



Introduction to snakemake

Jamie Gooding (adapted from Vukan Jevtić and Louis Gerken)

Fakultät Physik, Technische Universität Dortmund

Programmierkurs 2023 | 29th March 2023

 technische universität
dortmund

This Programmierkurs session provides a whistle-stop tour of Snakemake.

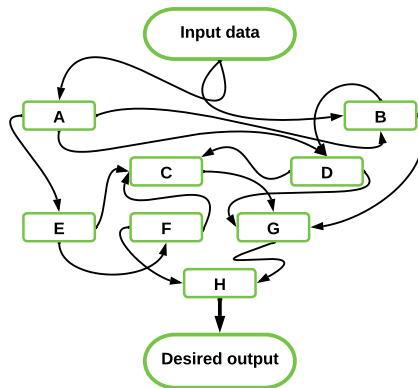
It is a good idea to keep the following questions in mind:

- What is Snakemake?
- Why is it useful?
- How do I use it?
- What can I do with it?
- Why might I want to use it in my work?

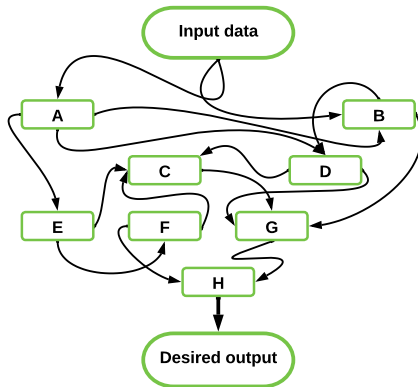
It is not possible to cover absolutely everything in a single session so please ask as many questions as you like throughout! 🙋

Introducing Snakemake

Throughout your work you will encounter workflows...

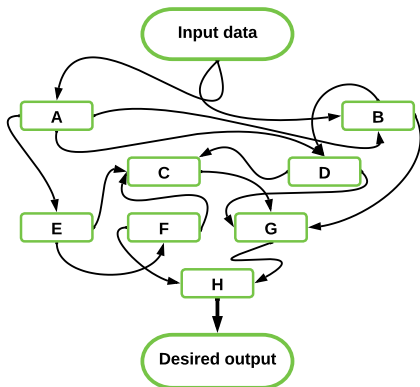


Throughout your work you will encounter workflows...

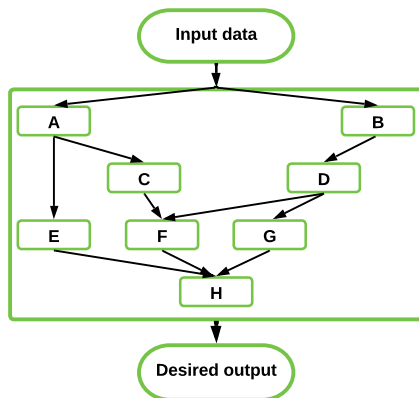


But how can we go from this?

Throughout your work you will encounter workflows...

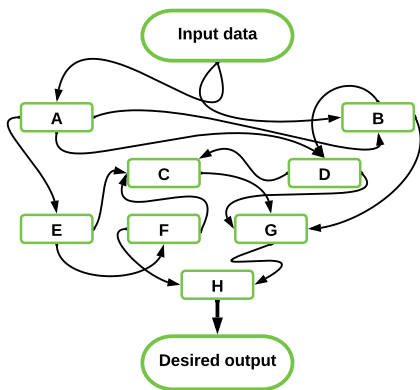


But how can we go from this?

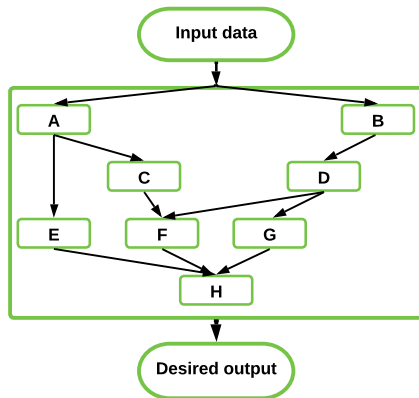


To this!

Throughout your work you will encounter workflows...



But how can we go from this?



To this!

Enter, the workflow manager—handy tool to, unsurprisingly, **manage workflows!**



Many workflow managers are available, though typically the following are used:

Make

Snakemake

Many workflow managers are available, though typically the following are used:

Make

- + Universal, always available
- + Supports abstraction
- Exclusively Bash
- Hard to read
- Hard to debug
- Hard to find rule for specific file (especially if abstraction used)

Snakemake

Many workflow managers are available, though typically the following are used:

Make

- + Universal, always available
- + Supports abstraction
- Exclusively Bash
- Hard to read
- Hard to debug
- Hard to find rule for specific file (especially if abstraction used)

Snakemake

- + Install via conda
- + Recipes can contain Bash/Python
- + Recipes are named
- + Much easier to read
- + Abstractions easy to understand
- + Can submit jobs to a cluster
- + If rule fails: output is deleted
- Additional `.snakemake` directory

Officially, snakemake is a...

“A scalable bioinformatics workflow engine”

...so why are we using a bioinformatics tool?

*Köster, Johannes and Rahmann, Sven. “Snakemake - A scalable bioinformatics workflow engine”.
Bioinformatics 2012.*

Officially, snakemake is a...

Applicable to problems
of all sizes

“A scalable bioinformatics workflow engine”

...so why are we using a bioinformatics tool? **Let's break down what this means!**

*Köster, Johannes and Rahmann, Sven. “Snakemake - A scalable bioinformatics workflow engine”.
Bioinformatics 2012.*

Officially, snakemake is a...

Applicable to problems
of all sizes

“A *scalable bioinformatics workflow engine*”

Designed with data
analysis in mind

...so why are we using a bioinformatics tool? **Let's break down what this means!**

Köster, Johannes and Rahmann, Sven. “Snakemake - A scalable bioinformatics workflow engine”.
Bioinformatics 2012.

What is snakemake?

Officially, snakemake is a...

Applicable to problems
of all sizes

Capable of handling
bespoke use-cases

“A scalable bioinformatics workflow engine”

Designed with data
analysis in mind

...so why are we using a bioinformatics tool? **Let's break down what this means!**

*Köster, Johannes and Rahmann, Sven. “Snakemake - A scalable bioinformatics workflow engine”.
Bioinformatics 2012.*

What is snakemake?

Officially, snakemake is a...

Applicable to problems
of all sizes

Capable of handling
bespoke use-cases

“A scalable bioinformatics workflow engine”

Designed with data
analysis in mind

Tool for running our
code (and more!)

...so why are we using a bioinformatics tool? **Let's break down what this means!**

Köster, Johannes and Rahmann, Sven. “Snakemake - A scalable bioinformatics workflow engine”.
Bioinformatics 2012.

If you're working on our cluster:

Good news!

Snakemake is already installed across the group **conda** environments 🎉

If you're working on your own machine/own **conda** environment:

You can install Snakemake in your own **conda** environment as follows:

```
$ conda config --add channels bioconda
$ conda install snakemake
```

Workflow: `raw_data.csv` \Rightarrow `data.csv` \Rightarrow `plot.pdf`

Workflow: raw_data.csv \Rightarrow data.csv \Rightarrow plot.pdf

Make

```
1 plot.pdf: data.csv
2     python plot.py
3
4 data.csv: raw_data.csv
5     python selection.py
```

Minimal console command

```
$ make
```

Snakemake

```
1 rule make_plot:
2     input: "data.csv"
3     output: "plot.pdf"
4     shell: "python plot.py"
5
6 rule select_data:
7     input: "raw_data.csv"
8     output: "data.csv"
9     shell: "python selection.py"
```

