snakemake

ORIAN LOUANT

University of Liège

EuroCC National Competence Center for Belgium
LUMI User Support Team

Snakemake is a workflow management system to create reproducible and scalable data analyses

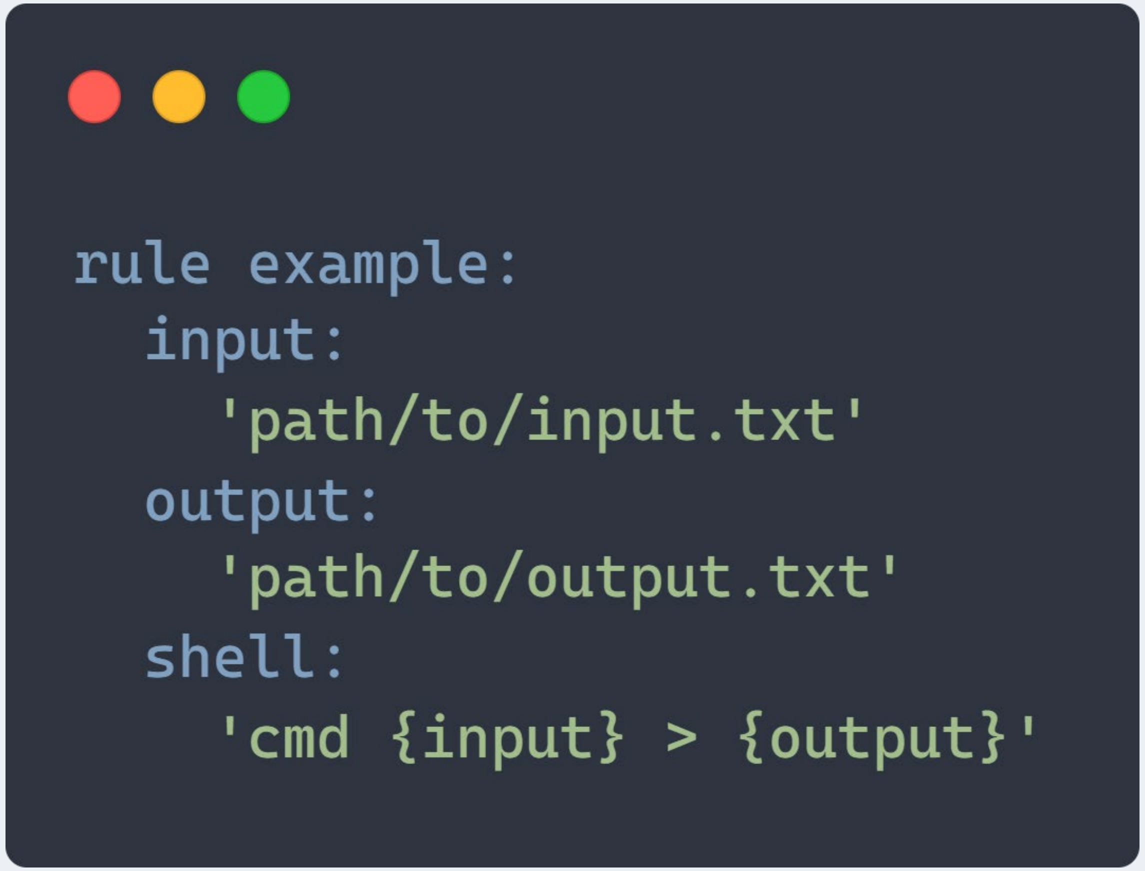
A Snakemake workflow is

- defined in terms of rules that represent the different steps of your data analysis. These rules are described via a human readable, Python based language
- can entail a description of required software with an integration with the Conda package manager and container virtualization
- independent of the available resources and computing platform

Workflow definition

Snakemake Rules

- Rules describe how to get output files from input files
- A shell command or a script to generate the output from the input
- By default, Snakemake will look for rules in a file named Snakefile
- Dependencies between the rules are determined automatically, creating a DAG (directed acyclic graph) of jobs



```
rule example:
  input:
    'path/to/input.txt'
  output:
    'path/to/output.txt'
  shell:
    'cmd {input} > {output}'
```

