

HPC-Pipes Software managers II

from the
Health Data Science
Sandbox



Alba Refoyo Martinez, PhD

Data scientist

Center for Health Data Science (HeaDS)

UNIVERSITY OF
COPENHAGEN





snakemake

I - Basics



II - Advanced

- Resources and parallelization
- Other cool snakemake features

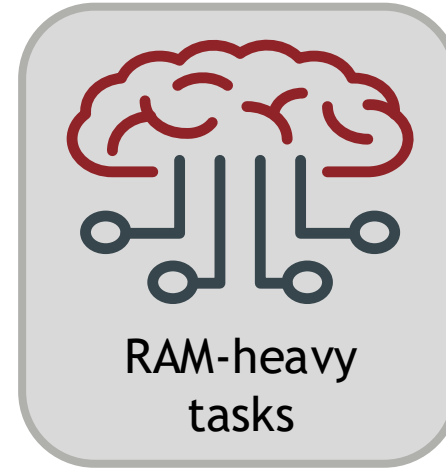
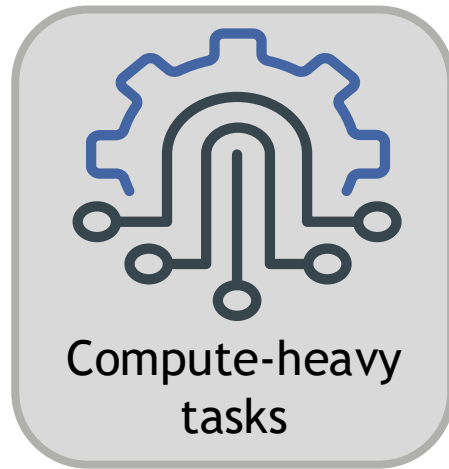
Parallelization

Disjoint paths in the DAG of jobs can be executed in parallel





Defining resources: what type of task are you handling?



Determine the task(s) type before designing the pipeline



Defining resources

Check if the software is multithreaded. Run tests to estimate the optimal #threads

Request what your task can actually use (allocated resources for you are effectively blocked from use by others)

Over-requesting wastes both system efficiency and funds



Defining resources

```
rule sort:
  input:
    "path/to/{dataset}.txt"
  output:
    "{dataset}.sorted.txt"
  threads: 4
  resources: mem_mb=100
  shell:
    "sort --parallel {threads} {input} > {output}"
```

Define
additional
resources

Refer to defined
thread number



Global vs. local resources

Global resources apply across all jobs (even if executed on different machines)

```
snakemake -c1 --resources  
mem_mb=1000
```

Local resources apply to a single job

```
rule ...:  
    output: ...  
    threads: 1  
    resources:  
        mem_mb=1000  
    shell: "..."
```

Without keeping track of resource usage, you won't actually know how to optimize it!



Exercise: parallelization



~ 3 min

How many jobs will Snakemake run, and how many cpus will be utilized in each scenario?

S1. Execute workflow with 8 cores

```
snakemake --cores 8
```

S2. Execute workflow with 2 cores

```
snakemake --cores 2
```

S3. Execute workflow with 10 cores

```
snakemake --cores 10 --resources  
mem_mb=100
```

```
rule sort:  
    input:  
        "path/to/{dataset}.txt"  
    output:  
        "{dataset}.sorted.txt"  
    threads: 4  
    resources: mem_mb=100  
    shell:  
        "sort --parallel  
{threads} {input} > {output}"
```



Scheduling

Available jobs are scheduled to

- maximize parallelization
- prefer high priority jobs
(**priority** directive)
- minimize temp file lifetime
temp()

while satisfying resource constraints

$$\max U_t \cdot 2S \cdot \sum_{j \in J} x_j \cdot p_j + 2S \cdot \sum_{j \in J} x_j \cdot u_{t,j} \\ + S \cdot \sum_{f \in F} \gamma_f \cdot S_f + \sum_{f \in F} \delta_f \cdot S_f$$

subject to:

$$\sum_{j \in J} x_j \cdot u_{r,j} \leq U_r \quad \forall r \in R$$

$$\delta_f \leq \frac{\sum_{j \in J} x_j \cdot z_{f,j}}{\sum_{j \in J} z_{f,j}} \quad \forall f \in F$$

$$\gamma_f \leq \delta_f \quad \forall f \in F$$

$$x_j \in \{0, 1\}$$

$$\gamma_f \in \{0, 1\}$$

$$\delta_f \in [0, 1]$$

job selection

job priority

job resource usage

job thread usage

job temp file consumption

temp file lifetime fraction

temp file deletion

temp file size

free resources



Exercise: scheduling problem

We have 2 completed jobs
(all jobs same priority)

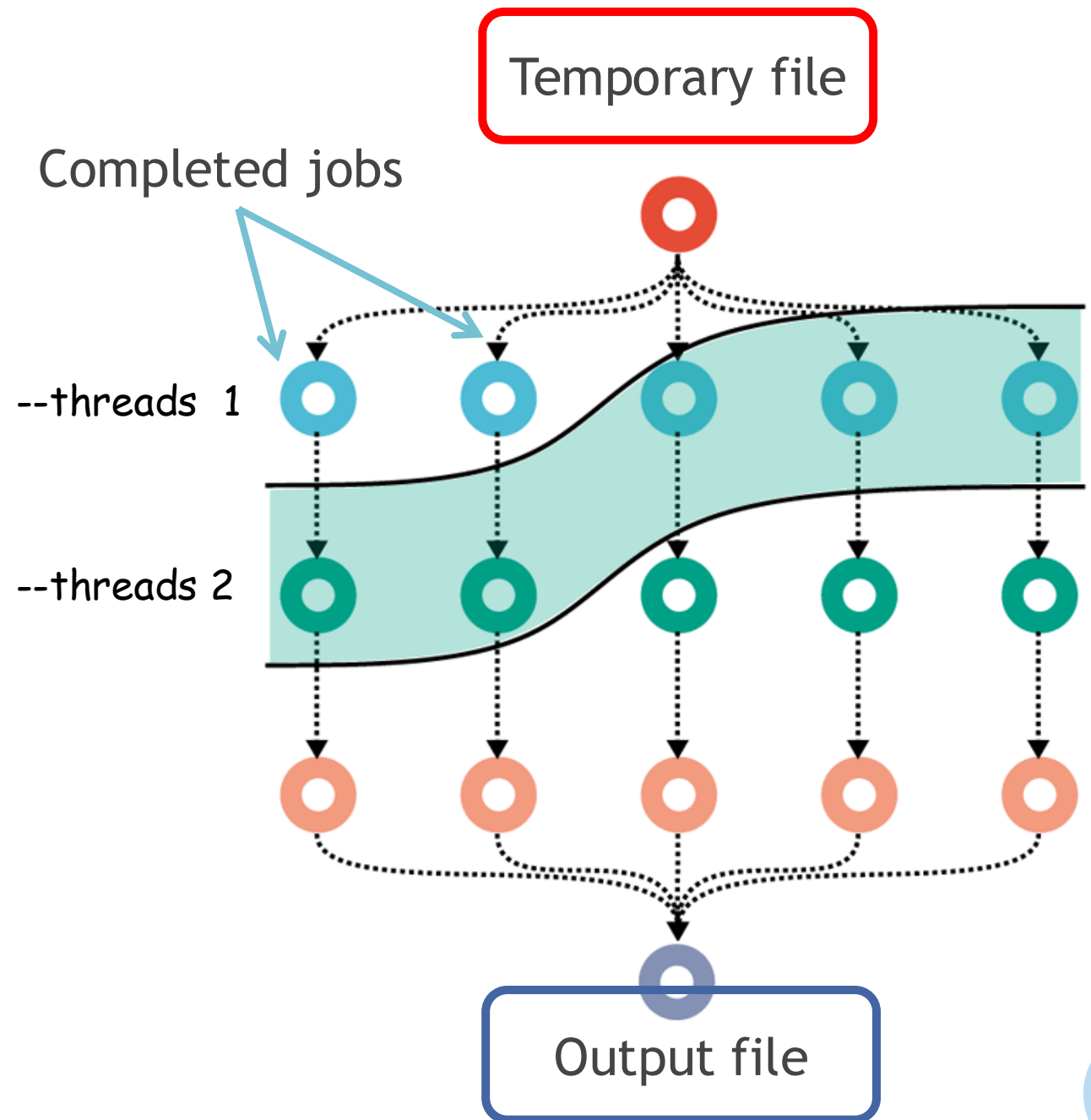
Blue rule: 1 cpus

Green rule: 2 cpus

5 Jobs ready for scheduling

5 cores requested!

