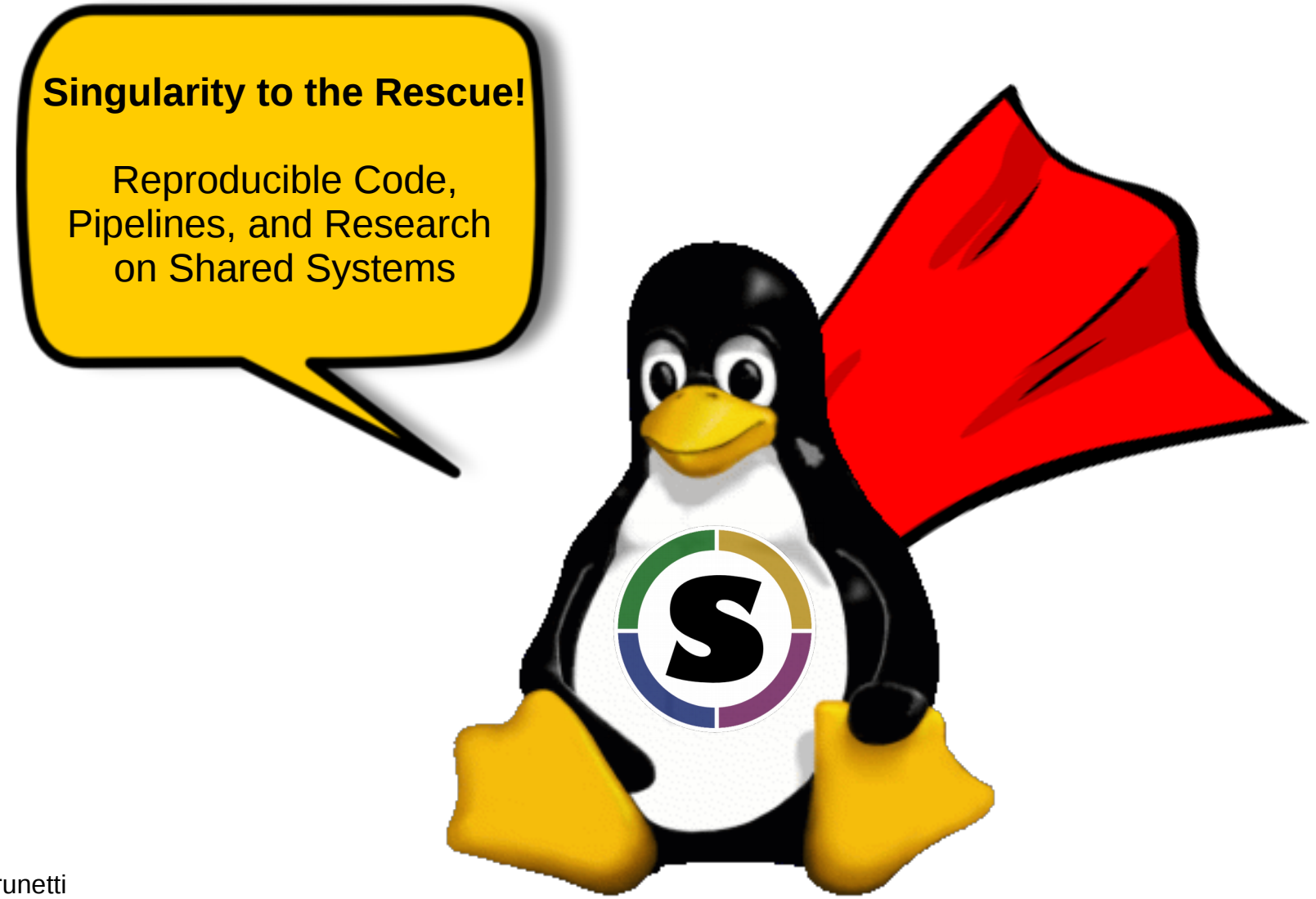
Singularity Containers Tutorial

Brought to you by Translational Informatics and Computational Resources (TICR)

a sub-division of the Colorado Center for Personalized Medicine
University of Colorado Anschutz Medical Center



Written by: Tonya Brunetti

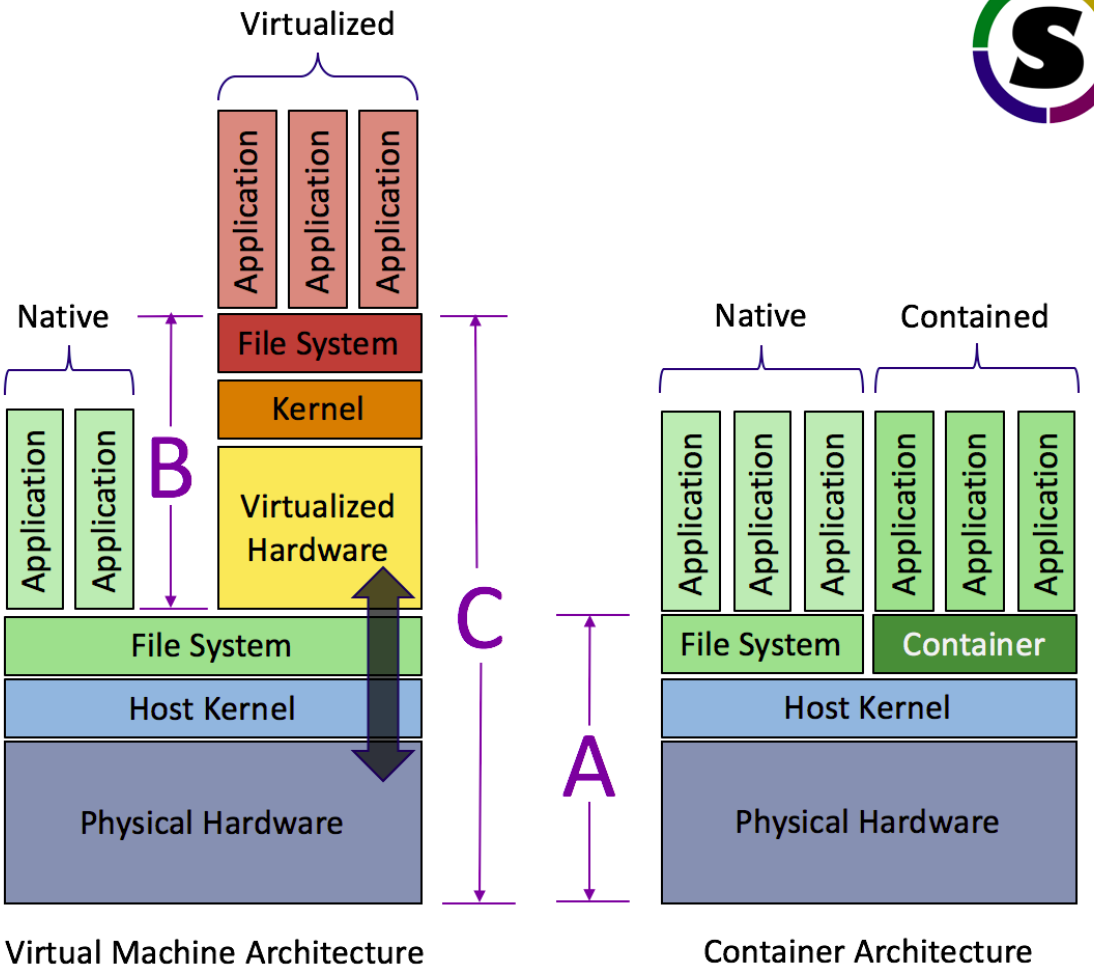
Overview

- What is Singularity?
  - What is a container and why use them?
  - Singularity vs Docker
- Building Custom Container (all local/root)
  - Installing Singularity
  - Singularity Recipe File Tutorial
  - Building (and Testing) a Container
- Running Container Images on a Host System (all non-root)
  - Move image to host system
  - Binding and mounting container images
  - Run container or write sbatch script
- Singularity Commands and Supported Functions
  - run vs exec
  - piping and redirections

What is a container and why should I use it?

VIRTUAL MACHINES VS. CONTAINERS

- Applications running within a container will always be “closer” to the physical hardware
  - Notice how close to native a container behaves
- Applications running through a virtual machine will always have multiple levels of indirection
- The container’s proximity to the physical hardware equates to less overhead, higher performance and lower latency

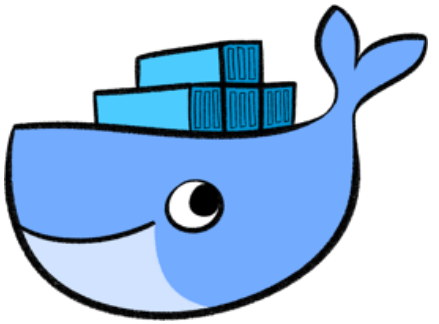


https://www.hpcwire.com

Containers allow for:

1) Easy collaboration
2) Reproducible code, pipelines, research, results, version control
3) Software installs without root privileges
4) Scalability on local and shared resources

Singularity vs Docker



- Almost ***all*** high performance compute systems do not directly support Docker but will directly support singularity

- **Why do we use Singularity on the HPC rather than Docker?**
  - Two reasons:
    1. Security
       - Presents a much smaller risk to shared systems like the HPC or the cloud
    2. Scalability
       - Singularity has the capacity to scale with almost all compute resources and is built to scale on shared and distributed systems (SLURM, PBS, etc…)

- **What if I already have Docker containers?**
  - This is not a problem! As a matter of fact, you may continue to use Docker, however, just be sure to wrap it in a Singularity container before importing it into the HPC or cloud
  - Singularity supports the use of importing existing Docker container images

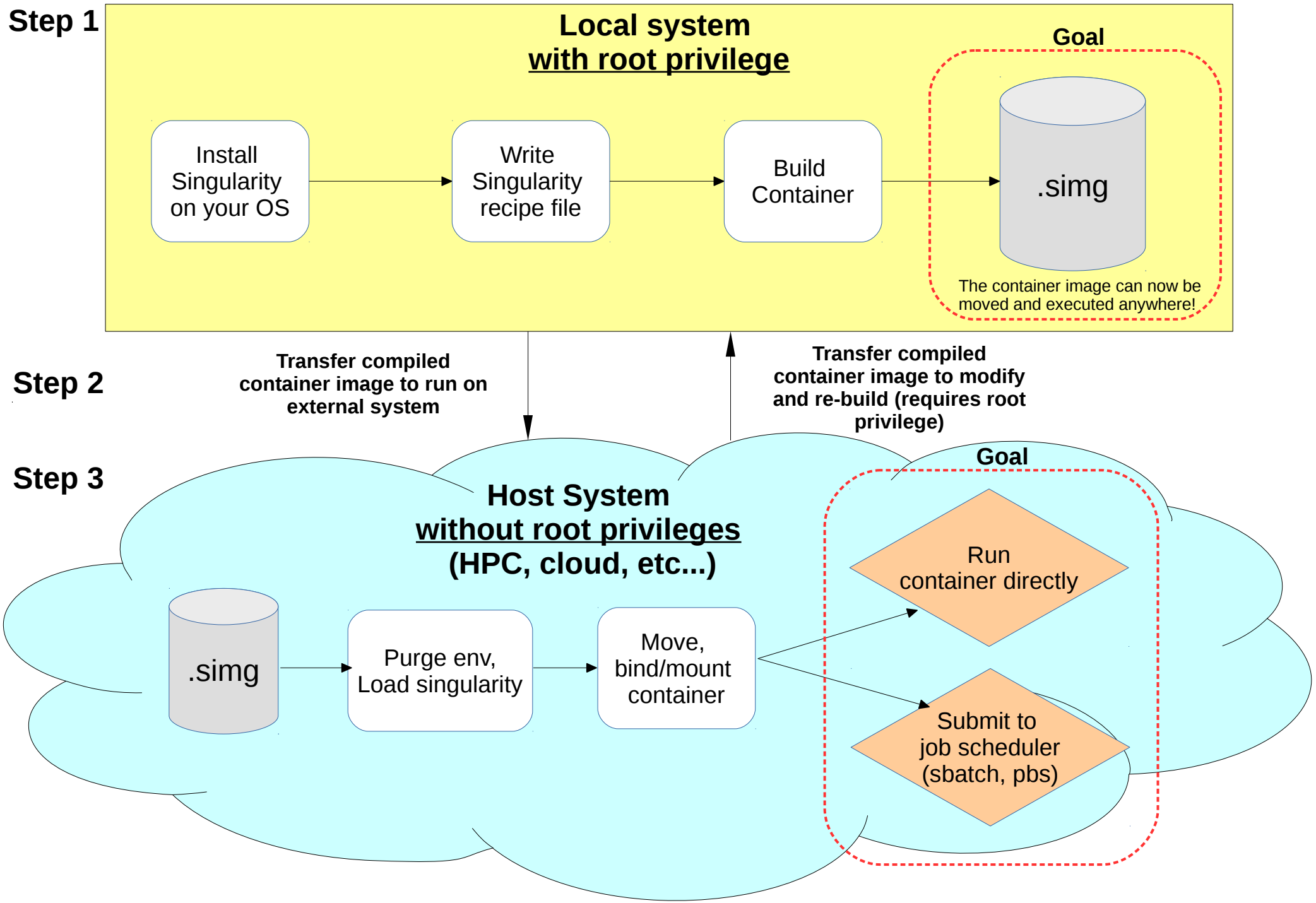
Singularity Security

The permissions of a container always equal the same permissions a user has on the host environment (i.e. the environment where the container is being used).