Demo time

binder

https://mybinder.org/v2/gh/olouant/enccb-workflow-snakemake/master

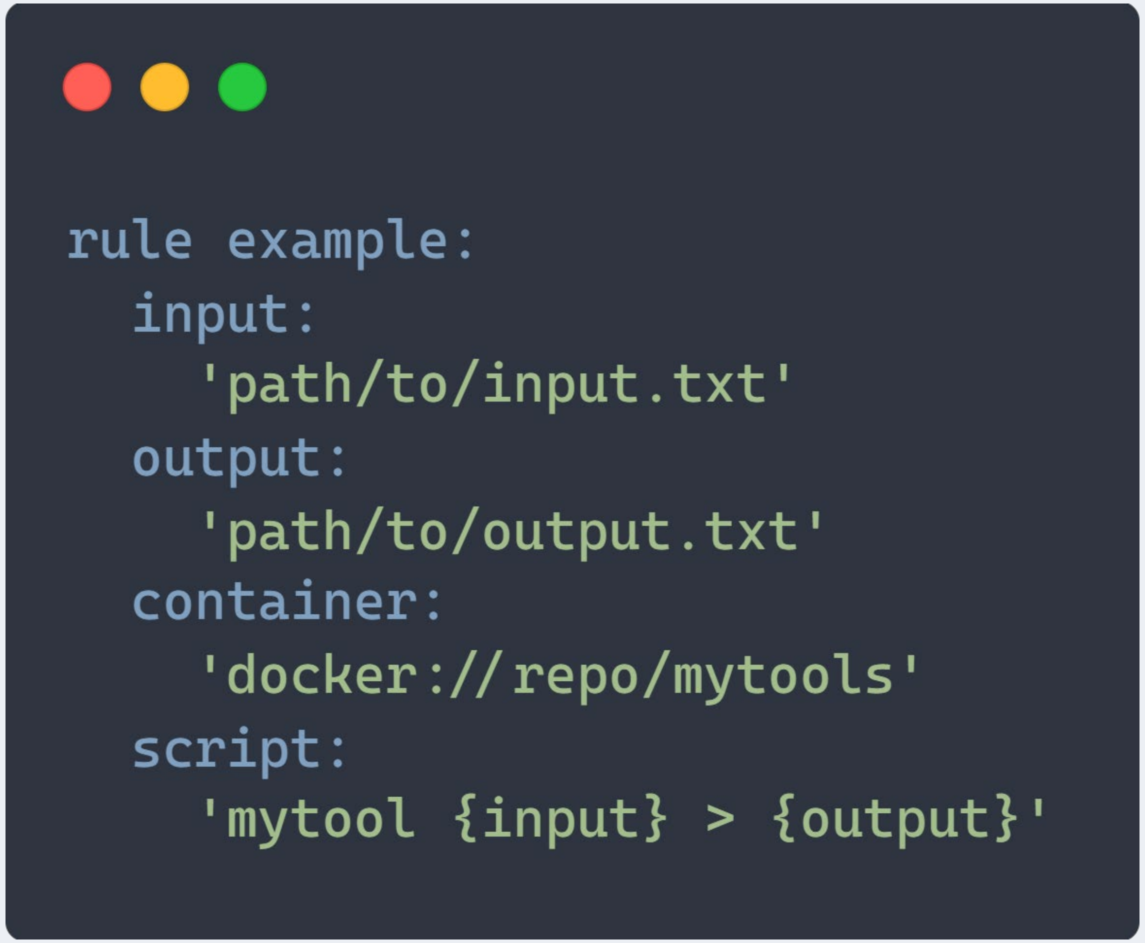
GitHub

https://github.com/olouant/enccb-workflow-snakemake

Reproducibility

Using Containers

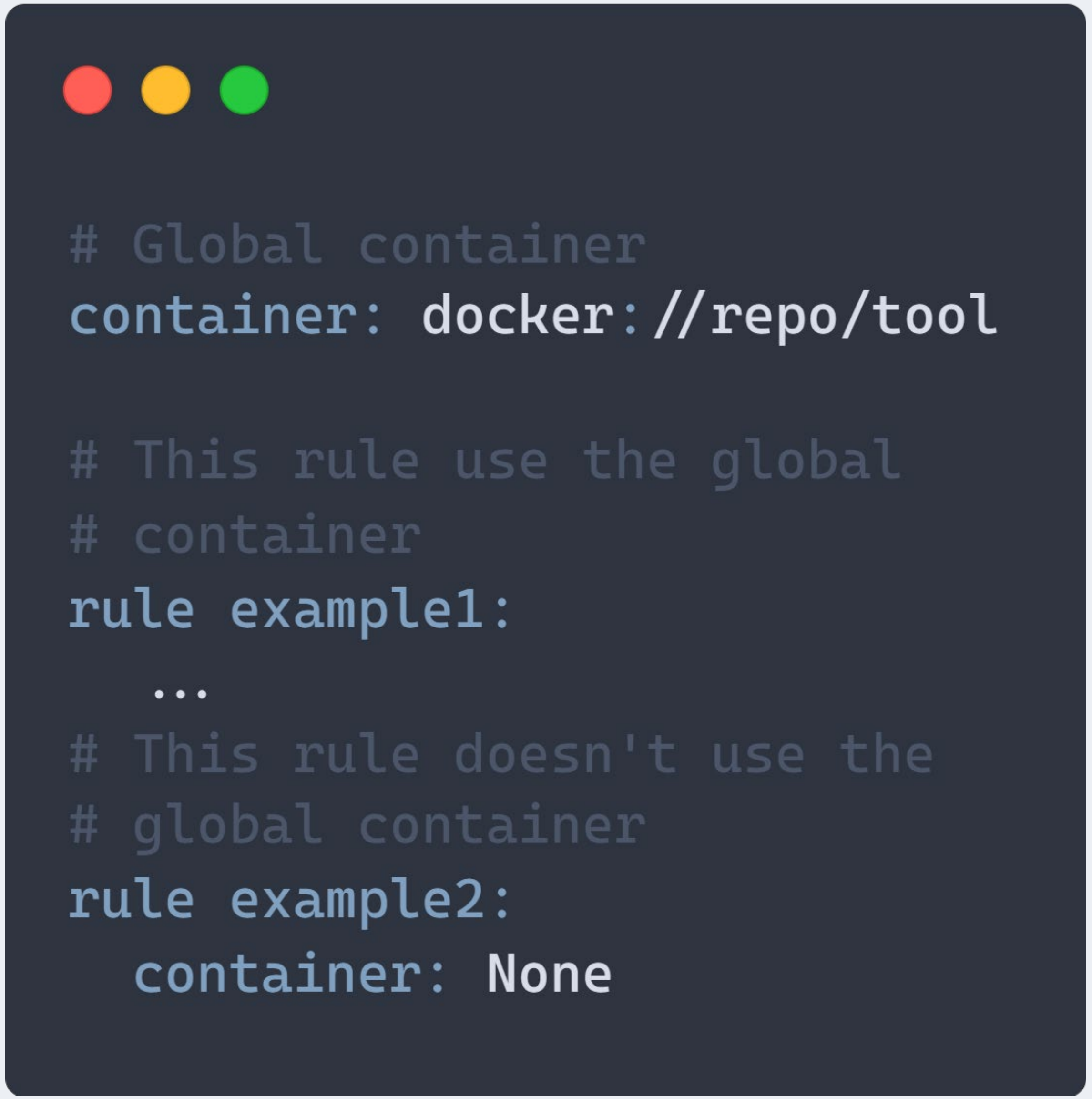
- For each rules in your workflow, you can specify a container to use with the **`container`** directive
- To run the rules that define a container within singularity, use the **`--use-singularity`** option
- You can use URLs starting with **`docker://`** or **`shub://`**



```
rule example:
  input:
    'path/to/input.txt'
  output:
    'path/to/output.txt'
  container:
    'docker://repo/mytools'
  script:
    'mytool {input} > {output}'
```

