can be seen in the Docker Dashboard, where they can also be run, stopped, deleted, opened in a port, or interacted with in a command line window. Currently running containers are green.
24. The container is given a randomly generated name by Docker unless you use the --name option, e.g., docker run -it --name mash2.2container mash2.2. It is necessary to use “mash2.2container” to refer to this container from here.
25. The -it option runs the container interactively, meaning it generates a command line interface for the user to give it instructions and produce output. This particular container can only run interactively and immediately stops running without this option.
26. A command line for the container now starts in Terminal (root@[containerID]:/data#), or you can use the command line available through the Docker Dashboard (the “CLI” button to the right of the container). To return to the host computer in Terminal, stop the container in the Dashboard and restart it. To run the image while retaining control of the Terminal for other commands, use the -d (detached) option with docker run. (e.g., docker run -it -d --name mash2.2container mash2.2).
27. Commands in this container follow basic Linux/Unix commands, unlike Terminal on the host, which require Command Prompt commands.
28. All containers can also be viewed in Terminal by entering docker ps -a.
29. By omitting the -a option, only running containers are displayed.
30. **Test that the Mash image was built properly by checking that the large file “RefSeqSketchesDefaults.msh” inside the Mash directory was downloaded without any changes.**
31. Do this by using the program “md5sum,” which calculates a mathematical description of the file’s contents and compares it to the same calculation done when the file was made and provided with the file. Enter the following commands inside the container:

md5sum /db/RefSeqSketchesDefaults.msh > hash.md5

md5sum -c hash.md5

1. This should result in the following message: /db/RefSeqSketchesDefaults.msh: OK.
2. **Copy the Mash data downloaded earlier to the container be entering the following command in Terminal, replacing “[container name]” with the name of the container (whether given by the “—name” option when running the image, as previously mentioned, or the random name created by Docker) and repeating it for the second data file (genome2.fna):**

docker cp Desktop/app/genome1.fna mash2.2container:/db/genome1.fna

1. **Now compute the distances between the first two genomes using Mash. These are Escherichia coli genomes, and they can be compared using this command:**

mash dist genome1.fna genome2.fna

1. The output will look like this:

genome1.fna genome2.fna 0.0222766 0 456/1000

and it shows the names of the two files being compared, then a standardized distance measure called a “Mash distance,” a p-value associated with seeing this particular distance by chance, and the number of matching hashes (pseudo-random identifiers).

1. **Now “sketch” the genome first to speed the comparison. This generates “.msh” files and stores them in the container’s directory “db.”**
2. Enter the following commands inside the container, which should produce the same output as above, only slightly faster:

mash sketch genome1.fna

mash sketch genome2.fna

mash dist genome1.fna.msh genome2.fna.msh

1. Now check that the .msh files have been generated by moving out of the “data” directory (the default directory where the command line started), and then navigating down to the “db” directory. To perform this, enter the following commands:

cd ..

cd db

dir

1. These mash files are inside the container’s Mash directory inside Docker’s virtual hard drive, so it is not possible to share these files unless they are copied to the host machine.
2. This is performed by using the following command in Docker (repeated for each file): docker cp mash2.2container:/db/genome1.fna.msh /Users/username/
3. The output will be unreadable on the host machine, since it is not running Mash, but they are visible and it’s possible to determine that content is present. The files can now easily be shared with colleagues.
4. If the container is stopped and deleted now, the imported data and the sketches produced from them will be lost unless they have been copied to the host machine; they will not appear when the newly built image is run again. However, the new changes can be committed to a new image by running the following command: docker commit mash2.2container mash2.2\_with\_data.
5. As always, the container name can be replaced by its ID, which can be seen by entering docker ps -a in the Terminal. Choose a name for the new image name. Here it is simply referred to as “\_with\_data.”
6. The new image will now appear in the list of images (seen in Terminal with docker images), and it can be run again like any other image, except it will have the data and output generated earlier.
7. Save the new image with the added data as tar file, or tarball, on the host machine, where it can be shared. Images are inside the Docker virtual hard drive, but Docker saves tar files to the User folder inside the machine’s file system. This is done by entering the following command in Terminal: docker save mash2.2\_with\_data mash\_w\_data.tar].
8. If you are in your User folder (/Users/your\_user\_name), then type dir to see its contents, including your new tar file.
9. If a tar file is received that was generated by someone else, it needs to be brought into the available images in Docker before it can be run, and this is done using the load command. The tar file will be moved into Docker’s virtual hard drive with the other images and given the original image name. This can be tested with the newly created tar file, using the following steps:
10. Delete previous containers. Containers can be easily stopped and deleted in the Docker Dashboard using the stop and trash icons, or by entering docker rm --force [container name] in Terminal.
11. Delete images using the docker rmi [image name] command (see section 6d above) in Terminal. Confirm this by entering docker images.
12. Enter the command docker load -i mash\_w\_data.tar.
13. Enter docker images. This will display “mash2.2\_with\_data” as an available image.
14. Run this image (docker run -it mash2.2\_with\_data), and inside the running container repeat steps 11b above. This will demonstrate that the imported data and generated outputs are in the db folder as before.

