Snakemake Vignette: Intro to Pipelines

The following documentation is to be used as an introduction to the understanding and development of Snakemake based pipelines which are run on a Sun Grid Engine (CentOS5) and support in Python (3.5.1).

Written By: Tim Boyarski, BSc

With assistance from: Lauren Chong, MSc

For: BC Cancer Agency – Lymphoid Cancer Research

Date of: March 21st, 2017

Contents

[I. Snakemake module organization 2](#_Toc478573393)

[II. Workspace organization 3](#_Toc478573394)

[III. Making a Snakefile building Python script (Aka, the build file). 4](#_Toc478573395)

[1) Activate your Conda environment and determine your working directory. 4](#_Toc478573396)

[2) Prepare your sample file, create the input/ directory, and store it in here. 4](#_Toc478573397)

[3) Create a file in the working directory called buildPipe.py and import the call function. 5](#_Toc478573398)

[4) Store in buildPipe.py the names of your files (YAML, JSON, Snakefile) and the path to the module directory. 5](#_Toc478573399)

[5) Copy line 5 from module/buildFile/buildFile.py into the build file. 5](#_Toc478573400)

[6) Read the ‘\_INCLUDE’ modules of *reBam*, *mPile*, and *varScan*, add Line 5 to buildPipe.py. 6](#_Toc478573401)

[7) Comment the build file. 6](#_Toc478573402)

[8) The resultant build file should look similar to Figure 11. 6](#_Toc478573403)

[9) Expected buildFile.py in Word QTP (UTF) format. 7](#_Toc478573404)

[IV. Build and configure the Snakefile 8](#_Toc478573405)

[10) Run the build file. 8](#_Toc478573406)

[11) Edit the Snakefile input directive. 8](#_Toc478573407)

[12) When finished, “Snakefile” should look similar to Figure 15. 9](#_Toc478573408)

[V. Configuring the pipeline 10](#_Toc478573409)

[13) Add the raw input into ‘input/rawBAM/’ and reference files into ‘input/ref/’ directory. 10](#_Toc478573410)

[14) Consider editing the ‘.JSON’ file. 10](#_Toc478573411)

[15) Edit the YAML file. 11](#_Toc478573412)

[Running a pipeline 12](#_Toc478573413)

[16) Dry-run the pipeline. 12](#_Toc478573414)

[17) Run pipeline 13](#_Toc478573415)

[VI. References 14](#_Toc478573416)

[VII. Appendices 14](#_Toc478573417)

## Snakemake module organization

All Snakemake modules are to be stored within the modules directory. Inside the directory, modules are to be developed with the intent of high cohesion. This means that we must attempt to keep modules small and focused. Each module will have at minimum three files. The structure is outlined in Figure 1 and Figure 2.

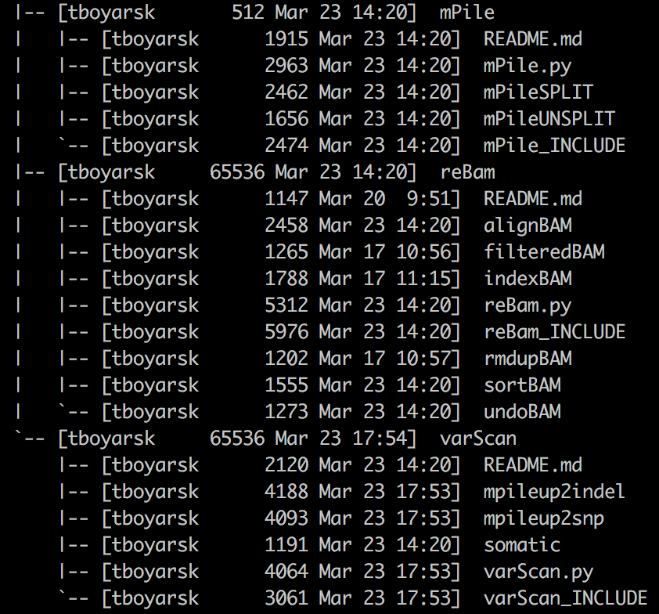


Figure 1. Tree structure of the module directory on Genesis

Snakemake/modules

/module0

/README.md

/module0.py

/module0\_INCLUDE

/module1

/README.md

/module1.py

/module1\_INCLUDE

/subModule1A

/subModule12B

…

/moduleN

/README.md

/moduleN.py

/moduleN\_INCLUDE

Figure 2. Hypothetical structure of modules directory

Naming: Modules should be named after their purpose. A *varScan* module uses varScan. A *reBAM* module re-assembles and processes ‘.BAM’ files. The modules themselves do not need to have the word ‘mod’ in them, it’s just used in the example above by chance. Sub-modules are then named by what they specifically accomplish. *sortBAM* sorts BAM files, *indexBAM* indexes ‘.BAM’ files, etc.