Using Containers

- The **container** directive can be used to define a global container
- When you define a global container, all the jobs will use this container
- You can disable the use of the global container for certain rules by setting value for the **container** directive to **None**



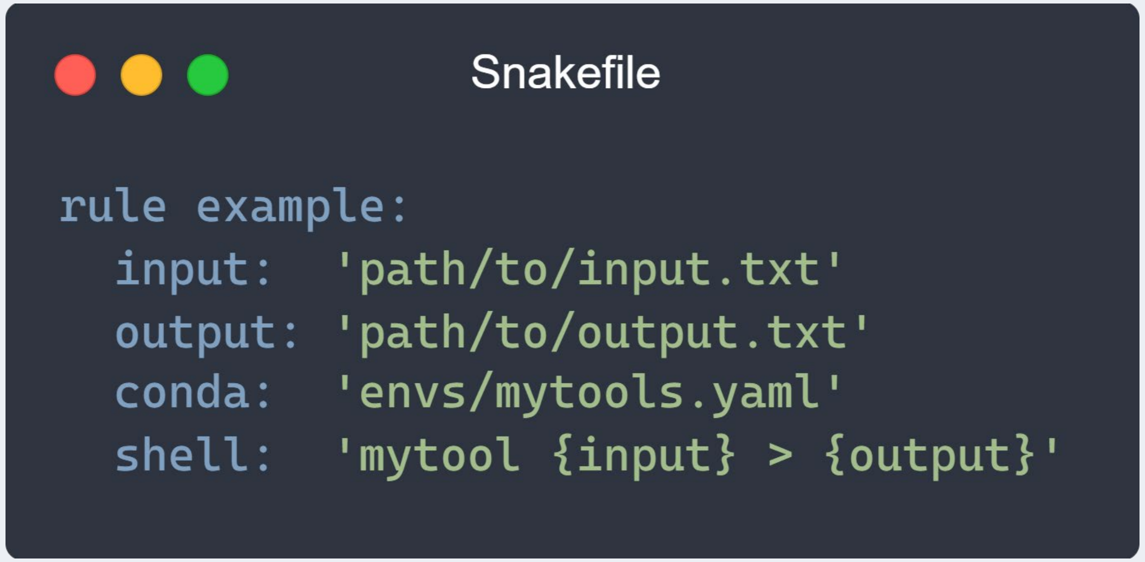
```
# Global container
container: docker://repo/tool

# This rule use the global
# container
rule example1:
  ...
# This rule doesn't use the
# global container
rule example2:
  container: None
```

Define a Conda Environment

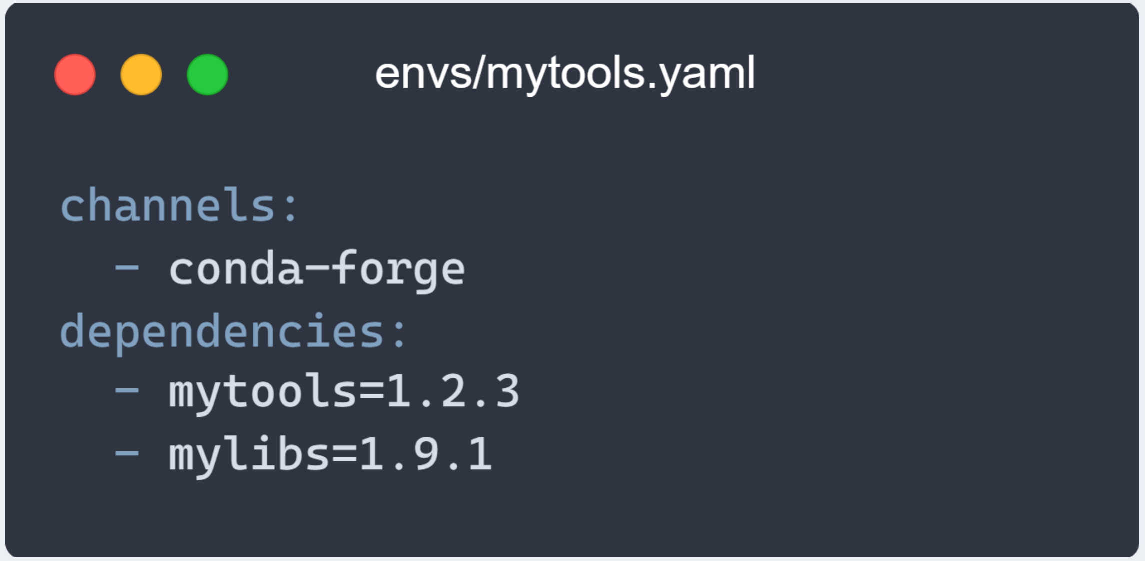
- You can define an isolated software environment per rule using the **conda** directive that takes as argument a YAML file describing the Conda packages to use

