

HPC-Pipes Workflow managers

from the
Health Data Science
Sandbox



Alba Refoyo Martinez, PhD

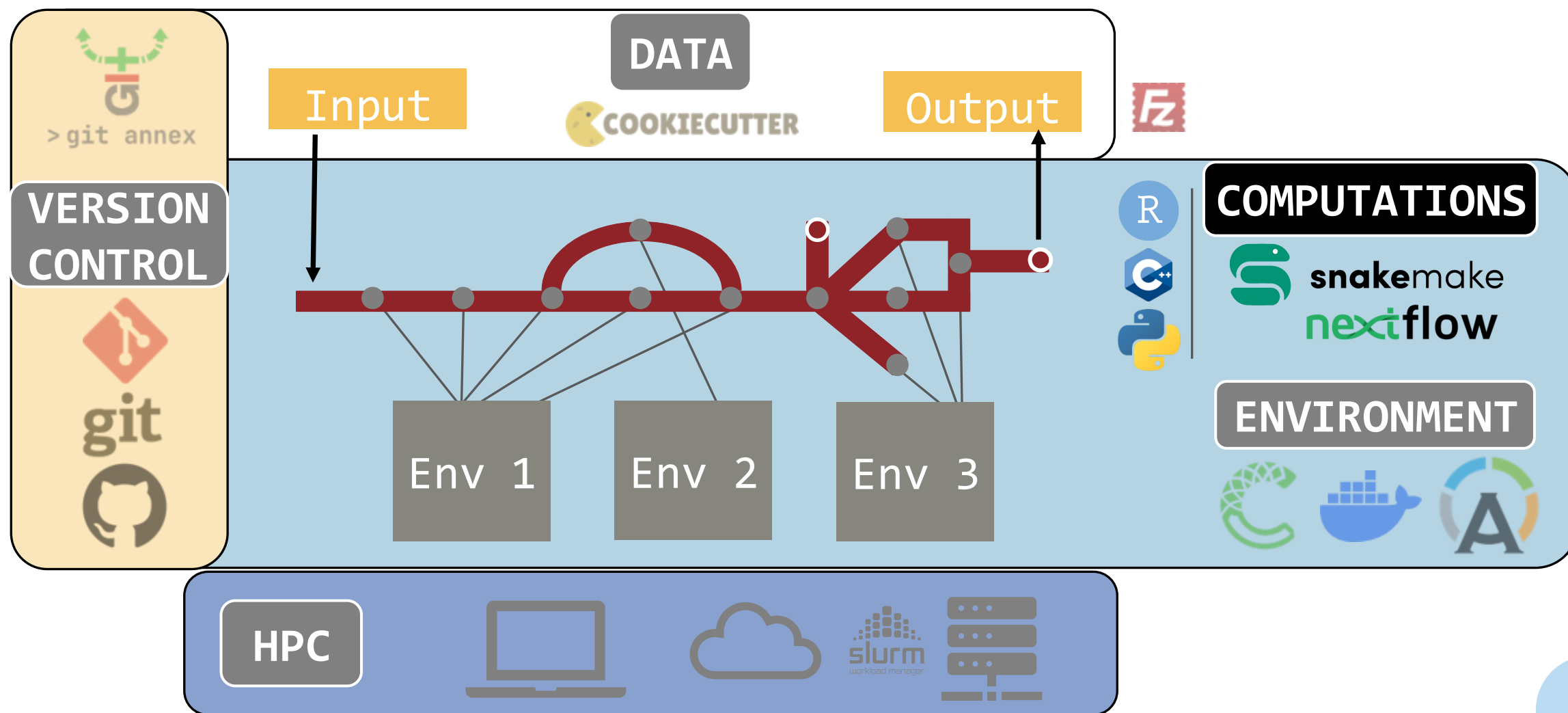
Data scientist

Center for Health Data Science (HeaDS)

UNIVERSITY OF
COPENHAGEN



Components of a bioinformatics pipeline



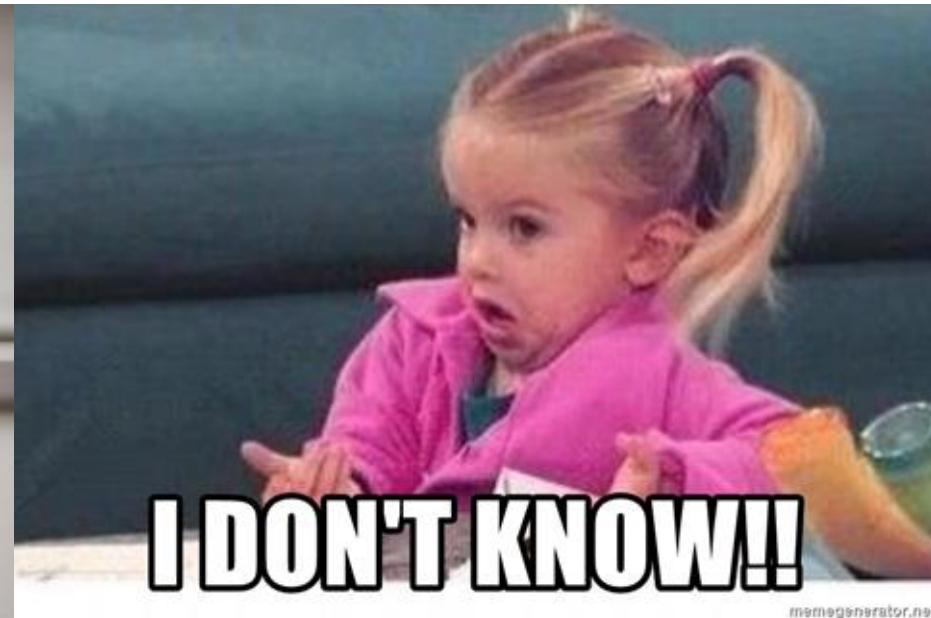
Part II - Computations management

Workflow
management
systems

Integration with
environment
managers

Understanding workflow management systems

What are they and why are they relevant to me?

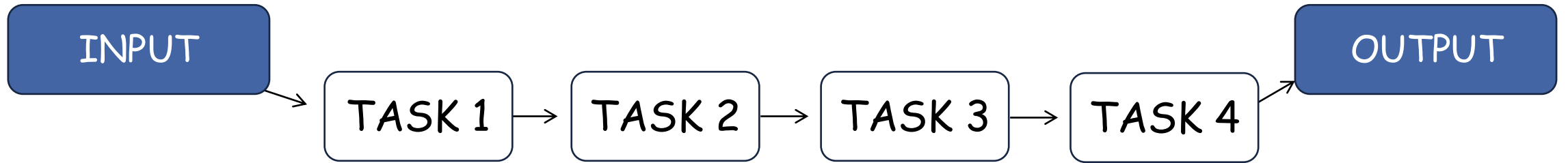


(YET!)