Multi-Rule Composition: Complex modules are decomposed into smaller events. We highly encourage that these small events are separated into their own Snakefile. They are all included into the core ‘\_INCLUDE’ file. For multi-part composition, the ‘\_INCLUDE’ should only contain the ‘include’ statements and a description about the submodules included. *When using submodules, there should be no rules in the ‘\_INCLUDE’ file.*

Core files:

* **README.md** – A read-me file written in markdown formatting.
* **moduleN.py** – A python script which accepts the ‘.YAML’ file, ‘JSON’ file, and Snakefile to be used. Acts as a script to populate the ‘.YAML’ file, ‘JSON’ file, and Snakefile.
* **moduleN\_INCLUDE** – The Snakefile to be included into the pipeline.
  + - **Single-Rule**: Contains only 1 rule.
    - **Multi**-**Rule**: Contains no rules. Contains submodule ‘include’ statements and a significant description for each submodule.
* **moduleN+1** – Contains only 1 rule. Used when module has multiple rules, each are isolated.

## Workspace organization

The code was developed in such a manner that the working directory may be located anywhere. The location should be on the network on which the computational processing will be occurring. The initially expected working directory structure is outlined in Figure 3 and Figure 4. ‘clean.sh’ is a file created out of personal preference to automatically cleanup the working directory, and is discussed in Appendix 1 in more detail.

workDIR/

clean.sh

/input

/sampleFILE.txt

/rawBam

/Pfeiffer2.bam

/Pfeiffer3.bam

/ref

/GRCh37-life.fa

/GRCh37-life.amb

/GRCh37-life.ann

/GRCh37-life.bwt

/GRCh37-life.pac

/GRCh37-life.sa

Figure 3. Hypothetical starting structure of the user working directory

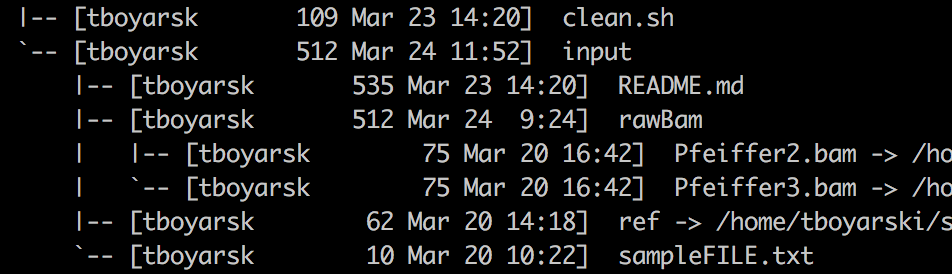


Figure 4. Starting tree structure of working directory on Genesis. Symlinks for files Pfeiffer2.bam, Pfeiffer3.bam, and the directory ref have all been truncated for ease of reading. The variety of linkages is to demonstrate what is acceptable and encouraged as to promote the sharing of files and stringent used of disk space.

**\*\*NOTE\*\***

* Users are expected to provide their own ‘.BAM’ files, and reference files ‘.fa’.
* The Snakefile will require only to be told about the ‘.fa’ file, but it will need the rest to exist.
* Sym-linked files or directories are acceptable and encouraged, see Figure 4.

All other files and folders within the workspace will be created by the python script. They assemble the pipeline. Users will first build this python script. When run, it will produce the hypothetical input as seen in Figure 5. The process of running it will produce output similar to what is seen in Figure 6. This is reporting for the user only, and it is not otherwise recorded by the system.

workDIR/

clean.sh

buildPipe.py

Snakefile

/input

/sampleFILE.txt

/config.yaml

/config.json

/rawBam

/Pfeiffer2.bam

/Pfeiffer3.bam

/ref

/GRCh37-life.fa

/GRCh37-life.amb

/GRCh37-life.ann

/GRCh37-life.bwt

/GRCh37-life.pac

/GRCh37-life.sa

/log

/module1

/module2

/log

/output

Figure 5. Hypothetical structure of the working directory after running the Snakefile building python script.

Logging redirected into module specific directories. Within the directory will be rule specific files. The python scripts have been coded to detect and create missing directories. Snakefile will create any missing directories as needed.



Figure 6. Example of output from running a Snakefile building python script.

## Making a Snakefile building Python script (Aka, the build file).

The first step in pipeline development is to create the python script which will assemble the pipeline automatically. The following steps describe how to generate a python file which when run, produces your Snakefile and all required configuration-based input files.

1. Activate your Conda environment and determine your working directory.   
   Snakemake pipelines have been developed to be run under the environmental management of an Anaconda environment. These environments can take up to a few minutes to load. Users can check what environments are available by using the call:

**$ conda info -e**

For the purpose of this tutorial, please use the recommended Anaconda environment. Anaconda environments are specific to the version of Anaconda being run. For this vignette, we will be using the following Anaconda environment, which can be activated directly or indirectly, noting version 4.3.0, as follows:

$ source ~/share/usr/anaconda/4.3.0/bin/activate CentOS5-Compatible

OR … (Depending how your path variables are setup)

$ source activate CentOS5-Compatible

There is no explicit reporting once the conda environment has been activated. Rather, users are notified to their active conda environment by a prefix addition to their bash prompt, as highlighted in the yellow box in Figure 7.



Figure 7. Text based notification of the Conda environment currently active for a single session within a single terminal window. The yellow box indicates the name of the active conda environment.