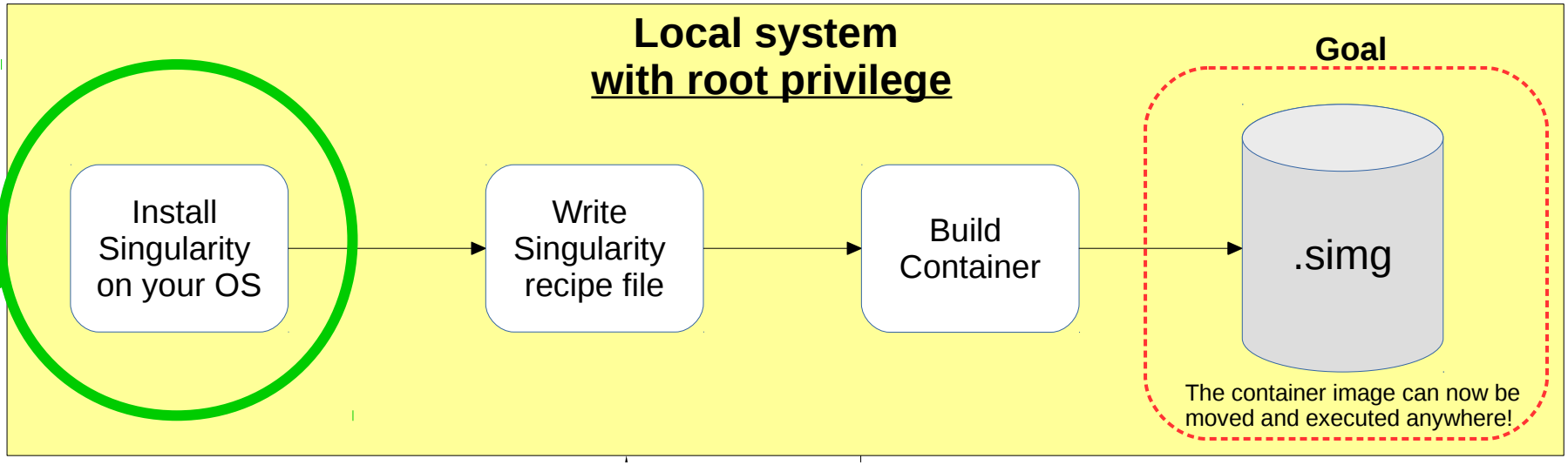
<u>This means a few things:</u>

1. Naturally, this means that a user can never have root privileges to any container within the context of the HPC environment, unless you are already granted root/sudo privileges within the HPC environment, which is usually restricted to only a limited number of system administrators.
2. This also means, that once a user has "activated" or is using his/her container a user cannot change the privileges within the container to exceed the privileges the user has on the host system.
3. Any changes to the container image (i.e. updating code within the image) **must** be performed on a local computer where the user has root access.

Singularity Workflow



Singularity Workflow



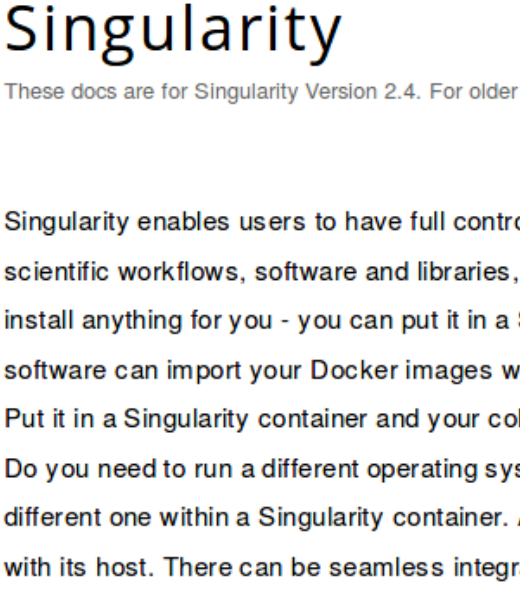
Focus our attention initially to the local system with root access

- Keep in mind to build or modify a container, the container must always be on a system where root access is available
- The next slide will address installing singularity on your personal OS

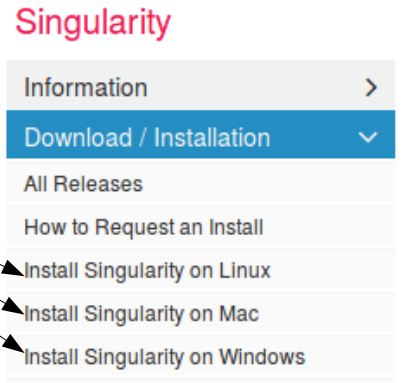
Singularity can be installed on multiple platforms

- We will not go into specific details about how to install Singularity on your own personal OS.
  - We expect if you are going to use Singularity and have HPC experience, you should be able to install this relatively easy.
  - If you need help, please contact us at CCPM-Rosalind@ucdenver.edu
- Here is the singularity URL:

http://singularity.lbl.gov/

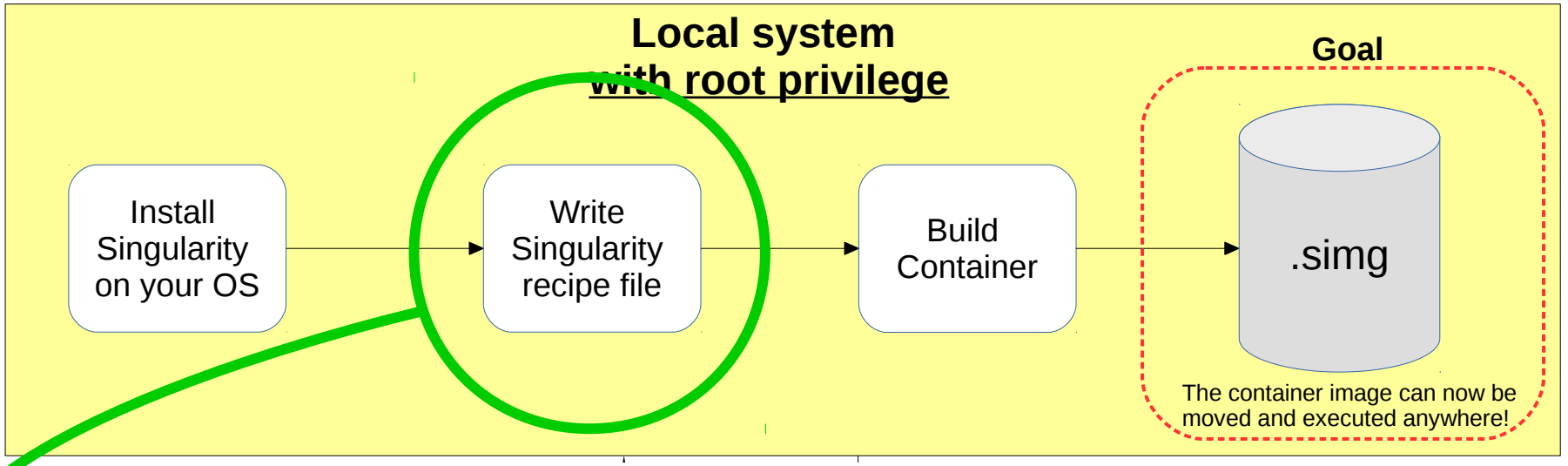


Select your OS from the main menu page for download and installation instructions



If you are using singularity on Rosalind please install version 2.4.2!

Singularity Workflow



Focus our attention initially to the local system with root access

- The next slides will address the how to write a singularity recipe file. This will be several slides, as we plan to show you a real-life example (that we have tested and used on Rosalind) of how we have used Singularity to package one of our GitHub repositories into a singularity container image.

- The recipe file is the most work intensive part of a container build and requires comprehension of how the code works. **This is all very important if you want your container to be funtional!**

Singularity Recipe File Tutorial

- A singularity recipe file is required when building a custom container
- This tutorial is not a comprehensive list of recipe file decorator options! (i.e. %apps is not covered)
- Additionally, this tutorial covers building a custom container only! (i.e. not importing and modifying existing images from shub or Docker)

What is a singularity recipe file?

- A plain text file (.txt) although the file extension is not required
- It is an organized set of instructions, designated with keywords and decorators, used by Singularity to build your container to the proper specifications
- Including the recipe file as part of your GitHub repository allows users to build and customize your repository on their local computer and store it as an image
- Additionally or alternatively, you may choose to also submit your container image to shub (Singularity's container image repository) so a user can download the image and run it immediately without building the container

Writing a recipe file workflow

Bootstrap:
OSVersion:
MirrorURL:
Include:

%help

%setup

%post

%files

%environment

%test

%runscript

HEADER

The words listed in the yellow section of the file are the keywords that Singularity looks for to determine which OS to install; a header is always required

SECTIONS

The decorators (words beginning with %) are the keywords that singularity looks for when determining how to build the bulk functionality of the container.

Only %runscript is required to build a simple functional container; however we will use all the decorators listed here, in order, to illustrate the functionality of Singularity for containerizing complex projects.

Case Study: GP3

Decription: GP3 (GWAS Pre-Processing Pipeline) is a custom GWAS pipeline that was built to help filter out samples that do not meet a series of GWAS criteria while also generating PCs and graphs, as well as prepare data for input into an imputation server all in one pipeline. This pipeline suite is located on GitHub in its own repository.

Case: This pipeline is an excellent case study for Singularity due to the complexity of the project. This pipeline has many moving parts:

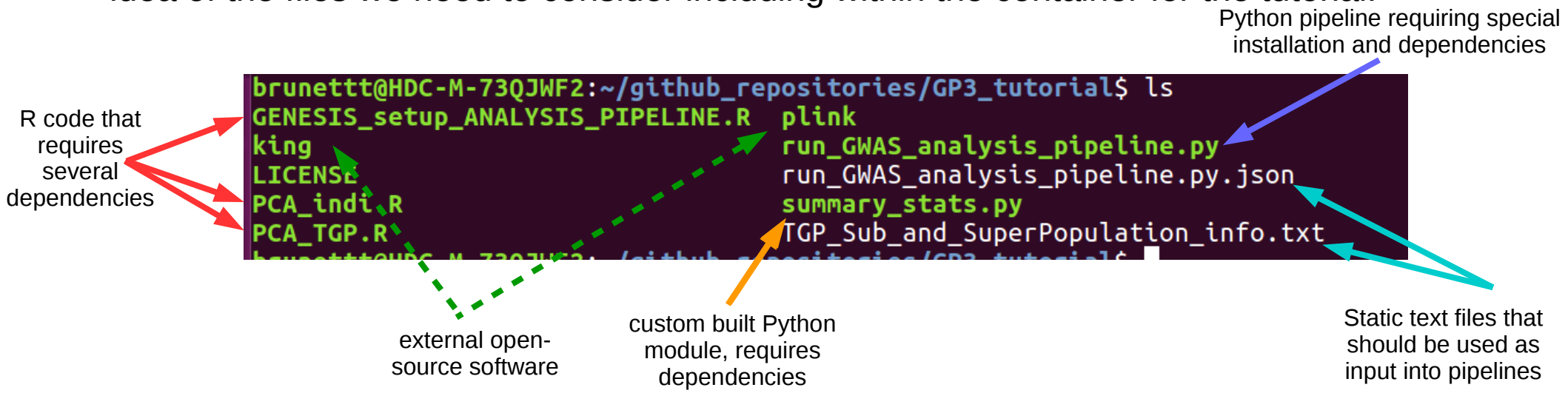
1. It is multi-language (R, Python, bash)
2. It has several package dependencies within R, Python, and Linux
3. It has custom built software that requires manual installation
4. It may be version sensitive in the near future
5. Most people using it will probably need to adapt it to the HPC

Problems and Assumptions: This pipeline may not be used by people for a few reasons, despite being well-documented and having a thorough set of installation instructions. Why?

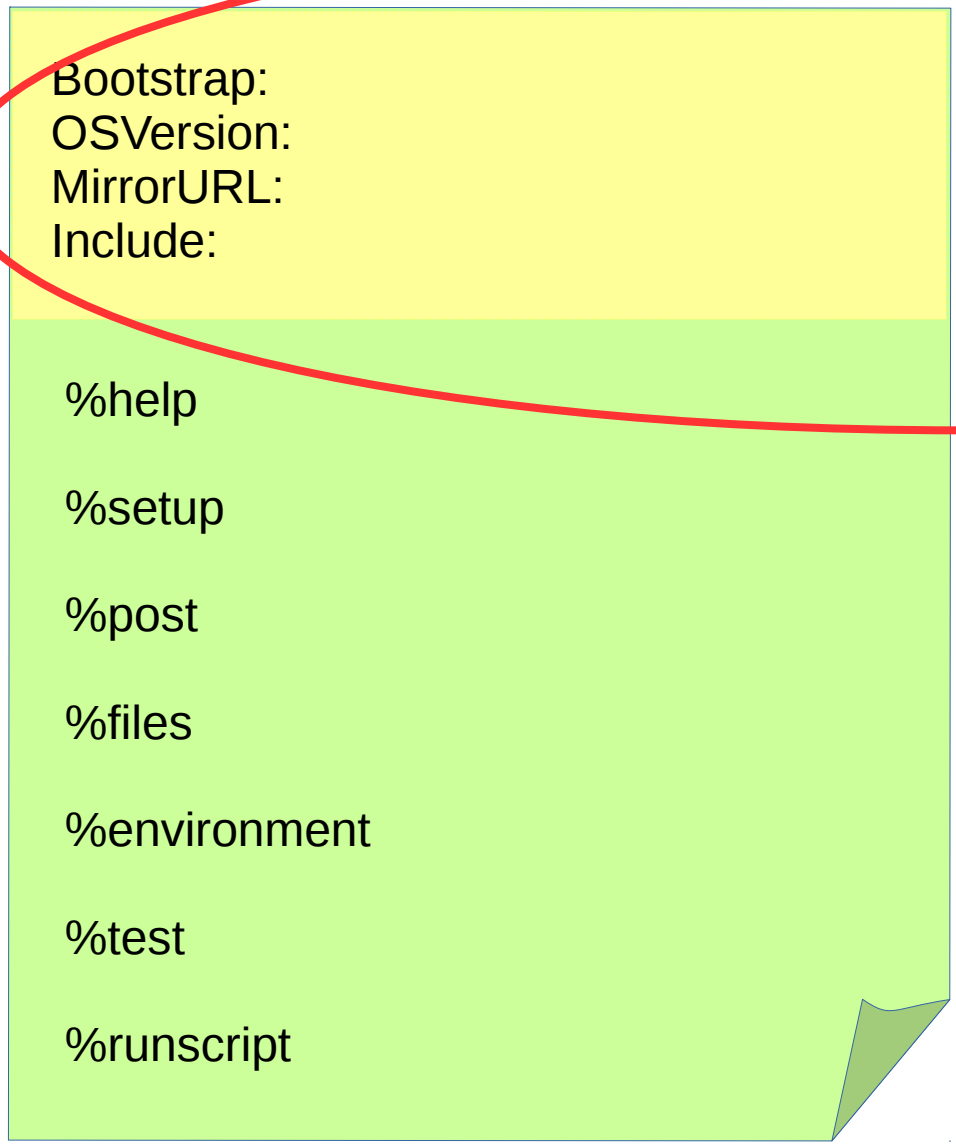
1. There may be users who do not know some of the languages well so they do not feel comfortable installing all the packages
2. Some packages may be difficult to install due to underlying OS and/or external package dependencies
3. The code may be specific to a particular version of software

Solution: Containerize!

- Since we coded the project, we have the best understanding of the best way to install all the software, dependencies, configure the OS and environment
- We, as the developers, can take all the guess work out of installing and configuring this pipeline by writing a recipe file and building the container ourselves.
  - We can then deliver the container to anyone and they can run it on any system/OS that has singularity installed on it (assuming they also have the available resources)
  - They have to just run it, one command! No installation and configuration necessary
  - When the user runs the image, all commands are contained within the container, which already has all the elements needed, including its own OS == **Easy to use, Reproducible Code and Research**
- Here is an image of the required files listed in the GP3 GitHub repository, to give you an idea of the files we need to consider including within the container for the tutorial:



Writing a recipe file workflow



HEADER

The words listed in the yellow section of the file are the keywords that Singularity looks for to determine which OS to install; a header is always required.

SECTIONS

The decorators (words beginning with %) are the keywords that singularity looks for when determining how to build the bulk functionality of the container.

Only %runscript is required to build a simple functional container; however we will use all the decorators listed here, in order, to illustrate the functionality of Singularity for containerizing complex projects.

Header Section: Debian/Ubuntu OS

1. We are going to create a singularity recipe file called “GP3_singularity_recipe”

```
brunettt@HDC-M-73QJWF2:~/github_repositories/GP3$ vi GP3_singularity_recipe
```

2. We are going to build a Debian/Ubuntu OS environment for the container, version “xenial”. We will tell Singularity to mirror this OS from the Ubuntu repository URL, and to be sure to also install the “apt” function during the OS install.

3. You can see that:
   - **Bootstrap** is specifiying the OS
   - **OSVersion** is specifying the OS version to use
   - **MirrorURL** is specifying the URL of the repo to mirror the OS
   - **Include** is specifying anything that should be automatically installed during the OS build; in this case “apt” is the Debian/Ubuntu function that installs libraries and packages directory from the Debian/Ubuntu repo

```
Bootstrap: debootstrap
OSVersion: xenial
MirrorURL: http://us.archive.ubuntu.com/ubuntu/
Include: apt
```

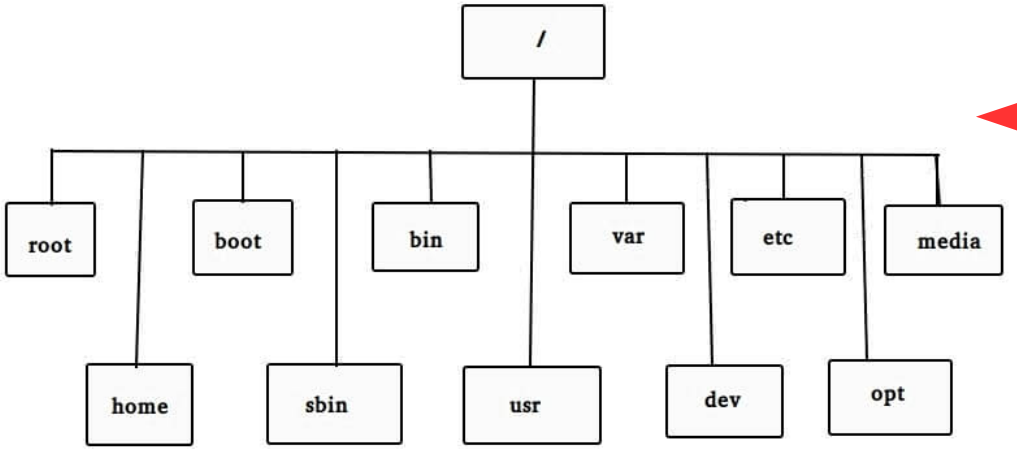
Header Section Options

Option Name	When you choose this option...

shub	Pull down an already existing container image from the Singularity Hub
	e*xample header in singularity recipe file:*
	`Bootstrap: shub`
	`From: shub://<registry>/<username>/<container-name>:<tag>@digest`

docker	Pull down an already existing container image from the Docker Hub e*xample header in singularity recipe file:* Bootstrap: docker From: <registry>/<namespace>/<container>:<tag>@<digest>
localimage	Load a container image that already exists on your local system/machine; if you use this option your header will only have the Bootstrap decorator and the From decorator fields in the header, where from is the URL/path to container
yum	Build yum-based CentOS Linux system e*xample header in singularity recipe file:* Bootstrap: yum OSVersion: 7 MirrorURL: http://mirror.centos.org/centos-%{OSVERSION}/%{OSVERSION}/os/$basearch/ Include: yum
debootstrap	Build apt-based Ubuntu/Debian Linux system
arch	Build Arch Linux
busybox	BusyBox-based system build
zypper	Build Suse and OpenSuse based system

Singularity adopts the *same* file system architecture of the OS selected for download



This is the traditional file system architecture within a Debian/Ubuntu OS

This is an example of the file structure within a singularity container we made based upon on the Debian/Ubuntu OS

A few things to mention:

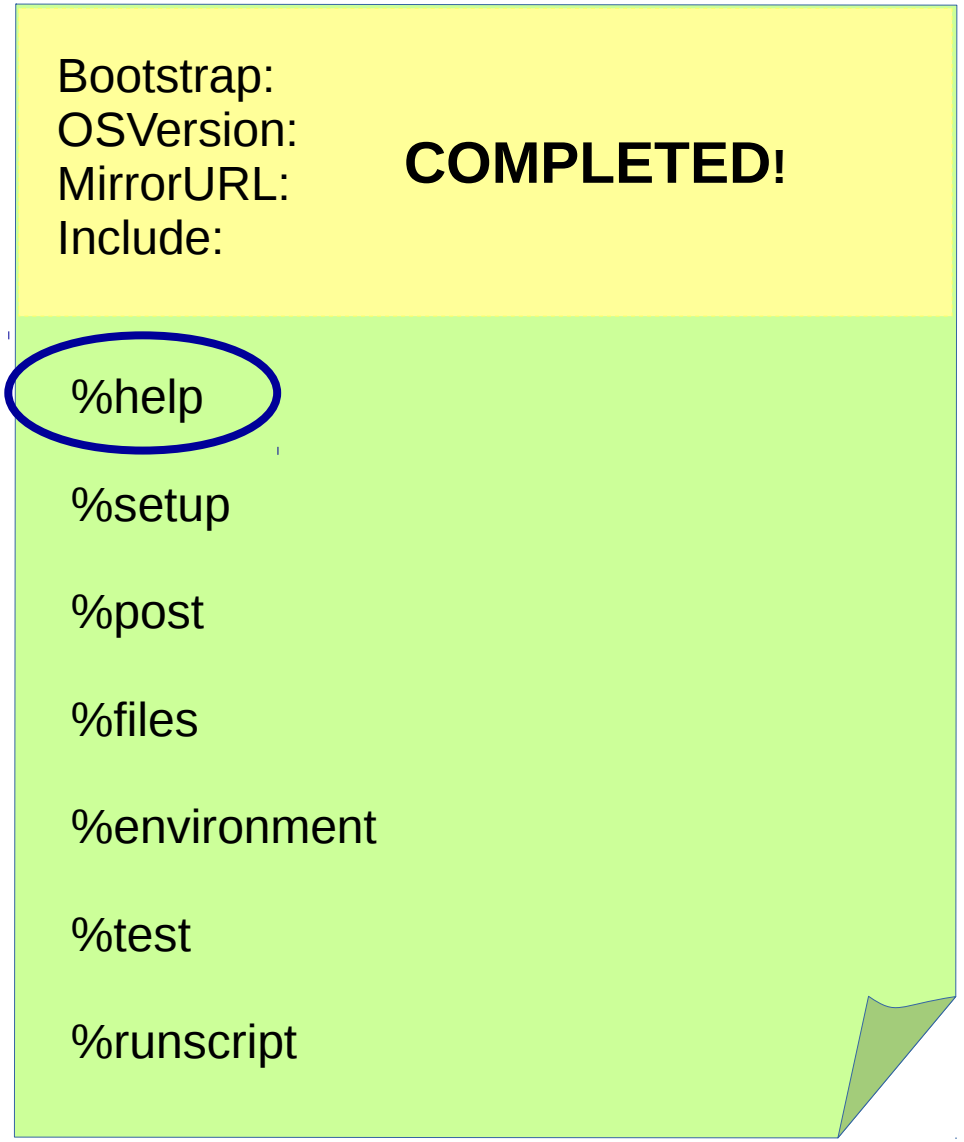
1. All folders are owned by root, hence the inability to mutate them on the HPC