> `snakemake --cores 5`

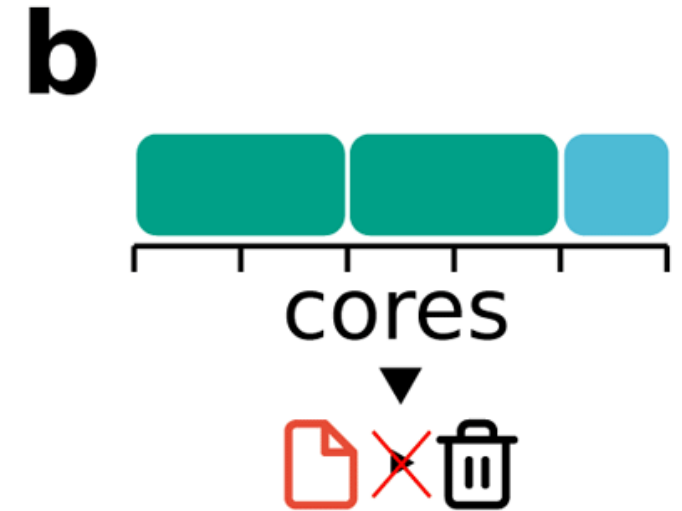
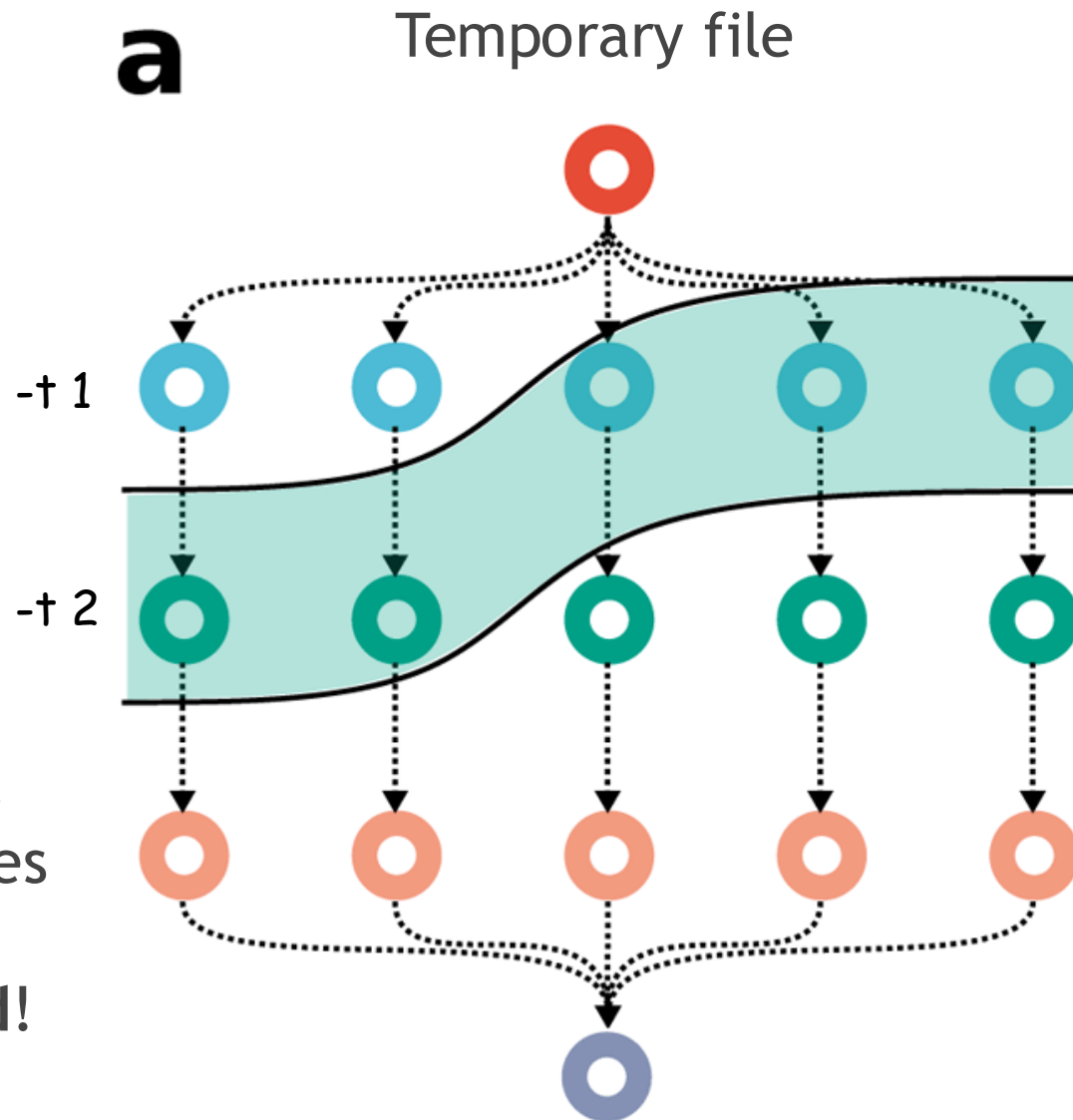


Scheduling problem

Jobs ready for scheduling

Blue rules: 1 cores
Green rules: 2 cores

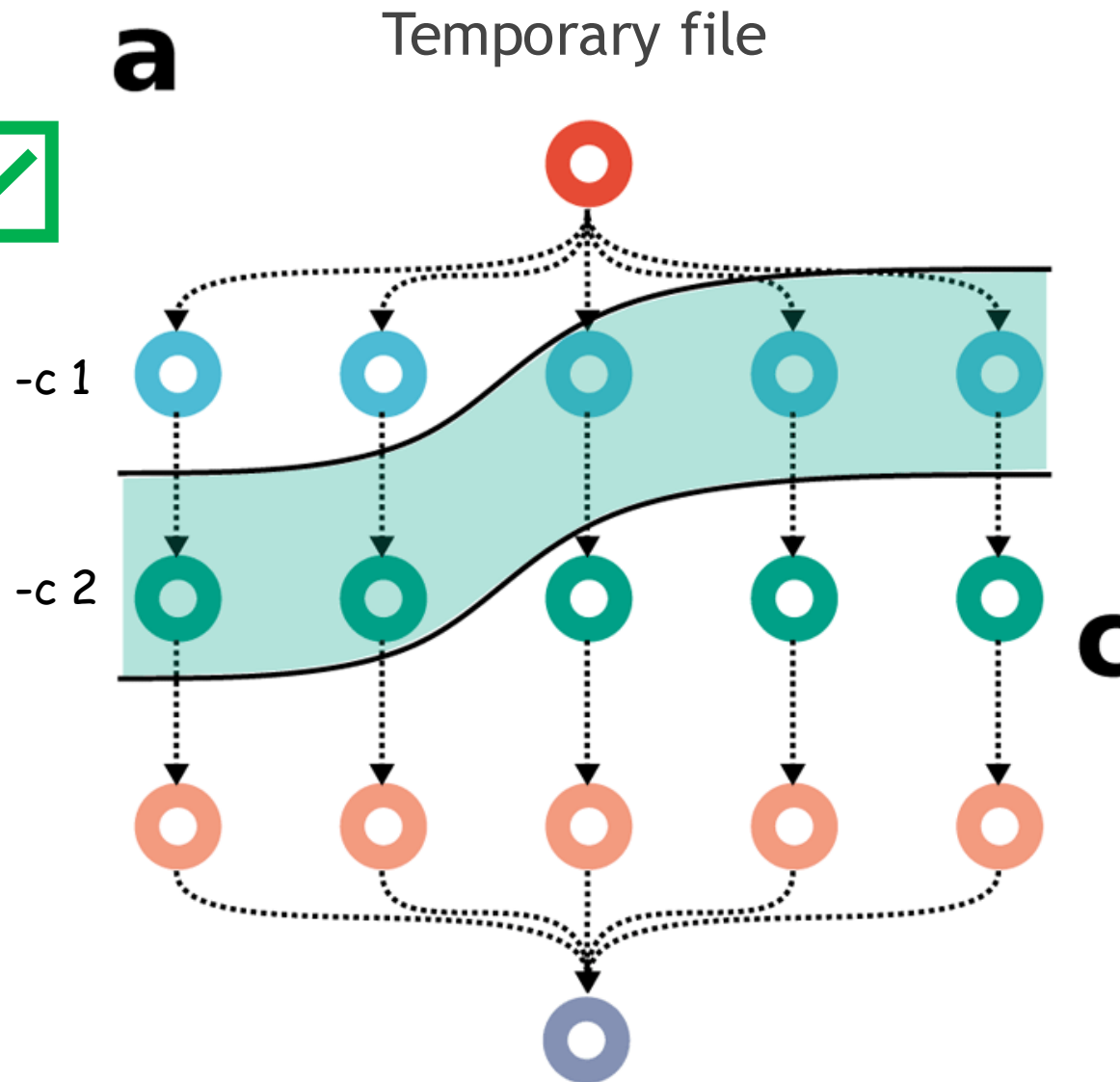
5 cores requested!



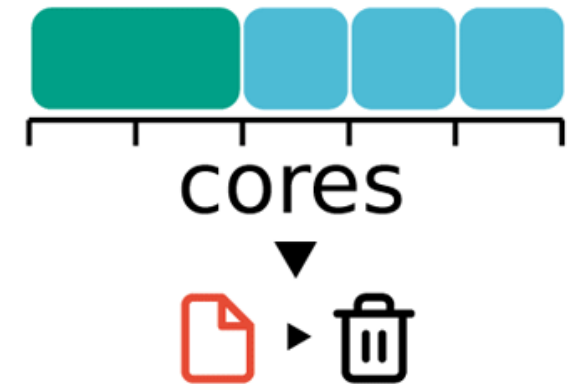
Scheduling
problem



Jobs ready for
scheduling



Snakemake will
determine the
dependencies for a
given set of targets
and schedule
accordingly (best
composition of rules
to create them!)



Resources - threads (defined locally, default 1)

Rule : ...

You ALWAYS need to define
--cores or --jobs globally

...

threads: 2

In Snakemake 'threads' is
equal to **cpus-per-task**

If no maximum for efficient parallelism exists for a tool:

rule NAME: ...

...

threads: workflow.cores * 0.75



Resources - memory

Defining locally

```
rule ...:
```

```
    resources: mem_mb=100
```

Defining globally (via command-line)

```
snakemake --resources mem_mb=100
```

The scheduler will ensure the given resources are not exceeded by **running jobs** (if benchmarking haven't done properly you might exceed the memory)

Resources are always meant to be specified as total per job, not by thread.



Resources - memory

Defining locally

```
rule ...:  
    resources: mem_mb=100
```

Dynamic resources

```
mem_mb=lambda wc, input:  
    max(2.5 * input.size_mb, 300)
```

```
mem_mb=threads * 150 ← If memory usage depends on # threads
```

Defining globally (via command-line)

```
snakemake --resources mem_mb=100
```

The scheduler will ensure the given resources are not exceeded by running jobs



What if you don't provide enough resources?

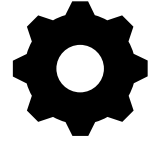
```
def get_mem_mb(wildcards, attempt):  
    return attempt * 100
```

```
rule:  
    input: ...  
    output: ...  
    resources:  
        mem_mb=get_mem_mb  
    shell: "..."
```

The first attempt will require 100 MB memory, the second attempt will require 200 MB memory and so on

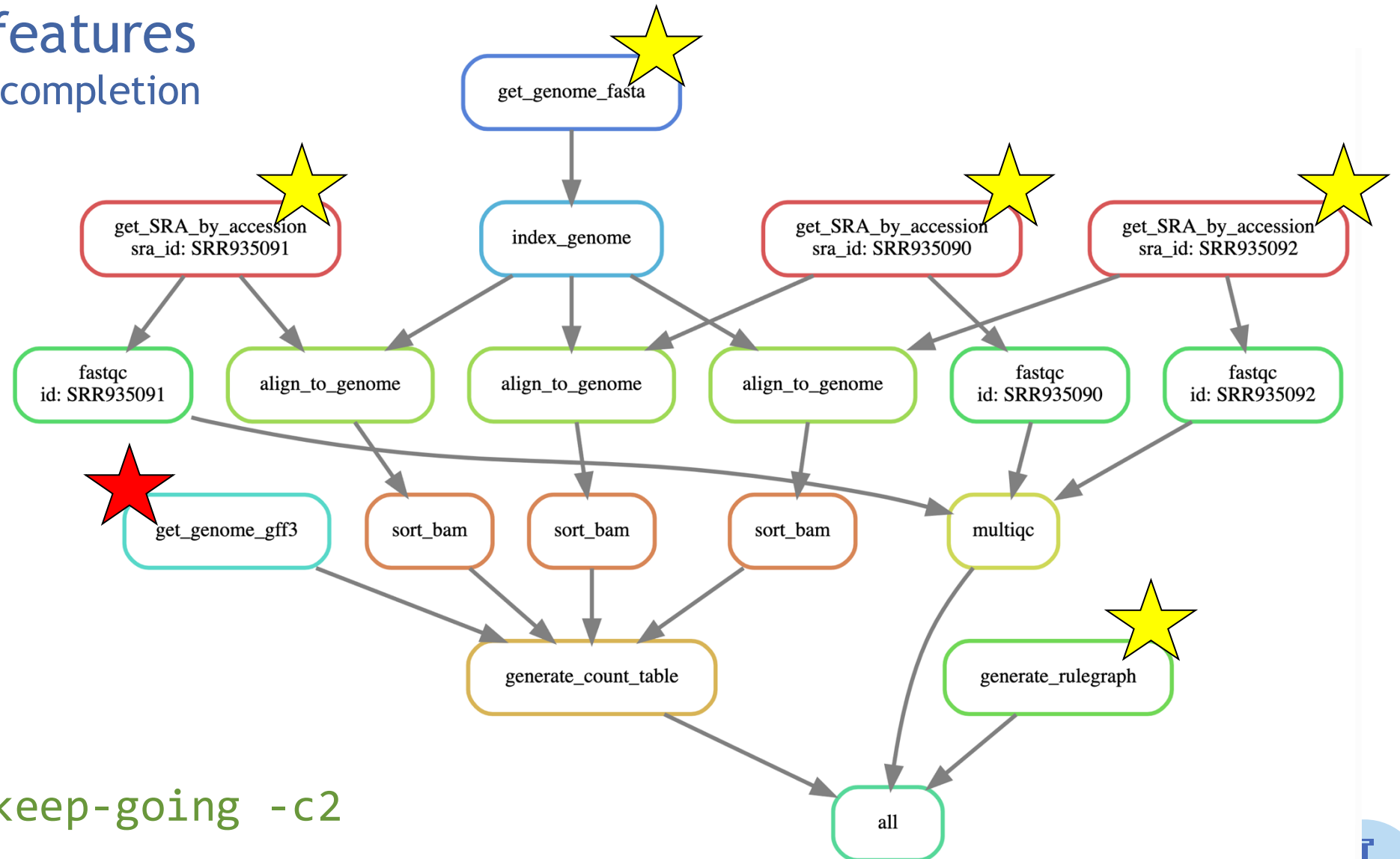


Additional cool features



Dev/testing features

Independent jobs completion



> `snakemake --keep-going -c2`



Other features: Output handling

Temporary files

Output file marked as temp is **deleted** after all rules that use it as an input are completed (e.g. intermediate files):

rule ...:

input: "path/to/infile"

output:

`temp("path/to/outfile")`

shell: "somecommand"

Protected files

Output file may require a huge amount of computation time. Hence one might want to **protect** it against accidental deletion or overwriting.

rule ...:

input: "path/to/infile"

output:

`protected("path/to/outfile")`

shell: "somecommand"

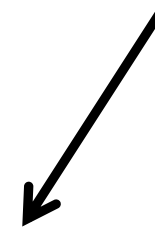


Params directive: non-file parameters

```
rule:
  input: ...
  params:
    prefix="somedir/{sample}"
  output: "somedir/{sample}.csv"
  shell:
    """
    somecommand -o {params.prefix}
    """
```

```
rule:
  input: ...
  params:
    cutoff: 1e-8
    minq=config["min_quality"]
  output: "somedir/{sample}.csv"
  shell:
    """
    CMD -p {params.cutoff} -q
    {params.minq}
    """
```

Key for testing,
benchmarking &
make it **configurable**



Config files

- **Centralized management:** of all parameters and settings in one place
- **Flexibility:** adaptable to varying needs
- **Improved readability:** enhancing clarity, easier to understand the structure of the pipeline
- **Collaboration:** collaborators can easily understand and modify parameters (even if they are not familiar with WfMS)
- Common formats: JSON or YAML

YAML: whitespaces (check formatting)



Configuration

Snakefile

A. Loaded to Snakefile as → `configfile: "path/to/config.yaml"`

```
rule all:
    input:
        expand("{sample}.{param}.output.pdf",
              sample=config["samples"],
              param=config["yourparam"])
```

Accessible via the variable `config`:

```
rule ...:
    input: config["ref"]
    params: config["pvalue"]
    output: "test.txt"
```

B. Use it from the command-line:

```
snakemake --configfile "/path/to/config.yml"
```



Config files

Snakefile

Loaded to Snakefile as →

```
configfile: "path/to/config.yaml"
```

config.yaml

samples:

- "sample1"
- "sample2"

yourparam: 1-e8

ref: "path/to/myfavgenome.fa"

```
rule all:
```

```
  input:
```

```
  expand("{sample}.{param}.output.pdf",  
        sample=config["samples"],  
        param=config["yourparam"])
```

```
rule ...:
```

```
  input: config["ref"]
```

```
  params: config["pvalue"]
```



Important: command-line overwrites infile statements

`snakemake --configfile` **overwrites** the value from the configfile statement in the Snakefile.

```
snakemake --configfile <new_config.yml>
```

Same applies to e.g.:

A. Change specific config values, not the entire file

```
snakemake --config yourparam=1.5 sample="newsample"
```

B. Specify resources

```
snakemake --resources mem_mb=8000
```



Constraining wildcards, what are they?

Use **regular expressions** to match output to input files and determine dependencies between jobs.

- **Avoid ambiguous rules** (e.g. two rules that can be applied to generate the same output file - maybe fixable by assigning different dirs?)
- Help to **guide** the regular expression-based matching so that wildcards are assigned to the right parts of the file name



Constraining wildcards, where can I define them?

Define them globally (==
apply for all rules) or
within the file pattern or
a rule

Snakefile

```
wildcard_constraints: id="\d+"
```

```
rule ...:
```

```
...
```

```
wildcard_constraints: id="\d+"
```

```
output: r"{id,\d+}.{group}.txt"
```



Exercise - constraining wildcards

rule all:
 input: "A.1.normal.txt" Target file
 ↙

rule ...:
 output: "{sample}.{group}.txt"
 shell: ...

Which one is the right assignment?

- dataset="A.1" and group="normal"
- dataset="A" and group="1.normal"



~ 2 min



Example

```
rule all:  
  input: "A.1.normal.txt"
```

```
rule ...:  
  input: "{sample}.{group}.txt"
```

Constraining the sample wildcard can solve the problem.

Assuming the answer was **b.**

```
{sample,[A-Z]+}.{group}
```



Be consistent with naming convention

You can define multiple wildcard_constraints:
pops="[^-]+",
samples="[^-]+"



Helper functions: `expand()`

Input files can be python lists:

rule aggregate:

```
input: [f"{dataset}/a.txt" for dataset in DATASETS]
```

```
output: "aggregated.txt"
```

```
shell: ...
```

↑
OR



```
expand("{dataset}/a.txt", dataset=DATASETS)
```



Semantic helper functions

```
rule ...:
    input:
        branch(
            lookup(dpath="switches/mark_duplicates", within=config),
            then="results/mapped/{sample}.bam",
            otherwise="results/mapped/{sample}.rmdup.bam")
    output: "results/{sample}.txt"
    script: "scripts/somestep.py"
```

Branch: choose different input files based on a given conditional

```
rule ...:
    input: "path/to/{dataset}.txt"
    output: "result/{dataset}.txt"
    params:
        some_threshold=lookup(
            dpath="some_tool/thresholds/{dataset}",
            within=config,
            default=0.1 )
    shell:
        "some-tool {input} > {output}"
```

Lookup: look up a value in a python mapping (e.g. a dict) or a pandas df or series



Modularization

- Reuse building blocks
- Help structure large workflows
- Extend previous analysis without modifying the pipeline

Use the 'include' directive to refer to another snakefile:

```
Snakefile
```

```
include: "path/to/other.smk"
```



My pipeline

myproject

- | -- Snakefile
- | -- config.yml
- | -- env.yaml
- | -- readme.txt
- | | rules
- | | -- pipe01.smk
- | | -- pipe02.smk
- | | -- pipe03.smk
- | | logs
- | | benchmarks

Snakefile

```
1  ### wrap up vg #####
2
3  import pandas as pd
4  import os
5  configfile: "config.yaml"
6
7  ##
8  wildcard_constraints:
9      vcf="[^~]+",
10     chrom="[~+\\.?$]+",
11     ext="[^.]+"
12
13  ## -----
14  include: 'rules/prep_vcf.smk'
15  include: 'rules/construct_graph_step1.smk'
16  include: 'rules/map_graph_step2.smk'
17
18  ## -----
19  ## global parameters
20  CHROMS = list(range(1, 23)) # + ['X']
21  # chromosome names
22  CHRS=[config['chr_prefix'] + ii for ii in str(CHROMS)]
23
24  # choose mapper
25  MAPPER=config['mapper']
26  if MAPPER == 'gaffe':
27      MAPPER = 'gaffe({k}{w}{N}'.format(config['mink'], config['minw'], config['covern'])
28  # choose vcf
29  VCFV=config['vcf_version']
30  VCFPATH=config['vcf'][VCFV]['minMAF']
31  # choose reference
32  REFV=config['ref_version']
33  REFPATH=config['ref'][REFV]
34
35  GRAPH=REFV+VCFV
36
37  ## -----
38  ## targets
39  # indexes used by the giraffe mapper and variant caller
40  rule construct_all_gaffe:
41      input:
42          expand('vg/graphs/{dat}/{genome}-{vcf}.{ext}', dat=VCFV, genome=REFV, vcf=VCFV, ext=['xg', 'trivial.snarls', 'dist']),
43          expand('vg/graphs/{dat}/{genome}-{vcf}.k{k}.w{w}.N{n}.min', dat=VCFV, genome=REFV, vcf=VCFV, k=config['mink'],
44              w=config['minw'], n=config['covern'])
45
46  # indexes used by the default mapper and variant caller
47  rule construct_all:
48      input:
49          expand('vg/graphs/{dat}/{genome}-{vcf}.{ext}', dat=VCFV, genome=REFV, vcf=VCFV, ext=['xg', 'snarls', 'gcsa'])
```



Using and combining workflows

Each task/step is **standalone** and can be used for other pipelines and it can be unit tested.

Example. Assuming we need the same preprocessing but different downstream analyses:

```
module some_workflow:  
    snakefile: "https://github.com/project/Snakefile"  
    config: config["some"]  
  
use rule * from some_workflow  
  
use rule simul_data from some_workflow with:  
    params: some_threshold = 1e-7
```



Complete example

configfile: "config/config.yaml"

```
rule all:
    input:
        "results/plots/vafs.svg"
```

```
module dna_seq:
    snakefile: "https://github.com/snakemake-workflows/dna-seq-gatk-variant-
calling/raw/v2.0.1/Snakefile"
    config:
        config["dna-seq"]
```

```
use rule * from dna_seq
```

easily extend the workflow

```
rule plot_vafs:
    input:
        "filtered/all.vcf.gz"
    output:
        "results/plots/vafs.svg"
    notebook:
        "notebooks/plot-vafs.py.ipynb"
```



Command line:
snakemake -c1



Logging

Use the programs' log file if possible: `--log {log}"`

If the software does have an explicit log parameter. Otherwise,

rule ...:

input: "input.txt"

output: "output.txt"

log:

stdout="logs/foo.stdout",

stderr="logs/foo.stderr"

shell:

"somecommand {input} {output} > "

"{log.stdout} 2> {log.stderr}"

OR `&> {log}`



Benchmarking

TIP: measure the wall clock time and memory usage (in MiB) of a job

```
rule bwa_map:
    input:
        "data/genome.fa",
        lambda wildcards: config["samples"][wildcards.sample]
    output: temp("mapped_reads/{sample}.bam")
    params: rg="@RG\tID:{sample}\tSM:{sample}"
    log: "logs/bwa_mem/{sample}.log"
    benchmark: "benchmarks/{sample}.bwa.benchmark.txt"
    threads: 8
    shell:
        "(bwa mem -R '{params.rg}' -t {threads} {input} |"
        "samtools view -Sb - > {output}) 2> {log}"
```



Output benchmarking (JSON)

- *s*: Wall clock time (in seconds),
- *h:m:s*: Wall clock time (in *hour:minutes:seconds*),
- *max_rss*: Max RSS memory usage (in megabytes),
- *max_vms*: Max VMS memory usage (in megabytes),
- *max_uss*: Max USS memory usage (in megabytes),
- *max_pss*: Max PSS memory usage (in megabytes),
- *io_in*: I/O read (in bytes),
- *io_out*: I/O written (in bytes),
- *mean_load*: CPU load = CPU time (*cpu_usage*) divided by wall clock time (s),
- *cpu_time*: CPU time user+system (seconds),



--benchmark-extended: somecommand

- *jobid*: Internal job ID,
- *rule_name*: Rule name,
- *wildcards*: Job wildcards,
- *params*: Job parameters,
- *threads*: Number of threads requested for this job,
- *cpu_usage*: Total CPU load,
- *resources*: Resources requested for this job,
- *input_size_mb*: Size of input files (MiB),



Summary

- Resource allocation
- Configuration
- Wildcard constraints
- Helper functions
 - `expand()`
 - New cool ones: lookup, branching, etc.
- Modularization
- Logging and benchmark (Bonus exercise)



 ~ 1.20h



`hds-sandbox.github.io/HPC-lab`

workshop > HPC Launch > Day 2

Snakemake advanced

Exercise I + II

Now is YOUR time!

Snakemake exercises

- Build a pipeline from scratch
- Extend your pipeline: make it more modular by adding parameters to your rules and making use of config files

If you have time, try the bonus exercise



Problems/Issues/Comments

Checkpoint

Special rules to **create dynamic** workflows. The structure of the pipeline recomputes after the checkpoint successfully executes.

Two main uses:

1. Used at critical junctures where the **downstream pipeline diverges for different outcomes of some step**, like reading dynamically generated files from a previous step.
2. For cases where the **output files are unknown before execution**



A. Data-dependent conditional execution (output files known, execution dependent on content)

```
checkpoint somestep:  
    input:  
        "samples/{sample}.txt"  
    output:  
        "somestep/{sample}.txt"  
    shell: "somecommand {input} > {output}"
```

The checkpoint that shall trigger
re-evaluation of the DAG

```
rule intermediate:  
    input: "somestep/{sample}.txt"  
    output: "post/{sample}.txt"  
    shell: "touch {output}"
```

```
rule alt_intermediate:  
    input: "somestep/{sample}.txt"  
    output: "alt/{sample}.txt"  
    shell: "touch {output}"
```

Input function for the rule aggregate.
Decision based on content of output file

```
def aggregate_input(wildcards):  
    with checkpoints.somestep.get(sample=wildcards.sample).output[0].open() as f:  
        if f.read().strip() == "a":  
            return "post/{sample}.txt"  
        else:  
            return "alt/{sample}.txt"
```

```
rule aggregate:  
    input: aggregate_input  
    output: "aggregated/{sample}.txt"  
    shell: "touch {output}"
```

```
rule all:  
    input: aggregated/mysample.txt
```



B. Data-dependent conditional execution (output files unknown before execution)

```
rule all:
    input: "aggregated.txt"    Target rule to define the desired final output
```

```
checkpoint stepX:
    output: directory("my_directory/")
    shell:
        """ mkdir my_directory/
        cd my_directory
        for i in 1 2 3; do touch $i.txt; done"""
```

The checkpoint that shall trigger re-evaluation of the DAG (# of files created in a defined directory)

```
def aggregate_input(wildcards):
    checkpointOut = checkpoints.stepX.get(**wildcards).output[0]
    return expand("my_directory/{i}.txt",
                 i=glob_wildcards(os.path.join(checkpointOut, "{i}.txt")).i)
```

Input function for rule aggregate, return paths to all files produced by the checkpoint 'stepX'

```
rule aggregate:
    input: aggregate_input
    output: "aggregated.txt"
    shell: "cat {input} > {output}"
```

As the wildcard *i* is only evaluated after completion of the checkpoint, we need to use `directory` to declare its output (instead of using wildcard patterns as output)



Checkpoints: data-dependent conditional execution

<https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#data-dependent-conditional-execution>

Examples:

- Split files by size or chromosome count (varies by organism) to improve processing speed
- Only apply downstream analyses to samples that pass a filter



Cluster or cloud execution

Executor plugin allows to use `slurm` in a seamless and straightforward way



executor plugins

[azure-batch](#)

[cluster-generic](#)

[cluster-sync](#)

[deeporigin](#)

[drmaa](#)

[flux](#)

[googlebatch](#)

[kubernetes](#)

[lsf](#)

[lsf-jobstep](#)

[slurm](#)

[slurm-gustave-roussy](#)

[slurm-jobstep](#)

[tes](#)

storage plugins

[azure](#)

[deeporigin](#)

Snakemake plugin catalog

This catalog collects information about and documentation of all Snakemake plugins published on [PyPI](#). Note that plugin support is available in [Snakemake 8.0](#) and later.

Contributing

- Improving PRs or issues with the plugin catalog (only the catalog not the plugins themselves) can be made [here](#).
- Improving PRs or issues with the plugins themselves can be made at the respective plugin repository (see the docs of each plugin here).
- Resources for contributing to new plugin development include the [snakemake-interface-executor-plugins](#) and [snakemake-interface-storage-plugins](#) (along with the [poetry-snakemake-plugin](#) for automatically generating skeleton code) and referring to existing plugin repositories.
- New plugins will be automatically found on PyPI (which implies that they have to be released to PyPI first) and added to this catalog. If you expect your plugin to appear here but do not see it, please check the [nightly github action](#) for errors with your plugin (will occur in the logs).

Option 1

Use executors plugin: [Snakemake plugin catalog](https://snakemake.github.io/snakemake-plugin-catalog/plugins/executor/slurm.html)

```
snakemake --executor slurm --default-resources --jobs N ...
```

<https://snakemake.github.io/snakemake-plugin-catalog/plugins/executor/slurm.html>

Example 2: cluster

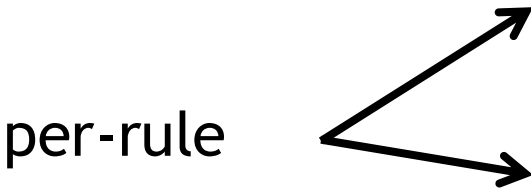
```
snakemake --jobs 200 --cluster \  
"sbatch -t {params.runtime} --mem={params.memory} --cpus-per-task={threads}"
```

Snakefile

```
rule samtools_sort:  
    input:  
        "results/{sample}/{sample}.sam"  
    output:  
        protected("results/{sample}/sort_{sample}.bam")  
    log:  
        "results/logs/samtools_sort/{sample}.log"  
    params:  
        runtime="01:00:00",  
        memory="16G"  
    threads: 16  
    shell:  
        ""  
        samtools sort -@ {threads} -o {output} {input} 2> {log}  
        ""
```

Example 3 - Using profile files (config.vX+.yaml)

```
# Tell snakemake to use Slurm & Maximum number of parallel jobs
executor: slurm
jobs: 10
# Set number of threads for rule(s)
set-threads:
  preprocess: 1
  train_and_plot: 2
# Set other resources (in this case memory and time) for rule(s). Formats are
described in:
https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#standard-resources
set-resources:
  preprocess:
    mem: 500MB
    runtime: 30m
  train_and_plot:
    mem: 1GB
    runtime: 1h
```



The diagram shows the text 'per-rule' on the left. Two arrows originate from it: one points to the 'preprocess' section of the 'set-resources' block, and the other points to the 'train_and_plot' section of the same block.

```
snakemake --profile profiles/slurm/ --software-deployment-method apptainer
```

<https://github.com/Snakemake-Profiles/slurm>

<https://github.com/jdblischak/smk-simple-slurm>

Additional features

Jupyter notebook integration (edit/run) `notebook: "mynotebook.ipynb"`

Directories as outputs `output: directory("path/to/dir")`

Ignoring timestamps `input: ancient("path/to/input")`

Ensuring output file `ensure("test.txt", sha256=my_checksum)`

Conde tracking (`--list-code-changes`)

Parameter space exploration

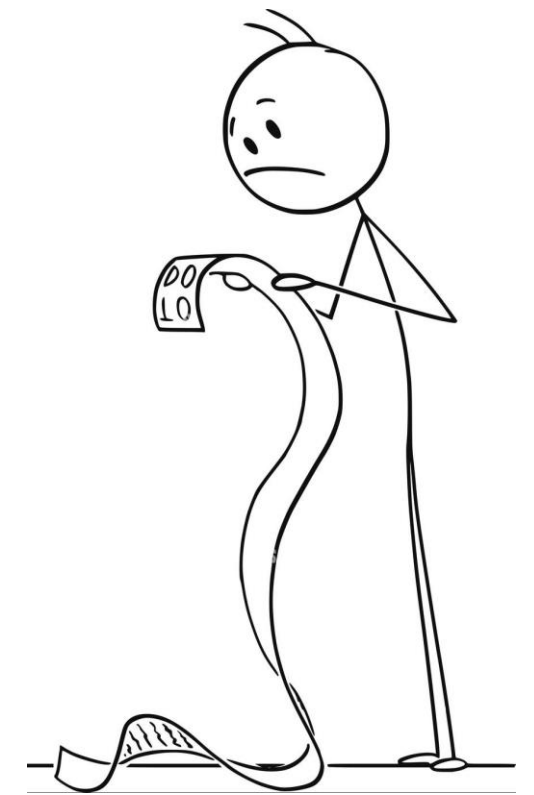
Properties like non-emptiness or checksum compliance output: Retries for fallible rules (e.g. download online resources) `retries: 3`

Local rules, not submitted to cluster and executes on the host (e.g. download `--localrules: dw`)

Group jobs

Defining groups of executions (executed together in same computing node)

Service rules: a resource that shall be kept available until all consuming jobs are finished



My setup: useful arguments when running a long pipeline

```
flags="--cores all "
```

```
flags+="--nolock "
```

```
flags+="--keep-going "
```

```
flags+="--printshellcmds "
```

```
flags+="--show-failed-logs "
```

```
flags+="--keep-incomplete "
```

```
flags+="--rerun-incomplete "
```

```
flags+="--reason "
```

```
flags+="--restart-times 1 "
```

```
snakemake ${flags}
```



Part II - Computations management

Workflow
management
systems

Integration with
environment
managers

A. Set up an isolated env for the whole workflow

Snakefile



envs/preprocessing.yml

```
conda: "envs/preprocessing.yml"
```

```
rule mytask:  
  input:  
    "data/{sample}.csv"  
  output:  
    "results/{sample}.csv"  
  script:  
    "somecommand {input}"
```

```
name: preprocessing  
channels:  
  - conda-forge  
dependencies  
  - coreutils
```



```
> snakemake --use-conda
```



A. Set up an isolated env for the whole workflow

conda: "envs/preprocessing.yml"

```
rule mytask:
    input:
        "data/{sample}.csv"
    output:
        "results/{sample}.csv"
    script:
        "somecommand {input}"
```

--software-deployment-method

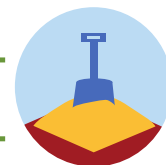
B. Specify conda envs per rule

Utilize conflicting software versions
(e.g. combine python 2 and 3)

```
rule mytask:
    input:
        "sorted_reads/{sample}.bam"
    output:
        "sorted_reads /{sample}.bam.bai"
    conda: "envs/samtools.yml"
    script:
        "samtools index {input}"
```



```
> snakemake --sdm conda -c1
> snakemake --use-conda -c1
```



If your HPC clusters do not allow using conda...

Best to specify env modules in a configuration file

```
rule ...:
    input:
    output:
    envmodules: "bio/VinaLC"
```



```
> snakemake --sdm env-modules
> snakemake --use-envmodules
```



Advantages integrating conda

Documenting software versions

No need to installing any prerequisites apart from snakemake and miniconda to re-execute in a different server (without admin rights)

Combine different software managers

```
--software-deployment-method <env>
```

Work with multiple environment types (e.g. conda, singularity, docker, environment modules)



Containers (HPC)

```
rule myothertask:
  input:
    "table.txt"
  output:
    "plots/myplot.pdf"
  singularity:
    "docker://joseespinoza/docker-r-ggplot2"
  shell:
    "some-rtool {input} > {output}"
```

When executing Snakemake with



```
snakemake --use-singularity (or --use-apptainer)
```



Containerization



```
> snakemake --containerize > Dockerfile
```

Snakefile

```
containerized: "docker://username/myworkflow:1.0.0"
```

rule NAME:

```
    input: "table.txt"
    output: "plots/myplot.pdf"
    conda: "envs/ggplot.yaml"
    script: "scripts/plot-stuff.R"
```

Each rule can have its
own env setup!



```
> snakemake --sdm conda apptainer
```



Advantages

Integrates with package manager conda and container engine singularity such that defining software becomes part of the workflow itself

Scales without modification from single core workstations & multi-core servers to batch systems (e.g. slurm)



Best practices

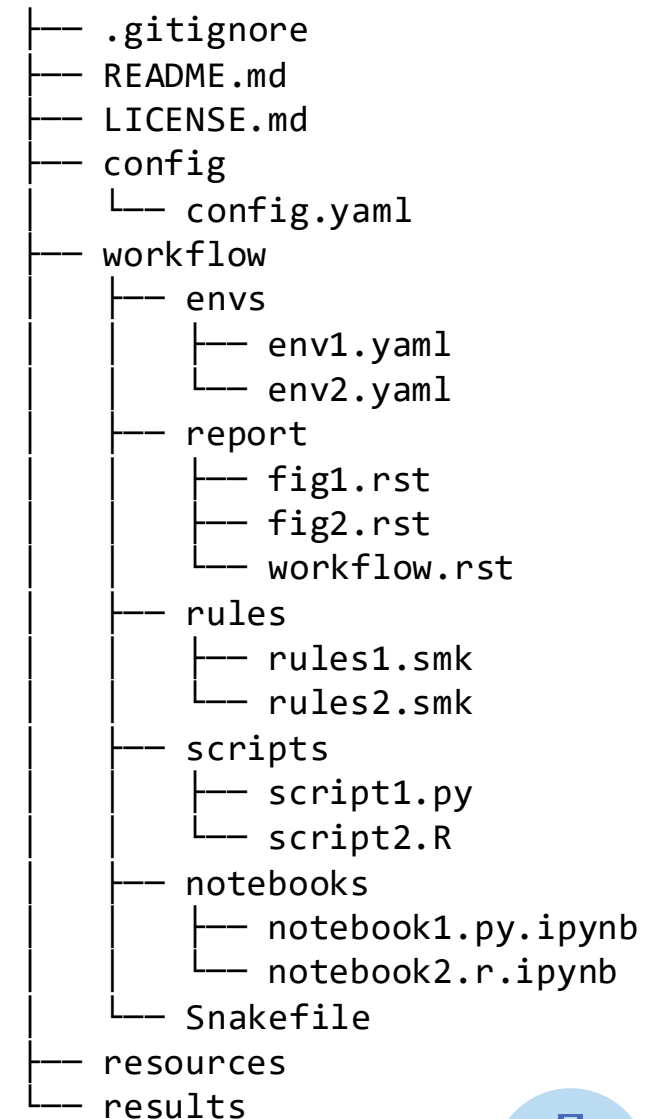
- simple, short, descriptive names
- reasonable directory structure
- use shell for short commands, use script and notebook for complex stuff
- put non-rule code (helper functions, config parsing) into separate module, e.g. *workflow/rules/common.smk*
- conda/singularity/envmodules - envs for every rule
- prefer small envs over large ones (maintainability, performance)
- define log (and benchmarking) files
- think about which rules' outputs shall be cached across workflows (modules)





Rules to get your workflow published

1. `snakemake --lint` -> quality control before publishing pipeline
2. Contained in a public GitHub repository (set up Github actions for continuously testing the workflow in each commit)
3. Have a README
4. Repository should contain (Snakefile + rules with *.smk)
5. Contain a YAML file configuring the usage instructions defined by workflow
6. Ensure portability: all rules with versioned conda or container
7. Small enough to be cloned into a GitHub actions job
8. Enable configurability
9. Use code formatters (Snakefmt)
10. Avoid duplication efforts (snakemake wrappers)



Snakemake workflow catalog



Snakemake workflow catalog

A comprehensive catalog of [standards compliant](#), public, [Snakemake](#) workflows

Standardized usage **293**

All workflows **3125**

Workflow	Description	Topics	QC	Stars	Watchers
Usage snakemake-workflows/rna-seq-star-deseq2	RNA-seq workflow using STAR and DESeq2	snakemake , sciworkflows , reproducibility , gene-expression-analysis , deseq2	<div>license MIT</div> <div>last commit june</div> <div>linting passed</div> <div>formatting passed</div>	321	11
Usage snakemake-workflows/dna-seq-gatk-variant-calling	This Snakemake pipeline implements the GATK best-practices workflow	reproducibility , snakemake , sciworkflows , genomic-variant-calling , gatk	<div>license MIT</div> <div>last commit may 2021</div> <div>linting passed</div> <div>formatting failed</div>	207	9
Usage franciscozorilla/metaGEM	:gem: An easy-to-use workflow for generating context specific genome-scale metabolic models and predicting metabolic interactions within microbial communities directly from metagenomic data	metagenomics , computational-biology , metabolic-models , gut-microbiome , snakemake , metagenome-assembled-genomes , mags , metabolism , bioinformatics , flux-balance-analysis , genome-scale-metabolic-model , metabolic-	<div>license MIT</div> <div>last commit july</div> <div>linting failed</div> <div>formatting failed</div>	189	5













Sources

[Home](#) » [Browse](#) » [Sustainable data analysis with Snakemake](#)

METHOD ARTICLE



REVISED Sustainable data analysis with Snakemake [version 2; peer review: 2 approved]

Felix Mölder ^{1,2}, Kim Philipp Jablonski ^{3,4}, Brice Letcher ⁵, Michael B. Hall ⁵,
Christopher H. Tomkins-Tinch ^{6,7}, Vanessa Sochat ⁸, Jan Forster^{1,9}, Soohyun Lee ¹⁰,
Sven O. Twardziok¹¹, Alexander Kanitz ^{12,13}, Andreas Wilm¹⁴, Manuel Holtgrewe^{11,15}, Sven
Rahmann¹⁶, Sven Nahnsen¹⁷,  [Johannes Köster](#) ^{1,18}

 [Author details](#)

<https://doi.org/10.12688/f1000research.29032.1>

Documentation: <https://snakemake.readthedocs.io>

Workflow catalog: <https://snakemake.github.io/snakemake-workflow-catalog>

Conda formatting: <https://github.com/snakemake/snakefmt>

Template: <https://github.com/snakemake-workflows/snakemake-workflow-template/tree/main>

Editors: vscode, pycharm, vim, theia, emacs





Now is YOUR time!

Complete Day 2 - snakemake envs integration or revisit any bonus exercises you've skipped.

If you need help setting up your **own pipeline** or want to work on it now, there's time for that too.



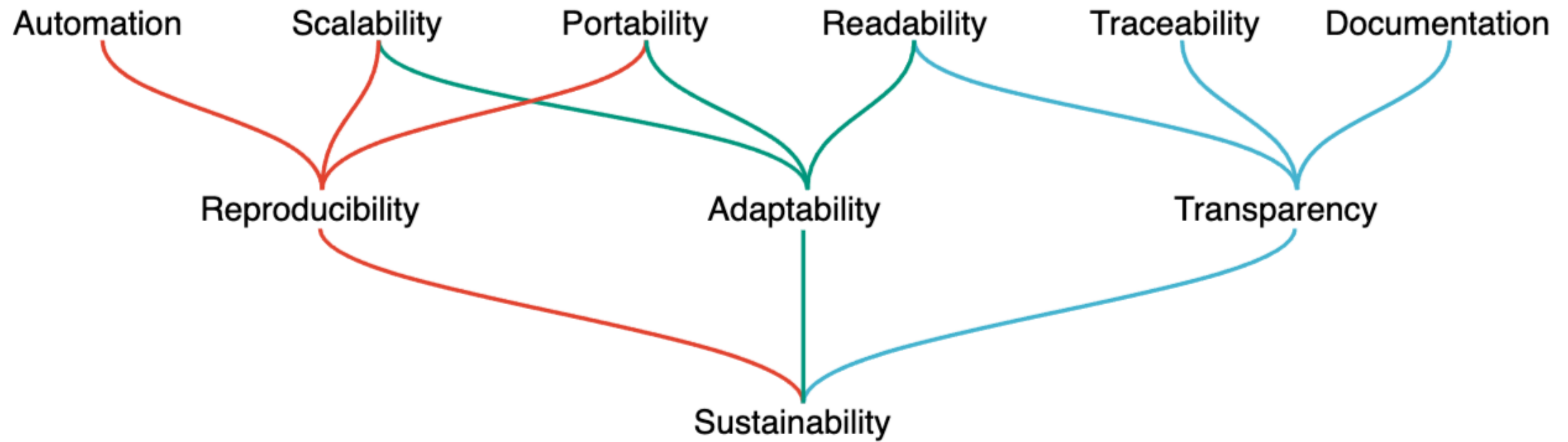
`hds-sandbox.github.io/HPC-lab`

workshop > HPC Launch > Day 2

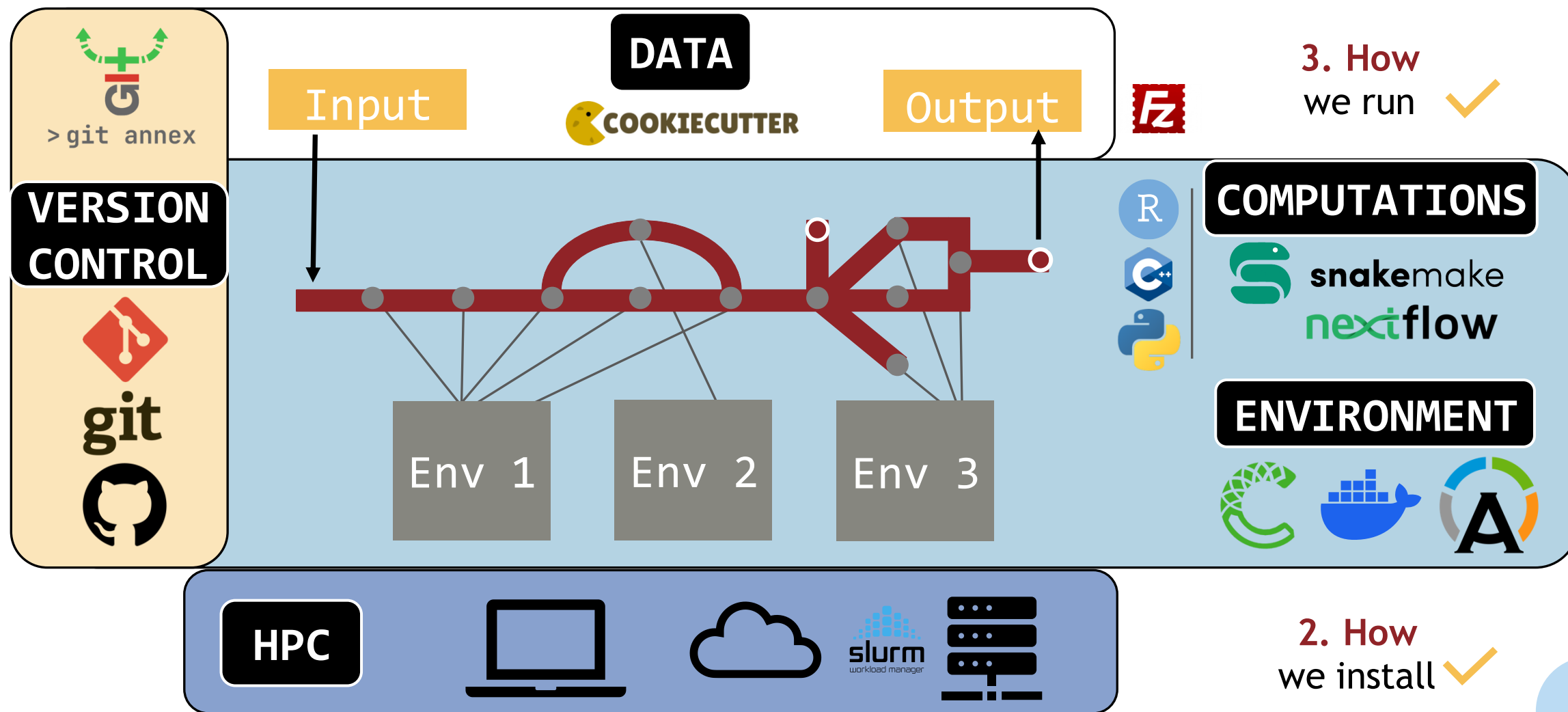
Snakemake - envs



Problems/Issues/Comments



HPC-Pipes is complete

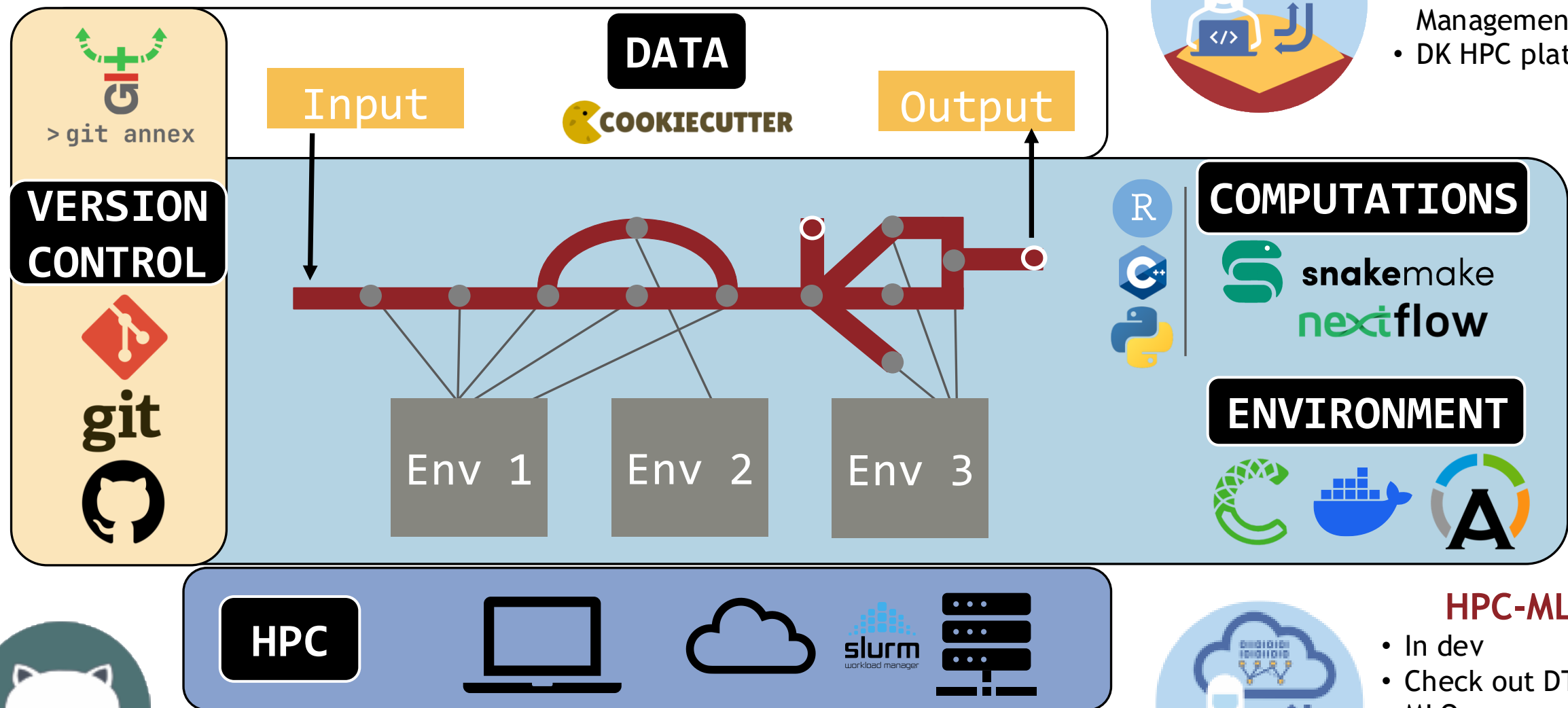


1. Where we set up ✓

2. How we install ✓



HPC-Pipes is complete - next?



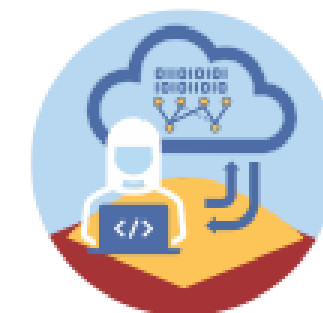
HPC-Launch

- Omics Data Management
- DK HPC platforms



Git & Github

- SUND Data Lab



HPC-ML

- In dev
- Check out DTU's MLOps course in the meantime

https://skaftenicki.github.io/dtu_mlops/