1. Prepare your sample file, create the input/ directory, and store it in here.  
   The scripts used to read your sample file expect a specific type of formatting to be used and for it to be in a directory named ‘input/’. There are three types of file formatting with a ‘.txt’ file which are currently acceptable. They are “single”, “pair”, and “csv”. To find more information about how to format your sample names input file, please refer to the ‘buildSample.py’ section of the Git-Repository link below:

<https://github.com/LCR-BCCRC/workflow_exploration/tree/master/Snakemake/modules/buildFile>

For the purposes of this vignette, please name your file “sampleFILE.txt”, it should contain “Pfeiffer2”. It should look identical to Figure 8.

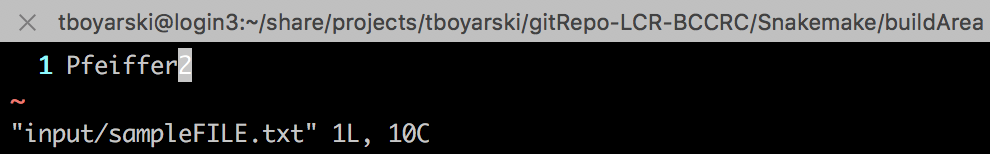


Figure 8. VIM screenshot of sampleFILE.txt. It contains a single line, and on that line is the sample name "Pfeiffer2".

### **Create a file in the working directory called buildPipe.py and import the call function.**

from subprocess import call

‘buildPipe.py’ is our Snakefile building python script. The above import statement gives Python access to the library and method required to make shell calls. This python script will be used to call the python scripts which exists in each module.

1. Store in buildPipe.py the names of your files (YAML, JSON, Snakefile) and the path to the module directory.  
   These variables are used to interact with each module’s python script. The “SAMPLE” parameter will only be used by one module (*buildFile*). The other four parameters are required by all modules. Please note that the sample, ‘.YAML’, and ‘.JSON’ file names have all been prefixed with the directory ‘input/’. Formatting should look similar to below:

SAMPLE="input/sampleFILE.txt"

YAMLFILE="input/config.yaml"

CLUSTERFILE="input/config.json"

SNAKEFILE="Snakefile"

snakeDIR="~/share/projects/path/to/snakeDIR/"

\*\*Warning: MS Word uses different Unicode quotation marks; Replace quotations if pasting the text above.

After Step 4, ‘buildFile.py’ should look similar to Figure 9.

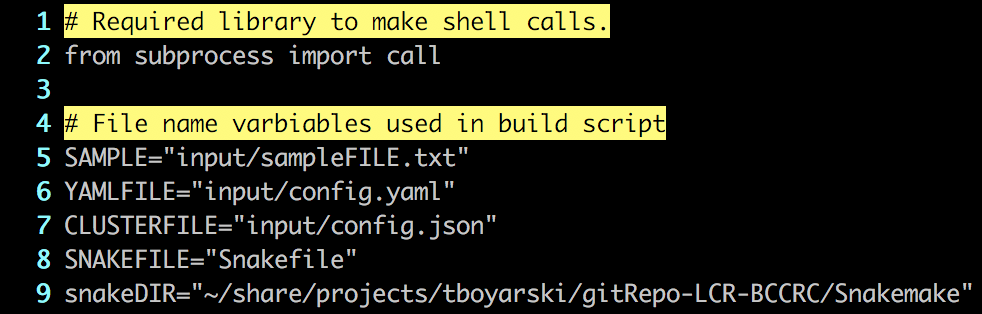


Figure 9. Example of what buildPipe.py should look like after completing 'Step 4'

### Copy line 5 from module/buildFile/buildFile.py into the build file.

Navigate to this file building module. It is the only module written purely in python. This call is used to automatically and consistently create the Snakefile, ‘.YAML’, and ‘.JSON’ file. Line 5 of ‘buildFile.py’ should be appended to the file. After Step 5, the buildFile.py should look similar to Figure 10.



Figure 10. VIM screenshot example of what buildPipe.py should look like after complete 'Step 5'.

### Read the ‘\_INCLUDE’ modules of *reBam*, *mPile*, and *varScan*, add Line 5 to buildPipe.py.

It would be unrealistic, and poor design, to include every possible module into your pipeline. Ideally, the user will navigate through the modules directory, reading the modules to better understand the purpose of each. Each “\_INCLUDE” file contains a line (Line #5) of python code which can be used to propagate the module into the user pipeline, much like what as performed in ‘Step 5’. This aids in the automation of Snakefile creation. Together, these lines constitute the core of the Snakefile building python script. This results in a robust record of how the Snakefile was assembled. We can easily regenerate the Snakefile at any time. Within each module, line 5 will look something similar to this:

call("python " + ROOT\_PATH + "/modules/reBam/reBam.py " \

+ YAMLFILE + " " + CLUSTERFILE + " " + SNAKEFILE, shell=True)

\*\*NOTE\*\* If you are familiar with module functions, just change the module names (E.g. XXXXXXXX = mPile)

call("python " + ROOT\_PATH + "/modules/XXXXXXX/XXXXXXXX.py " \

+ YAMLFILE + " " + CLUSTERFILE + " " + SNAKEFILE, shell=True)

These calls will add the relevant parameters to your ‘.YAML’ file, they will add the rule specific cluster arguments to the ‘.JSON’ file, and they will append directly into your Snakefile the comments and include statements of the respective module. The order in which these calls are listed inside of buildPipe.py will determine the order in which the information is appended to your input files. Typically, it is considered best practice to list the beginning modules first, and the later modules last, although this is not required. There is no limit to the number of modules you are able to include. Users should be mindful of duplication, as modules contain rules which can only be added once. Near the end of this build process we will need to manually edit small parts of the Snakefile. We will perform a quick manual check for duplicates at that time.