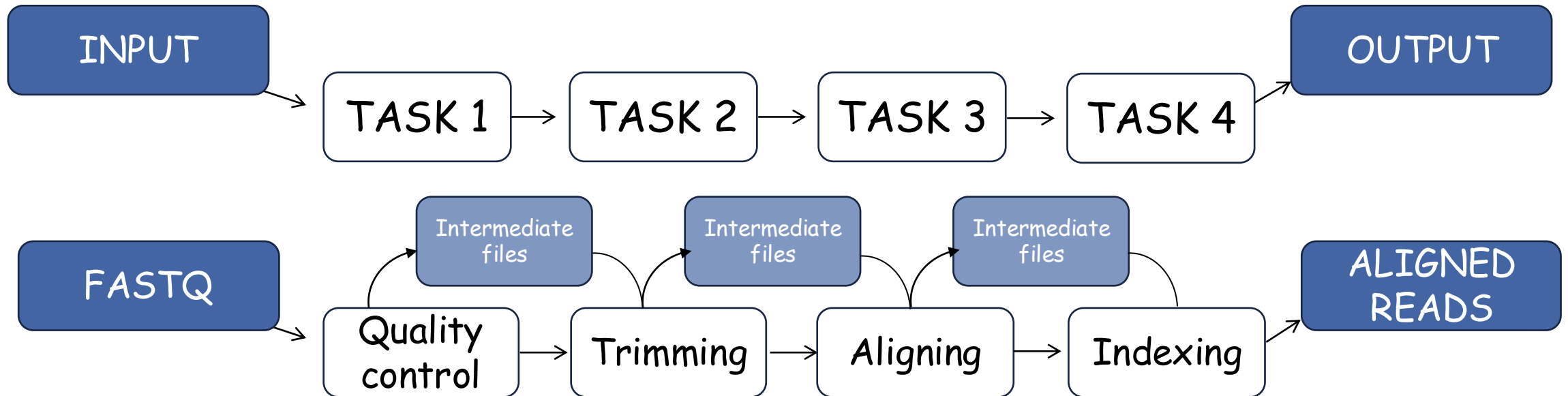
What is a bioinformatics workflow?

A series of programmatic steps to transform raw data into processed results, figures, and insights.



What is a bioinformatics workflow?

A series of programmatic steps to transform raw data into processed results, figures, and insights.

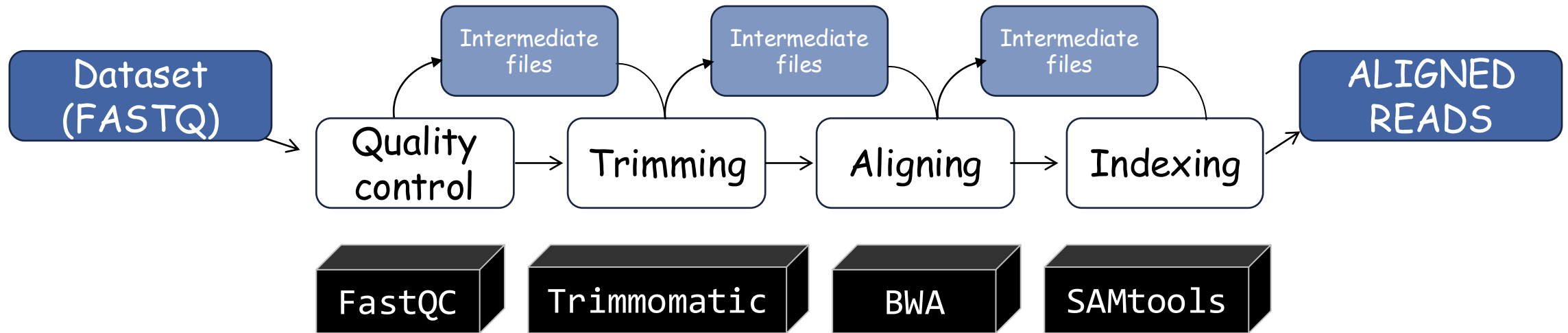


Several software are needed to generate the desired output



What is a bioinformatics workflow?

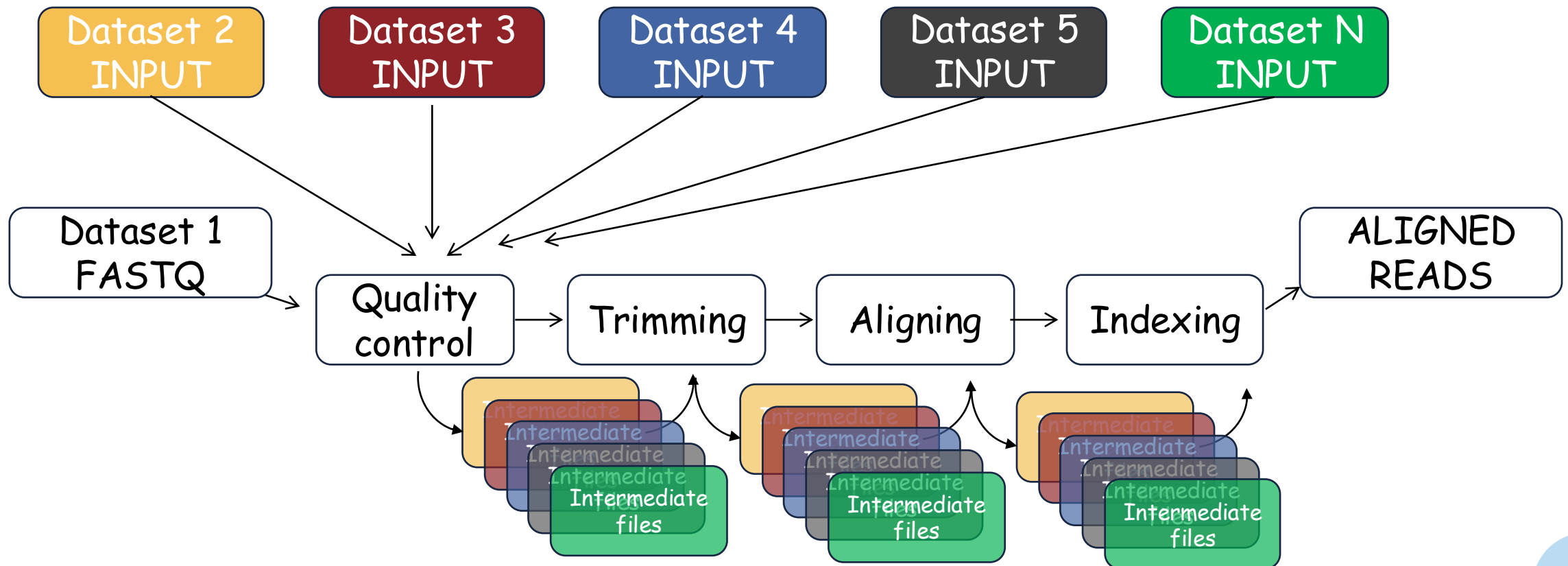
Each step involving **different tools, parameters, reference databases, and specific requirements.**



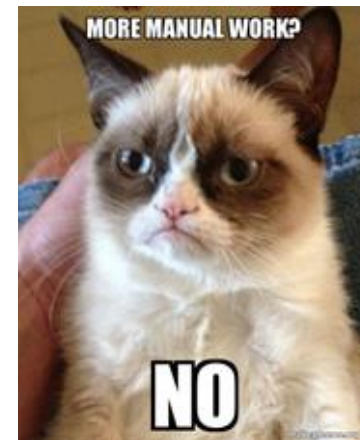
Let me do this by hand... 



Now apply the same analysis to new data...



Wait, why don't we test different software mapping tools?

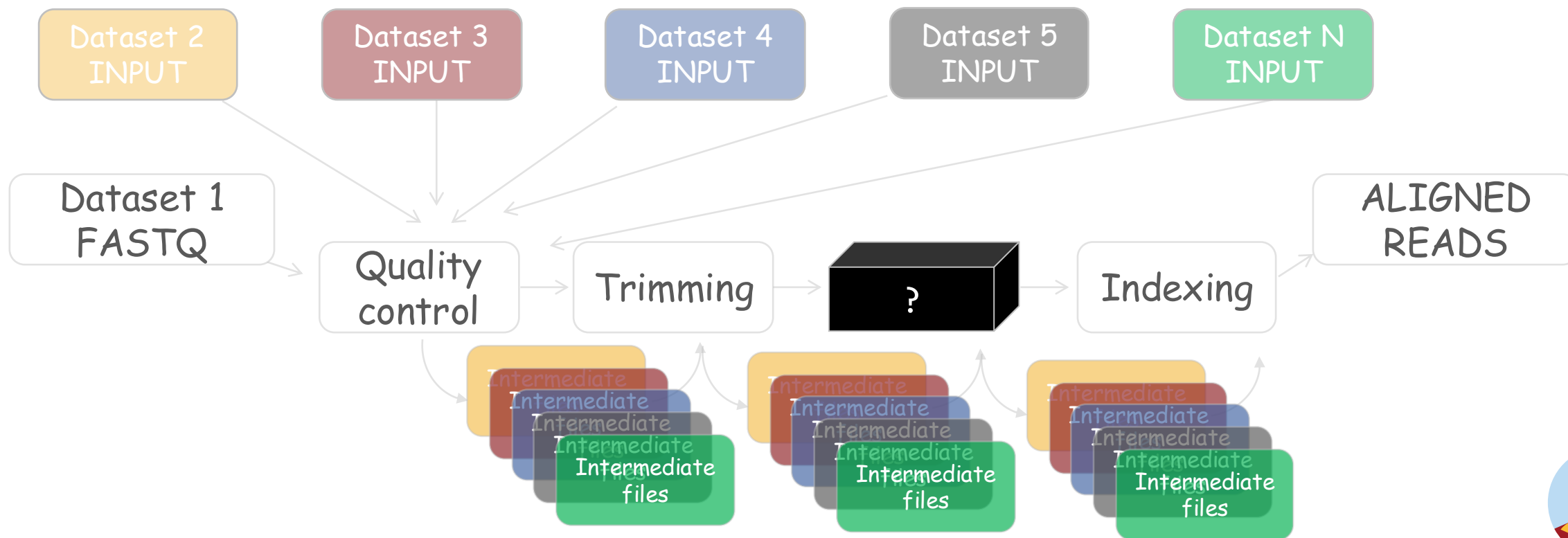


BWA

BOWTIE2

VG

MOSAİK



Bioinformatics workflows are complex... and reproducibility can be very challenging

SOFTWARE

- **Multiple software** required, and sometimes even more **parameters** to tweak
- Small changes in the parameters software can cause a large difference in the results
- Differences in **program resources** needs at each step (computational power, data inputs, software dependencies, etc.)



Bioinformatics workflows are complex... and reproducibility can be very challenging

DATA

- Many files are being generated (also intermediate files) and the size of the data files can be large
- Differences in **data type, shape and scale**



Authors of scientific computations must prepare complete and precise instructions to ensure replicability & identical results

Data Orchestration

The process of gathering, cleansing, organizing, and analysing data to make informed decisions

Workflow managers

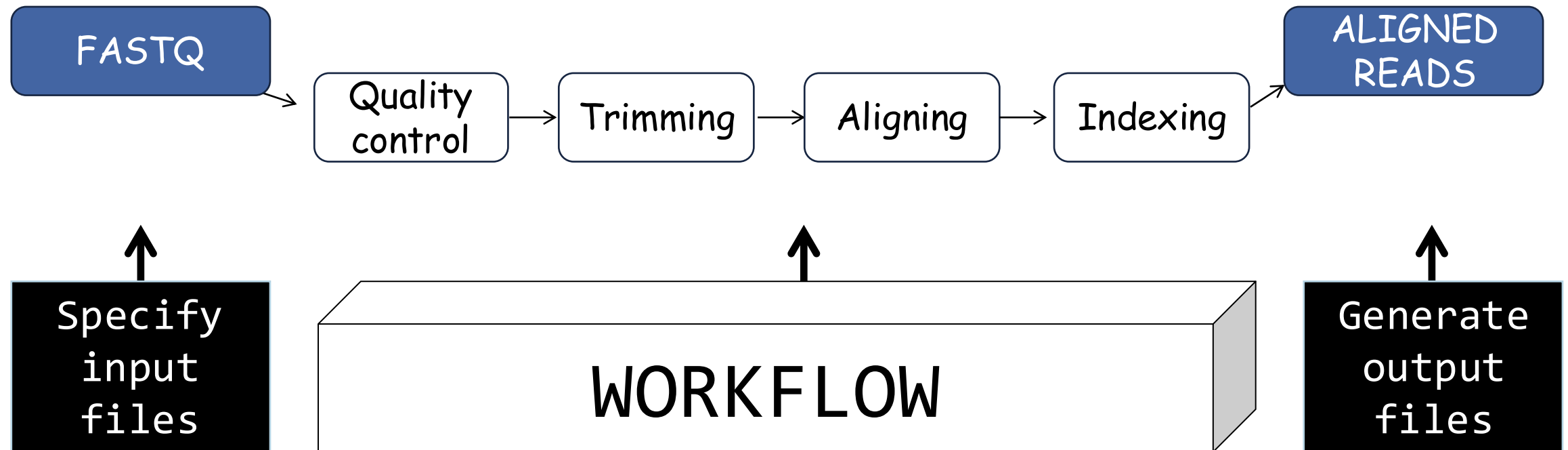
the tools that orchestrate the processes, dependencies, deployment and tracking of large bioinformatics pipelines



Workflow Management Systems

Execution without manual intervention

Choose the pipeline that
does the work for you! (or
define them yourself)



Understanding workflow management systems (WfMS)

- WfMS are software tools designed to **automate and optimize complex data analysis pipeline.**
- WfMS allow users to **define, run, and track tasks and dependencies** in a modular, scalable way, simplifying the **management and reproducibility** of analyses.

Why do we want them?

- Using WfMS helps **reduce errors and inconsistencies** in analyses, while enhancing the efficiency and reproducibility of research.



Summarising, they are key in **bioinformatics**...

- WfMS help **automate and streamline** analyses, simplifying the management and reproducibility of results.
- Workflows can be designed to be **modular, scalable, and reproducible**, using tools like **containerization** and **version control** to maintain consistency and transparency in the analysis pipeline.

Reproducibility

Portability

Efficiency
(parallelisation)

Scalability

Consistency

Collaboration

Automation

When are they particularly useful?

- **Re-run** the same analysis over and over, with different input parameters or datasets
- Benchmarking tools
- Re-run the work **partially**, easy recover from intermediate failures
- Combine together heterogeneous tooling in the same analysis (e.g. python, R, bash, Julia, C++, etc.)
- Dependency mgmt of incompatible dependencies between tasks



Exercise 1: why avoid using a shell script?

*“Bash scripts have been widely used for automation in the past and can handle many tasks effectively. Typically, running a bash script requires just one command, which executes all the steps in the script. However, a significant drawback is that it **re-runs all steps every time**. This can be problematic in certain situations.”*

Think about several scenarios in which that could be problematic.



Solution 1.

- **Changing input files:** if only some parts of the pipeline are affected by the changes (e.g. reference affecting intermediate files).
- **Code bugs:** such as incorrect paths or typos in your code (e.g last step of the pipeline).
- **Software updates:** newer version released.
- **Parameter updates:** test/update parameters in a software tool.
- **Script Modifications:** for example, if only the plotting section of your script is updated, re-running the entire pipeline could waste significant time and resources.



