



THE UNIVERSITY OF UTAH

# The Snakemake Workflow Manager

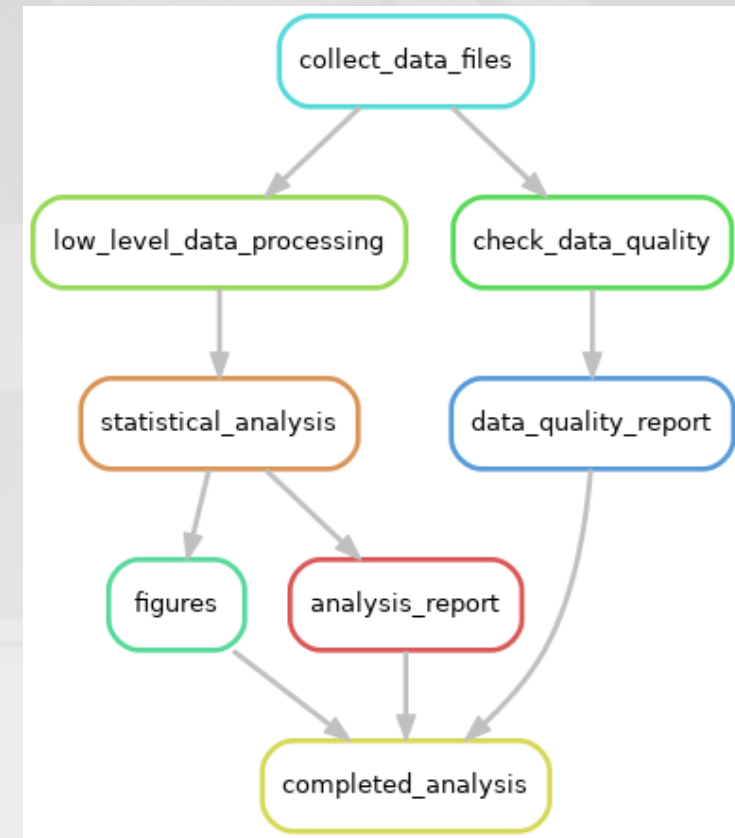
Brett Milash

Center for High Performance Computing

University of Utah

# A workflow manager is software that:

- Conducts a complex workflow or analysis
- Follows dependencies from results back to configuration and input files
- Executes commands step-by-step to carry out workflow



# Why use a workflow manager?

- Human efficiency and convenience
- Computational efficiency – only the required steps are executed
  - Great when your workflow needs to be restarted
  - Handles parallel execution
  - Handles execution on a compute cluster
- Improved reproducibility
- Portability between clusters, institutions
- Modularity – re-use and standardization

# Why choose snakemake?

Over 100 different workflow managers:

<https://github.com/pditommaso/awesome-pipeline>

Snakemake is:

- ☐ Actively used and developed
- ☐ Can be configured for local and/or cluster execution
- ☐ Native SLURM support
- ☐ No significant system administration support required
- ☐ General purpose (not just for bioinformatics, for example)
- ☐ Significant functionality bang for your learning buck