- To use the Conda integration add the **--use-conda** option when launching Snakemake



Snakefile

```
rule example:
  input:   'path/to/input.txt'
  output:  'path/to/output.txt'
  conda:   'envs/mytools.yaml'
  shell:   'mytool {input} > {output}'
```

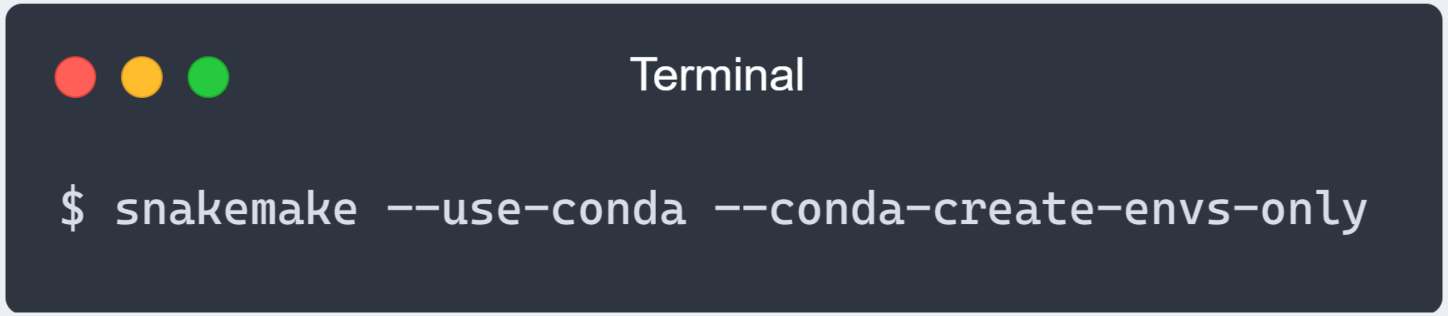


envs/mytools.yaml

```
channels:
  - conda-forge
dependencies:
  - mytools=1.2.3
  - mylibs=1.9.1
```

Create the Environment Before Running the Workflow

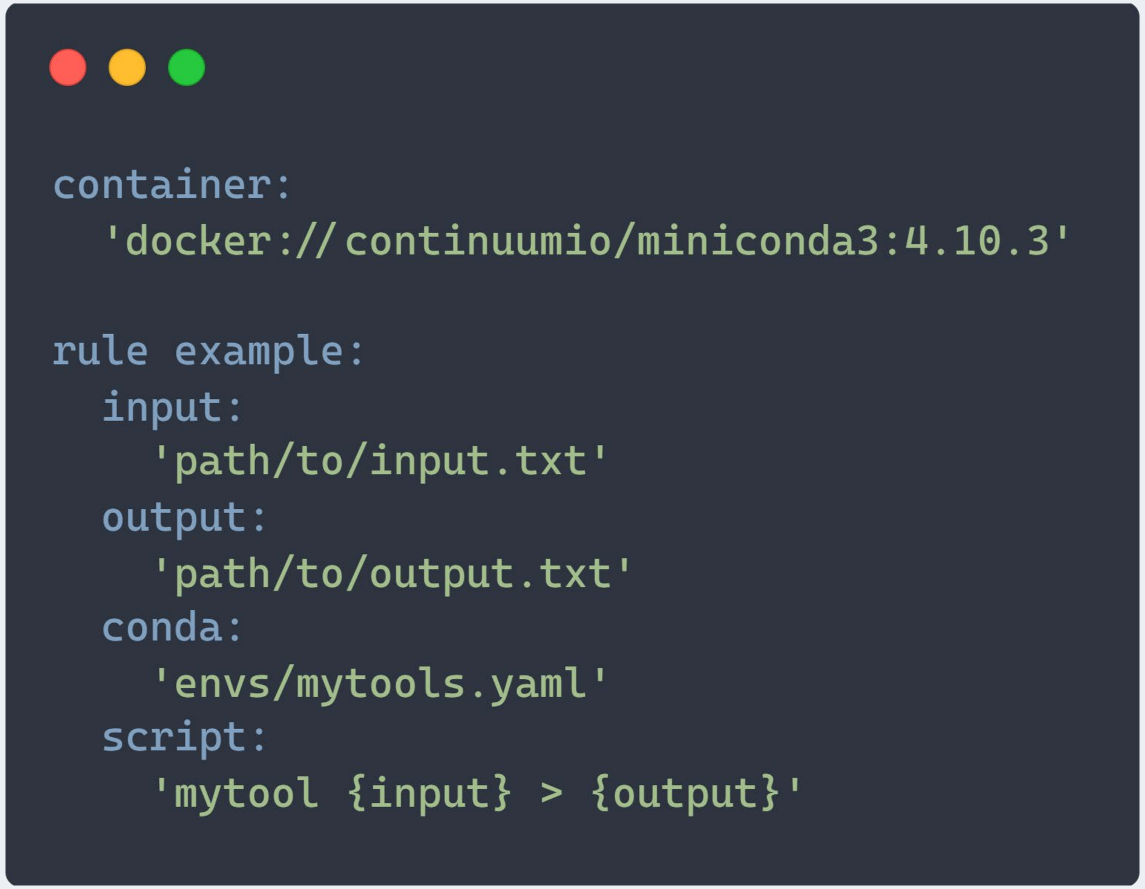
- In some cases, when running in a HPC environment the compute nodes do not have internet access and the creation of the Conda environment will fail
- Solution is to create the Conda environment on the login node but not run the full workflow



```
$ snakemake --use-conda --conda-create-envs-only
```