### Comment the build file.

Pipelines are often shared amongst individuals, as well, they are kept for long periods of time. It is important to comment your work such that those following or adopting your strategies can better understand the decisions that were made and the problems that were overcome.

### The resultant build file should look similar to Figure 11.

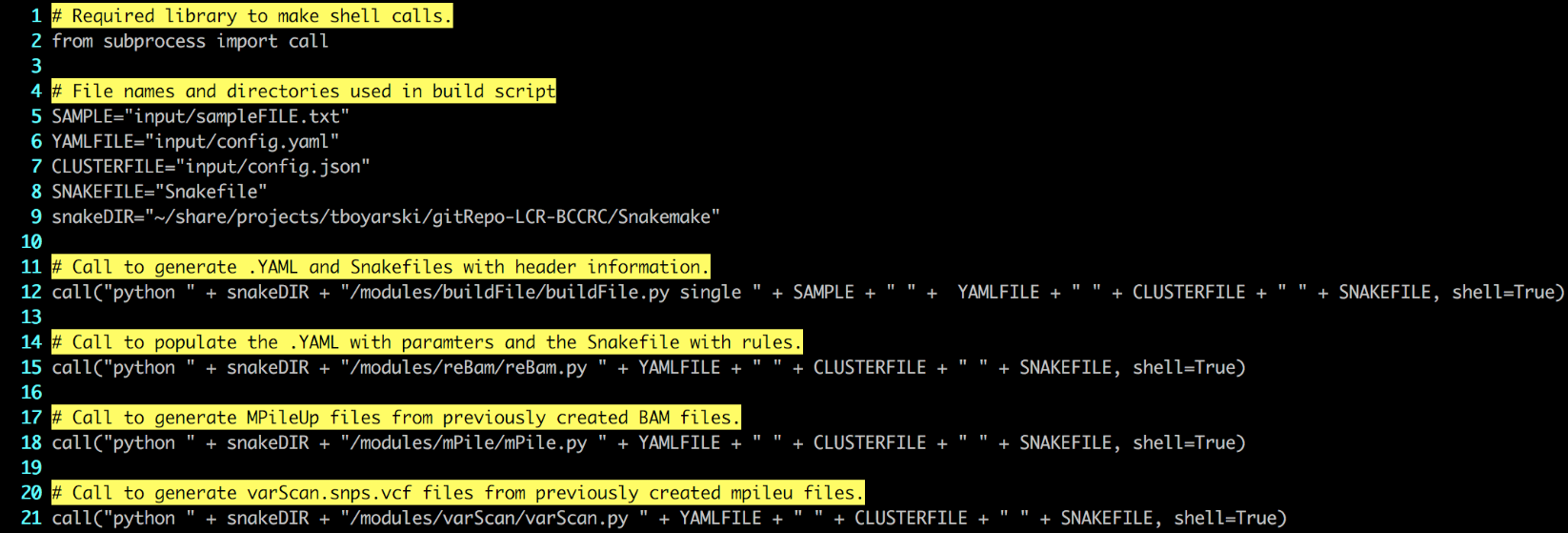


Figure 11. VIM screenshot of a finalized buildPipe.py file. This will generate a Snakefile capable of producing VCF annotations from a raw ‘.BAM’ file.

### Expected buildFile.py in Word QTP (UTF) format.

Just in case everything isn’t working and you need to copy from Figure 12 and paste.

\*\*Warning: MS Word uses different Unicode quotation marks; Replace quotations if pasting the text below.

# Required library to make shell calls.

from subprocess import call

# File name variables used in build script

SAMPLE="input/SampleSingleLines.txt"

YAMLFILE="input/yaml.yaml"

CLUSTERFILE="input/json.json"

SNAKEFILE="Snakefile"

ROOT\_PATH="~/share/projects/tboyarski/gitRepo-LCR-BCCRC/Snakemake"

# Generate .YAML, .JSON, and Snakefile with header information.

call("python " + ROOT\_PATH + "/modules/buildFile/buildFile.py single " + SAMPLE + " " + YAMLFILE + " " + CLUSTERFILE + " " + SNAKEFILE, shell=True)

# Populate .YAML with parameters and Snakefile with rules.

call("python " + ROOT\_PATH + "/modules/reBam/reBam.py " + YAMLFILE + " " + CLUSTERFILE + " " + SNAKEFILE, shell=True)

# Generate ‘.mpileup’ files from previously created ‘.BAM’ files.

call("python " + ROOT\_PATH + "/modules/mPile/mPile.py " + YAMLFILE + " " + CLUSTERFILE + " " + SNAKEFILE, shell=True)

# Generate ‘.varScan.snps.vcf’ files from ‘.mpileup’ files.

call("python " + ROOT\_PATH + "/modules/varScan/varScan.py " + YAMLFILE + " " + CLUSTERFILE + " " + SNAKEFILE, shell=True)

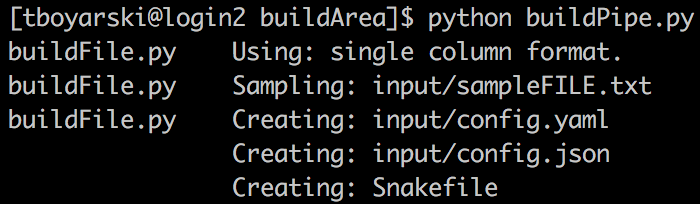
Figure 12. Text-based example of a finalized buildPipe.py file. This will generate a Snakefile capable of producing VCF annotations from a raw ‘.BAM’ file.