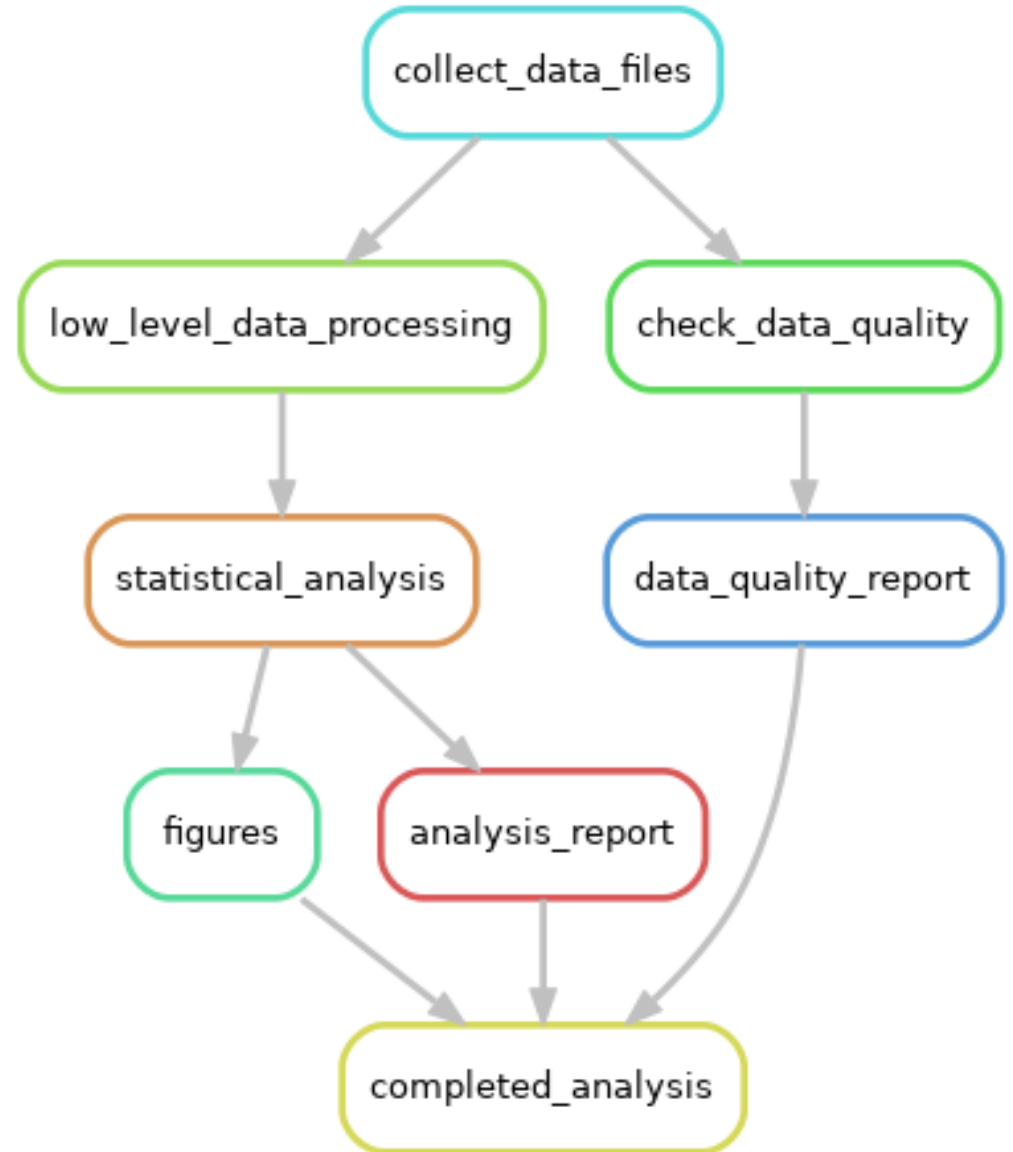
# Installation options

- Use the CHPC module:
  - `module load snakemake/6.4.1`
- Install your own using pip:
  - `pip install --user snakemake`
  - `export PATH=$HOME/.local/bin:$PATH`
- Install your own using anaconda:
  - `module load anaconda3`
  - `conda install -c bioconda -c conda-forge snakemake`

# Snakemake principles

---

- Based on same ideas as Linux “make” utility
- Workflow broken into steps, each governed by a “rule”
- Each rule executes one or more commands to transform input file(s) into output file(s)
- Rules linked together into a directed acyclic graph (DAG) when output files of one rule match input files of another rule
- The “ultimate goal” is the output of the first rule in the workflow
- Minimum work done to produce output of first rule
- Rule is triggered if output is missing or older than input







THE UNIVERSITY OF UTAH

# Simple example

See the example at:

<https://github.com/CHPC-UofU/workflows-with-snakemake/tree/main/Examples/HelloWorld>

# Snakemake workflows are built out of rules

```
rule link:  
    input: "hello_world.o"  
    output: "hello_world"  
    message: "Rule {rule} linking .o file {input}"  
    shell: "gcc -o {output} {input}"
```

Rules can have:

- names
- inputs
- outputs
- actions (shell or python)

Rules:

- are linked implicitly
- (or explicitly)
- can emit messages
- are executed in parallel if possible
- are executed locally or on a cluster

The first rule defines the default “target” for the workflow



# Snakefile syntax

- Snakemake workflows ("snakefiles") are python code
- All the python syntax rules apply:
  - Input and output file names in quotes
  - Shell commands in quotes
  - Whitespace / indentation is significant
  - Use either tabs or spaces (not both)
- Your snakefiles can include blocks of python code

# Rule inputs

- Inputs are one or more file names, in quotes, comma-separated
- Inputs are optional
- Inputs can have “symbolic” names

```
rule align:
  input: index="hg19", data="sample1.fastq"
  output: "sample1.sam"
  shell: "bwa mem {input.index} {input.data} -o {output}"
  message: "Rule {rule} aligning input file {input.data}"
```

# Rule outputs

- Just like inputs: one or more file names, in quotes, comma-separated
- Can have "symbolic names"
- Outputs are optional - common in top-level rule that simply checks if inputs are present.

```
rule align:
    input: index="hg19", data="sample1.fastq"
    output: "sample1.sam"
    shell: "bwa mem {input.index} {input.data} -o {output}"
    message: "Rule {rule} aligning input file {input.data}"
```

# Rule actions: the “shell:” section

- This is where you encode the actual work of the workflow
- By default: /bin/bash in strict mode (set –euo pipefail)
- Multi-line shell statements: use triple-quotes
- Can load modules, only affects the current rule.

```
rule link:
    input: "hello_world.o"
    output: "hello_world"
    shell: """
        module load gcc/6.1.0
        gcc -o {output} {input}
        """
```

# Rule “run:” section: action as python code

- Instead of bash, the action can be written in python
- Put this in the “run:” section of the rule
- Note there are no quotes around the python code

```
rule usercount:
    input: "userfile.txt"
    output: "users.count"
    run:
        users=set()
        with open(input[0]) as infile:
            for line in infile:
                unid=line.split()[0]
                users.add(unid)
        with open(output[0],'w') as outfile:
            print(f"There are {len(users)} users.",file=outfile)
```



# Rule messages

- Rules can emit messages with the “message:” section
- Messages are optional
- Really useful for monitoring your workflow
- Can access the inputs, outputs with {input}, {output}
- Can access the rule name as {rule}

```
rule align:
  input: index="hg19", data="sample1.fastq"
  output: "sample1.sam"
  shell: "bwa mem {input.index} {input.data} -o {output}"
  message: "Rule {rule} processing input file {input.data}"
```



# Snakemake command line arguments

First, need to load the module:

```
$ module load snakemake/6.4.1
```

Run snakemake using 1 core on default "Snakefile", default (i.e. first) rule:

```
$ snakemake -j1
```

Run snakemake using 1 core on non-default snakefile:

```
$ snakemake -j1 -s my_snakefile
```

Run snakemake using 1 core on non-default rule:

```
$ snakemake -j1 rule_name
```

Read the snakemake help:

```
$ snakemake --help
```



THE UNIVERSITY OF UTAH

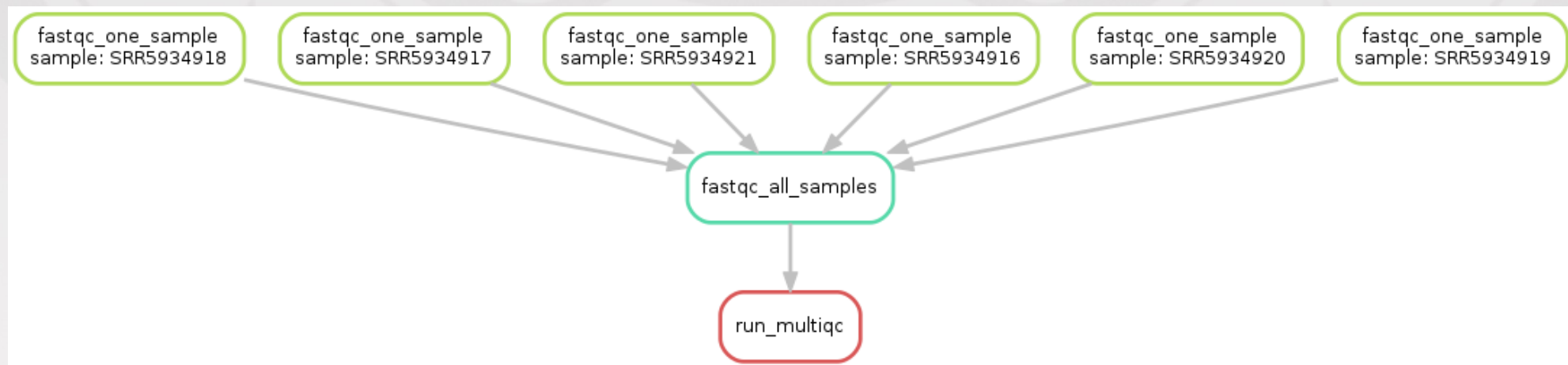
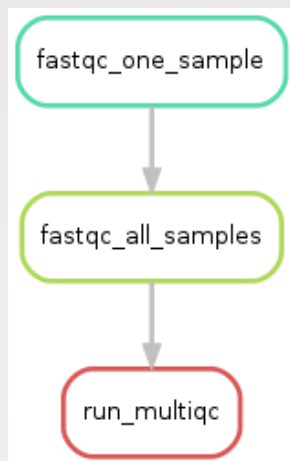
# Exercise1 - Simple workflow

See the exercise 1 instructions here:

<https://github.com/CHPC-UofU/workflows-with-snakemake/tree/main/Exercises/Exercise1>

# Graphical output

- Rule graph
  - Shows in general how rules depend on one another, but not the actual inputs/outputs
  - `snakemake -s snakefile --rulegraph | dot -Tpng > rulegraph.png`
- Directed Acyclic Graph (DAG)
  - all targets represented
  - Completed rules have dashed outline
  - `snakemake -s snakefile --dag | dot -Tpng > dag.png`



# Directories as input or output

- In snakemake version 5.0 or later:
  - Directories as input or output must be specified with `directory()`
    - input: `directory("data_directory"), "data_file"`
- In older version of snakemake:
  - Directories as input or output are just named like regular files
    - input: `"data_directory", "data_file"`

# Wildcards: filename patterns

- These make rules reusable, not tied to specific files
- Rules with wildcards are ideal for parallel execution

## How to do it:

- Create one rule that handles a single input -> output action using {variable} as a placeholder for the variable part of the input and output file name(s). This acts as a **template**.
  - Example: rule transforms input file "{sample}.data" to output file "{sample}\_report.pdf"
- Create another rule whose **input** lists all the template rule's output files.
  - You can use the `expand()` function for this.
  - Python lists and list comprehension are useful here.



# Snakemake wildcard example

```
# Given data files A.txt, B.txt, ..., F.txt, create a .md5 checksum file for
# each one, named A.md5, B.md5, ..., F.md5 .
#
# Here are the sample names that are embedded in the file names:
samples=[ 'A', 'B', 'C', 'D', 'E', 'F' ]

# These two rules create the .md5 checksum file for each sample's .txt file.
rule all_checksums:
    input: expand("{sample}.md5", sample=samples)
    # This produces the list ["A.md5", "B.md5", ... "F.md5"]

rule one_checksum:
    message: "Creating checksum file {output} for data file {input}."
    input: "{sample}.txt"
    output: "{sample}.md5"
    shell: "md5sum {input} > {output}"
```



# Exercise 2: Workflow with wildcards

See the exercise 2 instructions here:

<https://github.com/CHPC-UofU/workflows-with-snakemake/tree/main/Exercises/Exercise2>

# Snakemake on a cluster

- Any snakemake workflow can run on a cluster:  
`snakemake --cluster-config cluster.yaml --jobs 20 ...`
- Cluster configuration file can be in JSON or YAML format
- The catch is that we must tell snakemake how to start a job:
  - `--cluster "sbatch -A {cluster.account} -p {cluster.partition}"`

# Cluster configuration

- Basic cluster configuration file:

```
# cluster.yaml - cluster configuration for my snakemake job.
__default__:
  partition: slurm_partition
  account: slurm_account
  time: 1:00:00
  nodes: 1
```

- The `__default__` config applies to all rules
- Can override default with rule-specific configurations

```
image_processing:
  partition: kingspeak_gpu
  account: kingspeak_gpu
```

# Local rules

- When running on a cluster, may want to specify some rules NOT run on the cluster
- localrules: rule1, rule2, rule3
- Snakemake knows to run rules without an action (e.g. “shell:”) locally.

# Watching your workflow run on the cluster

- Run the squeue command to see your SLURM jobs:
  - `watch -n 3 squeue -M all -u $USER` # Check jobs on all clusters every 3s.
- You can get fancy with the output:
  - `watch -n 3 squeue -M all 3 -u $USER -o "%.6i %.10P %.7a %.20j %.2t %.6M %R"`





THE UNIVERSITY OF UTAH

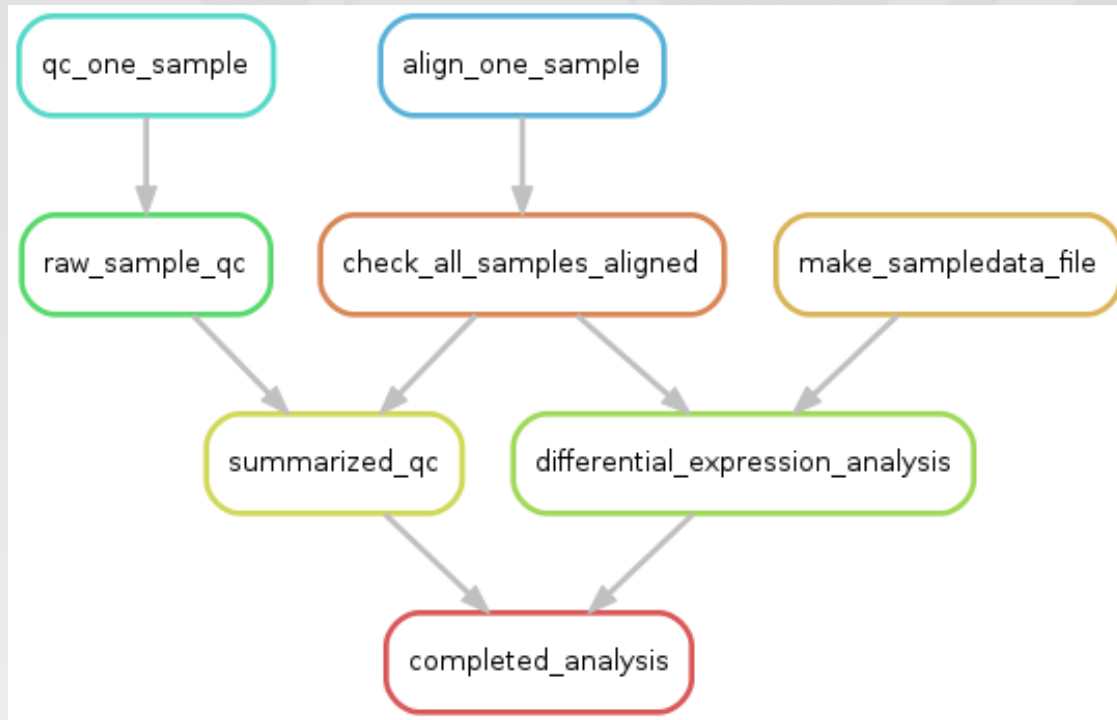
# Snakemake exercise 3

See the exercise 3 instructions here:

<https://github.com/CHPC-UofU/workflows-with-snakemake/tree/main/Exercises/Exercise3>



# Modular workflows



```
# In main Snakefile:  
include: "Snakefile.qc"
```

```
# Snakefile.qc  
rule summarized_qc:  
    input: ...  
    output: touch("qc.done")  
    shell: ...
```

```
rule qc_one_sample:  
    input: ...  
    output: ...  
    shell: ...
```

```
rule raw_sample_qc:  
    input: ...  
    output: ...
```

# Developing complex workflows

1. Define “skeleton” of workflow, link rules together using touch().
2. Start at beginning, implementing one rule at a time, testing as you go.
3. Use a small data set for testing, fast feedback
4. Implement the cluster configuration
5. Re-test
6. Run it with real data set



# Granularity

- Fine-grained
  - Many rules, simple shell statements
  - Efficient for local rules, easy debugging
  - Inefficient for cluster jobs, as each rule requires submitting a job
- Coarse-grained
  - Few rules, complex shell statements
  - More efficient on clusters

# Handling batches

- On a cluster, the snakemake paradigm maps the execution of one rule to one SLURM job – this may not fit your workflow well
  - Rule execution may be too small to fully occupy a node
  - Wait time in the SLURM queue on a busy cluster
- Solutions:
  - Write rules that process batches of samples or values
  - Use shared partitions in SLURM

# Snakemake is container-friendly

- Snakemake supports running code in containers using singularity
- See: <https://snakemake.readthedocs.io/...#running-jobs-in-containers>



# Snakemake may not be right for you

- What if your inputs and outputs aren't files?
- What if your cluster doesn't use SLURM or LSF?
  - HTCondor (Open Science Grid: > 1.2 billion core hours last year)
- What if your workflow changes?
  - nextflow: <https://www.nextflow.io/>
  - non-file inputs and outputs
  - support for HTCondor (OSG) and many other schedulers
  - workflow file is part of the workflow – when a rule changes, it gets re-run