Combining Conda and Container

- Snakemake allows you to combine the definition of a Conda environment with running jobs in containers
- In that case you need to invoke Snakemake with both the **`--use-conda`** and **`--use-singularity`** options

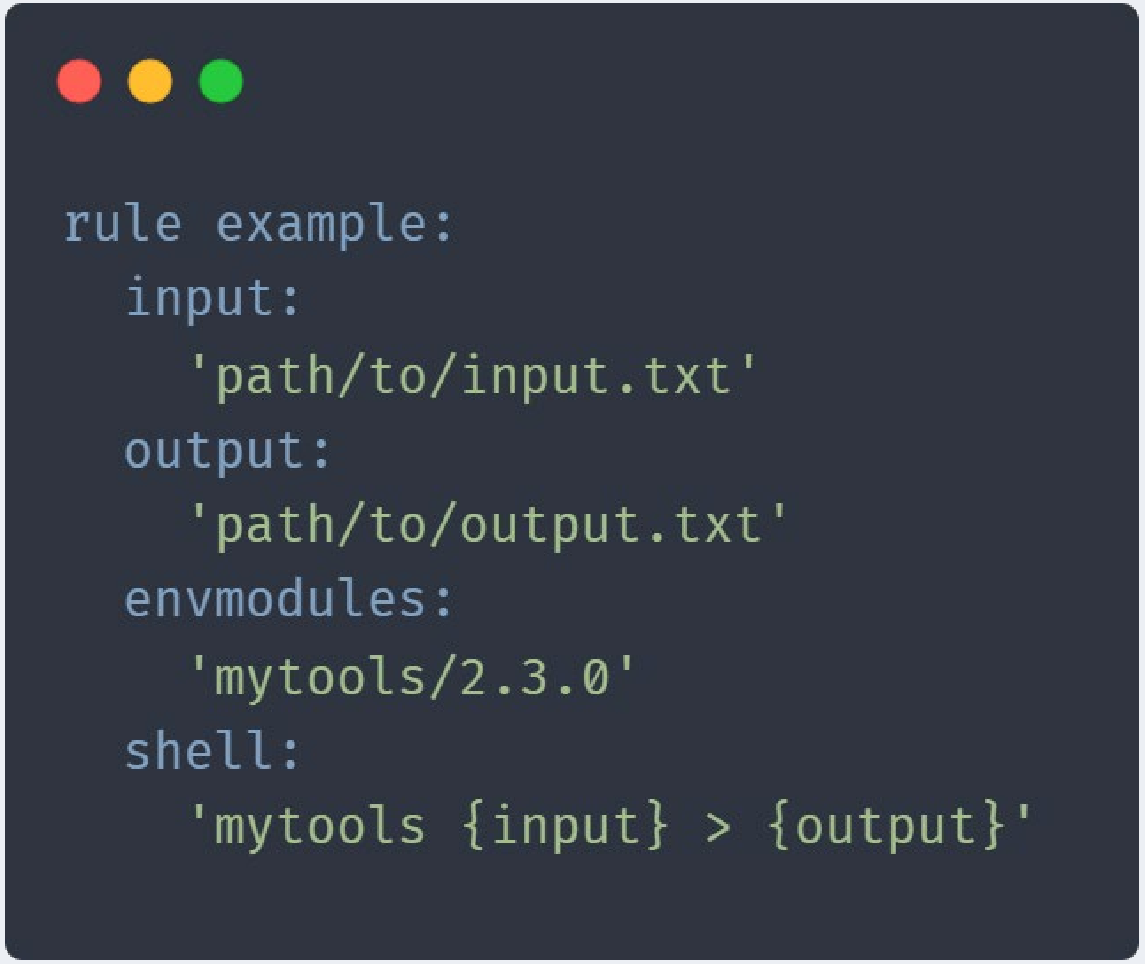


```
container:
  'docker://continuumio/miniconda3:4.10.3'

rule example:
  input:
    'path/to/input.txt'
  output:
    'path/to/output.txt'
  conda:
    'envs/mytools.yaml'
  script:
    'mytool {input} > {output}'
```

Using Environment Modules

- Snakemake allows to define environment modules per rule using the **`envsmodules`** directive to provide a list of modules that should be loaded in the environment
- To use environment modules, add the **`--use-envmodules`** option when launching Snakemake