**Appendix B – Running Three Networked Containers with a Shared Host Folder**

1. Below is an exercise that will use Docker Compose to run three containers in a networked environment, analyze a small DNA sequence data set, and use a shared host directory to pass data and results among the host and containers.
2. In this walk-through exercise, perform the following:

* Create a directory structure to store data, Dockerfiles, and output.
* Download DNA sequences of 31 SARS-CoV-2 isolates.
* Download Dockerfiles MAFFT (for DNA alignment), RAxML (phylogenetics), and FigTree (visualization).
* Create a docker-compose file to build the images and connect a host folder to each.
* Build the docker-compose network.
* Run the docker-compose network.
* Align the DNA sequences, find their evolutionary relationships, and output a pdf of the tree to the host folder.
* Shut down and remove the networked containers.

1. A common analysis performed using DNA sequence data is to create a tree of their historical relationships, and this is done by first aligning the sequences, which can be done in the program MAFFT.
2. This is done by introducing gaps in the sequences as needed to make them all the same length and put homologous bases at the same position in each sequence.
3. The aligned sequences can then be read by the program RAxML, which will conduct a search of a large number of possible connections between the sequences to find the optimal evolutionary tree.
4. The output of RAxML is a text file that describes the best tree using nested parentheses. It also outputs other tree files, including ones with statistical measures of support and alternate topologies with the same likelihood.
5. A large number of programs can convert these parenthetical trees into graphical trees, and FigTree can do this via a simple command line.
6. On the Desktop create a folder named “cova,” and within that folder create a subfolder named “shared.”
7. Download the full zipped “cova” directory from GitHub [here](http://github.com/A-Farhan/cova):
8. On the page click on the “Code” button and choose “Download ZIP.”
9. Move the file to the Desktop and expand it.
10. Open Terminal and navigate to the “cova/shared” folder created in Step 1 above.
11. Move the file “genomes.fna” from the expanded “cova-master” directory to “cova”:
12. On Windows use the move command like this: move C:\Users\<username>\Desktop\cova-master\cova-master\datasets\example\genomes.fna C:\Users\<username>\Desktop\cova\shared
13. Download (or copy-paste) MAFFT and RAxML Dockerfiles from the StaPH-B GitHub page and put them in “Desktop/cova.” Dockerfile content can be pasted into a new text file, which can be saved in /cova without a file extension using the names in parts 3b and 4b below. Your text editor may still insist on adding a file extension, so check the saved file and remove the extension if necessary.
14. Scroll down on the [StaPH-B GitHub Builds page](http://github.com/StaPH-B/docker-builds) to find the folder containing the Dockerfile.
15. Rename the Dockerfiles with the program name at the end:
16. Dockerfile\_mafft
17. Dockerfile\_raxml
18. The MAFFT Dockerfile should look similar to this (with additional metadata labels):

FROM ubuntu:bionic

RUN apt-get update && apt-get install -y wget

RUN wget https://mafft.cbrc.jp/alignment/software/mafft\_7.450-1\_amd64.deb && \

dpkg -i mafft\_7.450-1\_amd64.deb && \

mkdir /data

WORKDIR /data

1. The RAxML Dockerfile should look similar to this (with additional metadata labels):

FROM ubuntu:bionic

RUN apt-get update && \

apt-get -y install build-essential\

wget \

zip \

unzip && \

apt-get clean

RUN wget https://github.com/stamatak/standard-RAxML/archive/v8.2.12.tar.gz && \

tar -xvf v8.2.12.tar.gz && \

rm -rf v8.2.12.tar.gz && \

cd standard-RAxML-8.2.12/ && \

make -f Makefile.gcc && \

rm \*.o && \

make -f Makefile.SSE3.gcc && \

rm \*.o && \

make -f Makefile.AVX.gcc && \

rm \*.o && \

make -f Makefile.PTHREADS.gcc && \

rm \*.o && \

make -f Makefile.SSE3.PTHREADS.gcc && \

rm \*.o && \

make -f Makefile.AVX.PTHREADS.gcc && \

rm \*.o

RUN wget https://github.com/amkozlov/raxml-ng/releases/download/0.9.0/raxml-ng\_v0.9.0\_linux\_x86\_64.zip && \

unzip raxml-ng\_v0.9.0\_linux\_x86\_64.zip && \

mkdir raxml\_ng && \

mv raxml-ng raxml\_ng/ && \

rm -rf raxml-ng\_v0.9.0\_linux\_x86\_64.zip

ENV PATH="${PATH}:/standard-RAxML-8.2.12:/raxml\_ng"

WORKDIR /data

1. Notice that each Dockerfile directs its container to start in the directory named “data,” which will be visible upon later opening the running container.
2. Download (or copy-paste) the FigTree Dockerfile from the [GitHub Page for BioContainers](http://github.com/BioContainers/containers/blob/master/figtree/1.4.4-3-deb/Dockerfile) and save it to “Desktop/cova.”
3. Rename the file with the program name:
4. Dockerhub\_figtree
5. The FigTree Dockerfile should look like this (with additional metadata labels):

FROM biocontainers/biocontainers:vdebian-buster-backports\_cv1

USER root

ENV DEBIAN\_FRONTEND noninteractive

RUN apt-get update && (apt-get install -t buster-backports -y figtree || apt-get install -y figtree) && apt-get clean && apt-get purge && rm -rf /var/lib/apt/lists/\* /tmp/\*

USER biodocker

1. Build a Docker Compose file in a text editor.
2. Open a blank document and save it in /cova as “docker-compose.yml” and paste in the following content:

version: "3.7"

services:

mafft:

image: mafft

build:

context: /Users/qlu0/Desktop/cova/

dockerfile: Dockerfile\_mafft

stdin\_open: true

volumes:

- ./shared:/shared

raxml:

image: raxml

build:

context: /Users/qlu0/Desktop/cova/

dockerfile: Dockerfile\_raxml

stdin\_open: true

volumes:

- ./shared:/shared

figtree:

image: figtree

build:

context: /Users/qlu0/Desktop/cova/

dockerfile: Dockerfile\_figtree

stdin\_open: true

volumes:

- ./shared:/shared

1. Notice the “version” command at the top of the file. This is the version of Docker Compose, which can be found by simply entering docker-compose version. The third part of the version is not necessary, i.e., version 3.7.4 is covered by 3.7 in the docker-compose file. Failing to input the correct version can cause the dockerfile to fail at creating networked volumes and generating informative error messages.
2. The instruction “stdin\_open: true” is important for keeping the containers open for interactive control. It is equivalent to the -it option when using docker run for individual containers.
3. The docker-compose file will be executed from the Desktop/cova folder, which can be indicated simply by a period and slash (“./”). Subsequently, the “volumes” specifications in the docker-compose file tell Docker to map “/cova/shared” to a folder within each container also named “shared” (the colon indicates the container). If the folder specified in the docker-compose file does not yet exist on the host or container, Docker will create them automatically.
4. Check the docker-compose file by entering docker-compose config in Terminal. If the file is executable by Docker, it will simply display the file contents on the screen. If not, it will report the first error it finds with the line and character number. The format of .yml (or .yaml) is important, and improper indenting can cause errors. Docker-compose files can contain a large amount of information, especially when instructing complex networks, but the best practice is to start simply and add new components one-by-one while checking with the config command.
5. Built images can be verified by entering docker images.
6. Execute the docker-compose by using the up command in detached mode as follows: docker-compose up -d. Docker Compose will automatically use the file named docker-compose.yml in the working directory to set up the containers and network.
7. The resulting network can be seen on the Docker Desktop dashboard.
8. The network will have the name of the folder from which the docker-compose file was executed. It is possible to view the three containers that are running by clicking on the folder name.
9. The container names are prefixed with the network name and suffixed by a number. For example, the MAFFT container is named “cova\_mafft\_1.” The number suffix allows multiple containers for the same program to be run on the same network.
10. Each of the network containers can be interfaced in the dashboard by opening the command line window (via the “CLI” cursor icon to the right of the container name).

* Enter the container “cova\_figtree\_1” and explore the directory structure. Use pwd to see the current working directory (it should be “/data”), cd .. to go one level higher in the directory, and dir to see the directory contents.
* Confirm that there is a folder named “shared” and that it contains “genomes.fna.”
* Check that the FigTree container can communicate with the other containers by pinging them. For example, enter ping -c 1 cova\_raxml\_1 and confirm that the RAxML container has returned a packet.

1. The network that connects the three containers in “cova” can be viewed by entering docker network ls. This brings up a list of the networks, and there should be one named “cova\_default” that is being run by Docker’s “bridge” driver.
2. Inspect “cova\_default” using the inspect command: enter docker network inspect cova\_default. This will display details about the network, including the three containers running inside of it, as well as their networking addresses. The docker-compose file did not specify any ports, but if it had, those would be displayed here, too.
3. From the Terminal, enter each container in order using the exec (“execute”) command and produce a tree of the DNA sequences in “genomes.fna.” With exec, use the interactive option (-i) and the “pseudo-TTY,” or text telephone mimic, option (-t), and specify the interface platform, usually bash or ssh. The commands in each container will produce output files used by the next container.
4. In MAFFT, let the program choose the default optimal settings for the genomes.fna file and then align the sequences so that they are the same length and have homologous bases at the same position in each sequence. The program will then output an aligned file named “coronavirus.fasta.”
5. Enter docker exec -it cova\_mafft\_1 bash – Docker allows options to be combined in any order, so -it is equivalent to -ti, -i -t, and -t -i.
6. Enter mafft --auto /shared/genomes.fna > /shared/coronavirus.fasta.
7. Enter exit.
8. In RAxML, ask the program to read the aligned sequences, conduct a simple tree search, and output the various tree files to the “shared” folder. Each file will have “1” at the end of the file name, which can be used to identify the output from different runs of the same search. Other options (e.g., -m, -p) give starting specifications for the computational process needed to find the optimal tree.
9. Enter docker exec -it cova\_raxml\_1 bash.
10. Enter raxmlHPC -m GTRGAMMA -p 12345 -s /shared/coronavirus.fasta -n 1 -w /shared -f a -x 12345 -N 100 -T 12.
11. Enter exit.
12. In FigTree any one of the text trees output by RAxML can be read and transformed into graphical trees and output in various file formats. Here, read the file from RAxML that contains the “best tree” and output it as a pdf of a tree illustration with sequence identifiers at the terminals.
13. Enter docker exec -it cova\_figtree\_1 bash.
14. Enter figtree -graphic PDF -width 500 -height 800 /shared/RAxML\_bestTree.1 /shared/coronavirusTree.pdf.
15. Confirm that the pdf of a tree is now in the “/cova/shared” folder on the Desktop.
16. Enter exit
17. Dismantle the network and its containers by entering docker-compose down.
18. Confirm that the network “cova” is gone by entering docker network ls.
19. Confirm that the containers are now gone by entering docker ps.
20. Confirm that the images are still available by entering docker images.