## Building and configuring the Snakefile

At this stage, it is assumed that the previous steps have been followed and the user now has a finalized buildPipe.py file. The following steps will walk the user through using this file and how it will generate the desired Snakefile. This vignette will provide insight as to interpreting the output generated by the process. The process will result in the automatic generation of a Snakefile and the required ‘.YAML’ and ‘.JSON’ configuration-based input files.

### Run the build file, see Appendix for corresponding sequence diagram.

It will generate the log directories of the respective modules which were included in the build file you created. Everything that is created by this python script is mentioned in output statements; the statements are not recorded. The statements’ purpose is to allow the user to validate what they told the script to do. An example of the output which should be provided when running buildPipe.py is provided in Figure 13. It clearly reports the file from which the call was made, the action being performed, and the file the action is being performed on.



Call from

Action

Performed on

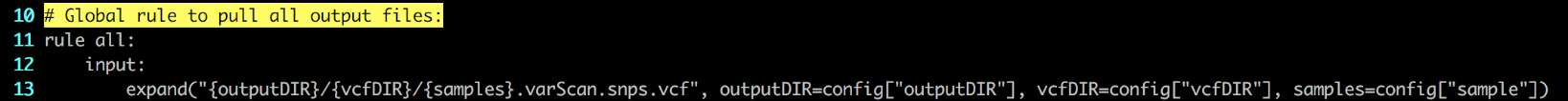
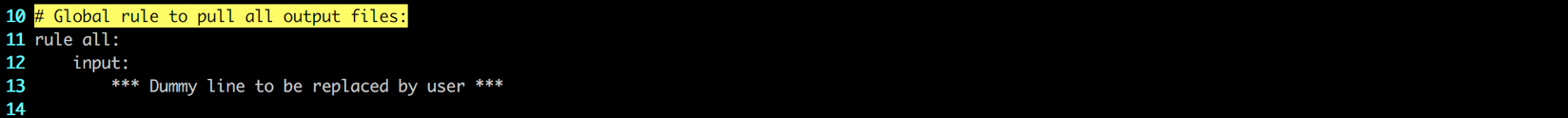
Figure 13. Example of the output provided when running the buildPipe.py script.

### Edit the Snakefile input directive.

When the Snakefile is generated, a dummy-value (Line #13) is provided in the input-field-line of the input-directive, this is intentional. It is to remind the user that they must to determine to which point the Snakefile pipeline is to progress. The replacement code will be from the desired endpoint module within the pipeline. This results in the Snakefile trying to generate this file, and all its dependencies. An example of replacing the dummy-value is provided in Diagram A of Figure 14. Its correct replacement is demonstrated in Diagram B Figure 14.

On the next page, Figure 15 outlines a production ready Snakefile. At the bottom of each included module (Figure 14; Line 28-35, 47, 61-62) are the input-field-line commands. One of these will be the call for Line #13.

\*\*Warning: *Users must remove the ‘#’ and indent 8 spaces; tabs are not allowed.*



…becomes…

Figure 14. Two VIM screenshots taken of code with the automatically generated Snakefile. This top image, A, is prior to the replacement of the dummy-value, the second image, B, as after the replacement of the dummy-value.

A

B

### When finished, “Snakefile” should look similar to Figure 15.



Figure 15. VIM screenshot on Genesis of a production ready Snakefile (A.K.A. Snakemake pipeline file). The module call from Line 61 is used on Line 13 to dictate that the pipeline should run from a ‘.BAM’ file to produce VCF annotated files of the samples’ Single Nucleotide Polymorphisms (SNPs).

## Configuring the pipeline

The configuration files for the pipeline have been created inside of the ‘input/’ directory.

### Add the raw input into ‘input/rawBAM/’ and reference files into ‘input/ref/’ directory.

The user may now populate the input directories using symlinks or by moving over the files to these directories. Create input directories as needed. To avoid the resource intensive task of moving files across networks or disks, it is recommended the user utilize bash symbolic links for both raw input and reference files. The resulting setup should appear similar to Figure 16.

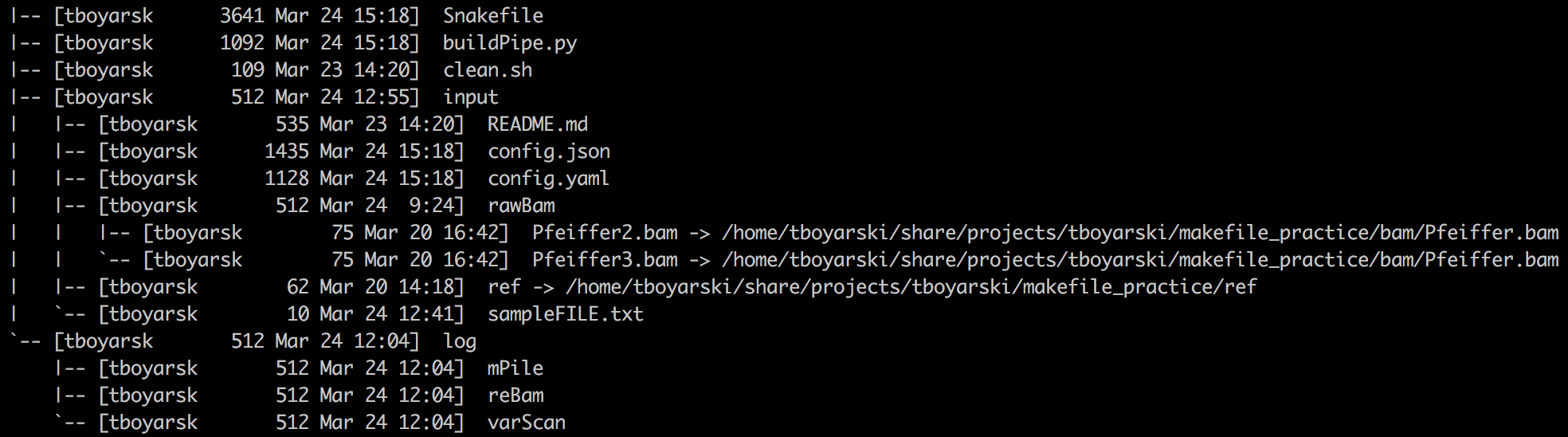


Figure 16. Example of the directory structure on Genesis after running a 'buildPipe.py', adding raw inputs by symlinking a file, and adding reference files by symlinking a directory.

\*\*\* Warning: The directory ‘ref/’ does not show its contents, it contains a reference genome (*XX1*.fa), and the following support files: *XX1*.fa.amb, *XX1*.fa.ann, *XX1*.fa.bwt *XX1*.fa.fai, *XX1*.fa.pac, *XX1*.fa.sa.

### Consider editing the ‘.JSON’ file.

The ‘.JSON’ file, typically, does not need to be edited. Users may edit the ‘.JSON’ file with the intent of increasing the resources of a specific rule, or to change the output directory of a specific rule, otherwise, very little should be changed in this file. Please note, in the example screenshot of Figure 17, the original file was whitespace-formatted from 37 Lines down to 14 Lines. Whitespace does not matter. Order of appearance does not matter.



Figure 17. Example of the 'config.json' file required by the Snakemake pipeline. This file provides the arguments required when submitting jobs to the cluster via DRMAA. The arguments '-o' and '-e' are for output redirection. Remaining arguments are provided by DRMAA via ‘qsub’ during job submission to the SGE scheduler.

### Edit the YAML file.

The ‘.YAML’ file contains a number of parameters which are either global or rule specific. It is worth checking out this file to become familiar with the optional granularity of the pipeline. Typically this involves setting the name and path of your reference file, toggling if a pipeline is to process ‘.BAM’ files by chromosome or as just a single file, which software is to be used to sort a ‘.BAM’ file, and other such options. When finished, “config.yaml” should look similar to Figure 18.

BAM Processing Tag: fileTAG

* “\_realigned” must be first, as this is recreating the ‘.BAM’ file which all other steps rely on.
* “\_sorted” should be last, as the other filtered remove data, doing this last may mean smaller file sizes.

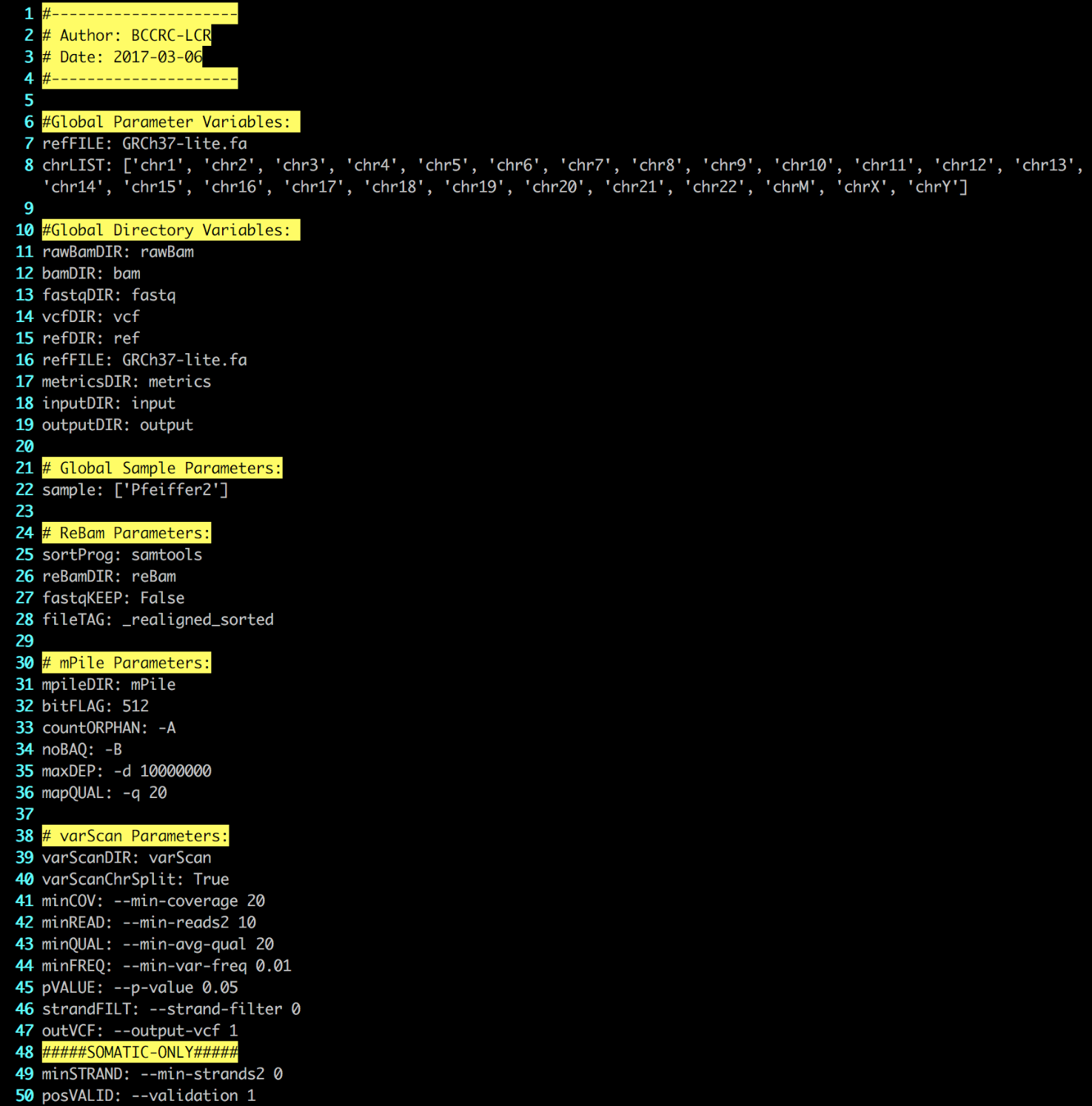


Figure 18. Genesis screenshot of the produced "config.yaml" file as generated by the python script "buildPipe.py". Due to consistent updates, the ".YAML" file produced may differ slightly from this image. The blue arrows indicate a few notable fields.

Reference Genome

Sample Names froenceGenome

BAM Processing Tags

Sample Names froenceGenome

## Running a pipeline

Your pipeline is now considered complete. There are a few more steps before we submit it to cluster.

### Dry-run the pipeline.

Before running the pipeline officially on the cluster, it is recommended that users first perform a dry run of their pipeline using the argument ‘-n’, as seen in Figure 19.

$ snakemake -n



Figure 19. Genesis screenshot of a Snakefile being dry-run. This is a linkage report which only determines anticipated outputs and jobs, it does not submit the jobs for processing.

### Run pipeline

If no errors are provided, we are then ready to make the cluster call to submit our pipeline. Users should note that Snakemake determines the working directory to be that from which the bash call to Snakemake was made. The modules were designed and organized in such a manner that the “Snakemake” file should be located in the root of the working directory, as outlined in this vignette. Other organizations are possible; however, they are outside the realms of this application. Please refer to the following steps below:

*Conditions:*

* 1. Be in the root of the working directory, where the “Snakefile” is located.
  2. If the Snakemake file is not named “Snakefile”, then the filename must be passed using “-s filename”.

Local Calls:

$ snakemake -s Snakefile 🡨 is equivalent to 🡪 $ snakemake

$ snakemake -s LCR1Pipe 🡨 is *not* equivalent to 🡪 $ snakemake

Or it can be submitted to the cluster using a Snakemake DRMAA submission process. Submitting to the cluster requires additional arguments. The call is saved for your convenience into the top (Line #5) of the generated Snakefile. Use the following bash command to have it printed out to terminal:

$ more Snakefile

The fifth line from the top contains the call required to submit the job to the cluster. Copy and paste it into terminal and hit enter. If the Snakemake pipeline file was called “LCR1Pipe”, the call should look like:

Cluster call:

$ snakemake -s LCR1Pipe --jobs 10 --cluster-config input/json.json

--drmaa "{cluster.clusterSpec}"

If successful users receive green and yellow text, as demonstrated in Figure 20. The call seen in Figure 20 was able to omit specifying the name of the Snakemake file because the file was named “Snakefile”.