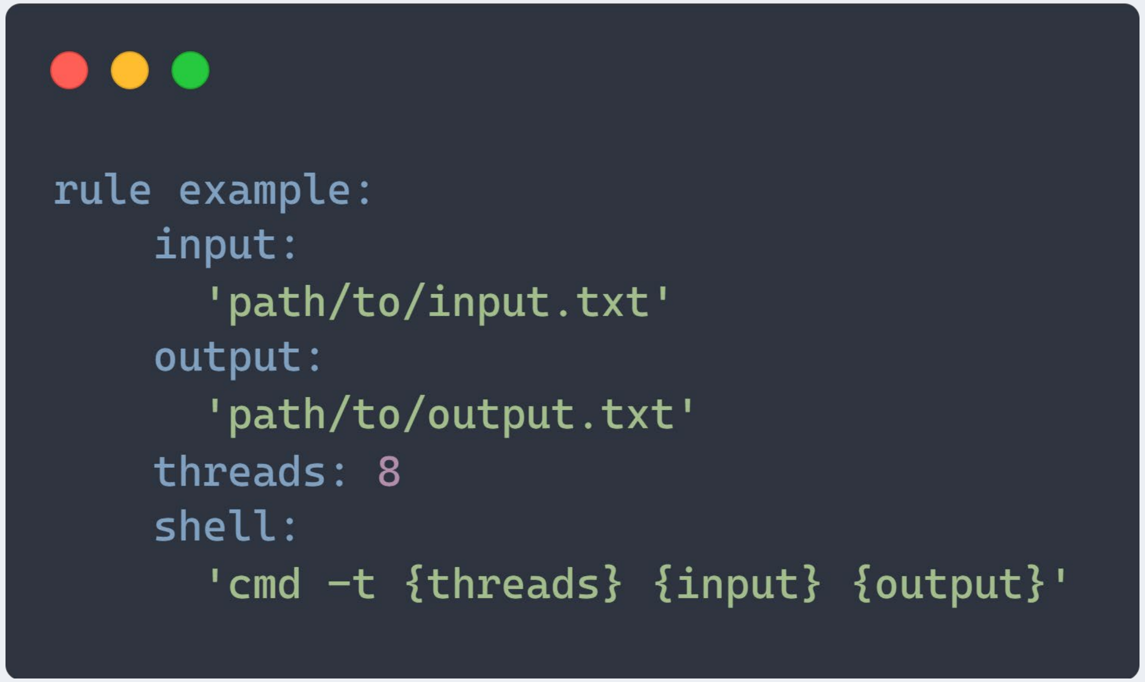
```
rule example:
  input:
    'path/to/input.txt'
  output:
    'path/to/output.txt'
  envmodules:
    'mytools/2.3.0'
  shell:
    'mytools {input} > {output}'
```

Scalability

Setting the Number of Threads

- You can specify the number of threads to use for a specific rule with the **`threads`** directive

- Snakemake will set common environment variables to the value given of the **`threads`** directive:
  - `OMP_NUM_THREADS`
  - `OPENBLAS_NUM_THREADS`
  - `MKL_NUM_THREADS`



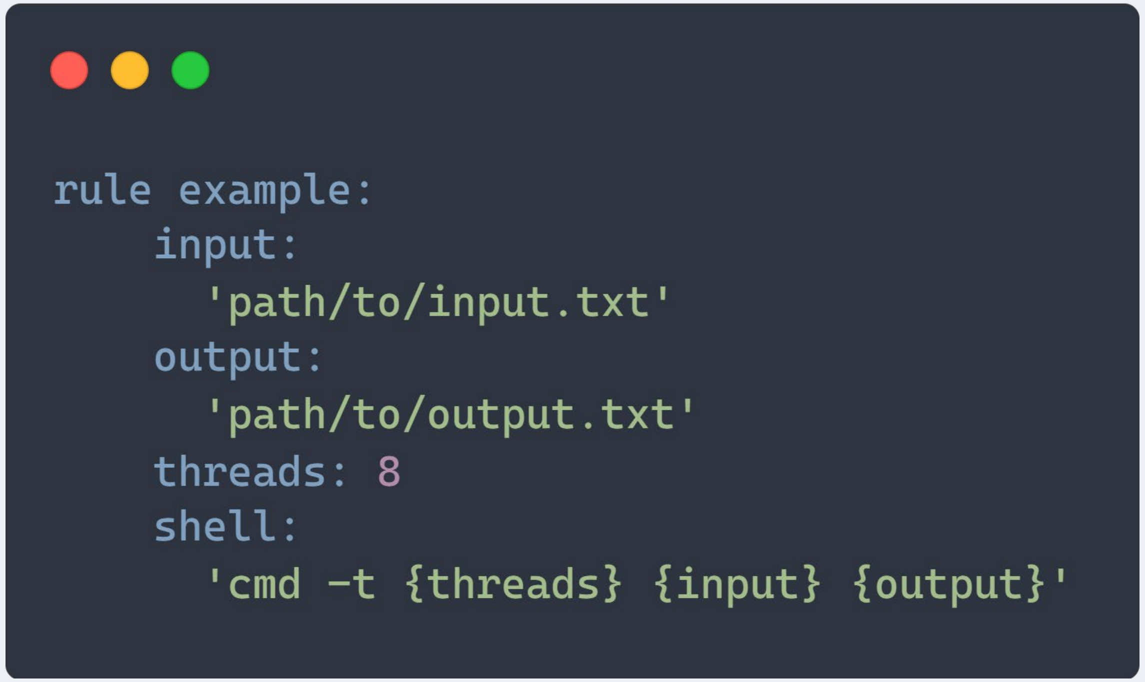
```
rule example:
    input:
      'path/to/input.txt'
    output:
      'path/to/output.txt'
    threads: 8
    shell:
      'cmd -t {threads} {input} {output}'
```

Setting the Number of Threads

- Specified threads must be seen as a maximum
- When Snakemake is executed with fewer cores, the number of threads will be adjusted:

```
threads = min(threads, cores)
```

with `cores`, the number of cores specified at the command line (`--cores` option)