Minimal console command

```
$ snakemake
```

Workflow: raw_data.csv \Rightarrow data.csv \Rightarrow plot.pdf

Make

```
1 plot.pdf: data.csv
2     python plot.py
3
4 data.csv: raw_data.csv
5     python selection.py
```

Running by specifying output file

```
$ make plot.pdf
```

Snakemake

```
1 rule make_plot:
2     input: "data.csv"
3     output: "plot.pdf"
4     shell: "python plot.py"
5
6 rule select_data:
7     input: "raw_data.csv"
8     output: "data.csv"
9     shell: "python selection.py"
```

Running by specifying output file

```
$ snakemake plot.pdf
```

Workflow: raw_data.csv \Rightarrow data.csv \Rightarrow plot.pdf

Make

```
1 plot.pdf: data.csv
2     python plot.py
3
4 data.csv: raw_data.csv
5     python selection.py
```

Running by specifying name of rule

```
$ ???
```

Snakemake

```
1 rule make_plot:
2     input: "data.csv"
3     output: "plot.pdf"
4     shell: "python plot.py"
5
6 rule select_data:
7     input: "raw_data.csv"
8     output: "data.csv"
9     shell: "python selection.py"
```

Running by specifying name of rule

```
$ snakemake make_plot
```

Workflow: raw_data.csv \Rightarrow data.csv \Rightarrow plot.pdf

Make

```
1 plot.pdf : data.csv
2     python script.py
3
4 data.csv : raw_data.csv
5     python selection.py
```

Output

Input

Recipe

Snakemake

```
1 rule make_plot:
2     input: "data.csv"
3     output: "plot.pdf"
4     shell: "python plot.py"
5
6 rule select_data:
7     input: "raw_data.csv"
8     output: "data.csv"
9     shell: "python selection.py"
```

+Self-Documentation

To check our pipeline we can dry run with `snakemake -nr`:

```
Building DAG of jobs...
Job counts:
   count  jobs
     1    make_plot
     1    select_data
     2
```

```
[Wed Feb 19 15:50:05 2020]
rule select_data:
  input: raw_data.csv
  output: data.csv
  jobid: 1
  reason: Missing output files: data.csv
```

```
[Wed Feb 19 15:50:05 2020]
rule make_plot:
  input: data.csv
  output: plot.pdf
  jobid: 0
  reason: Missing output files: plot.pdf; Input
  files updated by another job: data.csv
```

```
Job counts:
   count  jobs
     1    make_plot
     1    select_data
     2

This was a dry-run (flag -n). The order of jobs does
not reflect the order of execution.
```

We can also use `snakemake -n` if we don't care about seeing the reasons for each job being run.

Removing `-nr` runs the pipeline (i.e. just `snakemake`) runs the pipeline:

```
Building DAG of jobs...
Using shell: /usr/local/bin/bash
Provided cores: 256
Rules claiming more threads will be scaled down.
Job counts:
```

count	jobs
1	make_plot
1	select_data
2	

```
[Wed Feb 19 15:38:12 2020]
rule select_data:
  input: raw_data.csv
  output: data.csv
  jobid: 1
```

```
[Wed Feb 19 15:38:12 2020]
Finished job 1.
1 of 2 steps (50%) done
```

```
[Wed Feb 19 15:38:12 2020]
rule make_plot:
  input: data.csv
  output: plot.pdf
  jobid: 0
```

```
[Wed Feb 19 15:38:14 2020]
Finished job 0.
2 of 2 steps (100%) done
Complete log: /net/nfshome/home/somepath/.snakemake/
log/2020-02-19T153812.433826.snakemake.log
```

In fact there are many options we can choose when running `snakemake`!

In fact there are many options we can choose when running snakemake!

Just print scheduled rules without running

```
$ snakemake <rule> -n
```

Print reason for running each rule as well

```
$ snakemake <rule> -n -r
```

Force execution of target

```
$ snakemake <rule> -f
```

Force execution of a target and its workflow

```
$ snakemake <rule> -F
```

Force re-execution of rule and its workflow

```
$ snakemake <rule> -R
```

Run workflow until specified rule

```
$ snakemake <rule> --until <rule>
```

Update timestamps → force files up to date

```
$ snakemake <rule> --touch
```

Ignore errors

```
$ snakemake <rule> --keep-going
```

Rerun incomplete rules (in case of crash)

```
$ snakemake --rerun-incomplete
```

Print shell commands that snakemake runs

```
$ snakemake -p
```

To train a BDT, we usually need a selected data and mc file:

Example analysis pipeline

```
1 rule data_preselection:
2     input:  "data_raw.root"
3     output: "data_selected.root"
4     shell:  "python selection.py data_raw.root data_selected.root"
5
6 rule mc_preselection:
7     input:  "mc_raw.root"
8     output: "mc_selected.root"
9     shell:  "python selection.py mc_raw.root mc_selected.root"
10
11 rule train_bdt:
12     input:
13         "data_selected.root",
14         "mc_selected.root"
15     output: "bdt.model"
16     shell:  "python train_bdt.py data_selected.root mc_selected.root"
```

It would be nice to reduce amount of repetitions:

Example analysis pipeline

```
1 rule data_preselection:
2     input: "data_raw.root"
3     output: "data_selected.root"
4     shell: "python selection.py data_raw.root data_selected.root"
5
6 rule mc_preselection:
7     input: "mc_raw.root"
8     output: "mc_selected.root"
9     shell: "python selection.py mc_raw.root mc_selected.root"
10
11 rule train_bdt:
12     input:
13         "data_selected.root",
14         "mc_selected.root"
15     output: "bdt.model"
16     shell: "python train_bdt.py data_selected.root mc_selected.root"
```

input, output, shell etc. are optional

Example analysis pipeline

```
1 rule data_preselection:
2     input: "data_raw.root"
3     output: "data_selected.root"
4     shell: "python selection.py {input} {output}"
5
6 rule mc_preselection:
7     input: "mc_raw.root"
8     output: "mc_selected.root"
9     shell: "python selection.py {input} {output}"
10
11 rule train_bdt:
12     input:
13         data = rules.data_preselection.output,
14         mc    = rules.mc_preselection.output
15     output: "bdt.model"
16     shell: "python train_bdt.py {input.data} {input.mc}"
```

We can alias files \Rightarrow rules can reference their own parameters

Example analysis pipeline

```
1 rule data_preselection:
2     input: "data_raw.root"
3     output: "data_selected.root"
4     shell: "python selection.py {input} {output}"
5
6 rule mc_preselection:
7     input: "mc_raw.root"
8     output: "mc_selected.root"
9     shell: "python selection.py {input} {output}"
10
11 rule train_bdt:
12     input:
13         data = rules.data_preselection.output,
14         mc    = rules.mc_preselection.output
15     output: "bdt.model"
16     shell: "python train_bdt.py {input.data} {input.mc}"
```

Getting the most out of Snakemake

We keep mentioning the term “abstraction”, but what does this mean?

Let's say we have the following files containing data:

- `data.csv`,
- `dataset.csv`,
- `raw_data.csv`,

which we can pass to a script to generate the plots

- `data.pdf`,
- `dataset.pdf`,
- `raw_data.pdf`.

We keep mentioning the term “abstraction”, but what does this mean?

Let's say we have the following files containing data:

- `data.csv`,
- `dataset.csv`,
- `raw_data.csv`,

which we can pass to a script to generate the plots

- `data.pdf`,
- `dataset.pdf`,
- `raw_data.pdf`.

Abstraction allows us to leave the filename to the workflow manager to determine!

i.e. input file of

`*.pdf`

and output file of

`*.pdf`

where `*` must match between input and output.

A wildcard rule matches patterns in dependencies

Example of wildcards

```
1 rule single_selection:
2     input: "data_{num}.root"
3     output: "data_{num}_selected.root"
4     shell: "python run_selection.py {input} {output}"
5
6 rule select_files:
7     input: expand("data_{n}_selected.root", n=range(10))
```

Note:

1. Input and output *must* contain same wildcards
2. A wildcard rule cannot be called directly by its name
3. Two rules should not contain the same outputs

A wildcard rule matches patterns in dependencies

Example of wildcards

```
1 rule single_selection:
2     input: "data_{num}.root"
3     output: "data_{num}_selected.root"
4     shell: "python run_selection.py {input} {output}"
5
6 rule select_files:
7     input: expand("data_{n}_selected.root", n=range(10))
```

If one runs

```
$ snakemake select_files
```

rule `select_files` is going to call the wildcard rule for 10 different files

A wildcard rule matches patterns in dependencies

Example of wildcards

```
1 rule single_selection:
2     input: "data_{num}.root"
3     output: "data_{num}_selected.root"
4     shell: "python run_selection.py {input} {output}"
5
6 rule select_files:
7     input: expand("data_{n}_selected.root", n=range(10))
```

If one runs

```
$ snakemake data_7_selected.root
```

rule `select_files` is going to call the wildcard rule for case `num = 7`

Inside a wildcard rule, a variable named `wildcards` is defined

Example of the wildcards object

```
1 rule my_wildcard_rule:
2     input: "file_{channel}_{polarity}_{year}.root"
3     output: "file_{channel}_{polarity}_{year}_out.root"
4     message: "Reading {wildcards.year} file"
5     run:
6         print("Running the rule for year = {wildcards.year}")
7         if wildcards.year == "2017" and wildcards.channel == "B2JpsiKstar":
8             print("This is my favourite dataset!")
9         shell("python run_selection {input} {output} -year {wildcards.year}")
```

But what if a certain combination of wildcards needs to be treated differently?

⇒ Use wildcard constraints

Example of wildcard_constraints

```
1 rule somerule:
2     input: "data_{year}.root"
3     output: "massplot_{year}.pdf"
4     wildcard_constraints: year="201[578]"
5     shell: "python massfit.py {input}"
6
7 rule somerule_special_case:
8     input: rules.somerule.input
9     output: rules.somerule.output
10    wildcard_constraints: year="2016"
11    shell: "python massfit.py {input} -be_careful"
```

If you need to treat a wildcard value differently from the others, you need to constrain them for each relevant rule as shown here.

Here, regex can be quite useful: [regex tester at regex101.com](https://regex101.com).

Snakemake contains a method for generating lists of files, `expand(...)`:

Example of `expand(...)`

```
1 rule file_requester:  
2     input: expand("file_{cat}_{num}.txt", cat=["A", "B"], num=range(3))
```

The following list is created as input:

`file_A_0.txt`, `file_A_1.txt`, `file_A_2.txt`, `file_B_0.txt`, `file_B_1.txt`, `file_B_2.txt`

This method can also be used when considering

Parallelising
within jobs



Parallelising
in pipeline



If you have a rule wherein your script may use more than 1 CPU core, more cores can be assigned using the `threads` field:

Example of threads

```
1 rule somerule:
2     input: "data_{year}.root"
3     output: "massplot_{year}.pdf"
4     wildcard_constraints: year="201[578]"
5     threads: 4
6     shell: "python massfit.py {input}"
```

With the number of cores provided to snakemake determined by the flag `-jN`, where `N` many cores are then provided, e.g.

```
$ snakemake -j8
```


Snakemake automatically runs any jobs in parallel which can be run in parallel, provided sufficiently many cores have been provided.

Say we provided snakemake with 8 cores:

```
$ snakemake -j8
```

then these cores will be distributed across as many jobs as possible, e.g. 2 instances of our 4-CPU requiring **somerule**, at once.

Note that jobs which require more cores than are provided will be run with as many cores as are provided, rather than the specified amount in **threads**.

What if I need more cores?

Sometimes you might just need more cores, for these cases, we use the computing cluster:

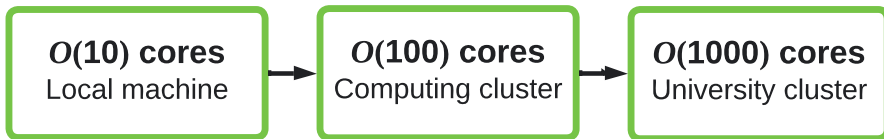


Local machine



Computing cluster

Generally we follow the following guidelines:



Submitting jobs to a HTCondor computing cluster is relatively straightforward ([see tutorial here](#)).

A profile has already been set up for submitting jobs to our cluster, so jobs can be submitted as:

```
$ snakemake <rule> -j999 --profile htcondor
```

It is always a good idea to set resource requirements for your rules, as to not take excessive resources from others. This can be done as follows:

Example rule with resource requirements

```
1 rule clusterrule:
2     input: "file.txt"
3     output: "outfile.txt"
4     threads: 12
5     resources:
6         MaxRunHours=24,      # Job takes up to a day
7         request_memory=1024  # Request RAM in MB
8         request_gpus=1,      # Submit to a machine with GPU
9         request_disk=1000000 # Disk requirement in kB
10    run:
11        print(f"This rule is allowed to use {threads} threads")
```

Rules with appropriate resource requirements will also typically be run sooner as the cluster will use your requirements instead of its higher default requirements (i.e. the job looks less needy).

Tips and Tricks

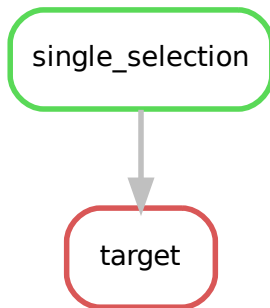
The first rule in your main Snakefile can be treated as a target rule:

Example of target rule

```
1 rule target:
2     input: expand("data_{n}_selected.root", n=range(4))
3
4 rule single_selection:
5     input: "data_{num}.root"
6     output: "data_{num}_selected.root"
7     shell: "python run_selection.py {input} {output}"
```

You can name this whatever you like, but conventionally this is named **target**, **main** or **all**.

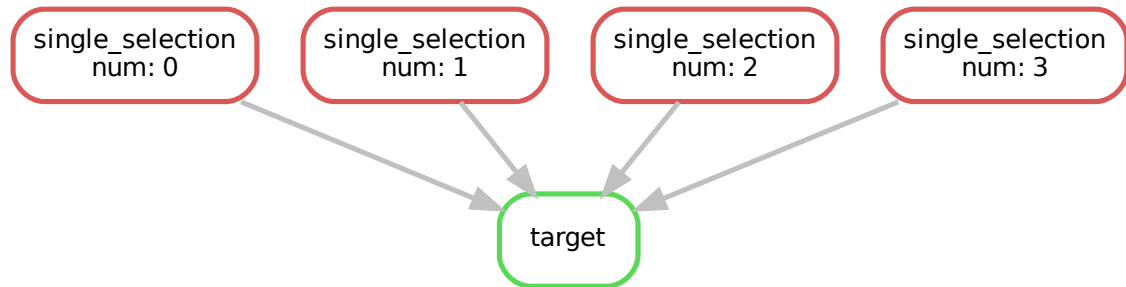
It is useful to be able to visualise our pipelines—Snakemake handles this too!
Using the pipeline from the previous slide:



Produce a rulegraph, i.e. flowchart of rules.

```
$ snakemake --rulegraph | dot -Tpdf > rulegraph.pdf
```

It is useful to be able to visualise our pipelines—Snakemake handles this too!
Using the pipeline from the previous slide:



Produce a directed acyclic graph (DAG), i.e. flowchart of jobs.

```
$ snakemake --dag | dot -Tpdf > dag.pdf
```


Snakefiles can be connected via subworkflows:

Main Snakefile

```
1  subworkflow another_worklow:
2      workdir: 'path/to/other/workdir'
3      snakefile: 'path/to/other/workdir/Snakefile'
4
5  rule master_rule:
6      input: another_worklow("text.txt")
```

Another Snakefile

```
1  rule create_file:
2      output: "text.txt"
3      shell: "touch text.txt"
```

There are many file wrappers to make your life easier!

- Timestamp of files wrapped in `ancient("filename")` is ignored
- Files wrapped in `protected("filename")` are not deleted by Snakemake
- A file wrapped in `temp("filename")` is deleted after rule is finished
- `touch("filename")` creates an empty file with that name as output

A Snakefile can be treated almost like a Python script:

Example of Python code running in Snakemake

```
1 import uproot
2 import pandas
3 import numpy as np
4
5 def say_hello(name):
6     print(f"Hello {name}!")
7
8 rule somerule:
9     input: files = [f"dataset_{num}.root" for num in range(100)]
10    run:
11        say_hello("E5")
12        for tfile in input.files:
13            ds = uproot.open(tfile)["DecayTree"]
14            data = ds.arrays("B_P[XY]", outputtype=pandas.DataFrame)
15            print(np.sqrt(data.B_PX**2 + data.B_PY**2))
```

It is possible to use functions to provide inputs to a rule:

Example with function as rule input

```
1  def get_files(wildcards):
2      return #[ A list of files according to wildcards]
3
4  rule arule:
5      input: get_files
```

Note: functions can only be used to specify the inputs, not other fields.

Variables can be specified by including a config `.json` file in the Snakefile.

config.json

```
1 {  
2   "param_a" : "362",  
3   "param_b" : "cat"  
4 }
```

Snakefile

```
1 configfile: "config.json"  
2  
3 param_a = config["param_a"]  
4 param_b = config["param_b"]
```

This can be really handy for modularising your pipeline and can make it much clearer.

Single rules (or the whole Snakefile) can be configured to run in an arbitrary virtual environment

Example with singularity

```
1 rule envrule:
2     input: "file.txt"
3     output: "outfile.txt"
4     singularity: "/path/to/singularity/container.simg"
5     shell: "SomeShellCommand"
```

This is limited to `shell` and `script` execution

When calling snakemake, singularity needs to be activated:

```
$ snakemake envrule --use-singularity --singularity-args
    "--bind /run,/ceph,/net"
```

Binding `/run` is obligatory, the rest is optional

When `singularity: ...` is defined outside of a rule it is implied for all rules

Example of ambiguous wildcard deduction

```
1 rule somerule:  
2     output: "afile_{year}_{polarity}.root"  
3     shell: "echo Running rule"  
4  
5 rule requester:  
6     input: "afile_2017_MagnetUp_garbage.root"
```

This is valid code: rule requester is calling somerule with (for example) `year="2017_MagnetUp"` and `polarity="garbage"`

Example of ambiguous wildcard deduction

```
1 rule somerule:
2     output: "afile_{year}_{polarity}.root"
3     shell: "echo Running rule"
4
5 rule requester:
6     input: "afile_2017_MagnetUp_garbage.root"
```

This is valid code: rule requester is calling somerule with (for example) `year="2017_MagnetUp"` and `polarity="garbage"`




This will eventually lead to an error → define what wildcard values are allowed

Constrain your wildcards!

```
1 wildcard_constraints:
2     year="201[5678]",
3     polarity="Magnet(Up|Down)"
```


Conclusions

Hopefully you now know what Snakemake is and how to use it!

- You don't have to use Snakemake, but doing so could:
 - Greatly simplify the workflow of your project 
 - Speed up both running and debugging of your code 
 - Make your results reproducible 
- If you have any questions feel free to get in touch (james.gooding@tu-dortmund.de, [@jgooding](https://www.mattermost.com/user/jgooding) on our Mattermost)!

Any questions? 

Slides adapted from previous Programmierkurs slides of Vukan Jevtić and Louis Gerken.