Exercise 2: are notebooks better than bash scripts? (Jupyter Notebook or R Markdown)

Discuss the advantages and disadvantages of using notebooks (instead of bash scripts)

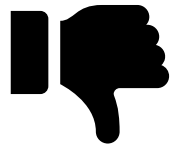
- I only have 4 steps to run
- I need to present my results to colleagues
- If my pipeline consist of 100 commands
- I want to benchmark and test several of the parameters of a new software



Solution 2.



- User-friendly interface for quick execution and visualization. Useful for debugging and great for prototyping and testing small workflows



- Notebooks allow you to run individual cells separately, however, it will become cumbersome and slow you down with intensive code blocks
- Inefficient for extensive benchmarking. Tracking & managing multiple parameters can become complex.
- Notebooks do not provide automation features



Perspective | Published: 23 September 2021



Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers

[Laura Wratten](#), [Andreas Wilm](#) & [Jonathan Göke](#) 

[Nature Methods](#) **18**, 1161–1168 (2021) | [Cite this article](#)

20k Accesses | **74** Citations | **224** Altmetric | [Metrics](#)

Tool	Class	Ease of use ^a	Expressiveness ^b	Portability ^c	Scalability ^d	Learning resources ^e	Pipeline initiatives ^f
Galaxy	Graphical	●●●	●○○	●●●	●●●	●●●	●●○
KNIME	Graphical	●●●	●○○	○○○	●●●	●●●	●●○
Nextflow	DSL	●●○	●●●	●●●	●●●	●●●	●●●
Snakemake	DSL	●●○	●●●	●●●	●●●	●●○	●●●
GenPipes	DSL	●●○	●●●	●●○	●●○	●●○	●●○
bPipe	DSL	●●○	●●●	●●○	●●●	●●○	●○○
Pachyderm	DSL	●●○	●●●	●○○	●●○	●●●	○○○
SciPipe	Library	●●○	●●●	○○○	○○○	●●○	○○○
Luigi	Library	●●○	●●●	●○○	●●●	●●○	○○○
Cromwell + WDL	Execution + workflow specification	●○○	●●○	●●●	●●●	●●○	●●○
cwltool + CWL	Execution + workflow specification	●○○	●●○	●●●	○○○	●●●	●●○
Toil + CWL/WDL/Python	Execution + workflow specification	●○○	●●●	●●○	●●●	●●○	●●○

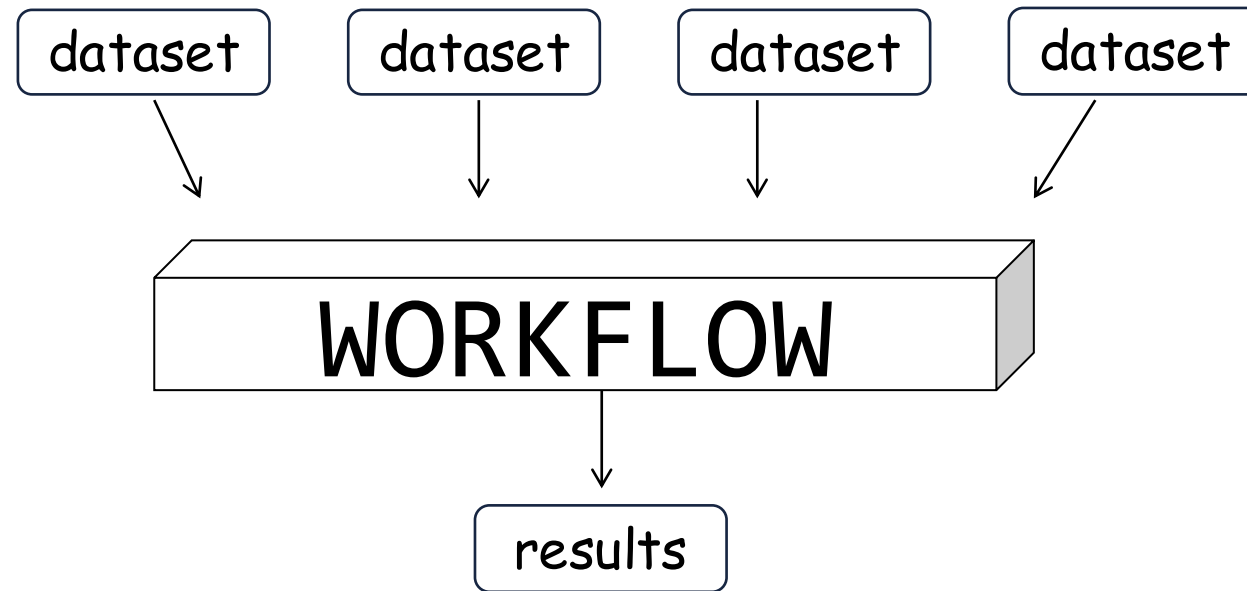
	 snakemake	 nextflow	 BROAD INSTITUTE
Readable/Popular Syntax	✓	✗	✗
Lightweight & Fast to Prototype	✓	⚠	✗
Reproducibility (envs, containers, checksums)	✓	✓	✓
Container Support (Docker/Singularity)	✓	✓	✓
Cloud-Native Execution Support	⚠	✓	✓
HPC/Cluster Integration	✓	✓	✓
Robust Run Resumption	⚠	✓	✓
Modular Workflow Structure	✓	✓	✓
Auditable / Structured Workflows	⚠	⚠	✓
Support for Regulated Environments	✗	⚠	✓
Strong Community Support	✓	✓	⚠
Easy to Debug & Maintain	✓	⚠	✗
Best Suited for Exploratory Research	✓	⚠	✗
Git Integrated	✓	✓	✗

✓ = strong support ⚠ = partial or situational ✗ = not the strongest of strengths

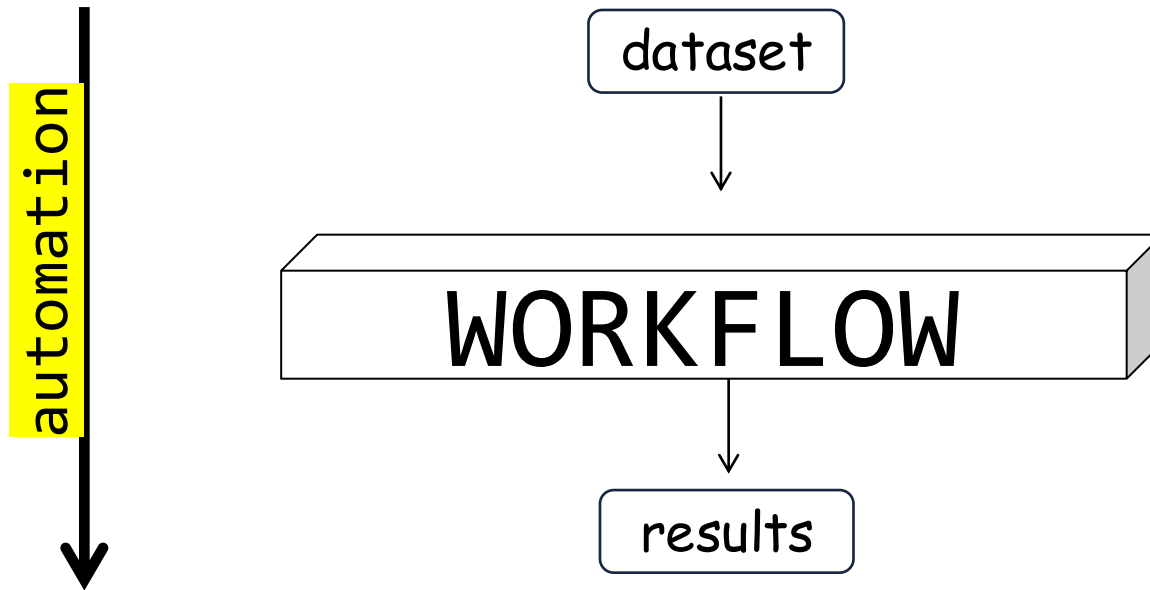




Reproducible data analysis

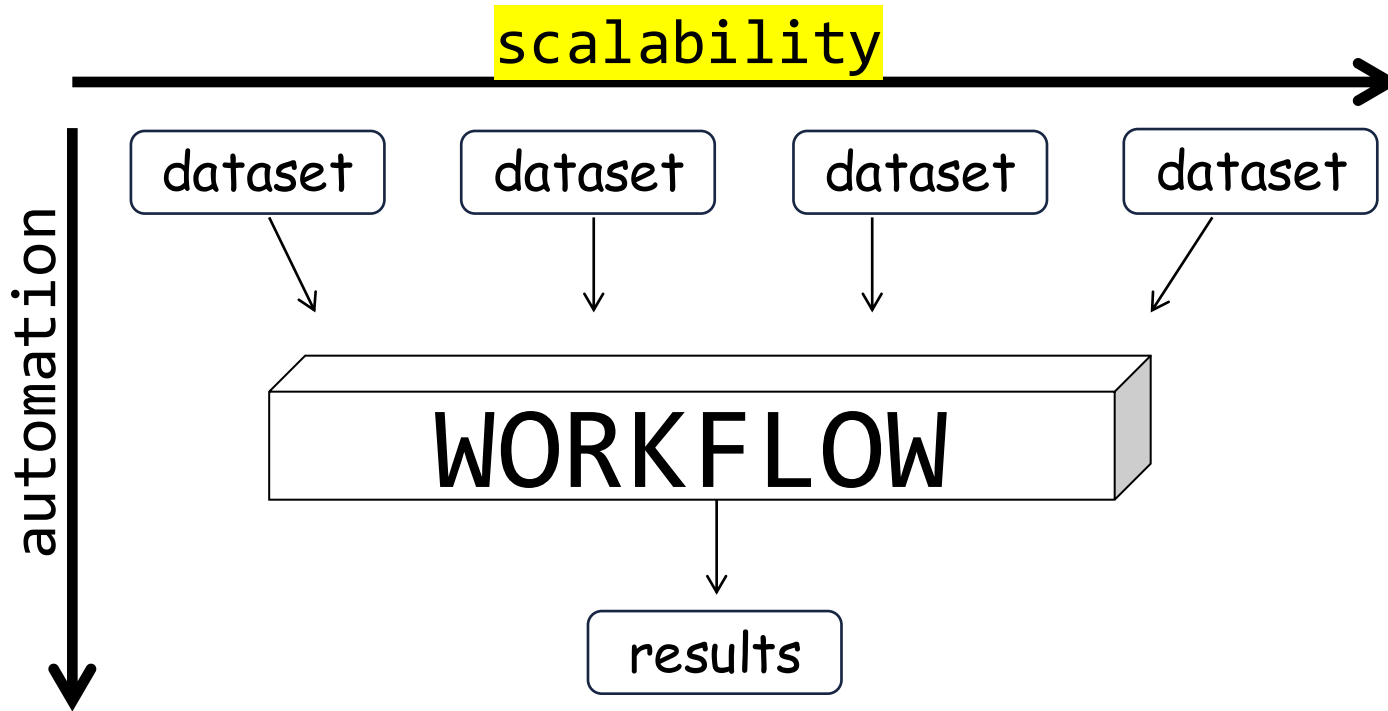


- Snakemake is a WfMS that employs a **Python-based domain-specific language (DSL)** to define and execute workflows.



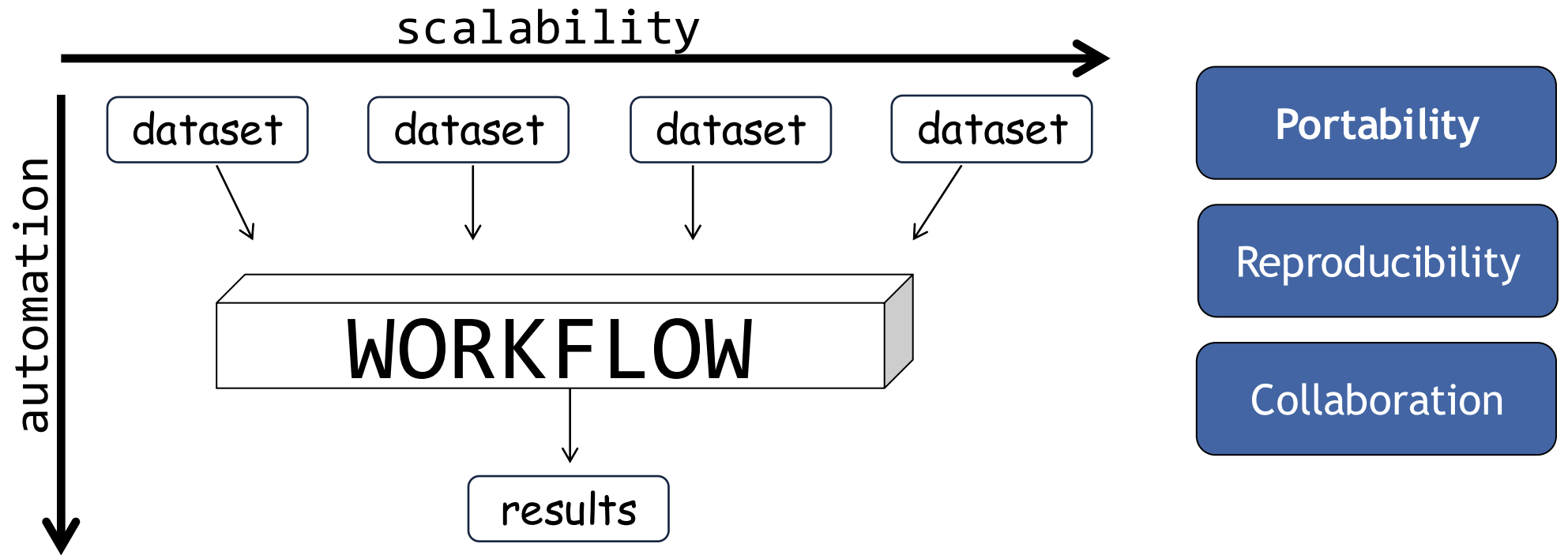
- Document parameters, tools and versions
- Handles complex dependencies (DAG)
- Execute without manual intervention





- Parallelize: execute for many datasets
- Efficiently in any computing platform





- It is widely used in bioinformatics for data processing and analysis, supported by a **large and active user community**.



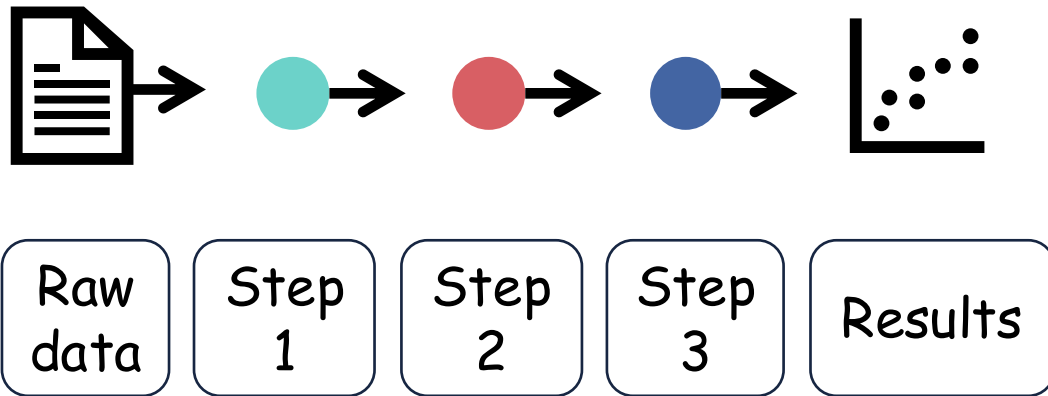
Snakemake

I - Basics

II - Advanced

Defining workflows in terms of rules

The workflow is defined in terms of **rules** (“tasks”) that define how to create output files from input files



- ```
rule mytask:
 input:
 "path/to/{dataset}.txt"
 output:
 "result/{dataset}.txt"
 script:
 "scripts/myscript.R"
```
- ```
rule myfiltration:
  input:
    "result/{dataset}.txt"
  output:
    "result/{dataset}.filtered.txt"
  shell:
    "mycommand {input} > {output}"
```
- ```
rule aggregate:
 input:
 "results/dataset1.filtered.txt",
 "results/dataset2.filtered.txt"
 output:
 "plots/myplot.pdf"
 script:
 "scripts/myplot.R"
```



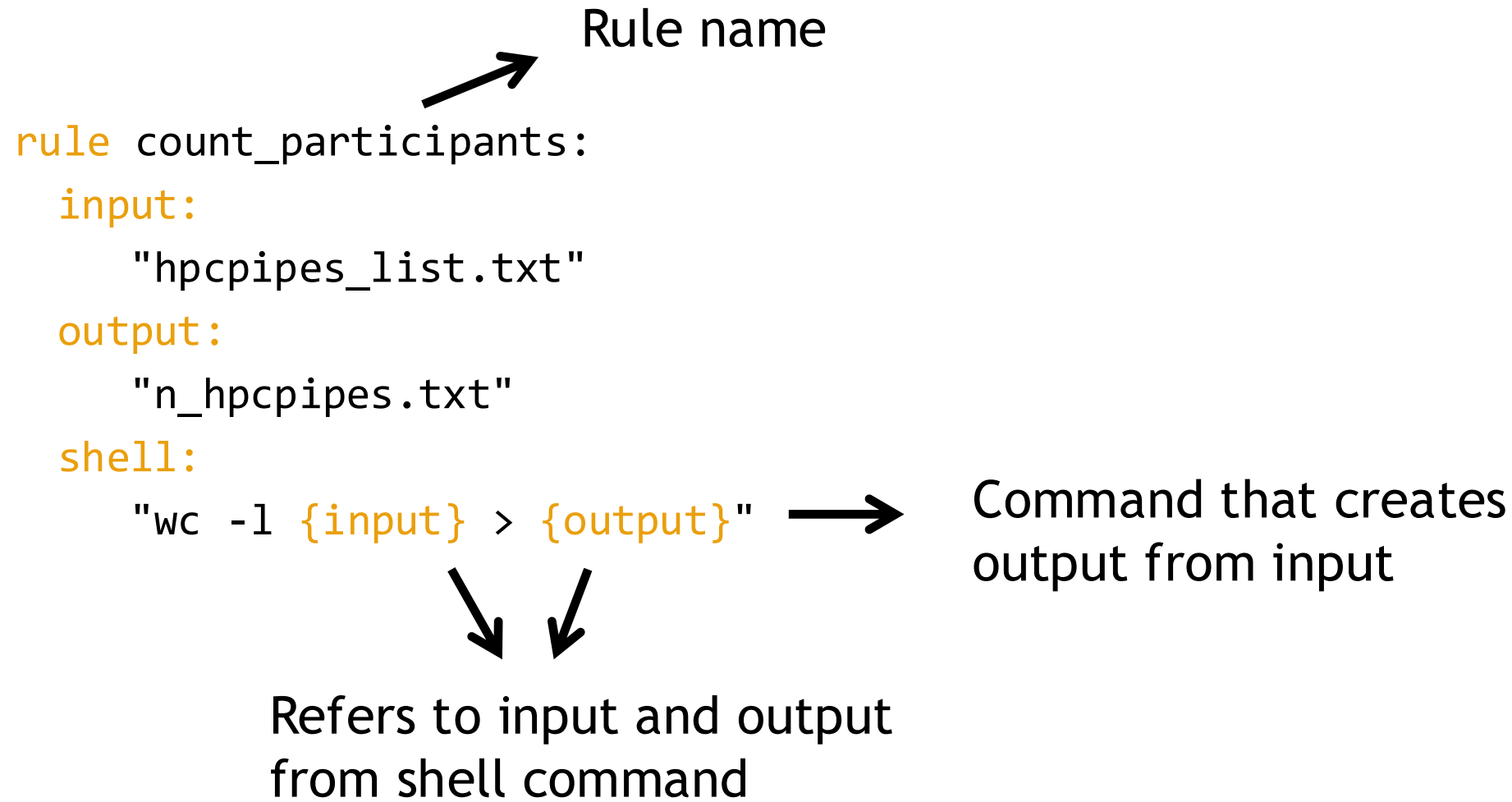
## A rule: structure

```
rule count_participants:
 input:
 "hpcpipes_list.txt"
 output:
 "n_hpcpipes.txt"
 shell:
 "wc -l {input} > {output}"
```

Rule name

Command that creates output from input

Refers to input and output from shell command



## Running a rule on the terminal: defining targets

### Snakefile

```
rule count_participants:
 input:
 "hpcpipes_list.txt"
 output:
 "n_hpcpipes.txt"
 shell:
 "wc -l {input} > {output}"
```

```
> cat hpcpipes_list.txt
```

```
Alba
Anne
Hugh
Monica
Steve
Marco
```



Snakemake will run first rule by default!

```
> snakemake --cores 1
> snakemake --cores 1 count_participants
> snakemake --cores 1 n_hpcpipes.txt
```

The target can be a rule or an output file



# Running a rule on the terminal: defining targets

## Snakefile

```
rule count_participants:
 input:
 "hpcpipes_list.txt"
 output:
 "n_hpcpipes.txt"
 shell:
 "wc -l {input}"
```

```
> cat hpcpipes_list.txt
```

Alba

Anne

Hugh

Monica

More on targets in a few slides

Hint: rule all

## Snakemake

```
> snakemake --cores 1
> Snakemake --cores 1 count_participants
> Snakemake --cores 1 n_hpcpipes.txt
```

The target can be a rule or an output file



# Modifying the input

Snakefile

```
rule count_participants:
 input:
 "hpcpipes_list.txt"
 output:
 "n_hpcpipes.txt"
 shell:
 "wc -l {input} > {output}"
```

```
> cat hpcpipes_list.txt
```

Alba  
Anne  
Hugh  
Steve  
Marco  
Carlo  
Mia



```
> snakemake --cores 1
```

This will rerun the pipeline as the input timestamp is newer than the output





# Organise your project structure

Snakefile

```
rule count_participants:
 input:
 "hpcpipes_list.txt"
 output:
 "stats/n_hpcpipes.txt"
 shell:
 "wc -l {input} > {output}"
```

Snakemake will create  
the dir for you

WD -> sandbox-courses

```
sandbox-courses
├── hpcpipes_list.txt
├── stats
│ └── n_hpcpipes.txt
└── figures
```



## Wildcards: how to generalise my rules?

We need to run the same stats for all our courses:

```
course_names = ["hpc-pipes", "hpc-launch", "genomics", "transcriptomics"]
```



# Wildcards: how to generalise my rules?

Snakefile

```
rule count_participants:
 input:
 "{course_name}_list.txt"
 output:
 "stats/n_{course_name}.txt"
 shell:
 "wc -l {input} > {output}"
```

Named wildcards

The rule can produce ANY files that follow the regular expression pattern:

`^stats/n_.+\.txt`

(Wildcards are replaced by the regular expression `.+` )

Naming conventions are important! use regular expressions to enforce them, ensuring consistency and ease of file management.



# Wildcards: how to generalise my rules?

## Snakefile

```
rule count_participants:
 input:
 "{course_name}_list.txt"
 output:
 "stats/n_{course_name}.txt"
 shell:
 "wc -l {input} > {output}"
```

## Named wildcards

If the rule's output matches a requested file, the **substrings** matched by the wildcards are **propagated** to the input files and to variable wildcards

requested files



```
> snakemake --cores 1 stats/n_hpcpipes.txt stats/n_hpcm1.txt
> snakemake --cores 1 stats/n_{hpcpipes, hpcm1}.txt
```



# Variable wildcards in the command-line

Snakefile

```
rule count_participants:
```

```
 input:
```

```
 "{course_name}_list.txt"
```

```
 output:
```

```
 "stats/n_{course_name}.txt"
```

```
 shell:
```

```
 "myscript -i {input} -j {wildcards.course_name} -o {output}"
```



# Variable wildcards in the command-line

Other examples shell:

```
plink --file {input} --out plink/{wildcards.dataset} --make-bed
```

```
samtools sort -O bam -o {wildcards.prefix} {input}
```

Some software required prefixed (as flags) to specify input and/or output filenames



# Targets

```
snakemake -c 1 <target>
```

- Filenames (output files)
- Rule names
- "None"
- Snakemake searches for a file named Snakefile in your current directory



## Targets: Snakefile

- A. snakemake will define the first rule of the snakefile as the target
- B. Target is defined in the Snakefile (**rule all**)

Does this mean I can't name it differently or place it in a different location on the server?



Snakefile

qc.smk

align.smk

Annotation.smk





# Targets

`snakemake -c 1 <target>`

- Filenames - output files
- Rule names
- "None"
  - snakemake will define the first rule of the snakefile as the target
  - Target is defined in the snakefile (rule all)

No file named “Snakefile”, use `--snakefile (-s)` argument

- `snakemake -c <cores> --snakefile /path/to/mypipe.smk <target>`



Target: rule all

Tell snakemake what  
files you want to be  
created

**rule all:**

input:

"plots/myplot.pdf"

rule mytask:

input:

"path/to/{dataset}.txt"

output:

"result/{dataset}.txt"

script:

"scripts/myscript.R"

Use wildcards to  
write general rule  
for all dataset

rule myfiltration:

input:

"result/{dataset}.txt"

output:

"result/{dataset}.filtered.txt"

shell:

"mycommand {input} > {output}"

rule aggregate:

input:

"results/dataset1.filtered.txt",

"results/dataset2.filtered.txt"

output:

"plots/myplot.pdf"

script:

"scripts/myplot.R"



## Target: rule all



### Best practice

Having a **rule all** at the top of the workflow (e.g. Snakefile) which has all typically desired target files as input files (e.g. output from last rule)

**rule all:**

**input:**

**"plots/myplot.pdf"**

rule mytask:

input:

"path/to/{dataset}.txt"

output:

"result/{dataset}.txt"

script:

"scripts/myscript.R"

rule myfiltration:

input:

"result/{dataset}.txt"

output:

"result/{dataset}.filtered.txt"

shell:

"mycommand {input} > {output}"

rule aggregate:

input:

"results/dataset1.filtered.txt",

"results/dataset2.filtered.txt"

output:

**"plots/myplot.pdf"**

script:

"scripts/myplot.R"



## Target rules using wildcards

Collecting all results using 3 different approaches:

```
DATASETS=["D1", "D2", "D3"]
```

```
rule all:
```

```
 input:
```

```
 [f"{dataset}.sorted.txt" for dataset in DATASETS],
```

```
 OR
```

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

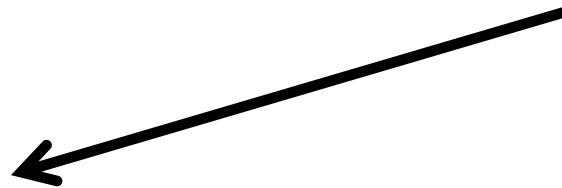
```
 OR
```

```
 "D1.sorted.txt",
```

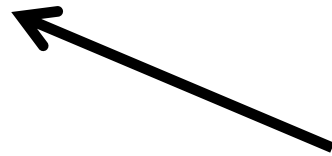
```
 "D2.sorted.txt",
```

```
 "D3.sorted.txt"
```

Use arbitrary python  
code in your workflow



Use helper functions!



# Named variables: labelling for easy reference within rules



Useful when command  
has many input files  
and/or params

## Refer to files by index

Snakefile

```
rule concat:
 input:
 "stats/n_hpcpipes.txt",
 "stats/n_hpclaunch.txt"
 output:
 "stats/allcourses.txt"
 shell:
 "cat {input[0]} {input[1]}
 {output}"
```

## Refer to files by name

Snakefile

```
rule concat:
 input:
 pipes="stats/n_hpcpipes.txt",
 launch="stats/n_hpclaunch.txt"
 output:
 all="stats/allcourses.txt"
 shell:
 "cat {input.pipes}
 {input.launch} > {output.all}"
```



## Internal scripts

Run **python code** in rules

```
rule sort:
 input: "{dataset}.txt",
 output: "{dataset}.sorted.txt"
 run:
 with open(output[0], "w") as out:
 for l in sorted(open(input[0])):
 print(l, file=out)
```



# Dependencies are determined top-down

Which rules create the target?

aggregate

For the input files of the job (aggregate),  
go on **recursively**


myfiltration -> mytask

```
rule all:
 input:
 "plots/myplot.pdf"

rule mytask:
 input:
 "path/to/{dataset}.txt"
 output:
 "result/{dataset}.txt"
 script:
 "scripts/myscript.R"

rule myfiltration:
 input:
 "result/{dataset}.txt"
 output:
 "result/{dataset}.filtered.txt"
 shell:
 "mycommand {input} > {output}"

rule aggregate:
 input:
 "results/dataset1.filtered.txt",
 "results/dataset2.filtered.txt"
 output:
 "plots/myplot.pdf"
 script:
 "scripts/myplot.R"
```



## Job execution (when)

- output file is target and does not exist
- output file needed by another (executed) job and does not exist
- input file newer than output file
- rule has been modified
- input file will be updated by other job
- execution is enforced (--rerun, --force-all)





# Command-line: test your pipeline! (and debug)

Assumption: rules defined in a Snakefile

# dry-run, print shell commands

```
snakemake -n -p
```

# dry-run, print execution reason for each job

```
snakemake -n -r
```

# visualize the DAG of jobs using Graphviz dot command

```
snakemake --dag | dot -Tsvg > dag.svg
```



## Directed acyclic graph (DAG) of jobs

Snakemake builds a DAG before execution to understand all job dependencies and task order. This is how it knows **what needs to run, when, and in what order.**

Workflows are executed in **three phases**

1. initialization phase (parsing)
2. DAG phase (DAG is built)
3. scheduling phase (execution of DAG)



## Directed acyclic graph (DAG) of jobs

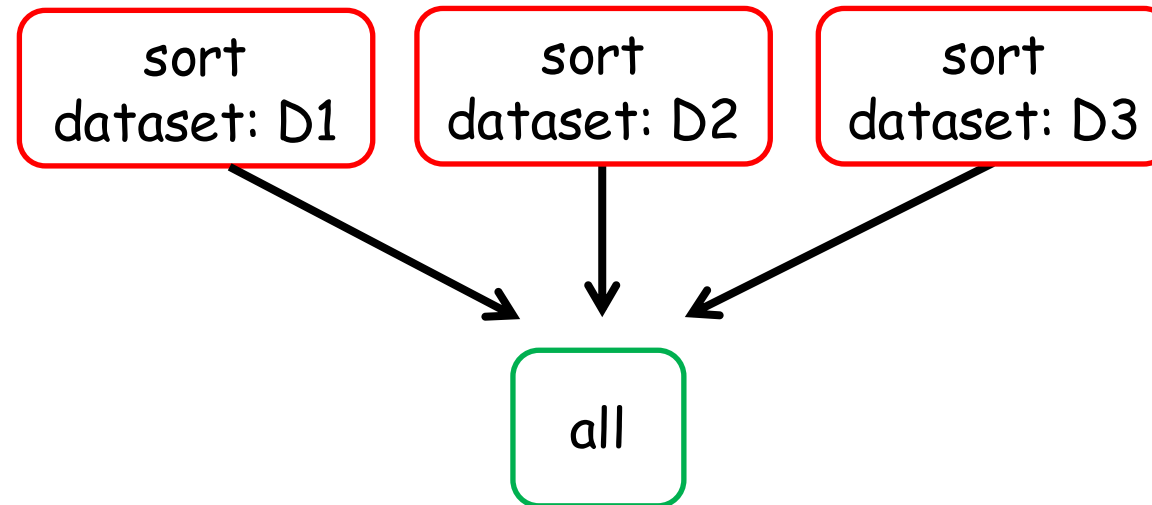
```
DATASETS=["D1", "D2", "D3"]
```

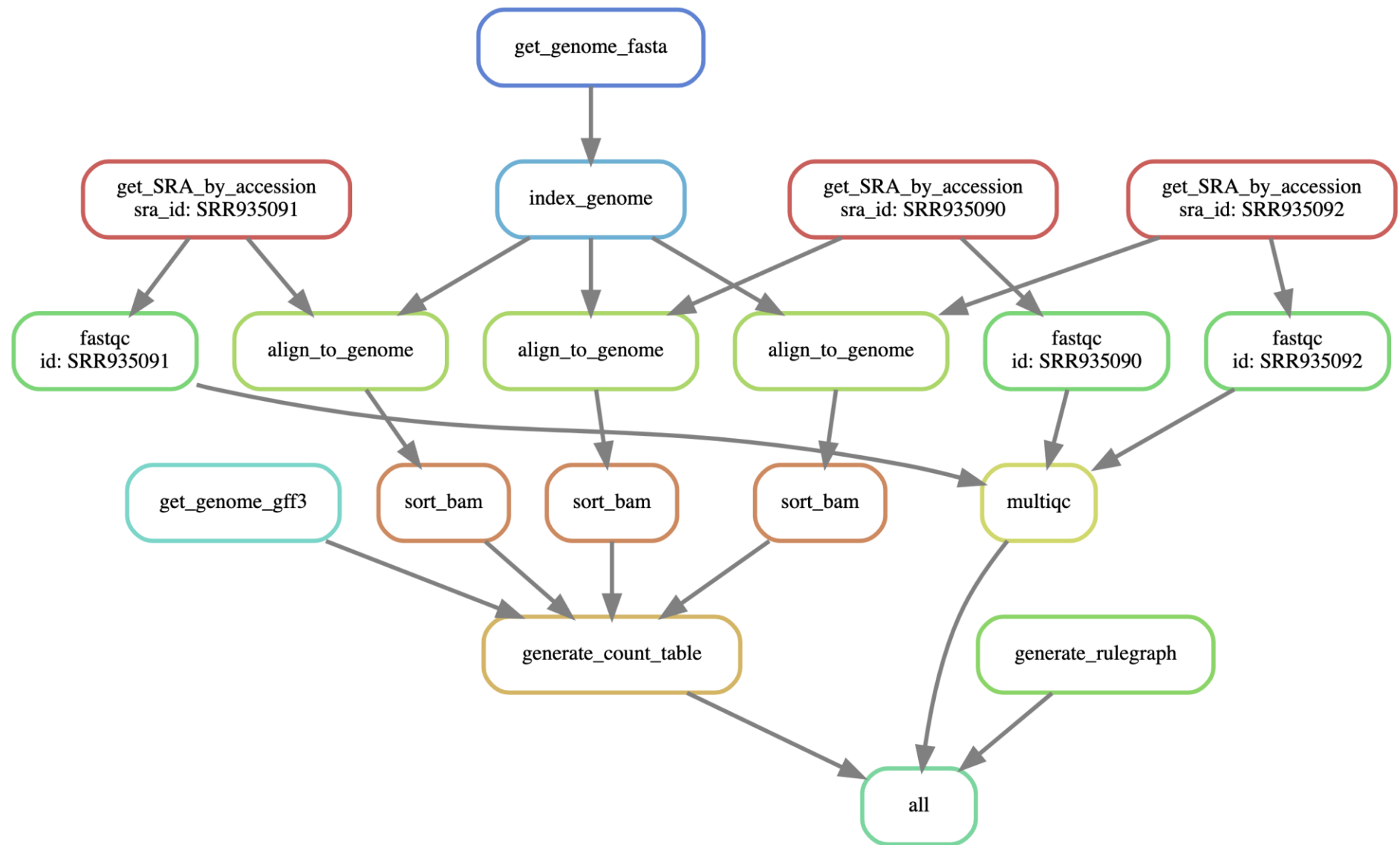
```
rule all:
```

```
 input:
```

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

Dependencies between rules are determined automatically, creating a DAG of jobs that can get automatically parallelized









~ 30 min



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

workshop > HPC Pipes > Day 1

**snakemake**

Exercise I and II

Now is YOUR time!

I- General knowledge about workflow managers

II - Snakemake exercise

*Imagine a colleague shares a snakemake pipeline with you—let's explore it and see if we understand how rules/processes are invoked in the workflow*



**snakemake**

# Problems/Issues/Comments

## Summary

- Rules syntax and structure
- Targets. Recommended: target rule all
- Wildcards to generalise our pipelines
- DAG. Check your pipeline makes sense!

BUT, there are plenty of other cool features.





## A bit more on external scripts...

You can use `script` directive when running **Julia**, **Python**, **R**, **Rust**, **Bash**

```
rule sort:
 input:
 "{dataset}.txt",
 output:
 "{dataset}.sorted.txt"
 script:
 "myscript.R {input[0]}"
```



## External scripts

myscript.py



```
import pandas as pd
```

```
data = pd.read_table(snakemake.input[0])
data = data.sort_values("id")
data.to_csv(snakemake.output[0], sep="\t")
```

myscript.r



```
data <- read.table(snakemake@input[[1]])
data <- data[order(data$id),]
write.table(data, file =
 snakemake@output[[1]])
```

script directive ties the workflow tightly to Snakemake-specific objects,  
less portable and reusable code



Personal preference, use shell and argument parser functions in the  
scripts (make them reusable outside the pipeline, faster dev and testing)



# Jupyter notebook integration

<https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#jupyter-notebook-integration>