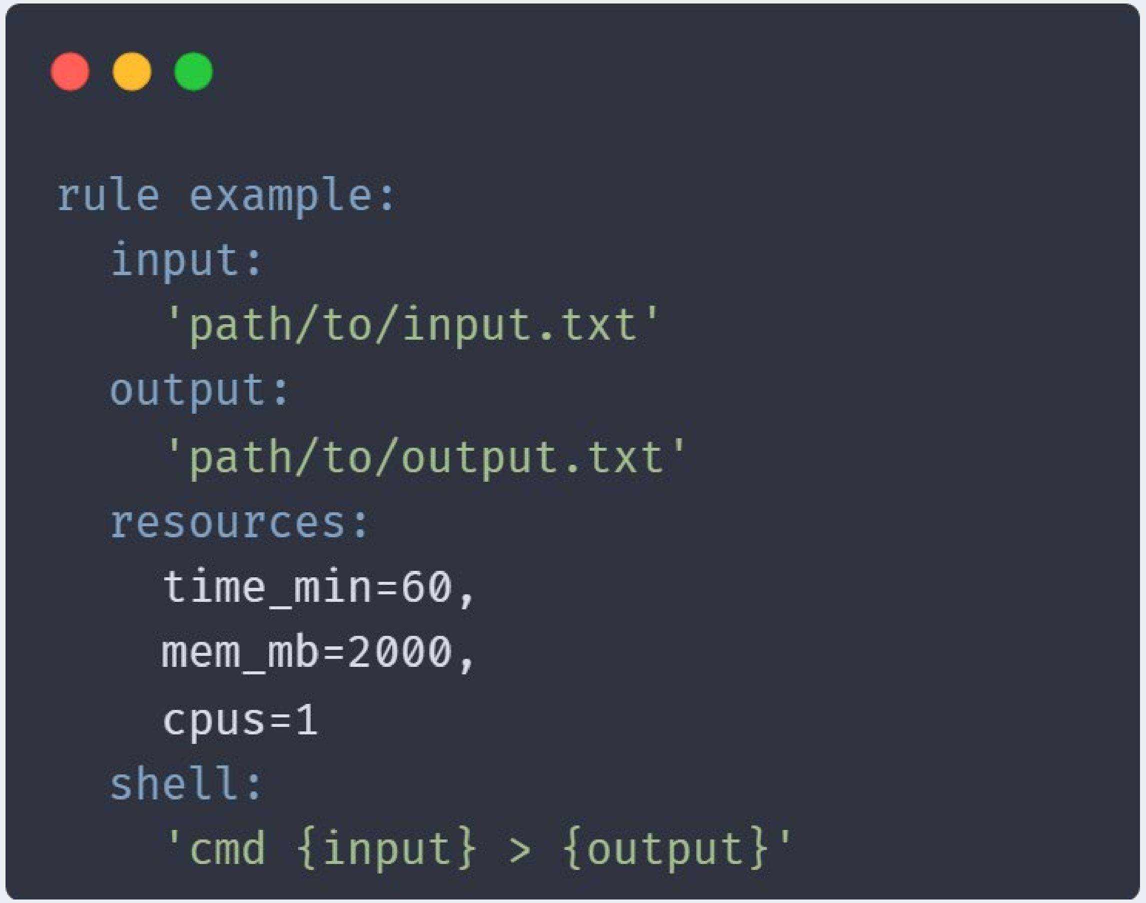


```
rule example:
    input:
      'path/to/input.txt'
    output:
      'path/to/output.txt'
    threads: 8
    shell:
      'cmd -t {threads} {input} {output}'
```

Managing Resources

- Running on in a cluster environment may require resources definition that are defined using the **`resources`** directive

- There are 3 standard resources used by Snakemake:
  - **`mem_mb`**: memory usage
  - **`disk_mb`**: disk usage
  - **`tmpdir`**: temporary directory



```
rule example:
  input:
    'path/to/input.txt'
  output:
    'path/to/output.txt'
  resources:
    time_min=60,
    mem_mb=2000,
    cpus=1
  shell:
    'cmd {input} > {output}'
```

- Resources that shall constrain the scheduling are specified using the **`--resources`** command line option

Snakemake Scheduling

- Available jobs are scheduled to maximize parallelization while satisfying resource constraints

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

```
$ snakemake --cores 2


$ snakemake --cores 8


$ snakemake --cores 8 --resources mem_mb=100
```

1 **sort** job using 2 threads

2 **sort** job using 4 threads each

1 **sort** job using 4 threads

Managing Resources and Cluster Execution

- Resources for rules that do not define explicitly resources can be provided through the command line option `--default-resources`

```
$ snakemake --default-resources time_min=600 mem_mb=2000 cpus=1 ...
```

- Resources defined within the rules can be overwitted using the command line option `--set-resources`

Running on an HPC cluster

- Snakemake can compile jobs into scripts that will run on the compute node of and HPC cluster
- The jobs are submitted to the cluster via a submission command that is provided by passing the `--cluster` option to the command line
- The maximum number of jobs to be submitted at once is set using the `--jobs` option

Slurm

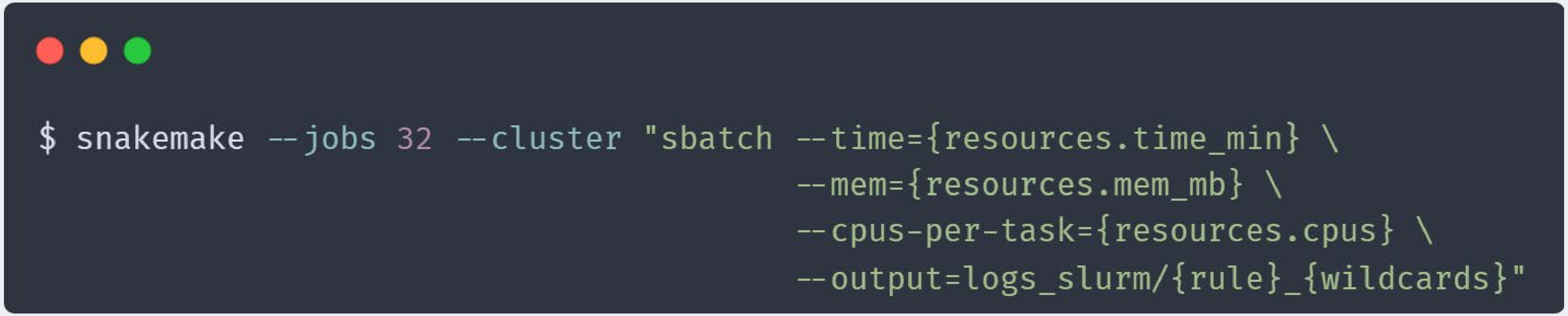
```
$ snakemake --jobs 32 --cluster "sbatch"
```