2. A few extra files are now present as symbolic links. These will add the environmental variables and runscript/executable calls to the container

grepitout.com

```
Singularity new_GP3.simg:~/github_repositories/GP3> ls -lh /
total 412K
-rwxrwxr-x   1 root root 1.1K Jan 25 21:43 LICENSE
drwxr-xr-x   2 root root 4.0K Jan 25 21:48 bin
drwxr-xr-x   2 root root 4.0K Apr 12  2016 boot
drwxr-xr-x  21 root root 4.3K Jan 25 23:55 dev
lrwxrwxrwx   1 root root   36 Jan 19 20:17 environment -> .singularity.d/env/90-environment.sh
drwxr-xr-x  68 root root 4.0K Jan 25 21:57 etc
drwxr-xr-x   1 root root   60 Jan 26 02:08 home
drwxr-xr-x   9 root root 4.0K Jan 25 21:48 lib
drwxr-xr-x   2 root root 4.0K Jan 25 21:43 lib64
drwx------   2 root root  16K Jan 25 21:57 lost+found
drwxr-xr-x   2 root root 4.0K Jan 25 21:43 media
drwxrwxrwx   4 root root 4.0K Jan 25 21:57 mnt
drwxr-xr-x   2 root root 4.0K Jan 25 21:43 opt
dr-xr-xr-x 331 root root    0 Jan 25 23:55 proc
drwx------   3 root root 4.0K Jan 25 21:50 root
drwxr-xr-x   4 root root 4.0K Jan 25 21:43 run
drwxr-xr-x   2 root root 4.0K Jan 25 21:48 sbin
lrwxrwxrwx   1 root root   24 Jan 19 20:17 singularity -> .singularity.d/runscript
drwxr-xr-x   2 root root 4.0K Jan 25 21:43 srv
dr-xr-xr-x  13 root root    0 Jan 26 02:07 sys
drwxrwxrwt  14 root root 336K Jan 26 02:07 tmp
drwxr-xr-x  10 root root 4.0K Jan 25 21:43 usr
drwxr-xr-x  11 root root 4.0K Jan 25 21:43 var
Singularity new_GP3.simg:~/github_repositories/GP3>
```

Writing a recipe file workflow



HEADER

We just completed this!

SECTIONS

Next, we will look at the **%help** section

Section: %help

- The **%help** decorator is a place to add a help section for you container.
- Everything that is under the **%help** section appears when a person runs the following singularity command:

```
singularity help <your container image>
```

- This section is optional, but for best practices should be included within each recipe file

Tutorial Example

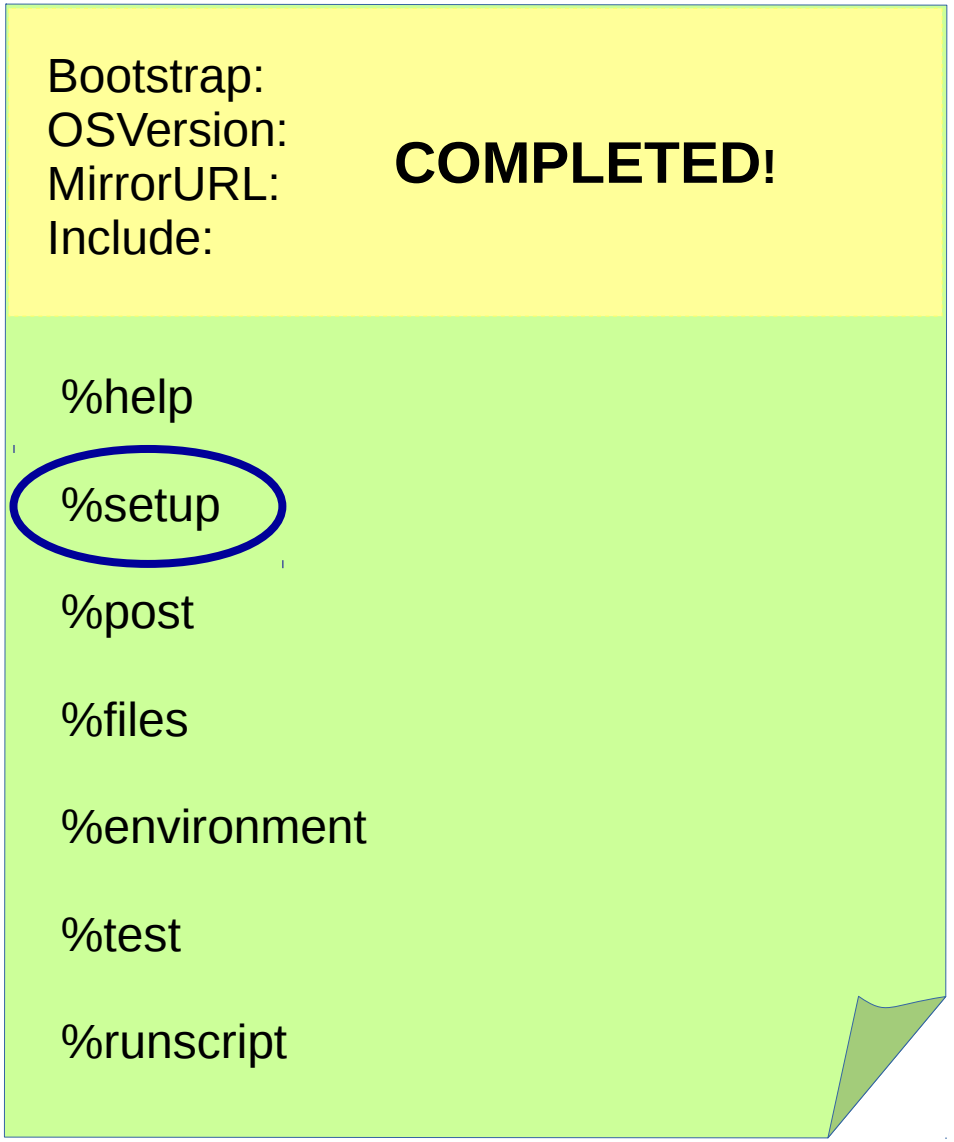
1. We will open our “GP3_singularity_recipe text file again and add a **%help** section:

```
Bootstrap: debootstrap
OSVersion: xenial
MirrorURL: http://us.archive.ubuntu.com/ubuntu/
Include: apt

%help
        This container is built to run the GP3 repo located on github at https://github.com/tbrunetti/GP3
```

2. Notice, that we must include the **%help** decorator to let Singularity know that everything following this decorator is part of **%help, until the next decorator is reached.**

Writing a recipe file workflow



HEADER

We just completed this!

SECTIONS

Next, we will look at the **%setup** section

Section: %setup

- The **%setup** decorator is a place to:
  - add files that need to be accessed during the building of the container
  - or export path variables that need to be accessed during the building of the container

  which should be copied into the **$SINGULARITY_ROOTFS** location within the Singularity container
- **%setup** is executed before the **%post** section (%post is where you will install all program, packages, libraries, and languages)
- Although this section is optional, for many cases, users will need this section in the event that certain files need to be accessed during the **%post** section

Tutorial Example

1. We will open our "GP3_singularity_recipe text file again and add a **%setup** section:

```
Bootstrap: debootstrap
OSVersion: xenial
MirrorURL: http://us.archive.ubuntu.com/ubuntu/
Include: apt

%help
        This container is built to run the GP3 repo located on github at https://github.com/tbrunetti/GP3

%setup
        cp run_GWAS_analysis_pipeline.py $SINGULARITY_ROOTFS
        cp run_GWAS_analysis_pipeline.py.json $SINGULARITY_ROOTFS
        cp king $SINGULARITY_ROOTFS
        cp plink $SINGULARITY_ROOTFS
        cp TGP_Sub_and_SuperPopulation_info.txt $SINGULARITY_ROOTFS
        echo $SINGULARITY_ROOTFS
        ls $SINGULARITY_ROOTFS
```

2. Notice, all the files and programs we know we will be using in the next section for installation, we have copied into **$SINGULARITY_ROOTFS**. It is **required** that these files get copied there if they need to be accessed in the next section since this is the only location that will be available for use during the **%post** section

Section: %setup, tutorial continued...

Tutorial Example

3. The **`cp`** command works as expected in Linux. The file to be copied should be the path to the file relative to where you call the **`singularity build`** function the container. In this tutorial, we are working within the GP3 repository and we plan to build the container here as well, therefore the absolute path is not required.

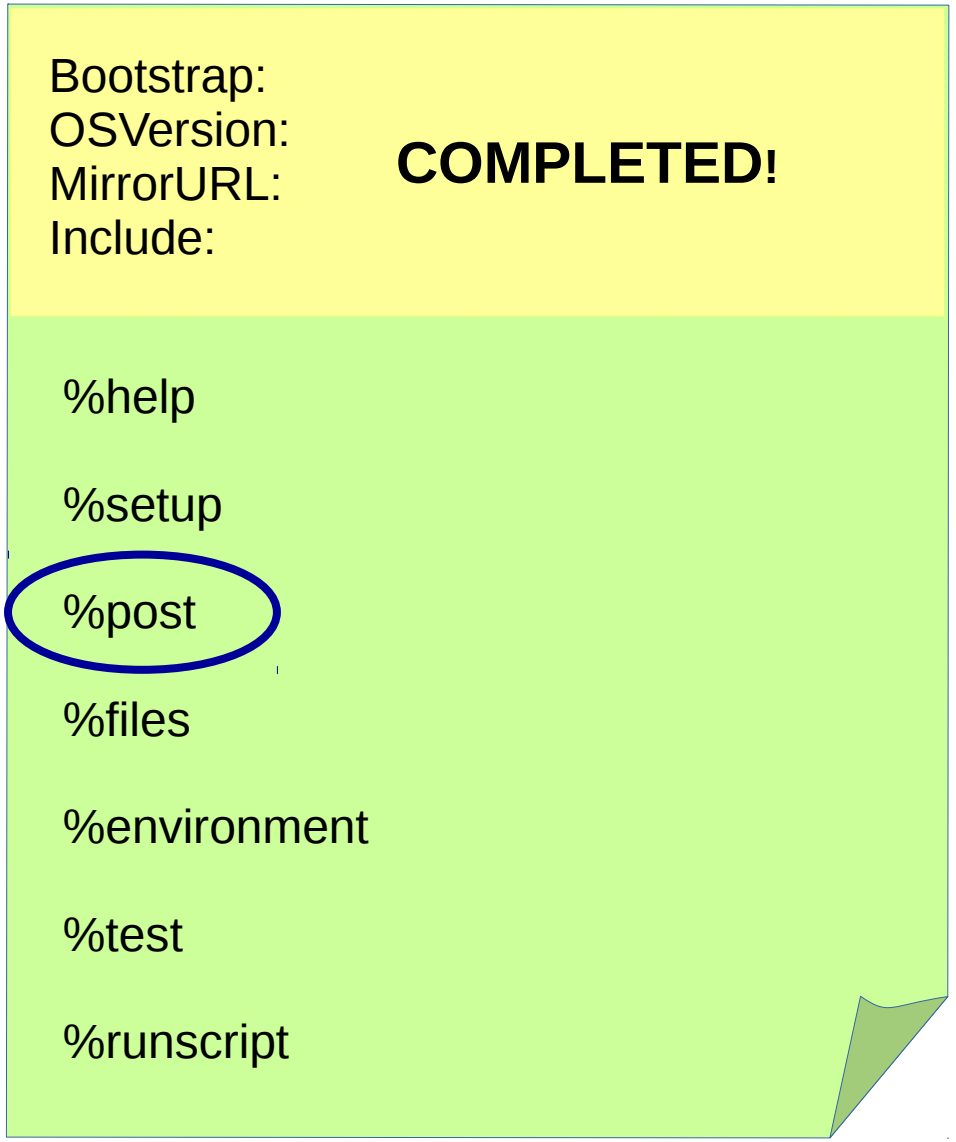
```
Bootstrap: debootstrap
OSVersion: xenial
MirrorURL: http://us.archive.ubuntu.com/ubuntu/
Include: apt

%help
        This container is built to run the GP3 repo located on github at https://github.com/tbrunetti/GP3

%setup
        cp run_GWAS_analysis_pipeline.py $SINGULARITY_ROOTFS
        cp run_GWAS_analysis_pipeline.py.json $SINGULARITY_ROOTFS
        cp king $SINGULARITY_ROOTFS
        cp plink $SINGULARITY_ROOTFS
        cp TGP_sub_and_superPopulation_info.txt $SINGULARITY_ROOTFS
        echo $SINGULARITY_ROOTFS
        ls $SINGULARITY_ROOTFS
```

4. All Linux commands work within the Singularity recipe files. We have included the **`echo`** and **`ls`** command on the **$SINGULARITY_ROOTFS** location to ensure that it we can confirm in the build log that the files copied in this section get transferred.

Writing a recipe file workflow



HEADER

We just completed this!

SECTIONS

Next, we will look at the **%post** section

Section: %post

- The **%post** decorator is the place where most of your container customization will happen
- This section will call all commands relating to the following:
  - **Software installation**
    - Custom scripts and programs that require installation
    - Languages
    - Libraries
    - Packages
  - **Software Configuration**
  - **Additional OS Configuration**

Tutorial Example

see the next slide for tutorial and explanation...

Section: %post, tutorial continued...

Tutorial Example

1. We will open our “GP3_singularity_recipe text file again and add a **%post** section:

```
%post
        echo "Installing all container dependencies!"
        apt-get -y update
        apt-get -y install software-properties-common
        apt-add-repository universe
        apt-get -y update
        apt-get -y install libnetcdf-dev netcdf-bin
        apt-get -y install python
        apt-get -y install python-pip
        echo "verifiying pip install"
        pip -V
        pip install chunkypipes
        pip install pandas
        pip install numpy
        pip install matplotlib
        pip install fpdf
        pip install Pillow
        pip install pypdf2
        pip install statistics
        pip install xlrd
        apt-get -y install r-base
        mkdir /mnt/R_packages
        R --slave -e 'system("defaults write org.R-project.R force.LANG en_US.UTF-8")'
        R --slave -e 'install.packages("ncdf", repos="https://cloud.r-project.org", lib="/mnt/R_packages", dependencies=T)'
        R --slave -e 'source("https://bioconductor.org/biocLite.R"); biocLite("GWASTools", lib="/mnt/R_packages")'
        R --slave -e 'source("https://bioconductor.org/biocLite.R"); biocLite("SNPRelate", lib="/mnt/R_packages")'
        R --slave -e 'source("https://bioconductor.org/biocLite.R"); biocLite("GENESIS", lib="/mnt/R_packages")'
        /usr/local/bin/chunky init /mnt
        mv run_GWAS_analysis_pipeline.py /mnt/.chunky/pipelines
        mv run_GWAS_analysis_pipeline.py.json /mnt/.chunky/configs
        mv king /mnt
        mv plink /mnt
        mv TGP_Sub_and_SuperPopulation_info.txt /mnt
        cd /mnt
        ls
        chmod a+rwx /mnt
        chmod -R a+rwx /mnt/.chunky
        apt-get clean
```

Section: %post, tutorial continued...

Tutorial Example

2. Here is an explanation of what is bring installed here under **%post**:
IMPORTANT NOTE: Anything installed must automatically be installed <u>without waiting for user input;</u> hence the -y, -e, and --slave options

```
%post
        echo "Installing all container dependencies!"
        apt-get -y update
        apt-get -y install software-properties-common
        apt-add-repository universe
        apt-get -y update
        apt-get -y install libnetcdf-dev netcdf-bin
        apt-get -y install python
        apt-get -y install python-pip
        echo "verifiying pip install"
        pip -V
        pip install chunkypipes
        pip install pandas
        pip install numpy
        pip install matplotlib
        pip install fpdf
        pip install Pillow
        pip install pypdf2
        pip install statistics
        pip install xlrd
        apt-get -y install r-base
        mkdir /mnt/R_packages
        R --slave -e 'system("defaults write org.R-project.R force.LANG en_US.UTF-8")'
        R --slave -e 'install.packages("ncdf", repos="https://cloud.r-project.org", lib="/mnt/R_packages", dependencies=T)'
        R --slave -e 'source("https://bioconductor.org/biocLite.R"); biocLite("GWASTools", lib="/mnt/R_packages")'
        R --slave -e 'source("https://bioconductor.org/biocLite.R"); biocLite("SNPRelate", lib="/mnt/R_packages")'
        R --slave -e 'source("https://bioconductor.org/biocLite.R"); biocLite("GENESIS", lib="/mnt/R_packages")'
        /usr/local/bin/chunky init /mnt
        mv run_GWAS_analysis_pipeline.py /mnt/.chunky/pipelines
        mv run_GWAS_analysis_pipeline.py.json /mnt/.chunky/configs
        mv king /mnt
        mv plink /mnt
        mv TGP_Sub_and_SuperPopulation_info.txt /mnt
        cd /mnt
        ls
        chmod a+rwx /mnt
        chmod -R a+rwx /mnt/.chunky
        apt-get clean
```

Installation of Linux libraries and repos

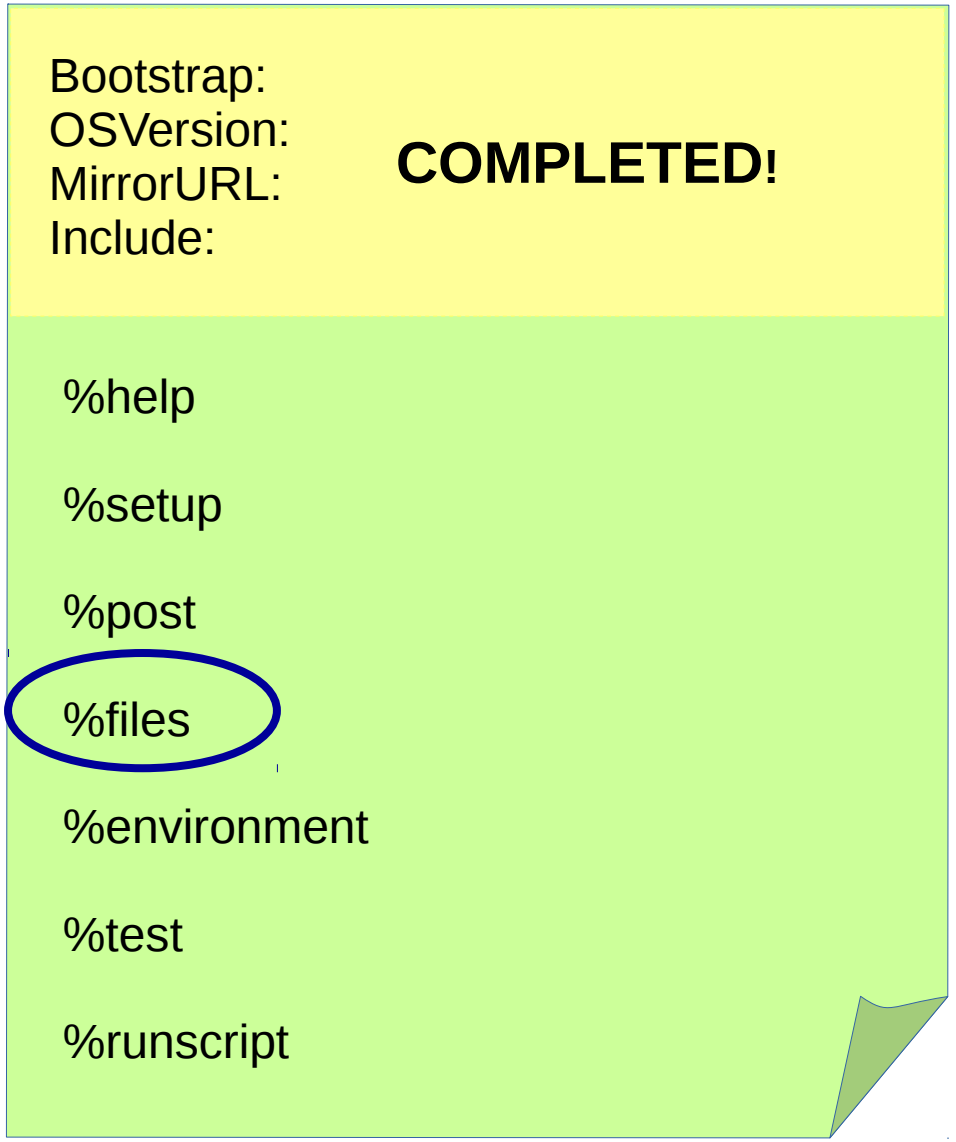
Installation of Python software and Python libraries
As expected, installations default to /usr/bin/ and site-packages directory

Installation of R, biocLite, and packages (yellow braces)
As expected, R/Rsciprt installations /usr/bin and normal library locations, however, I am making a new directory since I want to have a custom library installation location

Installation of Custom Scripts
note: we only have access to these local files due to %setup

Customizing file structure, permissions, and cleaning
note: we only have access to these local files due to %setup

Writing a recipe file workflow



HEADER

We just completed this!

SECTIONS

Next, we will look at the **%files** section

Section: %files

- The **%files** decorator is a place to **copy any local files** (location where you build the container image) into the container permanently for use at runtime

- This section is executed after the **%post** section, so only use this to copy files that need to be run when the container's **%runscript** is executed but not required when executing the **%post** section

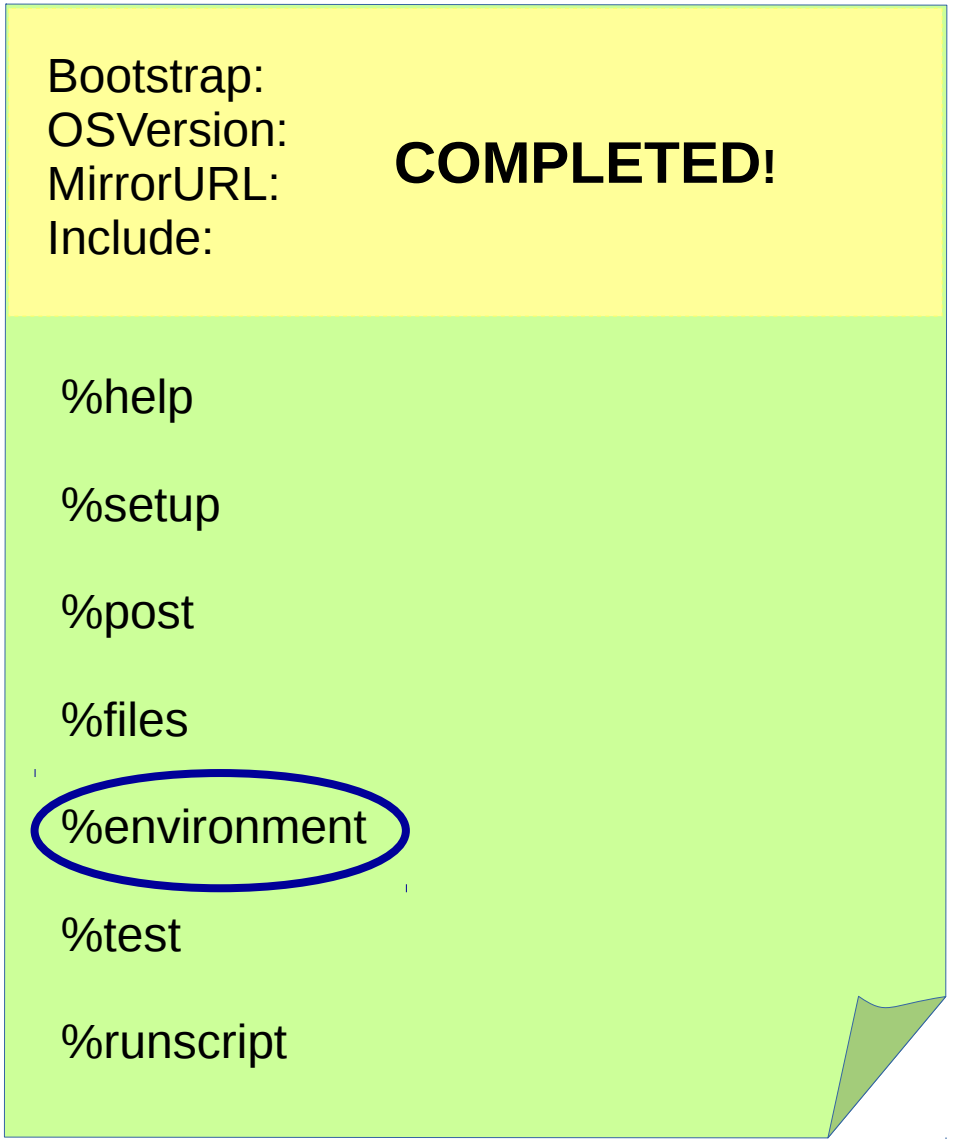
Tutorial Example

1. We will open our "GP3_singularity_recipe text file again and add a **%files** section:

```
%files
        LICENSE
        PCA_indi.R /mnt
        PCA_TGP.R /mnt
        GENESIS_setup_ANALYSIS_PIPELINE.R /mnt
        summary_stats.py /mnt
        run_GWAS_analysis_pipeline.py /mnt
```

2. There is no need to add the `cp` command since **%files** is only used for copying, it automatically adds the `cp` command to the listed files. You can see that most of the files we are copying are being copied into Singularity's /mnt directory within the container image. *None of these files can be accessed during **%post**!*

Writing a recipe file workflow



HEADER

We just completed this!

SECTIONS

Next, we will look at the **%environment** section

Section: %environment

- The **%environment** decorator is a place to list all the environmental variables and paths that needs to be added to your container environment.

- This section is executed after the **%files** section. Therefore, if environmental variables or paths are needed during the **%post** section, they should be added within the **%setup** section

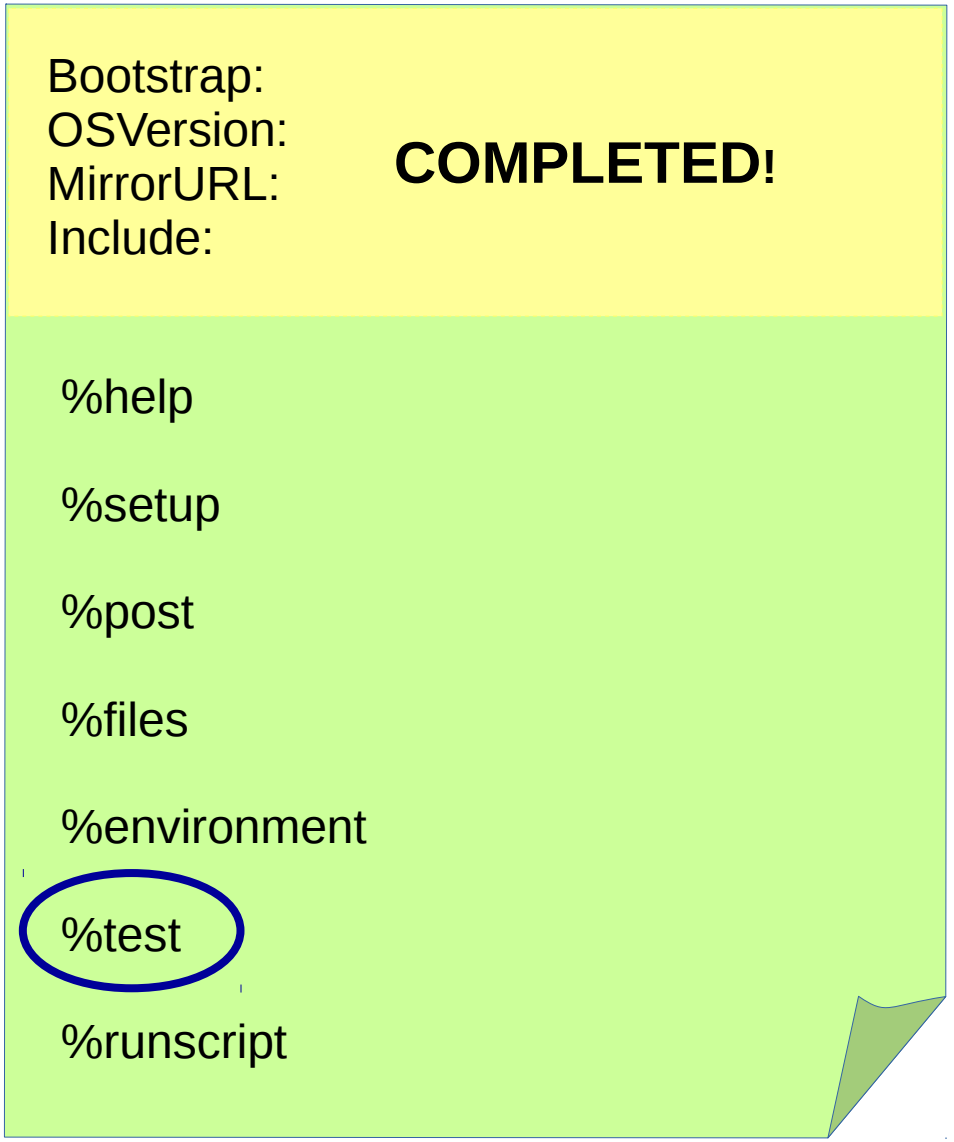
Tutorial Example

1. We will open our “GP3_singularity_recipe text file again and add a **%environment** section:

```
%environment
        CHUNKY_HOME=/mnt
        export CHUNKY_HOME
        export PYTHONPATH="${PYTHONPATH}:/mnt/"
        export NETCDF_INCLUDE=/usr/include
```

2. The variables we have listed will be available for use during container runtime. For example, I have added the container’s /mnt directory to the Python search path since we have a custom built Python module located in that directory.

Writing a recipe file workflow



HEADER

We just completed this!

SECTIONS

Next, we will look at the **%test** section

Section: %test

- The **%test** decorator is a place to call test scripts or commands to ensure everything listed in the recipe file thus far works as expected.
- This section is only executed when the container is being built.
- This section is optional, and each user is responsible for writing code or commands that test their container build.

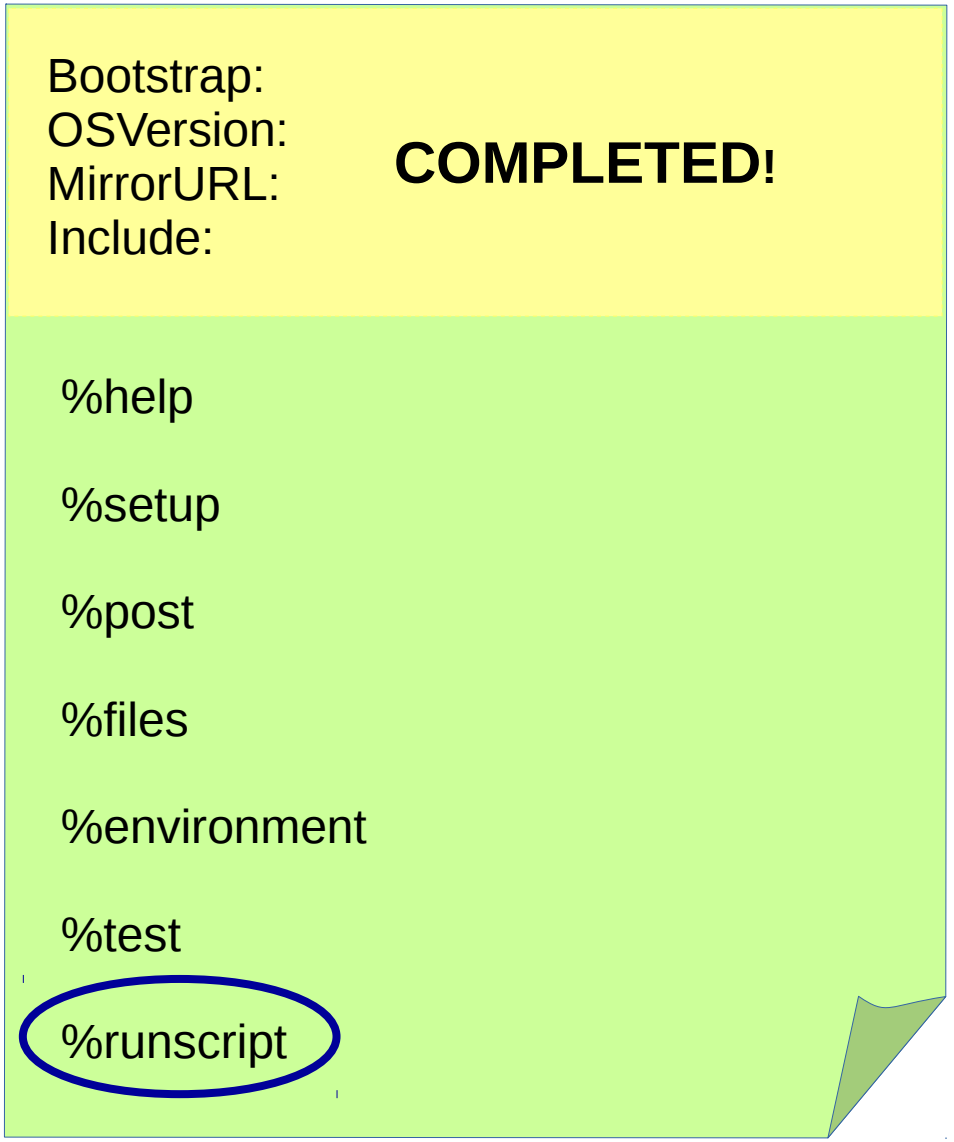
Tutorial Example

1. We will open our “GP3_singularity_recipe text file again and add a **%test** section:

```
%test
        #!/bin/bash
        echo "Testing: List of all R packages installed..."
        R --slave -e 'installed.packages()'
        echo "Testing: List of all Python packages installed..."
        pip freeze
        echo "List of all files in singularity /mnt file..."
        ls /mnt
        /usr/local/bin/chunky list
        /usr/local/bin/chunky show /mnt/run_GWAS_analysis_pipeline.py
```

2. We have used the **%test** to list: all the R packages installed, all the Python packages installed, all the files in the /mnt directory installed, all the python chunky pipelines installed and determine if those python chunky pipelines are configured.

Writing a recipe file workflow



HEADER

We just completed this!

SECTIONS

Next, we will look at the **%runscript** section

Section: %runscript

- The **%runscript** decorator is a place to list all the commands that will be run when Singularity runs your container image
- This section is required. This is the only section that will be executed when running your container:

```
singularity run <my container image>
```

- Every command listed in **%runscript** will always run in the order listed!

Tutorial Example

1. We will open our "GP3_singularity_recipe text file again and add a **%runscript** section:

```
%runscript
        echo "chunkypipes framework is ready!"
        cd /mnt
        exec /usr/local/bin/chunky run "$@"
```

2. Every time we run my container image, Singularity will print the "chunkypipes framework is ready!" statement, followed by changing to the /mnt directory within the Singularity container, and then running the command beginning with **exec**

Section: %runscript, tutorial continued...

Tutorial Example

3. The **exec** tells Singularity to spawn a new shell process within the container and run the **/usr/local/bin/chunky "$@"** command within the spawned shell

```
%runscript
        echo "chunkypipes framework is ready!"
        cd /mnt
        exec /usr/local/bin/chunky run "$@"
```

4. The **"$@"** part of the command means to expect and run all user specified inputs following the container run command. For example:

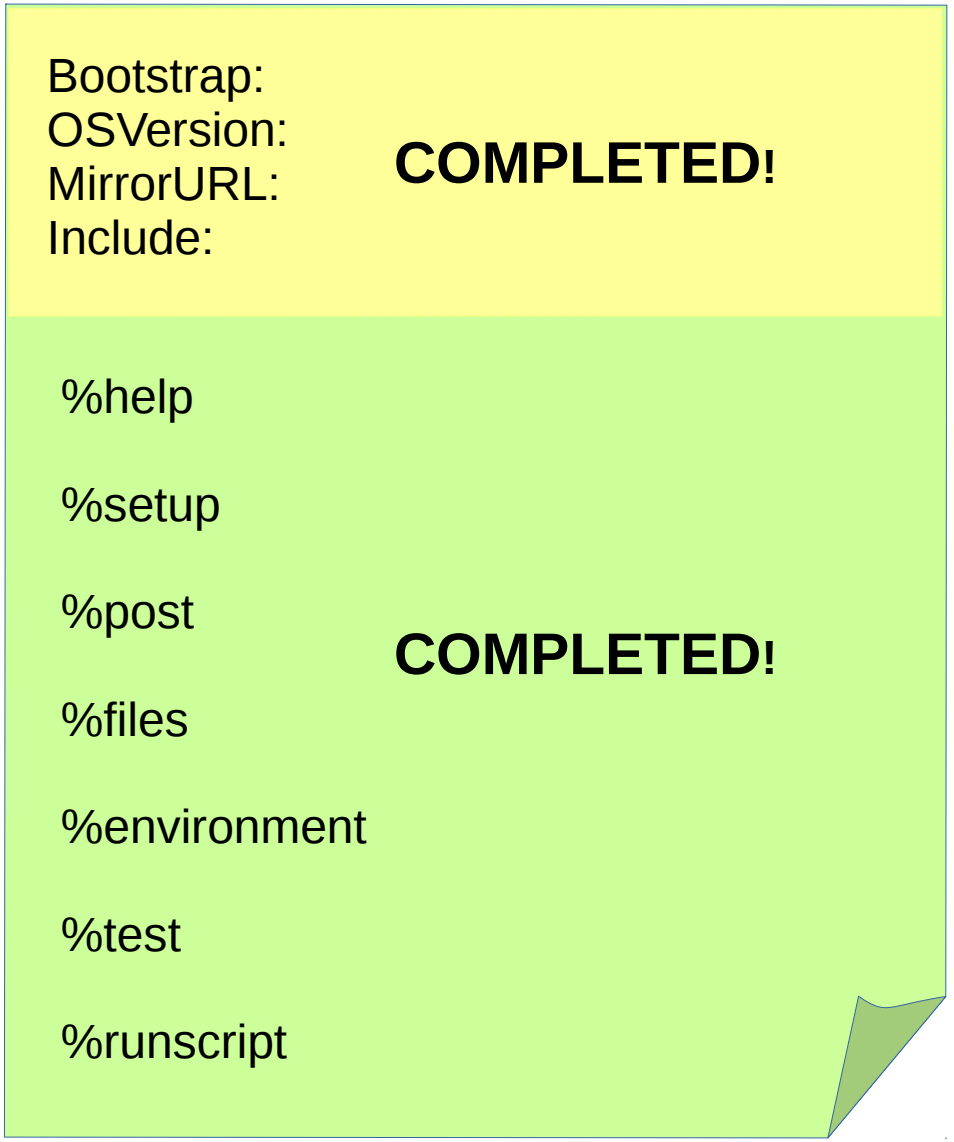
To run this particular program without using Singularity, we would normally input the following on the command line:

```
/usr/local/bin/chunky run run_GWAS_analysis_pipeline.py -inputPLINK
                    myfile.bed -phenoFile myFile.xlsx
```

To run this particular program using a container image from the Singularity recipe file we just created, we would input the following on the command line:

singularity run myContainer.simg run_GWAS_analysis_pipeline.py -inputPLINK myfile.bed -phenoFile myFile.xlsx

Writing a recipe file workflow



HEADER

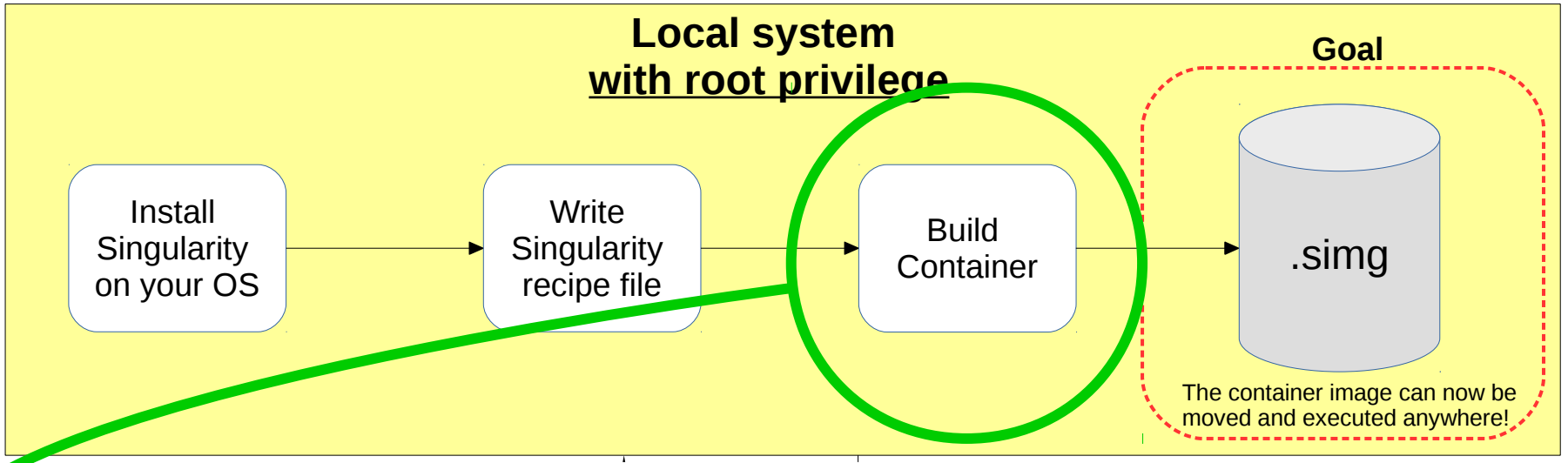
We just completed this!

SECTIONS

We just completed this!

Next, we need to build our container

Singularity Workflow

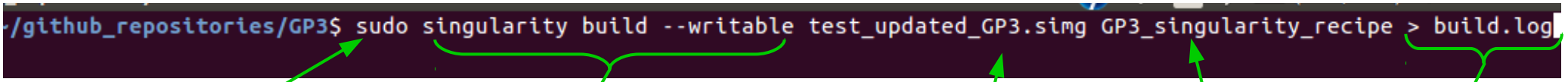


Focus our attention initially to the local system with root access

- The next slides will address the how to build a container image based on the Singularity recipe file that we just created

Building a container from a recipe file

- Once a singularity recipe file has been created, you can then run the **`build`** command to create a container image.
- Here is an example using the GP3 singularity recipe to build the GP3 container image:



Required to compile and build

singularity command
Build will create container, the --writable option makes it so that the container can be modified

Singularity recipe file to build the container image

Name of container to be created (doesn't have to end in .simg but the extension makes it easy to identify as a singularity image

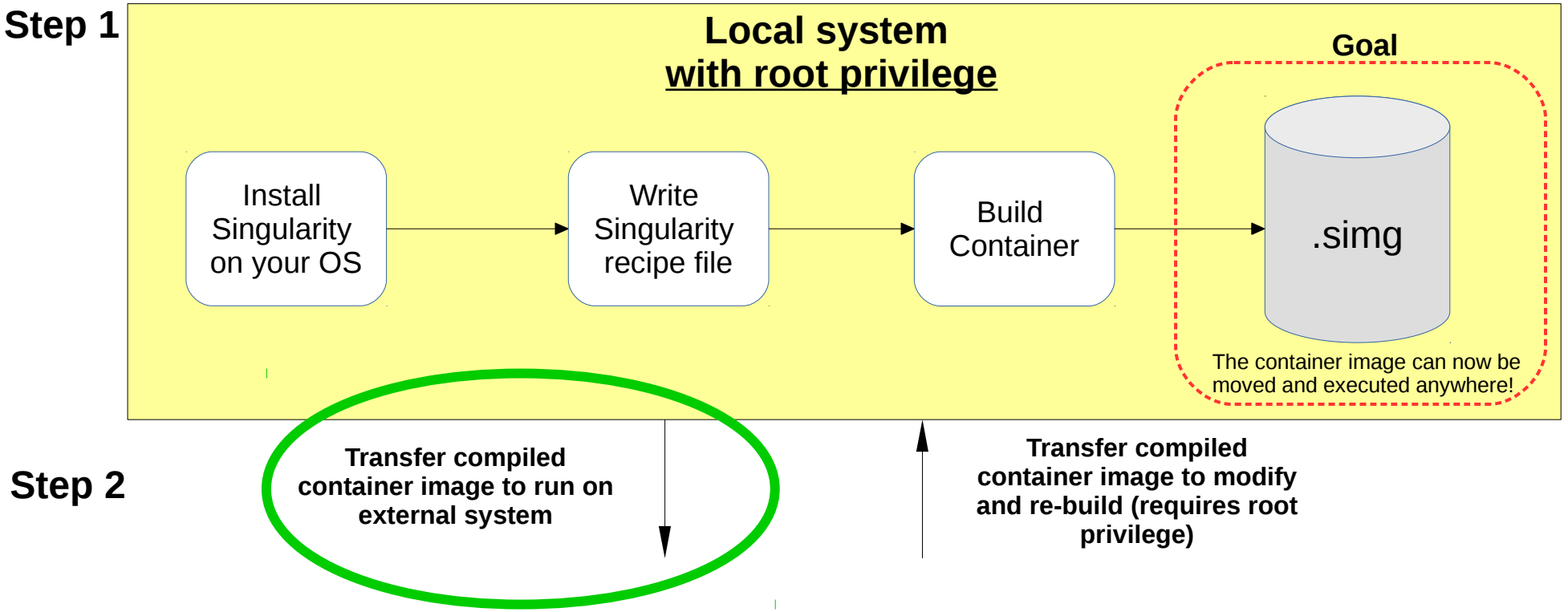
Not required, but recommended; lots of output and it makes it easy to parse for errors in building

Depending on how many installations you have and how large your OS is, it may take a little while. I suggest using **`screen`**.

<u>Upon successful completion:</u>
1. You should see your container name now created

<u>If not:</u>
1. check the log file for errors. Regardless, the log file should be checked to be sure the build was completed as expected

Singularity Workflow



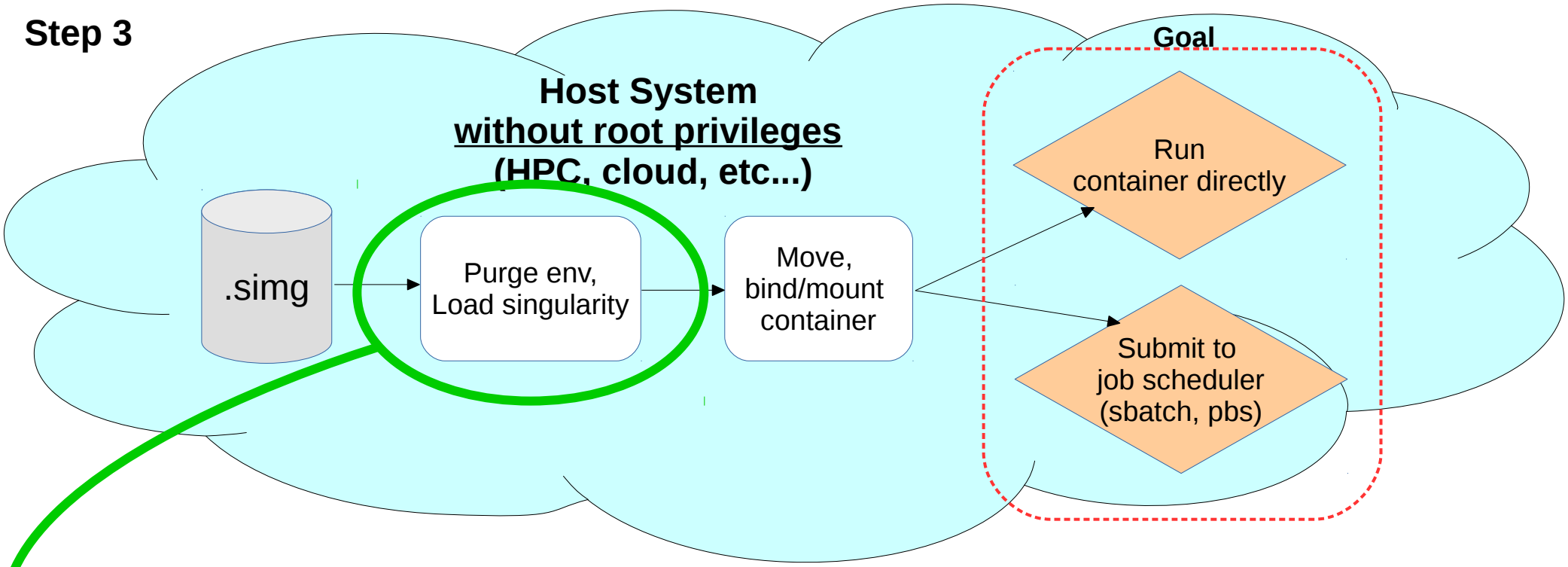
We have our container image completed, now we can move it or copy it or any external system that has Singularity installed on it and run it! (HPC, cloud)

If Rosalind is your host system:

1. Transfer the container via the staging area
2. Move container from staging area into project

For more information visit: https://github.com/tbrunetti/Rosalind_HPC

Singularity Workflow



Switch our attention to the host system without root access (HPC, cloud)

- It is important that when wanting to use the specific software compiled in the Singularity container, that you purge your HPC environment to not confuse software versions that may already be loaded, thereby, causing conflicts with software in your container!!

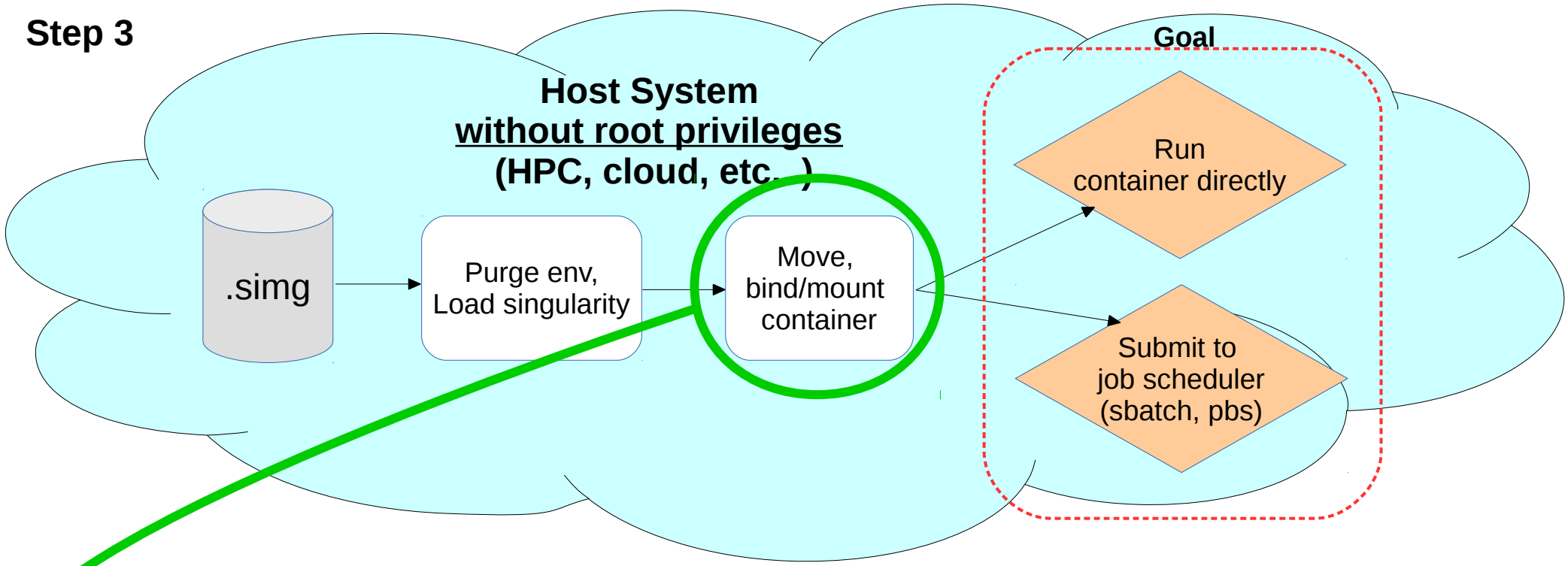
Cleaning Host System Environment

- On Rosalind, we highly recommend that you run the following commands before running your container:

```
module purge
module load slurm/16.05.8
module load singularity/2.4.2c
```

- Please note, that there are a few versions of Singularity available on Rosalind.
  - Load the version that matches the singularity version used to build the container to ensure compatibility.
  - DO NOT USE version 2.4.2, use version 2.4.2c, which is the patched version of 2.4.2

Singularity Workflow



Switch our attention to the host system without root access (HPC, cloud)

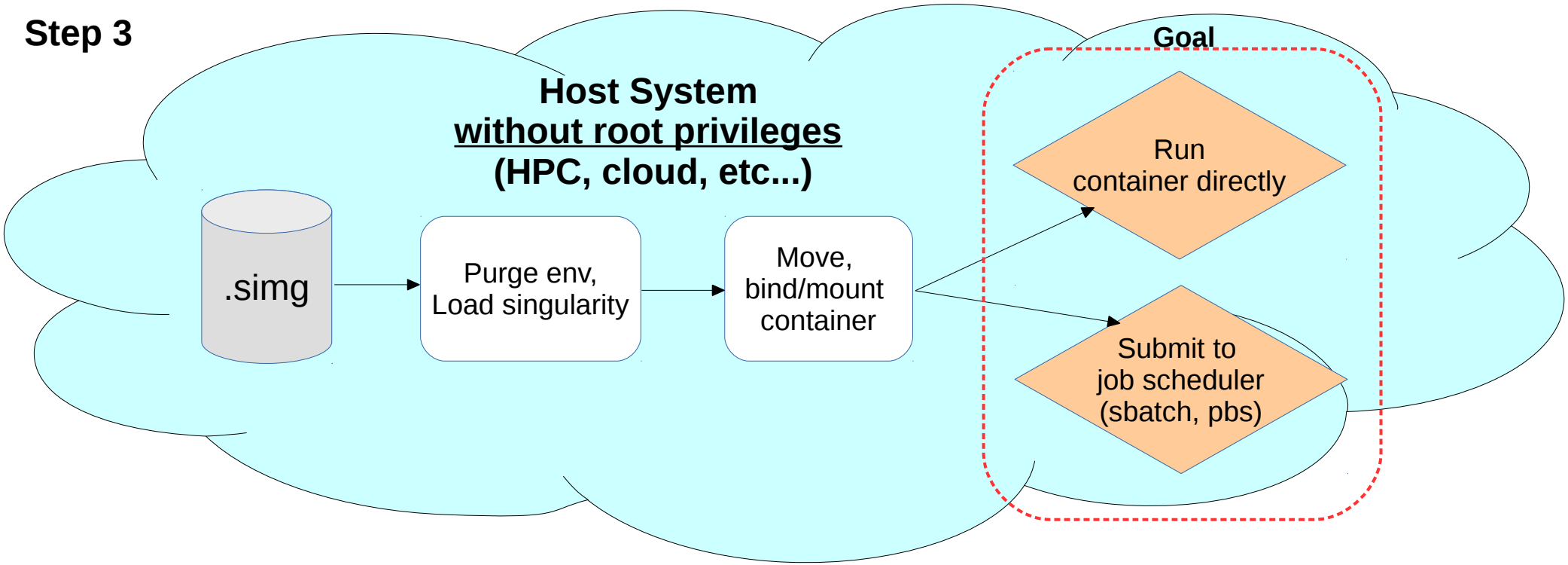
The next slide will discuss how to bind or mount your container and the benefits of doing so

Binding and mounting containers

- Any Singularity command has the option to use the `-B` argument

- This allows a container to be mounted to a specified path:
  - When bound, a container has access to any file or directory (assuming the user has permissions) beginning at the mount point and traversing through the full depth of the tree
  - Additionally, this allows the user to code output to be within the scope of the mount point on the host system and not within the Singularity container
  - Singularity DOES NOT support sharing or accessing files or directories between groups on Rosalind!

Singularity Workflow



Switch our attention to the host system without root access (HPC, cloud)

Running your container on Rosalind or other shared services on next slide

Submitting your container to a task scheduler

Example of a SBATCH script submitting the GP3 example container:

```
#!/bin/bash

#SBATCH --time=1000
#SBATCH --ntasks=6
#SBATCH --mem=90000
#SBATCH --job-name=singGP3
#SBATCH --output=singularity_test.out
#SBATCH --error=singularity_test.err

eval "#SBATCH --account=myGroup=${USER}"


module purge
module load slurm/16.05.8
module load singularity/2.4.2c


cd /gpfs/share/myProject/path_to_my_container



singularity run -B /gpfs/share/myProject example_GP3.simg run_GWAS_analysis_pipeline.py --config /gpfs/share/myProject/config.py.json -inputPLI
NK /gpfs/share/myProject/myPLINK.bed -phenoFile /gpfs/share/myProject/my_file.xlsx --outDir /gpfs/share/myProject/output_files_here --projectNa
me myGP3project
```

This sbatch script can be saved and submitted to SLURM as normal.

The run command in Singularity

General Singularity run command:

```
singularity run –B /path/to/mount/point myContiainer.simg
<any_optional_arguments, if container permits it>
```

It is worth mentioning again, that when the run command is called, all commands listed in the %runscript section of the recipe file are run in sequence

```
singularity run -B /gpfs/share/myProject example_GP3.simg run_GWAS_analysis_pipeline.py --config /gpfs/share/myProject/config.py.json -inputPLI
NK /gpfs/share/myProject/myPLINK.bed -phenoFile /gpfs/share/myProject/my_file.xlsx --outDir /gpfs/share/myProject/output_files_here --projectNa
me myGP3project
```

The mount point indicates that we can use this path followed by any subsequent file or path following this prefix and it will be visible to the container once the run command is called

Singularity Commands (not exhaustive) and Supported Functions

- `singularity shell myContainer.simg`
  - spawns interactive shell within container
- `singularity exec myContainer.simg <commands>`
  - allows you to run commands within the container
  - ex:

    ```
    singularity exec myContainer.simg ls /
    ```

    - lists all directories and files within the / directory within the container
- `singularity run myContainer.simg <optional commands>`
  - runs the %runscript section of the recipe file used to build myContainer.simg
- `singularity <command> myContainer.simg –help`
  - lists descriptions and arguments that can be used
- Singularity supports Linux piping and redirection commands
  - see “Compatibility with standard work-flows, pipes and IO” at http://singularity.lbl.gov/user-guide#compatibility-with-standard-work-flows-pipes-and-io

Resources and References

- Rosalind Home Page
  https://www.ucdenver.edu/Rosalind
- Rosalind GitHub
  https://github.com/tbrunetti/Rosalind_HPC
- Singularity Website and Documentation
  http://singularity.lbl.gov/
- Singularity Image Hub (shub)
  https://singularity-hub.org/
- Singularity GitHub
  https://github.com/singularityhub/
- Docker Documentation
  https://docs.docker.com/