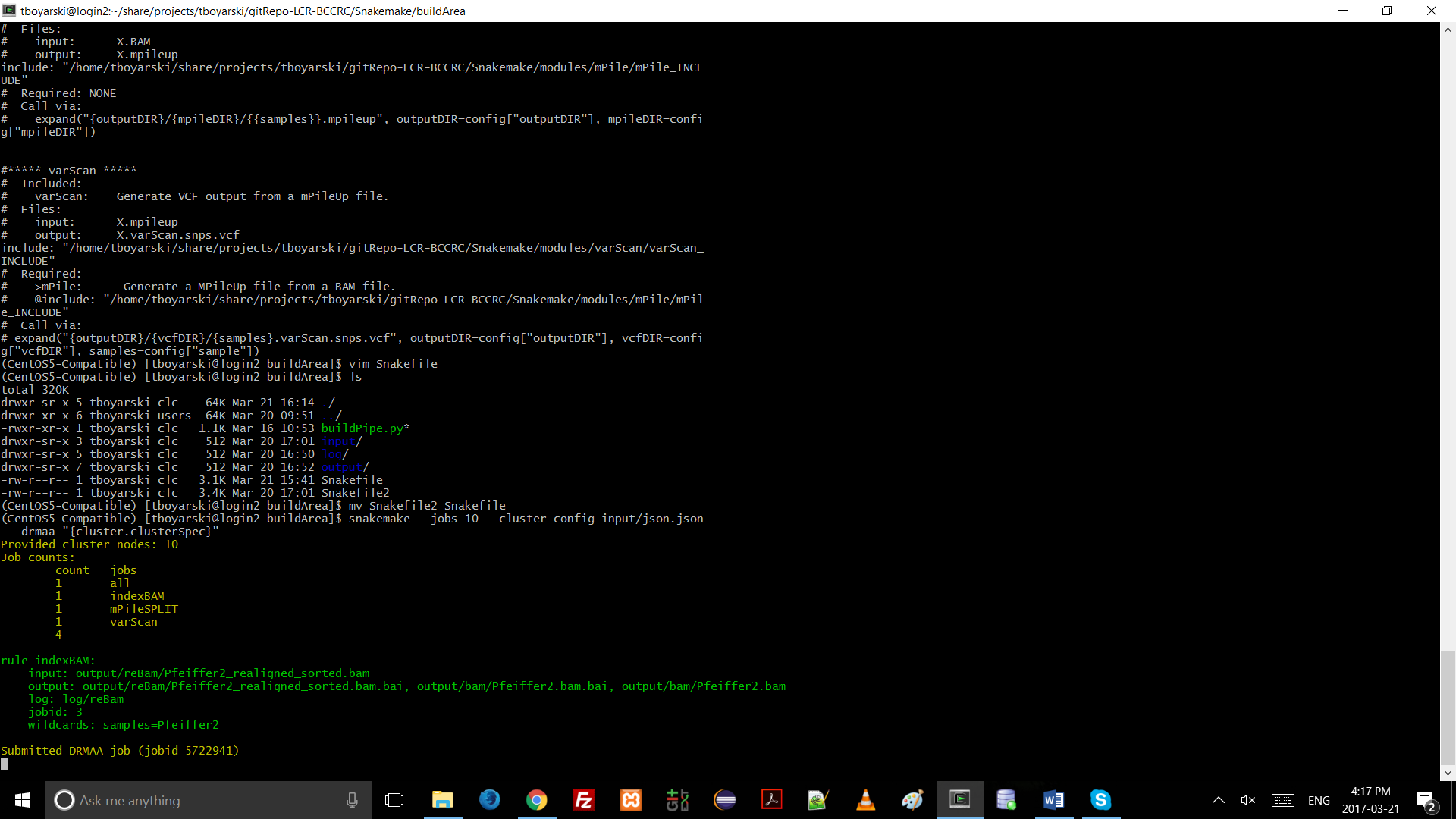


Figure 20. Genesis screenshot of a Snakemake pipeline file being submitted to the Sun Grid Engine job scheduler to be run. The "-s" argument is not provided because the Snakefile was named "Snakefile".

## References

Continuum Analytics (2017) Anaconda. [Available at: https://anaconda.org]

Koboldt, D. C., Chen, K., Wylie, T., Larson, D. E., McLellan, M. D., Mardis, E. R., ... & Ding, L. (2009). VarScan: variant detection in massively parallel sequencing of individual and pooled samples. *Bioinformatics*, *25*(17), 2283-2285. [Available at: http://varscan.sourceforge.net/using-varscan.html]

Köster, J., & Rahmann, S. (2012). Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, *28*(19), 2520-2522. [Available at: https://pypi.python.org/pypi/snakemake]

Python Software Foundation (2017) Python. [Available at: https://www.python.org/]

## Appendices

Clean.sh

This file is used when developing modules. It assists in cleaning out the working environment of all produced files and outputs so that both the python build file and the Snakemake calls can be performed again. This file is a convenience function and otherwise serves no purpose for this pipeline.

#!/bin/bash

rm -rf output/

rm -rf log/

rm -rf .snakemake/

rm Snakefile

rm input/config.yaml

rm input/config.json

Snakemake Pipeline - buildPile.py Sequence Diagram

Please refer to Figure 21 on the next page.

Aside from the core build module (buildFile.py), only three processing modules were included (reBam, mPile, and varScan). The system is not limited by the number of module it can accept, rather, it is limited by the number of modules which are able to be integrated with meaningful results. Additional of more modules would simply result in the continuation of the downward block cascade as seen in Figure 21.



Figure 21. Visio sequence diagram of the user interaction with buildPipe.py. The diagram demonstrates the interactions amongst files in the build process of a the Snakemake pipeline. Note the modularity and low coupling of the system. The user directly interacts with only a single file. Each file that buildPipe.py interacts with is given the ability to direct write information to the three configuration files YAML, JSON, and Snakefile. Message labels are not factual method calls, rather they are meant to describe the actions being performed. As indicated by the legend, light purple background shading at the top of the diagram indicates the file would be within the current working directory. Light green background sharing indicates the file is in the shared modules directory.