PBS

```
$ snakemake --jobs 32 --cluster "qsub"
```

Managing Resources and Cluster Execution

- The submit command can be decorated to make it aware of certain job properties: (name, rule name, input, output, params, wildcards, log, threads...)



```
$ snakemake --jobs 32 --cluster "sbatch --time={resources.time_min} \
                                       --mem={resources.mem_mb} \
                                       --cpus-per-task={resources.cpus} \
                                       --output=logs_slurm/{rule}_{wildcards}"
```

Cluster Execution: be nice to the scheduler...

... and your fellow users

- If too many requests are made at once to the job scheduler, the performance will suffer for all users

- If the rules in your Snakemake jobs take minutes to complete, it's overkill to check their status every second. In that case, it may be desirable to set the following option:

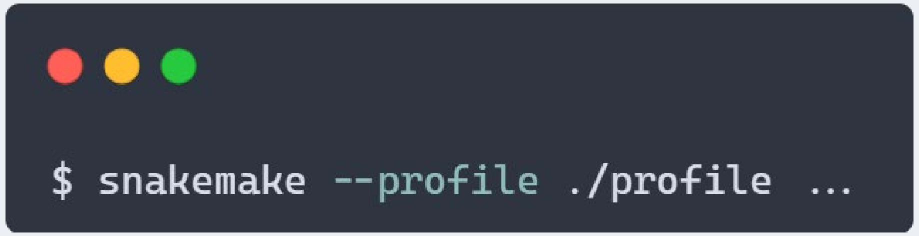
  - `--max-jobs-per-second`
  - `--max-status-checks-per-second`

Using a Configuration File

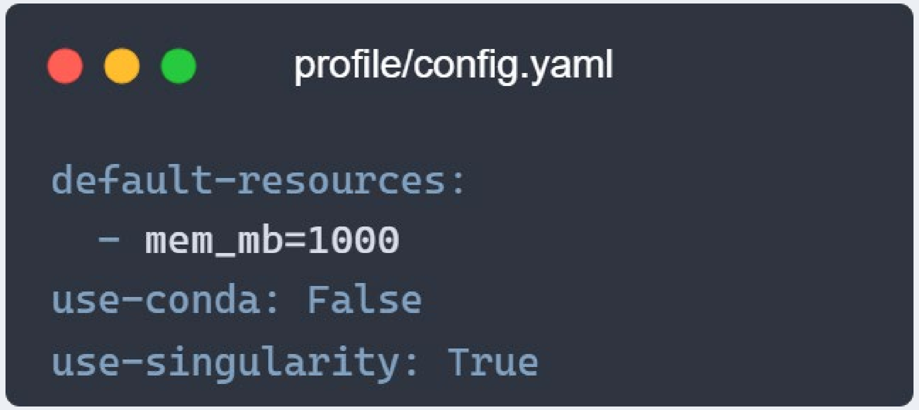
- When running workflows on a regular basis, it might be tedious to provide all the required flags every time
- It is possible to specify a configuration profile to specify the default options using the `--profile profile_name` command line option
- Snakemake will search for a folder named `profile_name` in the user and global configuration directories (`~/.config/snakemake`). You can also provide an absolute or relative path to a directory
- In the folder, Snakemake expect to find a file named `config.yaml`

Using a Configuration File

- The profile can be used to set a default for each option of the Snakemake command line
- In a profile, command-line option `--someoption` becomes `someoption`



```
$ snakemake --profile ./profile ...
```



profile/config.yaml

```yaml
default-resources:
  - mem_mb=1000
use-conda: False
use-singularity: True
```

See also: https://github.com/snakemake-profiles/doc

Using a Custom Job Script

- You can provide a custom job script for submission to a cluster using the `--jobscript` command line option or use a profile
- This allows you to perform additional operations in your job script
- For example, it can be used to add additionnal binding when using singularity

profile/config.yaml

```
cluster:
  mkdir -p logs/{rule} &&
  sbatch
    --cpus-per-task={threads}
    --job-name={rule}-{wildcards}
    --output=logs/{rule}/{rule}-{wildcards}-%j.out
jobs: 10
jobscript: job_script.sh
use-conda: False
use-singularity: True
```

profile/job_script.sh

```
#!/bin/bash
# properties = {properties}

export SINGULARITY_BIND=$SINGULARITY_BIND,$TMPDIR

{exec_job}
```

Wrapping up

- Snakemake is a workflow manager offering a simple Python-like syntax with high readability and flexibility
- Snakemake allows for easy mixing of shell commands and high-level language scripts (Python, Julia, R, Rust)
- Reproducibility and dependencies management of the workflow can be achieved through the integration with the Conda package manager and containers
- Snakemake provide mechanisms to manage resources and cluster execution

Get More Information About Snakemake

- Website: https://snakemake.github.io/
- Documentation: https://snakemake.readthedocs.io/en/stable/

The EuroCC project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union’s Horizon 2020 research and innovation program and Germany, Bulgaria, Austria, Croatia, Cyprus, the Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, the United Kingdom, France, the Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro.