Snakemake Documentation

Release 7.22.0

Johannes Koester

Feb 13, 2023

GETTING STARTED

The Snakemake workflow management system is a tool to create **reproducible and scalable** data analyses. Workflows are described via a human readable, Python based language. They can be seamlessly scaled to server, cluster, grid and cloud environments, without the need to modify the workflow definition. Finally, Snakemake workflows can entail a description of required software, which will be automatically deployed to any execution environment.

Snakemake is **highly popular**, with >7 new citations per week. For an introduction, please visit https://snakemake.github.io.

CHAPTER

ONE

GETTING STARTED

- To get a first impression, please visit https://snakemake.github.io.
- To properly understand what Snakemake can do for you please read our "rolling" paper.
- News about Snakemake are published via Twitter.
- To learn Snakemake, please do the *Snakemake Tutorial*, and see the *FAQ*.
- **Best practices** for writing Snakemake workflows can be found *here*.
- For more advanced usage on various platforms, see the *Snakemake Executor Tutorials*.

CHAPTER

TWO

SUPPORT

- For releases, see *Changelog*.
- Check *frequently asked questions (FAQ)*.
- In case of **questions**, please post on stack overflow.
- To **discuss** with other Snakemake users, use the discord server. **Please do not post questions there. Use stack overflow for questions.**
- For **bugs and feature requests**, please use the issue tracker.
- For **contributions**, visit Snakemake on Github and read the *guidelines*.

CHAPTER

THREE

CITATION

When using Snakemake, please cite our "rolling" paper

Mölder, F., Jablonski, K.P., Letcher, B., Hall, M.B., Tomkins-Tinch, C.H., Sochat, V., Forster, J., Lee, S., Twardziok, S.O., Kanitz, A., Wilm, A., Holtgrewe, M., Rahmann, S., Nahnsen, S., Köster, J., 2021. Sustainable data analysis with Snakemake. F1000Res 10, 33.

This paper will also be regularly updated when Snakemake receives new features. See *Citations* for more information.

CHAPTER

FOUR

RESOURCES

Snakemake Wrappers Repository
: The Snakemake Wrapper Repository is a collection of reusable wrappers that allow to quickly use popular tools from Snakemake rules and workflows.

Snakemake Workflow Catalog
: An automatically scraped catalog of publicly available Snakemake workflows for any kind of data analysis.

Snakemake Workflows Project
: This project provides a collection of high quality modularized and re-usable workflows. The provided code should also serve as a best-practices of how to build production ready workflows with Snakemake. Everybody is invited to contribute.

Snakemake Profiles Project
: This project provides Snakemake configuration profiles for various execution environments. Please consider contributing your own if it is still missing.

Snakemake API documentation
: The documentation of the Snakemake API for programmatic access and developement on Snakemake.

Conda-Forge
: Conda-Forge is a community driven distribution of Conda packages that can be used from Snakemake for creating completely reproducible workflows by defining the used software versions and providing binaries.

Bioconda
: Bioconda, a partner project of conda-forge, is a community driven distribution of bioinformatics-related Conda packages that can be used from Snakemake for creating completely reproducible workflows by defining the used software versions and providing binaries.

4.1 Installation

Snakemake is available on PyPi as well as through Bioconda and also from source code. You can use one of the following ways for installing Snakemake.

4.1.1 Installation via Conda/Mamba

This is the **recommended** way to install Snakemake, because it also enables Snakemake to *handle software dependencies of your workflow*.

First, you need to install a Conda-based Python3 distribution. The recommended choice is Mambaforge which not only provides the required Python and Conda commands, but also includes Mamba an extremely fast and robust replacement for the Conda package manager which is highly recommended. The default conda solver is a bit slow and sometimes has issues with selecting the latest package releases. Therefore, we recommend to in any case use Mamba.

In case you don't use Mambaforge you can always install Mamba into any other Conda-based Python distribution with

```
$ conda install -n base -c conda-forge mamba
```

Full installation

Snakemake can be installed with all goodies needed to run in any environment and for creating interactive reports via

```
$ conda activate base
$ mamba create -c conda-forge -c bioconda -n snakemake snakemake
```

from the Bioconda channel. This will install snakemake into an isolated software environment, that has to be activated with

```
$ conda activate snakemake
$ snakemake --help
```

Installing into isolated environments is best practice in order to avoid side effects with other packages.

Note that full installation is not possible from **Windows**, because some of the dependencies are Unix (Linux/MacOS) only. For Windows, please use the minimal installation below.

Minimal installation

A minimal version of Snakemake which only depends on the bare necessities can be installed with

```
$ conda activate base
$ mamba create -c bioconda -c conda-forge -n snakemake snakemake-minimal
```

In contrast to the full installation, which depends on some Unix (Linux/MacOS) only packages, this also works on Windows.

Notes on Bioconda as a package source

Note that Snakemake is available via Bioconda for historical, reproducibility, and continuity reasons (although it is not limited to biology applications at all). However, it is easy to combine Snakemake installation with other channels, e.g., by prefixing the package name with `::bioconda`, i.e.,

```
$ conda activate base
$ mamba create -n some-env -c conda-forge bioconda::snakemake bioconda::snakemake-
↪minimal ...
```

4.1.2 Installation via pip

Instead of conda, snakemake can be installed with pip. However, note that snakemake has non-python dependencies, such that the pip based installation has a limited functionality if those dependencies are not manually installed in addition.

A list of Snakemake's dependencies can be found within its meta.yaml conda recipe.

4.1.3 Installation of a development version via pip

If you want to quickly try out an unreleased version from the snakemake repository (which you cannot get via bioconda, yet), for example to check whether a bug fix works for you workflow, you can get the current state of the main branch with:

```
$ mamba create --only-deps -n snakemake-main snakemake
$ conda activate snakemake-main
$ pip install git+https://github.com/snakemake/snakemake
```

You can also install the current state of another branch or the repository state at a particular commit. For information on the syntax for this, see the pip documentation on git support.

4.2 Snakemake Tutorial

This tutorial introduces the text-based workflow system Snakemake. Snakemake follows the GNU Make paradigm: workflows are defined in terms of rules that define how to create output files from input files. Dependencies between the rules are determined automatically, creating a DAG (directed acyclic graph) of jobs that can be automatically parallelized.

Snakemake sets itself apart from other text-based workflow systems in the following way. Hooking into the Python interpreter, Snakemake offers a definition language that is an extension of Python with syntax to define rules and workflow specific properties. This allows Snakemake to combine the flexibility of a plain scripting language with a pythonic workflow definition. The Python language is known to be concise yet readable and can appear almost like pseudo-code. The syntactic extensions provided by Snakemake maintain this property for the definition of the workflow. Further, Snakemake's scheduling algorithm can be constrained by priorities, provided cores and customizable resources and it provides a generic support for distributed computing (e.g., cluster or batch systems). Hence, a Snakemake workflow scales without modification from single core workstations and multi-core servers to cluster or batch systems. Finally, Snakemake integrates with the package manager Conda and the container engine Singularity such that defining the software stack becomes part of the workflow itself.

The examples presented in this tutorial come from Bioinformatics. However, Snakemake is a general-purpose workflow management system for any discipline. We ensured that no bioinformatics knowledge is needed to understand the tutorial.

Also have a look at the corresponding slides.

4.2.1 Setup

Requirements

To go through this tutorial, you need the following software installed:

- Python ≥3.5
- Snakemake ≥5.24.1
- BWA 0.7

- SAMtools 1.9
- Pysam 0.15
- BCFtools 1.9
- Graphviz 2.42
- Jinja2 2.11
- NetworkX 2.5
- Matplotlib 3.3

However, don't install any of these this manually now, we guide you through better ways below.

Run tutorial for free in the cloud via Gitpod

Note

A common thing to happen while using the development environment in GitPod is to hit `Ctrl-s` while in the terminal window, because you wanted to save a file in the editor window. This will freeze up you terminal. To get it back, make sure you selected the terminal window by clicking on it and then hit `Ctrl-q`.

The easiest way to run this tutorial is to use Gitpod, which enables performing the excercises via your browser—including all required software, for free and in the cloud. In order to do this, simply open the predefined snakemake-tutorial GitPod workspace in your browser. GitPod provides you with a Theia development environment, which you can learn about in the linked documentation. Once you have a basic understanding of this environment, you can go on directy with *Basics: An example workflow*.

Running the tutorial on your local machine

If you prefer to run the tutorial on your local machine, please follow the steps below.

The easiest way to set these prerequisites up, is to use the Mambaforge Python 3 distribution (Mambaforge is a Conda based distribution like Miniconda, which however uses Mamba a fast and more robust replacement for the Conda package manager). The tutorial assumes that you are using either Linux or MacOS X. Both Snakemake and Mambaforge work also under Windows, but the Windows shell is too different to be able to provide generic examples.

Setup on Windows

If you already use Linux or MacOS X, go on with **Step 1**.

Windows Subsystem for Linux

If you use Windows 10, you can set up the Windows Subsystem for Linux (WSL) to natively run linux applications. Install the WSL following the instructions in the WSL Documentation. You can chose any Linux distribution available for the WSL, but the most popular and accessible one is Ubuntu. Start the WSL and set up your account; now, you can follow the steps of our tutorial from within your Linux environment in the WSL.

Vagrant virtual machine

If you are using a version of Windows older than 10 or if you do not wish to install the WSL, you can instead setup a Linux virtual machine (VM) with Vagrant. First, install Vagrant following the installation instructions in the Vagrant Documentation. Then, create a new directory you want to share with your Linux VM, for example, create a folder named `vagrant-linux` somewhere. Open a command line prompt, and change into that directory. Here, you create a 64-bit Ubuntu Linux environment with

```
> vagrant init hashicorp/precise64
> vagrant up
```

If you decide to use a 32-bit image, you will need to download the 32-bit version of Miniconda in the next step. The contents of the `vagrant-linux` folder will be shared with the virtual machine that is set up by vagrant. You can log into the virtual machine via

```
> vagrant ssh
```

If this command tells you to install an SSH client, you can follow the instructions in this Blogpost. Now, you can follow the steps of our tutorial from within your Linux VM.

Step 1: Installing Mambaforge

First, please **open a terminal** or make sure you are logged into your Vagrant Linux VM. Assuming that you have a 64-bit system, on Linux, download and install Miniconda 3 with

```
$ curl -L https://github.com/conda-forge/miniforge/releases/latest/download/
↪Mambaforge-Linux-x86_64.sh -o Mambaforge-Linux-x86_64.sh
$ bash Mambaforge-Linux-x86_64.sh
```

On MacOS with x86_64 architecture, download and install with

```
$ curl -L https://github.com/conda-forge/miniforge/releases/latest/download/
↪Mambaforge-MacOSX-x86_64.sh -o Mambaforge-MacOSX-x86_64.sh
$ bash Mambaforge-MacOSX-x86_64.sh
```

On MacOS with ARM/M1 architecture, download and install with

```
$ curl -L https://github.com/conda-forge/miniforge/releases/latest/download/
↪Mambaforge-MacOSX-arm64.sh -o Mambaforge-MacOSX-arm64.sh
$ bash Mambaforge-MacOSX-arm64.sh
```

When you are asked the question

```
Do you wish the installer to prepend the install location to PATH ...? [yes|no]
```

answer with **yes**. Along with a minimal Python 3 environment, Mambaforge contains the package manager Mamba. After closing your current terminal and opening a **new terminal**, you can use the new `conda` command to install software packages and create isolated environments to, for example, use different versions of the same package. We will later use Conda to create an isolated environment with all the required software for this tutorial.

Step 2: Preparing a working directory

First, **create a new directory** `snakemake-tutorial` at a **place you can easily remember** and change into that directory in your terminal:

```
$ mkdir snakemake-tutorial
$ cd snakemake-tutorial
```

If you use a Vagrant Linux VM from Windows as described above, create that directory under `/vagrant/`, so that the contents are shared with your host system (you can then edit all files from within Windows with an editor that supports Unix line breaks). Then, **change to the newly created directory**. In this directory, we will later create an example workflow that illustrates the Snakemake syntax and execution environment. First, we download some example data on which the workflow shall be executed:

```
$ curl -L https://github.com/snakemake/snakemake-tutorial-data/archive/v5.24.1.tar.gz␣
↪-o snakemake-tutorial-data.tar.gz
```

Next we extract the data. On Linux, run

```
$ tar --wildcards -xf snakemake-tutorial-data.tar.gz --strip 1 "*/data" "*/
↪environment.yaml"
```

On MacOS, run

```
$ tar -xf snakemake-tutorial-data.tar.gz --strip 1 "*/data" "*/environment.yaml"
```

This will create a folder `data` and a file `environment.yaml` in the working directory.

Step 3: Creating an environment with the required software

First, make sure to activate the conda base environment with

```
$ conda activate base
```

The `environment.yaml` file that you have obtained with the previous step (Step 2) can be used to install all required software into an isolated Conda environment with the name `snakemake-tutorial` via

```
$ mamba env create --name snakemake-tutorial --file environment.yaml
```

If you don't have the Mamba command because you used a different conda distribution than Mambaforge, you can also first install Mamba (which is a faster and more robust replacement for Conda) in your base environment with

```
$ conda install -n base -c conda-forge mamba
```

and then run the *mamba env create* command shown above.

Step 4: Activating the environment

To activate the `snakemake-tutorial` environment, execute

```
$ conda activate snakemake-tutorial
```

Now you can use the installed tools. Execute

```
$ snakemake --help
```

to test this and get information about the command-line interface of Snakemake. To exit the environment, you can execute

```
$ conda deactivate
```

but **don't do that now**, since we finally want to start working with Snakemake :-).

4.2.2 Basics: An example workflow

Please make sure that you have **activated** the environment we created before, and that you have an open terminal in the working directory you have created.

A Snakemake workflow is defined by specifying rules in a Snakefile. **Rules decompose the workflow into small steps** (for example, the application of a single tool) by specifying how to create sets of **output files** from sets of **input files**. Snakemake automatically **determines the dependencies** between the rules by matching file names.

The Snakemake language extends the Python language, adding syntactic structures for rule definition and additional controls. All added syntactic structures begin with a keyword followed by a code block that is either in the same line or indented and consisting of multiple lines. The resulting syntax resembles that of original Python constructs.

In the following, we will introduce the Snakemake syntax by creating an example workflow. The workflow comes from the domain of genome analysis. It maps sequencing reads to a reference genome and calls variants on the mapped reads. The tutorial does not require you to know what this is about. Nevertheless, we provide some background in the following paragraph.

Background

The genome of a living organism encodes its hereditary information. It serves as a blueprint for proteins, which form living cells, carry information and drive chemical reactions. Differences between species, populations or individuals can be reflected by differences in the genome. Certain variants can cause syndromes or predisposition for certain diseases, or cause cancerous growth in the case of tumour cells that have accumulated changes with respect to healthy cells. This makes the genome a major target of biological and medical research. Today, it is often analyzed with DNA sequencing, producing gigabytes of data from a single biological sample (for example a biopsy of some tissue). For technical reasons, DNA sequencing cuts the DNA of a sample into millions of small pieces, called **reads**. In order to recover the genome of the sample, one has to map these reads against a known **reference genome** (for example, the human one obtained during the famous human genome project). This task is called **read mapping**. Often, it is of interest where an individual genome is different from the species-wide consensus represented with the reference genome. Such differences are called **variants**. They are responsible for harmless individual differences (like eye color), but can also cause diseases like cancer. By investigating the differences between the mapped reads and the reference sequence at a particular genome position, variants can be detected. This is a statistical challenge, because they have to be distinguished from artifacts generated by the sequencing process.

Step 1: Mapping reads

Our first Snakemake rule maps reads of a given sample to a given reference genome (see *Background*). For this, we will use the tool bwa, specifically the subcommand `bwa mem`. In the working directory, **create a new file** called `Snakefile` with an editor of your choice. We propose to use the Atom editor, since it provides out-of-the-box syntax highlighting for Snakemake. In the Snakefile, define the following rule:

```
rule bwa_map:
    input:
        "data/genome.fa",
        "data/samples/A.fastq"
    output:
        "mapped_reads/A.bam"
    shell:
        "bwa mem {input} | samtools view -Sb - > {output}"
```

Note

A common error is to forget the comma between the input or output items. Since Python concatenates subsequent strings, this can lead to unexpected behavior.

A Snakemake rule has a name (here `bwa_map`) and a number of directives, here `input`, `output` and `shell`. The `input` and `output` directives are followed by lists of files that are expected to be used or created by the rule. In the simplest case, these are just explicit Python strings. The `shell` directive is followed by a Python string containing the shell command to execute. In the shell command string, we can refer to elements of the rule via braces notation (similar to the Python format function). Here, we refer to the output file by specifying `{output}` and to the input files by specifying `{input}`. Since the rule has multiple input files, Snakemake will concatenate them, separated by a whitespace. In other words, Snakemake will replace `{input}` with `data/genome.fa data/samples/A.fastq` before executing the command. The shell command invokes `bwa mem` with reference genome and reads, and pipes the output into `samtools` which creates a compressed BAM file containing the alignments. The output of `samtools` is redirected into the output file defined by the rule with >.

Note

It is best practice to have subsequent steps of a workflow in separate, unique, output folders. This keeps the working directory structured. Further, such unique prefixes allow Snakemake to quickly discard most rules in its search for rules that can provide the requested input. This accelerates the resolution of the rule dependencies in a workflow.

When a workflow is executed, Snakemake tries to generate given **target** files. Target files can be specified via the command line. By executing

```
$ snakemake -np mapped_reads/A.bam
```

in the working directory containing the Snakefile, we tell Snakemake to generate the target file `mapped_reads/A.bam`. Since we used the `-n` (or `--dry-run`) flag, Snakemake will only show the execution plan instead of actually performing the steps. The `-p` flag instructs Snakemake to also print the resulting shell command for illustration. To generate the target files, **Snakemake applies the rules given in the Snakefile in a top-down way**. The application of a rule to generate a set of output files is called **job**. For each input file of a job, Snakemake again (i.e. recursively) determines rules that can be applied to generate it. This yields a directed acyclic graph (DAG) of jobs where the edges represent dependencies. So far, we only have a single rule, and the DAG of jobs consists of a single node. Nevertheless, we can **execute our workflow** with

```
$ snakemake --cores 1 mapped_reads/A.bam
```

Whenever executing a workflow, you need to specify the number of cores to use. For this tutorial, we will use a single core for now. Later you will see how parallelization works. Note that, after completion of above command, Snakemake will not try to create `mapped_reads/A.bam` again, because it is already present in the file system. Snakemake **only re-runs jobs if one of the input files is newer than one of the output files or one of the input files will be updated by another job**.

Step 2: Generalizing the read mapping rule

Obviously, the rule will only work for a single sample with reads in the file `data/samples/A.fastq`. However, Snakemake allows **generalizing rules by using named wildcards**. Simply replace the `A` in the second input file and in the output file with the wildcard `{sample}`, leading to

```
rule bwa_map:
    input:
        "data/genome.fa",
        "data/samples/{sample}.fastq"
    output:
        "mapped_reads/{sample}.bam"
    shell:
        "bwa mem {input} | samtools view -Sb - > {output}"
```

Note

Note that if a rule has multiple output files, Snakemake requires them to all have exactly the same wildcards. Otherwise, it could happen that two jobs running the same rule in parallel want to write to the same file.

When Snakemake determines that this rule can be applied to generate a target file by replacing the wildcard `{sample}` in the output file with an appropriate value, it will propagate that value to all occurrences of `{sample}` in the input files and thereby determine the necessary input for the resulting job. Note that you can have multiple wildcards in your file paths, however, to avoid conflicts with other jobs of the same rule, **all output files** of a rule have to **contain exactly the same wildcards**.

When executing

```
$ snakemake -np mapped_reads/B.bam
```

Snakemake will determine that the rule `bwa_map` can be applied to generate the target file by replacing the wildcard `{sample}` with the value `B`. In the output of the dry-run, you will see how the wildcard value is propagated to the input files and all filenames in the shell command. You can also **specify multiple targets**, for example:

```
$ snakemake -np mapped_reads/A.bam mapped_reads/B.bam
```

Some Bash magic can make this particularly handy. For example, you can alternatively compose our multiple targets in a single pass via

```
$ snakemake -np mapped_reads/{A,B}.bam
```

Note that this is not a special Snakemake syntax. Bash is just applying its brace expansion to the set `{A,B}`, creating the given path for each element and separating the resulting paths by a whitespace.

In both cases, you will see that Snakemake only proposes to create the output file `mapped_reads/B.bam`. This is because you already executed the workflow before (see the previous step) and no input file is newer than the output file

mapped_reads/A.bam. You can update the file modification date of the input file data/samples/A.fastq via

```
$ touch data/samples/A.fastq
```

and see how Snakemake wants to re-run the job to create the file mapped_reads/A.bam by executing

```
$ snakemake -np mapped_reads/A.bam mapped_reads/B.bam
```

Step 3: Sorting read alignments

For later steps, we need the read alignments in the BAM files to be sorted. This can be achieved with the samtools sort command. We add the following rule beneath the bwa_map rule:

```
rule samtools_sort:
    input:
        "mapped_reads/{sample}.bam"
    output:
        "sorted_reads/{sample}.bam"
    shell:
        "samtools sort -T sorted_reads/{wildcards.sample} "
        "-O bam {input} > {output}"
```

Note

In the shell command above we split the string into two lines, which are however automatically concatenated into one by Python. This is a handy pattern to avoid too long shell command lines. When using this, make sure to have a trailing whitespace in each line but the last, in order to avoid arguments to become not properly separated.

This rule will take the input file from the mapped_reads directory and store a sorted version in the sorted_reads directory. Note that Snakemake **automatically creates missing directories** before jobs are executed. For sorting, samtools requires a prefix specified with the flag -T. Here, we need the value of the wildcard sample. Snakemake allows to access wildcards in the shell command via the wildcards object that has an attribute with the value for each wildcard.

When issuing

```
$ snakemake -np sorted_reads/B.bam
```

you will see how Snakemake wants to run first the rule bwa_map and then the rule samtools_sort to create the desired target file: as mentioned before, the dependencies are resolved automatically by matching file names.

Step 4: Indexing read alignments and visualizing the DAG of jobs

Note

Snakemake uses the Python format mini language to format shell commands. Sometimes you have to use braces ({}) for something else in a shell command. In that case, you have to escape them by doubling, for example when relying on the bash brace expansion we mentioned above: ls {{A,B}}.txt.

Next, we need to use samtools again to index the sorted read alignments so that we can quickly access reads by the genomic location they were mapped to. This can be done with the following rule:

```
rule samtools_index:
    input:
        "sorted_reads/{sample}.bam"
    output:
        "sorted_reads/{sample}.bam.bai"
    shell:
        "samtools index {input}"
```

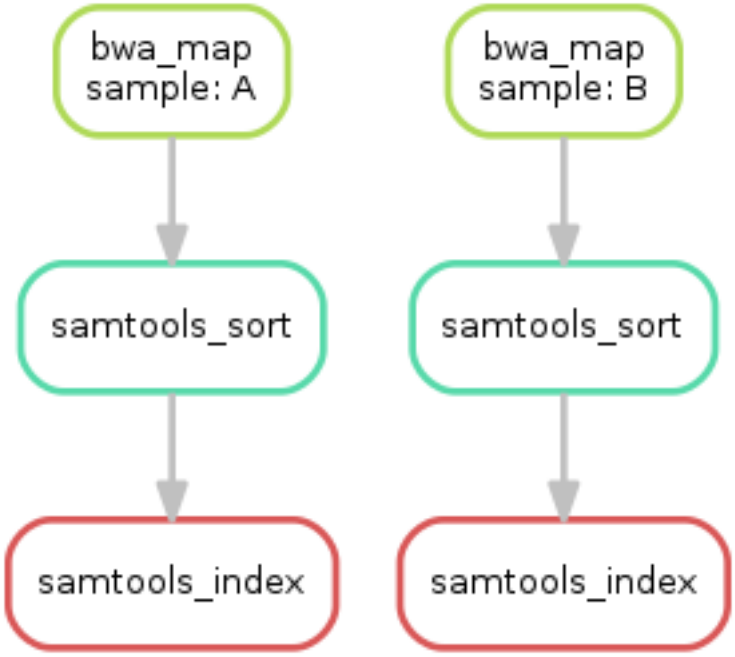
Having three steps already, it is a good time to take a closer look at the resulting directed acyclic graph (DAG) of jobs. By executing

```
$ snakemake --dag sorted_reads/{A,B}.bam.bai | dot -Tsvg > dag.svg
```

Note

If you went with: *Run tutorial for free in the cloud via Gitpod*, you can easily view the resulting `dag.svg` by right-clicking on the file in the explorer panel on the left and selecting `Open With -> Preview`.

we create a **visualization of the DAG** using the `dot` command provided by Graphviz. For the given target files, Snakemake specifies the DAG in the dot language and pipes it into the `dot` command, which renders the definition into SVG format. The rendered DAG is piped into the file `dag.svg` and will look similar to this:



The DAG contains a node for each job with the edges connecting them representing the dependencies. The frames of jobs that don't need to be run (because their output is up-to-date) are dashed. For rules with wildcards, the value of the wildcard for the particular job is displayed in the job node.

Exercise

- Run parts of the workflow using different targets. Recreate the DAG and see how different rules' frames become dashed because their output is present and up-to-date.

Step 5: Calling genomic variants

The next step in our workflow will aggregate the mapped reads from all samples and jointly call genomic variants on them (see *Background*). For the variant calling, we will combine the two utilities samtools and bcftools. Snakemake provides a **helper function for collecting input files** that helps us to describe the aggregation in this step. With

```
expand("sorted_reads/{sample}.bam", sample=SAMPLES)
```

we obtain a list of files where the given pattern `"sorted_reads/{sample}.bam"` was formatted with the values in a given list of samples `SAMPLES`, i.e.

```
["sorted_reads/A.bam", "sorted_reads/B.bam"]
```

The function is particularly useful when the pattern contains multiple wildcards. For example,

```
expand("sorted_reads/{sample}.{replicate}.bam", sample=SAMPLES, replicate=[0, 1])
```

would create the product of all elements of `SAMPLES` and the list `[0, 1]`, yielding

```
["sorted_reads/A.0.bam", "sorted_reads/A.1.bam", "sorted_reads/B.0.bam", "sorted_
↪reads/B.1.bam"]
```

Here, we use only the simple case of `expand`. We first let Snakemake know which samples we want to consider. Remember that Snakemake works backwards from requested output, and not from available input. Thus, it does not automatically infer all possible output from, for example, the fastq files in the data folder. Also remember that Snakefiles are in principle Python code enhanced by some declarative statements to define workflows. Hence, we can define the list of samples ad-hoc in plain Python at the top of the Snakefile:

```
SAMPLES = ["A", "B"]
```

Note

If you name input or output files like above, their order won't be preserved when referring to them as `{input}`. Further, note that named and unnamed (i.e., positional) input and output files can be combined, but the positional ones must come first, equivalent to Python functions with keyword arguments.

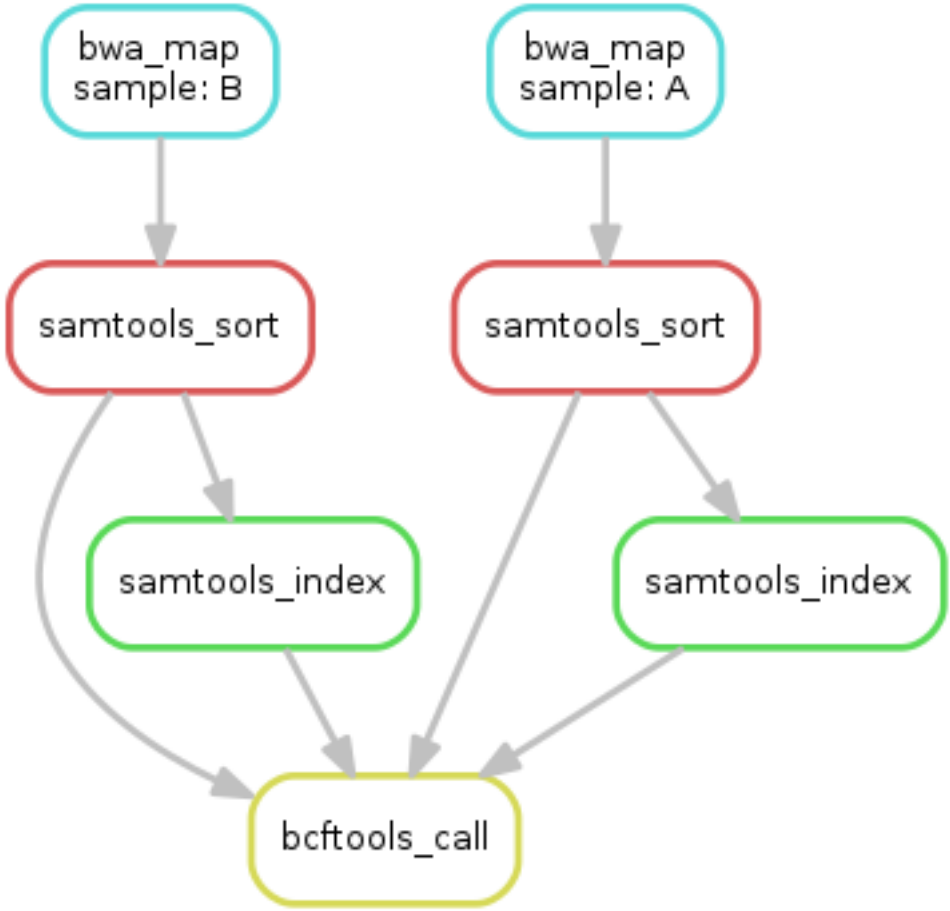
Later, we will learn about more sophisticated ways like **config files**. But for now, this is enough so that we can add the following rule to our Snakefile:

```
rule bcftools_call:
    input:
        fa="data/genome.fa",
        bam=expand("sorted_reads/{sample}.bam", sample=SAMPLES),
        bai=expand("sorted_reads/{sample}.bam.bai", sample=SAMPLES)
    output:
        "calls/all.vcf"
    shell:
        "bcftools mpileup -f {input.fa} {input.bam} | "
        "bcftools call -mv - > {output}"
```

With multiple input or output files, it is sometimes handy to refer to them separately in the shell command. This can be done by **specifying names for input or output files**, for example with `fa=...`. The files can then be referred to in the shell command by name, for example with `{input.fa}`. For **long shell commands** like this one, it is advisable to **split the string over multiple indented lines**. Python will automatically merge it into one. Further, you will notice that the **input or output file lists can contain arbitrary Python statements**, as long as it returns a string, or a list of strings. Here, we invoke our `expand` function to aggregate over the aligned reads of all samples.

Exercise

- obtain the updated DAG of jobs for the target file `calls/all.vcf`, it should look like this:



Step 6: Using custom scripts

Usually, a workflow not only consists of invoking various tools, but also contains custom code to for example calculate summary statistics or create plots. While Snakemake also allows you to directly write Python code inside a rule, it is usually reasonable to move such logic into separate scripts. For this purpose, Snakemake offers the `script` directive. Add the following rule to your Snakefile:

```
rule plot_quals:
    input:
        "calls/all.vcf"
    output:
```

(continues on next page)

(continued from previous page)

```
        "plots/quals.svg"
    script:
        "scripts/plot-quals.py"
```

Note

`snakemake.input` and `snakemake.output` always contain a list of file names, even if the lists each contain only one file name. Therefore, to refer to a particular file name, you have to index into that list. `snakemake.output[0]` will give you the first element of the output file name list, something that always has to be there.

With this rule, we will eventually generate a histogram of the quality scores that have been assigned to the variant calls in the file `calls/all.vcf`. The actual Python code to generate the plot is hidden in the script `scripts/plot-quals.py`. Script paths are always relative to the referring Snakefile. In the script, all properties of the rule like `input`, `output`, `wildcards`, etc. are available as attributes of a global `snakemake` object. Create the file `scripts/plot-quals.py`, with the following content:

```
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
from pysam import VariantFile

quals = [record.qual for record in VariantFile(snakemake.input[0])]
plt.hist(quals)

plt.savefig(snakemake.output[0])
```

Note

It is best practice to use the script directive whenever an inline code block would have more than a few lines of code.

Although there are other strategies to invoke separate scripts from your workflow (for example, invoking them via shell commands), the benefit of this is obvious: the script logic is separated from the workflow logic (and can even be shared between workflows), but **boilerplate code like the parsing of command line arguments is unnecessary**.

Apart from Python scripts, it is also possible to use R scripts. In R scripts, an S4 object named `snakemake` analogous to the Python case above is available and allows access to input and output files and other parameters. Here, the syntax follows that of S4 classes with attributes that are R lists, for example we can access the first input file with `snakemake@input[[1]]` (note that the first file does not have index 0 here, because R starts counting from 1). Named input and output files can be accessed in the same way, by just providing the name instead of an index, for example `snakemake@input[["myfile"]]`.

For details and examples, see the *External scripts* section in the Documentation.

Step 7: Adding a target rule

So far, we always executed the workflow by specifying a target file at the command line. Apart from filenames, Snakemake **also accepts rule names as targets** if the requested rule does not have wildcards. Hence, it is possible to write target rules collecting particular subsets of the desired results or all results. Moreover, if no target is given at the command line, Snakemake will define the **first rule** of the Snakefile as the target. Hence, it is best practice to have a rule `all` at the top of the workflow which has all typically desired target files as input files.

Here, this means that we add a rule

```
rule all:
    input:
        "plots/quals.svg"
```

to the top of our workflow. When executing Snakemake with

```
$ snakemake -n
```

Note

In case you have mutliple reasonable sets of target files, you can add multiple target rules at the top of the Snakefile. While Snakemake will execute the first per default, you can target any of them via the command line (for example, `snakemake -n mytarget`).

the execution plan for creating the file `plots/quals.svg`, which contains and summarizes all our results, will be shown. Note that, apart from Snakemake considering the first rule of the workflow as the default target, **the order of rules in the Snakefile is arbitrary and does not influence the DAG of jobs**.

Exercise

- Create the DAG of jobs for the complete workflow.
- Execute the complete workflow and have a look at the resulting `plots/quals.svg`.
- Snakemake provides handy flags for forcing re-execution of parts of the workflow. Have a look at the command line help with `snakemake --help` and search for the flag `--forcerun`. Then, use this flag to re-execute the rule `samtools_sort` and see what happens.
- Snakemake displays the reason for each job (under `reason:`). Perform a dry-run that forces some rules to be reexecuted (using the `--forcerun` flag in combination with some rulename) to understand the decisions of Snakemake.

Summary

In total, the resulting workflow looks like this:

```
SAMPLES = ["A", "B"]


rule all:
    input:
        "plots/quals.svg"
```

(continues on next page)

(continued from previous page)

```
rule bwa_map:
    input:
        "data/genome.fa",
        "data/samples/{sample}.fastq"
    output:
        "mapped_reads/{sample}.bam"
    shell:
        "bwa mem {input} | samtools view -Sb - > {output}"


rule samtools_sort:
    input:
        "mapped_reads/{sample}.bam"
    output:
        "sorted_reads/{sample}.bam"
    shell:
        "samtools sort -T sorted_reads/{wildcards.sample} "
        "-O bam {input} > {output}"


rule samtools_index:
    input:
        "sorted_reads/{sample}.bam"
    output:
        "sorted_reads/{sample}.bam.bai"
    shell:
        "samtools index {input}"


rule bcftools_call:
    input:
        fa="data/genome.fa",
        bam=expand("sorted_reads/{sample}.bam", sample=SAMPLES),
        bai=expand("sorted_reads/{sample}.bam.bai", sample=SAMPLES)
    output:
        "calls/all.vcf"
    shell:
        "bcftools mpileup -f {input.fa} {input.bam} | "
        "bcftools call -mv - > {output}"


rule plot_quals:
    input:
        "calls/all.vcf"
    output:
        "plots/quals.svg"
    script:
        "scripts/plot-quals.py"
```

4.2.3 Advanced: Decorating the example workflow

Now that the basic concepts of Snakemake have been illustrated, we can introduce some advanced functionality.

Step 1: Specifying the number of used threads

For some tools, it is advisable to use more than one thread in order to speed up the computation. **Snakemake can be made aware of the threads a rule needs** with the `threads` directive. In our example workflow, it makes sense to use multiple threads for the rule `bwa_map`:

```
rule bwa_map:
    input:
        "data/genome.fa",
        "data/samples/{sample}.fastq"
    output:
        "mapped_reads/{sample}.bam"
    threads: 8
    shell:
        "bwa mem -t {threads} {input} | samtools view -Sb - > {output}"
```

The number of threads can be propagated to the shell command with the familiar braces notation (i.e. `{threads}`). If no `threads` directive is given, a rule is assumed to need 1 thread.

When a workflow is executed, **the number of threads the jobs need is considered by the Snakemake scheduler**. In particular, the scheduler ensures that the sum of the threads of all jobs running at the same time does not exceed a given number of available CPU cores. This number is given with the `--cores` command line argument, which is mandatory for `snakemake` calls that actually run the workflow. For example

```
$ snakemake --cores 10
```

Note

Apart from the very common thread resource, Snakemake provides a `resources` directive that can be used to **specify arbitrary resources**, e.g., memory usage or auxiliary computing devices like GPUs. Similar to threads, these can be considered by the scheduler when an available amount of that resource is given with the command line argument `--resources` (see *Resources*).

would execute the workflow with 10 cores. Since the rule `bwa_map` needs 8 threads, only one job of the rule can run at a time, and the Snakemake scheduler will try to saturate the remaining cores with other jobs like, e.g., `samtools_sort`. The threads directive in a rule is interpreted as a maximum: when **less cores than threads** are provided, the number of threads a rule uses will be **reduced to the number of given cores**.

If `--cores` is given without a number, all available cores are used.

Exercise

- With the flag `--forceall` you can enforce a complete re-execution of the workflow. Combine this flag with different values for `--cores` and examine how the scheduler selects jobs to run in parallel.

Step 2: Config files

So far, we specified which samples to consider by providing a Python list in the Snakefile. However, often you want your workflow to be customizable, so that it can easily be adapted to new data. For this purpose, Snakemake provides a config file mechanism. Config files can be written in JSON or YAML, and are used with the `configfile` directive. In our example workflow, we add the line

```
configfile: "config.yaml"
```

to the top of the Snakefile. Snakemake will load the config file and store its contents into a globally available dictionary named `config`. In our case, it makes sense to specify the samples in `config.yaml` as

```
samples:
    A: data/samples/A.fastq
    B: data/samples/B.fastq
```

Now, we can remove the statement defining `SAMPLES` from the Snakefile and change the rule `bcftools_call` to

```
rule bcftools_call:
    input:
        fa="data/genome.fa",
        bam=expand("sorted_reads/{sample}.bam", sample=config["samples"]),
        bai=expand("sorted_reads/{sample}.bam.bai", sample=config["samples"])
    output:
        "calls/all.vcf"
    shell:
        "bcftools mpileup -f {input.fa} {input.bam} | "
        "bcftools call -mv - > {output}"
```

Step 3: Input functions

Since we have stored the path to the FASTQ files in the config file, we can also generalize the rule `bwa_map` to use these paths. This case is different to the rule `bcftools_call` we modified above. To understand this, it is important to know that Snakemake workflows are executed in three phases.

1. In the **initialization** phase, the files defining the workflow are parsed and all rules are instantiated.
2. In the **DAG** phase, the directed acyclic dependency graph of all jobs is built by filling wildcards and matching input files to output files.
3. In the **scheduling** phase, the DAG of jobs is executed, with jobs started according to the available resources.

The expand functions in the list of input files of the rule `bcftools_call` are executed during the initialization phase. In this phase, we don't know about jobs, wildcard values and rule dependencies. Hence, we cannot determine the FASTQ paths for rule `bwa_map` from the config file in this phase, because we don't even know which jobs will be generated from that rule. Instead, we need to defer the determination of input files to the DAG phase. This can be achieved by specifying an **input function** instead of a string as inside of the input directive. For the rule `bwa_map` this works as follows:

```
def get_bwa_map_input_fastqs(wildcards):
    return config["samples"][wildcards.sample]

rule bwa_map:
    input:
        "data/genome.fa",
        get_bwa_map_input_fastqs
    output:
        "mapped_reads/{sample}.bam"
    threads: 8
    shell:
        "bwa mem -t {threads} {input} | samtools view -Sb - > {output}"
```

Note

Snakemake does not automatically rerun jobs when new input files are added as in the excercise below. However, you can get a list of output files that are affected by such changes with `snakemake --list-input-changes`. To trigger a rerun, this bit of bash magic helps:

```
snakemake -n --forcerun $(snakemake --list-input-changes)
```

Any normal function would work as well. Input functions take as **single argument** a `wildcards` object, that allows to access the wildcards values via attributes (here `wildcards.sample`). They have to **return a string or a list of strings**, that are interpreted as paths to input files (here, we return the path that is stored for the sample in the config file). Input functions are evaluated once the wildcard values of a job are determined.

Exercise

- In the `data/samples` folder, there is an additional sample `C.fastq`. Add that sample to the config file and see how Snakemake wants to recompute the part of the workflow belonging to the new sample, when invoking with `snakemake -n --forcerun bcftools_call`.

Step 4: Rule parameters

Sometimes, shell commands are not only composed of input and output files and some static flags. In particular, it can happen that additional parameters need to be set depending on the wildcard values of the job. For this, Snakemake allows to **define arbitrary parameters** for rules with the `params` directive. In our workflow, it is reasonable to annotate aligned reads with so-called read groups, that contain metadata like the sample name. We modify the rule `bwa_map` accordingly:

```
rule bwa_map:
    input:
        "data/genome.fa",
        get_bwa_map_input_fastqs
    output:
        "mapped_reads/{sample}.bam"
    params:
        rg=r"@RG\tID:{sample}\tSM:{sample}"
    threads: 8
    shell:
        "bwa mem -R '{params.rg}' -t {threads} {input} | samtools view -Sb - >
↪{output}"
```

Note

The `params` directive can also take functions like in Step 3 to defer initialization to the DAG phase. In contrast to input functions, these can optionally take additional arguments `input`, `output`, `threads`, and `resources`.

Similar to input and output files, `params` can be accessed from the shell command, the Python based `run` block, or the script directive (see *Step 6: Using custom scripts*).

Exercise

- Variant calling can consider a lot of parameters. A particularly important one is the prior mutation rate (1e-3 per default). It is set via the flag `-P` of the `bcftools call` command. Consider making this flag configurable via adding a new key to the config file and using the `params` directive in the rule `bcftools_call` to propagate it to the shell command.

Step 5: Logging

When executing a large workflow, it is usually desirable to store the logging output of each job into a separate file, instead of just printing all logging output to the terminal—when multiple jobs are run in parallel, this would result in chaotic output. For this purpose, Snakemake allows to **specify log files** for rules. Log files are defined via the `log` directive and handled similarly to output files, but they are not subject of rule matching and are not cleaned up when a job fails. We modify our rule `bwa_map` as follows:

```
rule bwa_map:
    input:
        "data/genome.fa",
        get_bwa_map_input_fastqs
    output:
        "mapped_reads/{sample}.bam"
    params:
        rg=r"@RG\tID:{sample}\tSM:{sample}"
    log:
        "logs/bwa_mem/{sample}.log"
    threads: 8
    shell:
        "(bwa mem -R '{params.rg}' -t {threads} {input} | "
        "samtools view -Sb - > {output}) 2> {log}"
```

Note

It is best practice to store all log files in a subdirectory `logs/`, prefixed by the rule or tool name.

The shell command is modified to collect STDERR output of both `bwa` and `samtools` and pipe it into the file referred to by `{log}`. Log files must contain exactly the same wildcards as the output files to avoid file name clashes between different jobs of the same rule.

Exercise

- Add a log directive to the `bcftools_call` rule as well.
- Time to re-run the whole workflow (remember the command line flags to force re-execution). See how log files are created for variant calling and read mapping.
- The ability to track the provenance of each generated result is an important step towards reproducible analyses. Apart from the `report` functionality discussed before, Snakemake can summarize various provenance information for all output files of the workflow. The flag `--summary` prints a table associating each output file with the rule used to generate it, the creation date and optionally the version of the tool used for creation is provided. Further, the table informs about updated input files and changes to the source code of the rule after creation of the output file. Invoke Snakemake with `--summary` to examine the information for our example.

Step 6: Temporary and protected files

In our workflow, we create two BAM files for each sample, namely the output of the rules `bwa_map` and `samtools_sort`. When not dealing with examples, the underlying data is usually huge. Hence, the resulting BAM files need a lot of disk space and their creation takes some time. To save disk space, you can **mark output files as temporary**. Snakemake will delete the marked files for you, once all the consuming jobs (that need it as input) have been executed. We use this mechanism for the output file of the rule `bwa_map`:

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

This results in the deletion of the BAM file once the corresponding `samtools_sort` job has been executed. Since the creation of BAM files via read mapping and sorting is computationally expensive, it is reasonable to **protect** the final BAM file **from accidental deletion or modification**. We modify the rule `samtools_sort` to mark its output file as `protected`:

```
rule samtools_sort:
    input:
        "mapped_reads/{sample}.bam"
    output:
        protected("sorted_reads/{sample}.bam")
    shell:
        "samtools sort -T sorted_reads/{wildcards.sample} "
        "-O bam {input} > {output}"
```

After successful execution of the job, Snakemake will write-protect the output file in the filesystem, so that it can't be overwritten or deleted by accident.

Exercise

- Re-execute the whole workflow and observe how Snakemake handles the temporary and protected files.
- Run Snakemake with the target `mapped_reads/A.bam`. Although the file is marked as temporary, you will see that Snakemake does not delete it because it is specified as a target file.
- Try to re-execute the whole workflow again with the dry-run option. You will see that it fails (as intended) because Snakemake cannot overwrite the protected output files.

Summary

For this advanced part of the tutorial, we have now created a `config.yaml` configuration file:

```
samples:
    A: data/samples/A.fastq
    B: data/samples/B.fastq

prior_mutation_rate: 0.001
```

With this, the final version of our workflow in the `Snakefile` looks like this:

```
configfile: "config.yaml"


rule all:
    input:
        "plots/quals.svg"


def get_bwa_map_input_fastqs(wildcards):
    return config["samples"][wildcards.sample]


rule bwa_map:
    input:
        "data/genome.fa",
        get_bwa_map_input_fastqs
    output:
        temp("mapped_reads/{sample}.bam")
    params:
        rg=r"@RG\tID:{sample}\tSM:{sample}"
    log:
        "logs/bwa_mem/{sample}.log"
    threads: 8
    shell:
        "(bwa mem -R '{params.rg}' -t {threads} {input} | "
        "samtools view -Sb - > {output}) 2> {log}"


rule samtools_sort:
    input:
        "mapped_reads/{sample}.bam"
    output:
        protected("sorted_reads/{sample}.bam")
    shell:
        "samtools sort -T sorted_reads/{wildcards.sample} "
```

(continues on next page)

(continued from previous page)

```
        "-O bam {input} > {output}"


rule samtools_index:
    input:
        "sorted_reads/{sample}.bam"
    output:
        "sorted_reads/{sample}.bam.bai"
    shell:
        "samtools index {input}"


rule bcftools_call:
    input:
        fa="data/genome.fa",
        bam=expand("sorted_reads/{sample}.bam", sample=config["samples"]),
        bai=expand("sorted_reads/{sample}.bam.bai", sample=config["samples"])
    output:
        "calls/all.vcf"
    params:
        rate=config["prior_mutation_rate"]
    log:
        "logs/bcftools_call/all.log"
    shell:
        "(bcftools mpileup -f {input.fa} {input.bam} | "
        "bcftools call -mv -P {params.rate} - > {output}) 2> {log}"


rule plot_quals:
    input:
        "calls/all.vcf"
    output:
        "plots/quals.svg"
    script:
        "scripts/plot-quals.py"
```

4.2.4 Additional features

In the following, we introduce some features that are beyond the scope of above example workflow. For details and even more features, see *Writing Workflows*, *Frequently Asked Questions* and the command line help (`snakemake --help`).

Benchmarking

With the `benchmark` directive, Snakemake can be instructed to **measure the wall clock time of a job**. We activate benchmarking for the rule `bwa_map`:

```
rule bwa_map:
    input:
        "data/genome.fa",
        lambda wildcards: config["samples"][wildcards.sample]
    output:
        temp("mapped_reads/{sample}.bam")
    params:
```

(continues on next page)

(continued from previous page)

```
        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}"
```

The `benchmark` directive takes a string that points to the file where benchmarking results shall be stored. Similar to output files, the path can contain wildcards (it must be the same wildcards as in the output files). When a job derived from the rule is executed, Snakemake will measure the wall clock time and memory usage (in MiB) and store it in the file in tab-delimited format. It is possible to repeat a benchmark multiple times in order to get a sense for the variability of the measurements. This can be done by annotating the benchmark file, e.g., with `repeat("benchmarks/{sample}.bwa.benchmark.txt", 3)` Snakemake can be told to run the job three times. The repeated measurements occur as subsequent lines in the tab-delimited benchmark file.

Modularization

In order to re-use building blocks or simply to structure large workflows, it is sometimes reasonable to **split a workflow into modules**. For this, Snakemake provides the `include` directive to include another Snakefile into the current one, e.g.:

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

Alternatively, Snakemake allows to **define sub-workflows**. A sub-workflow refers to a working directory with a complete Snakemake workflow. Output files of that sub-workflow can be used in the current Snakefile. When executing, Snakemake ensures that the output files of the sub-workflow are up-to-date before executing the current workflow. This mechanism is particularly useful when you want to extend a previous analysis without modifying it. For details about sub-workflows, see the *documentation*.

Exercise

- Put the read mapping related rules into a separate Snakefile and use the `include` directive to make them available in our example workflow again.

Automatic deployment of software dependencies

In order to get a fully reproducible data analysis, it is not sufficient to be able to execute each step and document all used parameters. The used software tools and libraries have to be documented as well. In this tutorial, you have already seen how Conda can be used to specify an isolated software environment for a whole workflow. With Snakemake, you can go one step further and specify Conda environments per rule. This way, you can even make use of conflicting software versions (e.g. combine Python 2 with Python 3).

In our example, instead of using an external environment we can specify environments per rule, e.g.:

```
rule samtools_index:
  input:
      "sorted_reads/{sample}.bam"
  output:
```

(continues on next page)

(continued from previous page)

```
        "sorted_reads/{sample}.bam.bai"
    conda:
        "envs/samtools.yaml"
    shell:
        "samtools index {input}"
```

with `envs/samtools.yaml` defined as

```
channels:
  - bioconda
  - conda-forge
dependencies:
  - samtools =1.9
```

Note

The conda directive does not work in combination with `run` blocks, because they have to share their Python environment with the surrounding snakefile.

When Snakemake is executed with

```
snakemake --use-conda --cores 1
```

it will automatically create required environments and activate them before a job is executed. It is best practice to specify at least the major and minor version of any packages in the environment definition. Specifying environments per rule in this way has two advantages. First, the workflow definition also documents all used software versions. Second, a workflow can be re-executed (without admin rights) on a vanilla system, without installing any prerequisites apart from Snakemake and Miniconda.

Tool wrappers

In order to simplify the utilization of popular tools, Snakemake provides a repository of so-called wrappers (the Snakemake wrapper repository). A wrapper is a short script that wraps (typically) a command line application and makes it directly addressable from within Snakemake. For this, Snakemake provides the `wrapper` directive that can be used instead of `shell`, `script`, or `run`. For example, the rule `bwa_map` could alternatively look like this:

```
rule bwa_mem:
    input:
        ref="data/genome.fa",
        sample=lambda wildcards: config["samples"][wildcards.sample]
    output:
        temp("mapped_reads/{sample}.bam")
    log:
        "logs/bwa_mem/{sample}.log"
    params:
        "-R '@RG\tID:{sample}\tSM:{sample}'"
    threads: 8
    wrapper:
        "0.15.3/bio/bwa/mem"
```

Note

Updates to the Snakemake wrapper repository are automatically tested via continuous integration.

The wrapper directive expects a (partial) URL that points to a wrapper in the repository. These can be looked up in the corresponding database. The first part of the URL is a Git version tag. Upon invocation, Snakemake will automatically download the requested version of the wrapper. Furthermore, in combination with `--use-conda` (see *Automatic deployment of software dependencies*), the required software will be automatically deployed before execution.

Cluster execution

By default, Snakemake executes jobs on the local machine it is invoked on. Alternatively, it can execute jobs in **distributed environments, e.g., compute clusters or batch systems**. If the nodes share a common file system, Snakemake supports three alternative execution modes.

In cluster environments, compute jobs are usually submitted as shell scripts via commands like `qsub`. Snakemake provides a **generic mode** to execute on such clusters. By invoking Snakemake with

```
$ snakemake --cluster qsub --jobs 100
```

each job will be compiled into a shell script that is submitted with the given command (here `qsub`). The `--jobs` flag limits the number of concurrently submitted jobs to 100. This basic mode assumes that the submission command returns immediately after submitting the job. Some clusters allow to run the submission command in **synchronous mode**, such that it waits until the job has been executed. In such cases, we can invoke e.g.

```
$ snakemake --cluster-sync "qsub -sync yes" --jobs 100
```

The specified submission command can also be **decorated with additional parameters taken from the submitted job**. For example, the number of used threads can be accessed in braces similarly to the formatting of shell commands, e.g.

```
$ snakemake --cluster "qsub -pe threaded {threads}" --jobs 100
```

Alternatively, Snakemake can use the Distributed Resource Management Application API (DRMAA). This API provides a common interface to control various resource management systems. The **DRMAA support** can be activated by invoking Snakemake as follows:

```
$ snakemake --drmaa --jobs 100
```

If available, **DRMAA is preferable over the generic cluster modes** because it provides better control and error handling. To support additional cluster specific parametrization, a Snakefile can be complemented by a *Cluster Configuration (deprecated)* file.

Using –cluster-status

Sometimes you need specific detection to determine if a cluster job completed successfully, failed or is still running. Error detection with `--cluster` can be improved for edge cases such as timeouts and jobs exceeding memory that are silently terminated by the queueing system. This can be achieved with the `--cluster-status` option. The value of this option should be a executable script which takes a job id as the first argument and prints to stdout only one of [running|success|failed]. Importantly, the job id snakemake passes on is captured from the stdout of the cluster submit tool. This string will often include more than the job id, but snakemake does not modify this string and will pass this string to the status script unchanged. In the situation where snakemake has received more than the job id these are 3 potential solutions to consider: parse the string received by the script and extract the job id within the script, wrap the submission tool to intercept its stdout and return just the job code, or ideally, the cluster may offer an option to only return

the job id upon submission and you can instruct snakemake to use that option. For sge this would look like `snakemake --cluster "qsub -terse"`.

The following (simplified) script detects the job status on a given SLURM cluster (>= 14.03.0rc1 is required for `--parsable`).

```
#!/usr/bin/env python
import subprocess
import sys

jobid = sys.argv[1]

output = str(subprocess.check_output("sacct -j %s --format State --noheader | head -1 | awk '{print $1}'" % jobid, shell=True).strip())

running_status=["PENDING", "CONFIGURING", "COMPLETING", "RUNNING", "SUSPENDED"]
if "COMPLETED" in output:
  print("success")
elif any(r in output for r in running_status):
  print("running")
else:
  print("failed")
```

To use this script call snakemake similar to below, where `status.py` is the script above.

```
$ snakemake all --jobs 100 --cluster "sbatch --cpus-per-task=1 --parsable" --cluster-status ./status.py
```

Using –cluster-cancel

When snakemake is terminated by pressing `Ctrl-C`, it will cancel all currently running node when using `--drmaa`. You can get the same behaviour with `--cluster` by adding `--cluster-cancel` and passing a command to use for canceling jobs by their jobid (e.g., `scancel` for SLURM or `qdel` for SGE). Most job schedulers can be passed multiple jobids and you can use `--cluster-cancel-nargs` to limit the number of arguments (default is 1000 which is reasonable for most schedulers).

Using –cluster-sidecar

In certain situations, it is necessary to not perform calls to cluster commands directly and instead have a "sidecar" process, e.g., providing a REST API. One example is when using SLURM where regular calls to `scontrol show job JOBID` or `sacct -j JOBID` puts a high load on the controller. Rather, it is better to use the `squeue` command with the `-i/--iterate` option.

When using `--cluster`, you can use `--cluster-sidecar` to pass in a command that starts a sidecar server. The command should print one line to stdout and then block and accept connections. The line will subsequently be available in the calls to `--cluster`, `--cluster-status`, and `--cluster-cancel` in the environment variable `SNAKEMAKE_CLUSTER_SIDECAR_VARS`. In the case of a REST server, you can use this to return the port that the server is listening on and credentials. When the Snakemake process terminates, the sidecar process will be terminated as well.

Constraining wildcards

Snakemake uses regular expressions to match output files to input files and determine dependencies between the jobs. Sometimes it is useful to constrain the values a wildcard can have. This can be achieved by adding a regular expression that describes the set of allowed wildcard values. For example, the wildcard `sample` in the output file `"sorted_reads/{sample}.bam"` can be constrained to only allow alphanumeric sample names as `"sorted_reads/{sample,[A-Za-z0-9]+}.bam"`. Constraints may be defined per rule or globally using the `wildcard_constraints` keyword, as demonstrated in *Wildcards*. This mechanism helps to solve two kinds of ambiguity.

- It can help to avoid ambiguous rules, i.e. two or more rules that can be applied to generate the same output file. Other ways of handling ambiguous rules are described in the Section *Handling Ambiguous Rules*.
- It can help to guide the regular expression based matching so that wildcards are assigned to the right parts of a file name. Consider the output file `{sample}.{group}.txt` and assume that the target file is `A.1.normal.txt`. It is not clear whether `dataset="A.1"` and `group="normal"` or `dataset="A"` and `group="1.normal"` is the right assignment. Here, constraining the dataset wildcard by `{sample,[A-Z]+}.{group}` solves the problem.

When dealing with ambiguous rules, it is best practice to first try to solve the ambiguity by using a proper file structure, for example, by separating the output files of different steps in different directories.

4.3 Short tutorial

Here we provide a short tutorial that guides you through the main features of Snakemake. Note that this is not suited to learn Snakemake from scratch, rather to give a first impression. To really learn Snakemake (starting from something simple, and extending towards advanced features), use the main *Snakemake Tutorial*.

This document shows all steps performed in the official Snakemake live demo, such that it becomes possible to follow them at your own pace. Solutions to each step can be found at the bottom of this document.

The examples presented in this tutorial come from Bioinformatics. However, Snakemake is a general-purpose workflow management system for any discipline. For an explanation of the steps you will perform here, have a look at *Background*. More thorough explanations are provided in the full *Snakemake Tutorial*.

4.3.1 Prerequisites

First, install Snakemake via Conda, as outlined in *Installation via Conda/Mamba*. The minimal version of Snakemake is sufficient for this demo.

Second, download and unpack the test data needed for this example from here, e.g., via

```
mkdir snakemake-demo
cd snakemake-demo
wget https://github.com/snakemake/snakemake-tutorial-data/archive/v5.4.5.tar.gz
tar --wildcards -xf v5.4.5.tar.gz --strip 1 "*/data"
```

4.3.2 Step 1

First, create an empty workflow in the current directory with:

```
mkdir workflow
touch workflow/Snakefile
```

Once a Snakefile is present, you can perform a dry run of Snakemake with:

```
snakemake -n
```

Since the Snakefile is empty, it will report that nothing has to be done. In the next steps, we will gradually fill the Snakefile with an example analysis workflow.

4.3.3 Step 2

The data folder in your working directory looks as follows:

```
data
├── genome.fa
├── genome.fa.amb
├── genome.fa.ann
├── genome.fa.bwt
├── genome.fa.fai
├── genome.fa.pac
├── genome.fa.sa
└── samples
    ├── A.fastq
    ├── B.fastq
    └── C.fastq
```

You will create a workflow that maps the sequencing samples in the `data/samples` folder to the reference genome `data/genome.fa`. Then, you will call genomic variants over the mapped samples, and create an example plot.

First, create a rule called `map_reads`, with input files

- `data/genome.fa`
- `data/samples/A.fastq`

and output file

- `results/mapped/A.bam`

To generate output from input, use the shell command

```
"bwa mem {input} | samtools view -Sb - > {output}"
```

Providing a shell command is not enough to run your workflow on an unprepared system. For reproducibility, you also have to provide the required software stack and define the desired version. This can be done with the Conda package manager, which is directly integrated with Snakemake: add a directive `conda: "envs/mapping.yaml"` that points to a Conda environment definition, with the following content

```
channels:
  - bioconda
  - conda-forge
dependencies:
  - bwa =0.7.17
  - samtools =1.9
```

Upon execution, Snakemake will automatically create that environment, and execute the shell command within.

Now, test your workflow by simulating the creation of the file `results/mapped/A.bam` via

```
snakemake --use-conda -n results/mapped/A.bam
```

to perform a dry-run and

```
snakemake --use-conda results/mapped/A.bam --cores 1
```

to perform the actual execution.

4.3.4 Step 3

Now, generalize the rule `map_reads` by replacing the concrete sample name `A` with a wildcard `{sample}` in input and output file the rule `map_reads`. This way, Snakemake can apply the rule to map any of the three available samples to the reference genome.

Test this by creating the file `results/mapped/B.bam`.

4.3.5 Step 4

Next, create a rule `sort_alignments` that sorts the obtained `.bam` file by genomic coordinate. The rule should have the input file

- `results/mapped/{sample}.bam`

and the output file

- `results/mapped/{sample}.sorted.bam`

and uses the shell command

```
samtools sort -o {output} {input}
```

to perform the sorting. Moreover, use the same `conda:` directive as for the previous rule.

Test your workflow with

```
snakemake --use-conda -n results/mapped/A.sorted.bam
```

and

```
snakemake --use-conda results/mapped/A.sorted.bam --cores 1
```

4.3.6 Step 5

Now, we aggregate over all samples to perform a joint calling of genomic variants. First, we define a variable

```
samples = ["A", "B", "C"]
```

at the top of the `Snakefile`. This serves as a definition of the samples over which we would want to aggregate. In real life, you would want to use an external sample sheet or a config file for things like this.

For aggregation over many files, Snakemake provides the helper function `expand` (see the docs). Create a rule `call` with input files

- `fa="data/genome.fa"`
- `bam=expand("results/mapped/{sample}.sorted.bam", sample=samples)`

output file

- `"results/calls/all.vcf"`

and shell command

```
bcftools mpileup -f {input.fa} {input.bam} | bcftools call -mv - > {output}
```

Further, define a new conda environment file with the following content:

```
channels:
  - bioconda
  - conda-forge
dependencies:
  - bcftools =1.9
```

4.3.7 Step 6

Finally, we strive to calculate some exemplary statistics. This time, we don't use a shell command, but rather employ Snakemake's ability to integrate with scripting languages like R and Python, and Jupyter notebooks.

First, we create a rule `plot_quals` with input file

- `"results/calls/all.vcf"`

and output file

- `"results/plots/quals.svg"`.

Instead of a shell command, we use Snakemake's Jupyter notebook integration by specifying

```
notebook:
    "notebooks/plot-quals.py"
```

instead of using the `shell` directive as before.

Next, we have to define a conda environment for the rule, say `workflow/envs/stats.yaml`, that provides the required Python packages to execute the script:

```
channels:
  - bioconda
  - conda-forge
dependencies:
  - pysam =0.17
  - altair =4.1
  - altair_saver =0.5
  - pandas =1.3
  - jupyter =1.0
```

Then, we let Snakemake generate a skeleton notebook for us with

```
snakemake --draft-notebook results/plots/quals.svg --cores 1 --use-conda
```

Snakemake will print instructions on how to open, edit and execute the notebook.

We open the notebook in the editor and add the following content

```
import pandas as pd
import altair as alt
from pysam import VariantFile

quals = pd.DataFrame({"qual": [record.qual for record in VariantFile(snakemake.
↪input[0])]})

chart = alt.Chart(quals).mark_bar().encode(
    alt.X("qual", bin=True),
    alt.Y("count()")
)

chart.save(snakemake.output[0])
```

As you can see, instead of writing a command line parser for passing parameters like input and output files, you have direct access to the properties of the rule via a magic `snakemake` object, that Snakemake automatically inserts into the notebook before executing the rule.

Make sure to test your workflow with

```
snakemake --use-conda --force results/plots/quals.svg --cores 1
```

Here, the force ensures that the readily drafted notebook is re-executed even if you had already generated the output plot in the interactive mode.

4.3.8 Step 7

So far, we have always specified a target file at the command line when invoking Snakemake. When no target file is specified, Snakemake tries to execute the first rule in the `Snakefile`. We can use this property to define default target files.

At the top of your `Snakefile` define a rule `all`, with input files

- `"results/calls/all.vcf"`
- `"results/plots/quals.svg"`

and neither a shell command nor output files. This rule simply serves as an indicator of what shall be collected as results.

4.3.9 Step 8

As a last step, we strive to annotate our workflow with some additional information.

Automatic reports

Snakemake can automatically create HTML reports with

```
snakemake --report report.html
```

Such a report contains runtime statistics, a visualization of the workflow topology, used software and data provenance information.

In addition, you can mark any output file generated in your workflow for inclusion into the report. It will be encoded directly into the report, such that it can be, e.g., emailed as a self-contained document. The reader (e.g., a collaborator of yours) can at any time download the enclosed results from the report for further use, e.g., in a manuscript you write

together. In this example, please mark the output file `"results/plots/quals.svg"` for inclusion by replacing it with `report("results/plots/quals.svg", caption="report/calling.rst")` and adding a file `report/calling.rst`, containing some description of the output file. This description will be presented as caption in the resulting report.

Threads

The first rule `map_reads` can in theory use multiple threads. You can make Snakemake aware of this, such that the information can be used for scheduling. Add a directive `threads: 8` to the rule and alter the shell command to

```
bwa mem -t {threads} {input} | samtools view -Sb - > {output}
```

This passes the threads defined in the rule as a command line argument to the `bwa` process.

Temporary files

The output of the `map_reads` rule becomes superfluous once the sorted version of the `.bam` file is generated by the rule `sort`. Snakemake can automatically delete the superfluous output once it is not needed anymore. For this, mark the output as temporary by replacing `"results/mapped/{sample}.bam"` in the rule `bwa` with `temp("results/mapped/{sample}.bam")`.

4.3.10 Solutions

Only read this if you have a problem with one of the steps.

Step 2

The rule should look like this:

```
rule map_reads:
    input:
        "data/genome.fa",
        "data/samples/A.fastq"
    output:
        "results/mapped/A.bam"
    conda:
        "envs/mapping.yaml"
    shell:
        "bwa mem {input} | samtools view -b - > {output}"
```

Step 3

The rule should look like this:

```
rule map_reads:
    input:
        "data/genome.fa",
        "data/samples/{sample}.fastq"
    output:
        "results/mapped/{sample}.bam"
    conda:
```

(continues on next page)

(continued from previous page)

```
        "envs/mapping.yaml"
    shell:
        "bwa mem {input} | samtools view -b - > {output}"
```

Step 4

The rule should look like this:

```
rule sort_alignments:
    input:
        "results/mapped/{sample}.bam"
    output:
        "results/mapped/{sample}.sorted.bam"
    conda:
        "envs/mapping.yaml"
    shell:
        "samtools sort -o {output} {input}"
```

Step 5

The rule should look like this:

```
samples = ["A", "B", "C"]

rule call_variants:
    input:
        fa="data/genome.fa",
        bam=expand("results/mapped/{sample}.sorted.bam", sample=SAMPLES)
    output:
        "results/calls/all.vcf"
    conda:
        "envs/calling.yaml"
    shell:
        "bcftools mpileup -f {input.fa} {input.bam} | bcftools call -mv - > {output}"
```

Step 6

The rule should look like this:

```
rule plot_quals:
    input:
        "results/calls/all.vcf"
    output:
        "results/plots/quals.svg"
    conda:
        "envs/stats.yaml"
    notebook:
        "notebooks/plot-quals.py.ipynb"
```

Step 7

The rule should look like this:

```
rule all:
    input:
        "results/calls/all.vcf",
        "results/plots/quals.svg"
```

It has to appear as first rule in the `Snakefile`.

Step 8

The complete workflow should look like this:

```
SAMPLES = ["A", "B", "C"]

rule all:
    input:
        "results/calls/all.vcf",
        "results/plots/quals.svg"

rule map_reads:
    input:
        "data/genome.fa",
        "data/samples/{sample}.fastq"
    output:
        "results/mapped/{sample}.bam"
    conda:
        "envs/mapping.yaml"
    shell:
        "bwa mem {input} | samtools view -b - > {output}"


rule sort_alignments:
    input:
        "results/mapped/{sample}.bam"
    output:
        "results/mapped/{sample}.sorted.bam"
    conda:
        "envs/mapping.yaml"
    shell:
        "samtools sort -o {output} {input}"


rule call_variants:
    input:
        fa="data/genome.fa",
        bam=expand("results/mapped/{sample}.sorted.bam", sample=SAMPLES)
    output:
        "results/calls/all.vcf"
    conda:
        "envs/calling.yaml"
    shell:
        "bcftools mpileup -f {input.fa} {input.bam} | bcftools call -mv - > {output}"
```

(continues on next page)

(continued from previous page)

```
rule plot_quals:
    input:
        "results/calls/all.vcf"
    output:
        "results/plots/quals.svg"
    conda:
        "envs/stats.yaml"
    notebook:
        "notebooks/plot-quals.py.ipynb"
```

4.4 Snakemake Executor Tutorials

This set of tutorials are intended to introduce you to executing cloud executors. We start with the original Snakemake tutorial and expand upon it to be run in different cloud environments. For each run, we show you how to:

- authenticate with credentials, if required
- prepare your workspace
- submit a basic job
- generate an error and debug

The examples presented in these tutorials come from Bioinformatics. However, Snakemake is a general-purpose workflow management system for any discipline. We ensured that no bioinformatics knowledge is needed to understand the tutorial.

4.4.1 Google Life Sciences Tutorial

Setup

To go through this tutorial, you need the following software installed:

- Python ≥3.5
- Snakemake ≥5.16
- git

First, you have to install the Miniconda Python3 distribution. See here for installation instructions. Make sure to …

- Install the **Python 3** version of Miniconda.
- Answer yes to the question whether conda shall be put into your PATH.

The default conda solver is a bit slow and sometimes has issues with selecting the latest package releases. Therefore, we recommend to install Mamba as a drop-in replacement via

```
$ conda install -c conda-forge mamba
```

Then, you can install Snakemake with

```
$ mamba create -c conda-forge -c bioconda -n snakemake snakemake
```

from the Bioconda channel. This will install snakemake into an isolated software environment, that has to be activated with

```
$ conda activate snakemake
$ snakemake --help
```

Credentials

Using the Google Life Sciences executor with Snakemake requires the environment variable *GOOGLE_APPLICATION_CREDENTIALS* exported, which should point to the full path of the file on your local machine. To generate this file, you can refer to the page under iam-admin to download your service account key and export it to the environment.

```
export GOOGLE_APPLICATION_CREDENTIALS="/home/[username]/credentials.json"
```

The suggested, minimal permissions required for this role include the following:

- Compute Storage Admin(Can potentially be restricted further)
- Compute Viewer
- Service Account User
- Cloud Life Sciences Workflows Runner
- Service Usage Consumer

Step 1: Upload Your Data

We will be obtaining inputs from Google Cloud Storage, as well as saving outputs there. You should first clone the repository with the Snakemake tutorial data:

```
git clone https://github.com/snakemake/snakemake-lsh-tutorial-data
cd snakemake-lsh-tutorial-data
```

And then either manually create a bucket and upload data files there, or use the provided script and instructions to do it programatically from the command line. The script generally works like:

```
python upload_google_storage.py <bucket>/<subpath>   <folder>
```

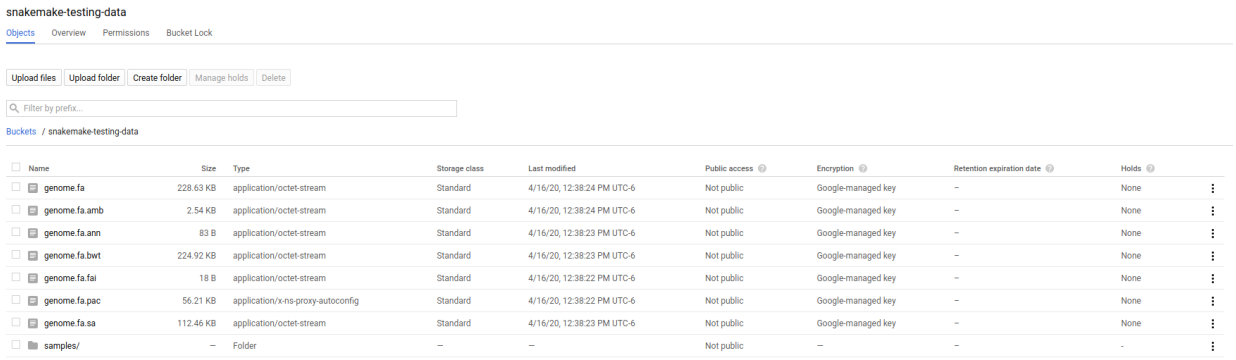
And you aren't required to provide a subfolder path if you want to upload to the root of a bucket. As an example, for this tutorial we upload the contents of "data" to the root of the bucket *snakemake-testing-data*

```
export GOOGLE_APPLICATION_CREDENTIALS="/path/to/credentials.json"
python upload_google_storage.py snakemake-testing-data data/
```

If you wanted to upload to a "subfolder" path in a bucket, you would do that as follows:

```
export GOOGLE_APPLICATION_CREDENTIALS="/path/to/credentials.json"
python upload_google_storage.py snakemake-testing-data/subfolder data/
```

Your bucket (and the folder prefix) will be referred to as the *–default-remote-prefix* when you run snakemake. You can visually browse your data in the *storage browser <https://console.cloud.google.com/storage/>_*.



Step 2: Write your Snakefile, Environment File, and Scripts

Now that we've exported our credentials and have all dependencies installed, let's get our workflow! This is the exact same workflow from the *basic tutorial*, so if you need a refresher on the design or basics, please see those pages. You can find the Snakefile, supporting scripts for plotting and environment in the snakemake-lsh-tutorial-data repository.

First, how does a working directory work for this executor? The present working directory, as identified by Snakemake that has the Snakefile, and where a more advanced setup might have a folder of environment specifications (env) a folder of scripts (scripts), and rules (rules), is considered within the context of the build. When the Google Life Sciences executor is used, it generates a build package of all of the files here (within a reasonable size) and uploads those to storage. This package includes the .snakemake folder that would have been generated locally. The build package is then downloaded and extracted by each cloud executor, which is a Google Compute instance.

We next need an *environment.yaml* file that will define the dependencies that we want installed with conda for our job. If you cloned the "snakemake-lsh-tutorial-data" repository you will already have this, and you are good to go. If not, save this to *environment.yaml* in your working directory:

```
channels:
  - conda-forge
  - bioconda
dependencies:
  - python =3.6
  - jinja2 =2.10
  - networkx =2.1
  - matplotlib =2.2.3
  - graphviz =2.38.0
  - bcftools =1.9
  - samtools =1.9
  - bwa =0.7.17
  - pysam =0.15.0
```

Notice that we reference this *environment.yaml* file in the Snakefile below. Importantly, if you were optimizing a pipeline, you would likely have a folder "envs" with more than one environment specification, one for each step. This workflow uses the same environment (with many dependencies) instead of this strategy to minimize the number of files for you.

The Snakefile (also included in the repository) then has the following content. It's important to note that we have not customized this file from the basic tutorial to hard code any storage. We will be telling snakemake to use the remote bucket as storage instead of the local filesystem.

```
SAMPLES = ["A", "B"]

rule all:
```

(continues on next page)

(continued from previous page)

```
    input:
        "plots/quals.svg"

rule bwa_map:
    input:
        fastq="samples/{sample}.fastq",
        idx=multiext("genome.fa", ".amb", ".ann", ".bwt", ".pac", ".sa")
    conda:
        "environment.yaml"
    output:
        "mapped_reads/{sample}.bam"
    params:
        idx=lambda w, input: os.path.splitext(input.idx[0])[0]
    shell:
        "bwa mem {params.idx} {input.fastq} | samtools view -Sb - > {output}"

rule samtools_sort:
    input:
        "mapped_reads/{sample}.bam"
    output:
        "sorted_reads/{sample}.bam"
    conda:
        "environment.yaml"
    shell:
        "samtools sort -T sorted_reads/{wildcards.sample} "
        "-O bam {input} > {output}"

rule samtools_index:
    input:
        "sorted_reads/{sample}.bam"
    output:
        "sorted_reads/{sample}.bam.bai"
    conda:
        "environment.yaml"
    shell:
        "samtools index {input}"

rule bcftools_call:
    input:
        fa="genome.fa",
        bam=expand("sorted_reads/{sample}.bam", sample=SAMPLES),
        bai=expand("sorted_reads/{sample}.bam.bai", sample=SAMPLES)
    output:
        "calls/all.vcf"
    conda:
        "environment.yaml"
    shell:
        "samtools mpileup -g -f {input.fa} {input.bam} | "
        "bcftools call -mv - > {output}"

rule plot_quals:
    input:
        "calls/all.vcf"
    output:
        "plots/quals.svg"
    conda:
        "environment.yaml"
```

(continues on next page)

(continued from previous page)

```
    script:
        "plot-quals.py"
```

And make sure you also have the script *plot-quals.py* in your present working directory for the last step. This script will help us to do the plotting, and is also included in the snakemake-lsh-tutorial-data repository.

```
import matplotlib

matplotlib.use("Agg")
import matplotlib.pyplot as plt
from pysam import VariantFile

quals = [record.qual for record in VariantFile(snakemake.input[0])]
plt.hist(quals)

plt.savefig(snakemake.output[0])
```

Step 3: Run Snakemake

Now let's run Snakemake with the Google Life Sciences Executor.

```
snakemake --google-lifesciences --default-remote-prefix snakemake-testing-data --use-
↪conda --google-lifesciences-region us-west1
```

The flags above refer to:

- *–google-lifesciences*: to indicate that we want to use the Google Life Sciences API
- *–default-remote-prefix*: refers to the Google Storage bucket. The bucket name is "snakemake-testing-data" and the "subfolder" (or path) (not defined above) would be a subfolder, if needed.
- *–google-lifesciences-region*: the region that you want the instances to deploy to. Your storage bucket should be accessible from here, and your selection can have a small influence on the machine type selected.

Once you submit the job, you'll immediately see the familiar Snakemake console output, but with additional lines for inspecting google compute instances with gcloud:

```
Building DAG of jobs...
Unable to retrieve additional files from git. This is not a git repository.
Using shell: /bin/bash
Rules claiming more threads will be scaled down.
Job counts:
    count   jobs
    1       all
    1       bcftools_call
    2       bwa_map
    1       plot_quals
    2       samtools_index
    2       samtools_sort
    9

[Thu Apr 16 19:16:24 2020]
rule bwa_map:
    input: snakemake-testing-data/genome.fa, snakemake-testing-data/samples/B.fastq
    output: snakemake-testing-data/mapped_reads/B.bam
    jobid: 8
```

(continues on next page)

(continued from previous page)

```
    wildcards: sample=B
    resources: mem_mb=15360, disk_mb=128000

Get status with:
gcloud config set project snakemake-testing
gcloud beta lifesciences operations describe 13586583122112209762
gcloud beta lifesciences operations list
```

Take note of those last three lines to describe and list operations - this is how you get complete error and output logs for the run, which we will demonstrate using later.

And you'll see a block like that for each rule. Here is what the entire workflow looks like after completion:

```
Building DAG of jobs...
Unable to retrieve additional files from git. This is not a git repository.
Using shell: /bin/bash
Rules claiming more threads will be scaled down.
Job counts:
    count   jobs
    1       all
    1       bcftools_call
    2       bwa_map
    1       plot_quals
    2       samtools_index
    2       samtools_sort
    9

[Fri Apr 17 20:27:51 2020]
rule bwa_map:
    input: snakemake-testing-data/samples/B.fastq, snakemake-testing-data/genome.fa.
↪amb, snakemake-testing-data/genome.fa.ann, snakemake-testing-data/genome.fa.bwt,␣
↪snakemake-testing-data/genome.fa.pac, snakemake-testing-data/genome.fa.sa
    output: snakemake-testing-data/mapped_reads/B.bam
    jobid: 8
    wildcards: sample=B
    resources: mem_mb=15360, disk_mb=128000

Get status with:
gcloud config set project snakemake-testing
gcloud beta lifesciences operations describe projects/snakemake-testing/locations/us-
↪west2/operations/16135317625786219242
gcloud beta lifesciences operations list
[Fri Apr 17 20:31:16 2020]
Finished job 8.
1 of 9 steps (11%) done

[Fri Apr 17 20:31:16 2020]
rule bwa_map:
    input: snakemake-testing-data/samples/A.fastq, snakemake-testing-data/genome.fa.
↪amb, snakemake-testing-data/genome.fa.ann, snakemake-testing-data/genome.fa.bwt,␣
↪snakemake-testing-data/genome.fa.pac, snakemake-testing-data/genome.fa.sa
    output: snakemake-testing-data/mapped_reads/A.bam
    jobid: 7
    wildcards: sample=A
    resources: mem_mb=15360, disk_mb=128000

Get status with:
```

(continues on next page)

(continued from previous page)

```
gcloud config set project snakemake-testing
gcloud beta lifesciences operations describe projects/snakemake-testing/locations/us-
↪west2/operations/5458247376121133509
gcloud beta lifesciences operations list
[Fri Apr 17 20:34:30 2020]
Finished job 7.
2 of 9 steps (22%) done

[Fri Apr 17 20:34:30 2020]
rule samtools_sort:
    input: snakemake-testing-data/mapped_reads/B.bam
    output: snakemake-testing-data/sorted_reads/B.bam
    jobid: 4
    wildcards: sample=B
    resources: mem_mb=15360, disk_mb=128000

Get status with:
gcloud config set project snakemake-testing
gcloud beta lifesciences operations describe projects/snakemake-testing/locations/us-
↪west2/operations/13750029425473765929
gcloud beta lifesciences operations list
[Fri Apr 17 20:37:34 2020]
Finished job 4.
3 of 9 steps (33%) done

[Fri Apr 17 20:37:35 2020]
rule samtools_sort:
    input: snakemake-testing-data/mapped_reads/A.bam
    output: snakemake-testing-data/sorted_reads/A.bam
    jobid: 3
    wildcards: sample=A
    resources: mem_mb=15360, disk_mb=128000

Get status with:
gcloud config set project snakemake-testing
gcloud beta lifesciences operations describe projects/snakemake-testing/locations/us-
↪west2/operations/15643873965497084056
gcloud beta lifesciences operations list
[Fri Apr 17 20:40:37 2020]
Finished job 3.
4 of 9 steps (44%) done

[Fri Apr 17 20:40:38 2020]
rule samtools_index:
    input: snakemake-testing-data/sorted_reads/B.bam
    output: snakemake-testing-data/sorted_reads/B.bam.bai
    jobid: 6
    wildcards: sample=B
    resources: mem_mb=15360, disk_mb=128000

Get status with:
gcloud config set project snakemake-testing
gcloud beta lifesciences operations describe projects/snakemake-testing/locations/us-
↪west2/operations/6525320566174651173
gcloud beta lifesciences operations list
[Fri Apr 17 20:43:41 2020]
Finished job 6.
```

(continues on next page)

(continued from previous page)

```
5 of 9 steps (56%) done

[Fri Apr 17 20:43:41 2020]
rule samtools_index:
    input: snakemake-testing-data/sorted_reads/A.bam
    output: snakemake-testing-data/sorted_reads/A.bam.bai
    jobid: 5
    wildcards: sample=A
    resources: mem_mb=15360, disk_mb=128000

Get status with:
gcloud config set project snakemake-testing
gcloud beta lifesciences operations describe projects/snakemake-testing/locations/us-
↪west2/operations/9175497885319251567
gcloud beta lifesciences operations list
[Fri Apr 17 20:46:44 2020]
Finished job 5.
6 of 9 steps (67%) done

[Fri Apr 17 20:46:44 2020]
rule bcftools_call:
    input: snakemake-testing-data/genome.fa, snakemake-testing-data/sorted_reads/A.
↪bam, snakemake-testing-data/sorted_reads/B.bam, snakemake-testing-data/sorted_reads/
↪A.bam.bai, snakemake-testing-data/sorted_reads/B.bam.bai
    output: snakemake-testing-data/calls/all.vcf
    jobid: 2
    resources: mem_mb=15360, disk_mb=128000

Get status with:
gcloud config set project snakemake-testing
gcloud beta lifesciences operations describe projects/snakemake-testing/locations/us-
↪west2/operations/622600526583374352
gcloud beta lifesciences operations list
[Fri Apr 17 20:49:57 2020]
Finished job 2.
7 of 9 steps (78%) done

[Fri Apr 17 20:49:57 2020]
rule plot_quals:
    input: snakemake-testing-data/calls/all.vcf
    output: snakemake-testing-data/plots/quals.svg
    jobid: 1
    resources: mem_mb=15360, disk_mb=128000

Get status with:
gcloud config set project snakemake-testing
gcloud beta lifesciences operations describe projects/snakemake-testing/locations/us-
↪west2/operations/9350722561866518561
gcloud beta lifesciences operations list
[Fri Apr 17 20:53:10 2020]
Finished job 1.
8 of 9 steps (89%) done

[Fri Apr 17 20:53:10 2020]
localrule all:
    input: snakemake-testing-data/plots/quals.svg
    jobid: 0
```

(continues on next page)

(continued from previous page)

```
    resources: mem_mb=15360, disk_mb=128000

Downloading from remote: snakemake-testing-data/plots/quals.svg
Finished download.
[Fri Apr 17 20:53:10 2020]
Finished job 0.
9 of 9 steps (100%) done
Complete log: /home/vanessa/snakemake-work/tutorial/.snakemake/log/2020-04-17T202749.
↪218820.snakemake.log
```

We've finished the run, great! Let's inspect our results.

Step 4: View Results

The entirety of the log that was printed to the terminal will be available on your local machine where you submit the job in the hidden *.snakemake* folder under "log" and timestamped accordingly. If you look at the last line in the output above, you'll see the full path to this file.

You also might notice a line about downloading results:

```
Downloading from remote: snakemake-testing-data/plots/quals.svg
```

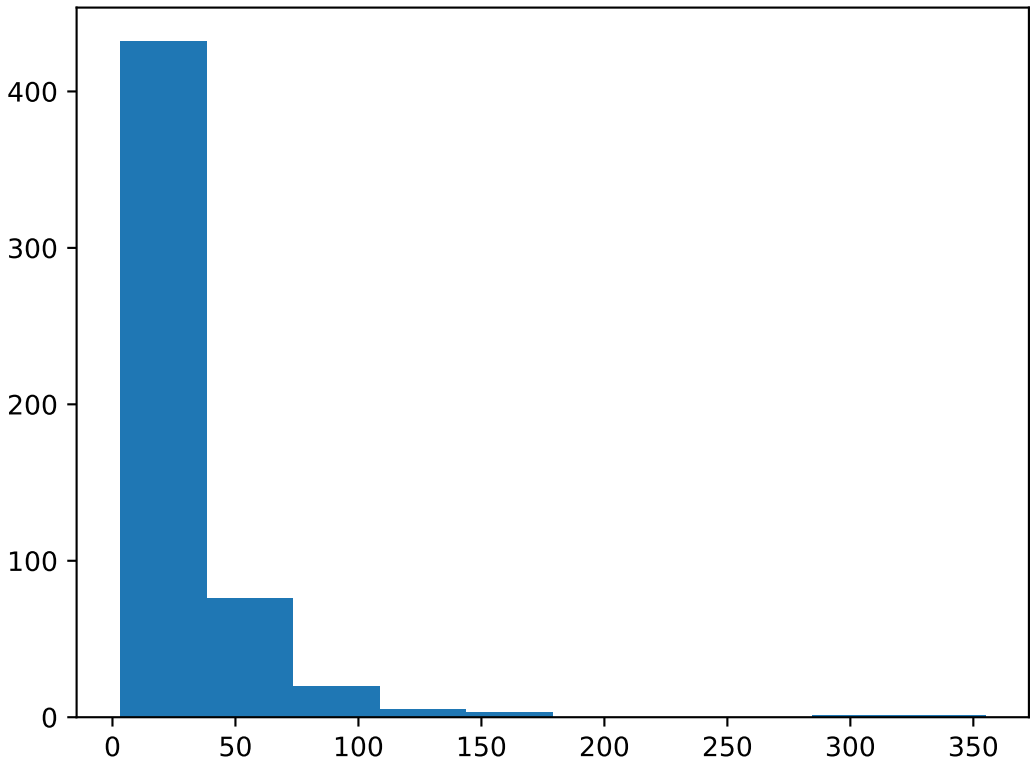
Since we defined this to be the target of our run

```
rule all:
    input:
        "plots/quals.svg"
```

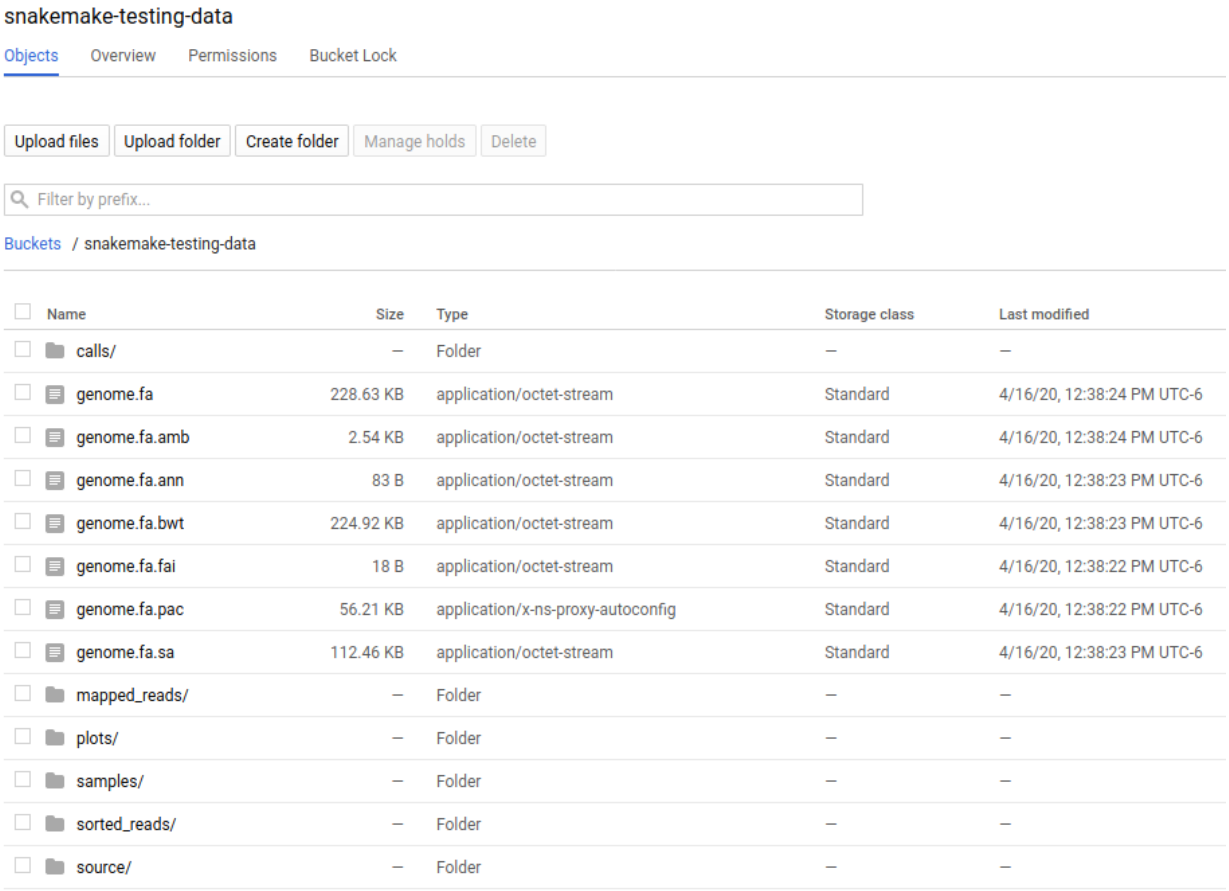
this fill is downloaded to our host too. Actually, you'll notice that paths in storage are mirrored on your filesystem (this is what the workers do too):

```
$ tree snakemake-testing-data/
snakemake-testing-data/
└── plots
    └── quals.svg
```

We can see the result of our run, quals.svg, below:



And if we look at the remote storage, we see that the result file (under plots) and intermediate results (under sorted_reads and calls) are saved there too!



Name	Size	Type	Storage class	Last modified
calls/	–	Folder	–	–
genome.fa	228.63 KB	application/octet-stream	Standard	4/16/20, 12:38:24 PM UTC-6
genome.fa.amb	2.54 KB	application/octet-stream	Standard	4/16/20, 12:38:24 PM UTC-6
genome.fa.ann	83 B	application/octet-stream	Standard	4/16/20, 12:38:23 PM UTC-6
genome.fa.bwt	224.92 KB	application/octet-stream	Standard	4/16/20, 12:38:23 PM UTC-6
genome.fa.fai	18 B	application/octet-stream	Standard	4/16/20, 12:38:22 PM UTC-6
genome.fa.pac	56.21 KB	application/x-ns-proxy-autoconfig	Standard	4/16/20, 12:38:22 PM UTC-6
genome.fa.sa	112.46 KB	application/octet-stream	Standard	4/16/20, 12:38:23 PM UTC-6
mapped_reads/	–	Folder	–	–
plots/	–	Folder	–	–
samples/	–	Folder	–	–
sorted_reads/	–	Folder	–	–
source/	–	Folder	–	–

The source folder contains a cache folder with archives that contain your working directories that are extracted on the worker instances. You can safely delete this folder, or keep it if you want to reproduce the run in the future.

Step 5: Debugging

Let's introduce an error (purposefully) into our Snakefile to practice debugging. Let's remove the conda environment.yaml file for the first rule, so we would expect that Snakemake won't be able to find the executables for bwa and samtools. In your Snakefile, change this:

```
rule bwa_map:
    input:
        fastq="samples/{sample}.fastq",
        idx=multiext("genome.fa", ".amb", ".ann", ".bwt", ".pac", ".sa")
    conda:
        "environment.yaml"
    output:
        "mapped_reads/{sample}.bam"
    params:
        idx=lambda w, input: os.path.splitext(input.idx[0])[0]
    shell:
        "bwa mem {params.idx} {input.fastq} | samtools view -Sb - > {output}"
```

to this:

```
rule bwa_map:
    input:
        fastq="samples/{sample}.fastq",
        idx=multiext("genome.fa", ".amb", ".ann", ".bwt", ".pac", ".sa")
    output:
        "mapped_reads/{sample}.bam"
    params:
        idx=lambda w, input: os.path.splitext(input.idx[0])[0]
    shell:
        "bwa mem {params.idx} {input.fastq} | samtools view -Sb - > {output}"
```

And then for the same command to run everything again, you would need to remove the plots, mapped_reads, and calls folders. Instead, we can make this request more easily by adding the argument *–forceall*:

```
snakemake --google-lifesciences --default-remote-prefix snakemake-testing-data --use-
↪conda --google-lifesciences-region us-west1 --forceall
```

Everything will start out okay as it did before, and it will pause on the first step when it's deploying the first container image. The last part of the log will look somethig like this:

```
[Fri Apr 17 22:01:38 2020]
rule bwa_map:
    input: snakemake-testing-data/samples/B.fastq, snakemake-testing-data/genome.fa.
↪amb, snakemake-testing-data/genome.fa.ann, snakemake-testing-data/genome.fa.bwt,␣
↪snakemake-testing-data/genome.fa.pac, snakemake-testing-data/genome.fa.sa
    output: snakemake-testing-data/mapped_reads/B.bam
    jobid: 8
    wildcards: sample=B
    resources: mem_mb=15360, disk_mb=128000

Get status with:
gcloud config set project snakemake-testing
gcloud beta lifesciences operations describe projects/snakemake-testing/locations/us/
↪operations/11698975339184312706
gcloud beta lifesciences operations list
```

Since we removed an important dependency to install libraries with conda, we are definitely going to hit an error! That looks like this:

```
[Fri Apr 17 22:03:08 2020]
Error in rule bwa_map:
    jobid: 8
    output: snakemake-testing-data/mapped_reads/B.bam
    shell:
        bwa mem snakemake-testing-data/genome.fa snakemake-testing-data/samples/B.
↪fastq | samtools view -Sb - > snakemake-testing-data/mapped_reads/B.bam
        (one of the commands exited with non-zero exit code; note that snakemake uses␣
↪bash strict mode!)
    jobid: 11698975339184312706

Shutting down, this might take some time.
```

Oh no! How do we debug it? The error above just indicates that "one of the commands exised with a non-zero exit code," and that isn't really enough to know what happened, and how to fix it. Debugging is actually quite simple, we can copy paste the gcloud command to describe our operation into the console. This will spit out an entire structure that shows every step of the rule running, from pulling a container, to downloading the working directory, to running the step.

```
gcloud beta lifesciences operations describe projects/snakemake-testing/locations/us/
↪operations/11698975339184312706
done: true
error:
  code: 9
  message: 'Execution failed: generic::failed_precondition: while running "snakejob-
↪bwa_map-8":
    unexpected exit status 1 was not ignored'
metadata:
  '@type': type.googleapis.com/google.cloud.lifesciences.v2beta.Metadata
  createTime: '2020-04-17T22:01:39.642966Z'
  endTime: '2020-04-17T22:02:59.149914114Z'
  events:
  - description: Worker released
    timestamp: '2020-04-17T22:02:59.149914114Z'
    workerReleased:
      instance: google-pipelines-worker-b1cdd36c743c3b477af8114d2511e37e
      zone: us-west1-c
  - description: 'Execution failed: generic::failed_precondition: while running
↪"snakejob-bwa_map-8":
      unexpected exit status 1 was not ignored'
    failed:
      cause: 'Execution failed: generic::failed_precondition: while running "snakejob-
↪bwa_map-8":
        unexpected exit status 1 was not ignored'
      code: FAILED_PRECONDITION
    timestamp: '2020-04-17T22:02:57.950752682Z'
  - description: Unexpected exit status 1 while running "snakejob-bwa_map-8"
    timestamp: '2020-04-17T22:02:57.842529458Z'
    unexpectedExitStatus:
      actionId: 1
      exitStatus: 1
  - containerStopped:
      actionId: 1
      exitStatus: 1
      stderr: |
        me.fa.bwt
        Finished download.
        /bin/bash: bwa: command not found
        /bin/bash: samtools: command not found
        [Fri Apr 17 22:02:57 2020]
        Error in rule bwa_map:
            jobid: 0
            output: snakemake-testing-data/mapped_reads/B.bam
            shell:
                bwa mem snakemake-testing-data/genome.fa snakemake-testing-data/
↪samples/B.fastq | samtools view -Sb - > snakemake-testing-data/mapped_reads/B.bam
                (one of the commands exited with non-zero exit code; note that␣
↪snakemake uses bash strict mode!)

        Removing output files of failed job bwa_map since they might be corrupted:
        snakemake-testing-data/samples/B.fastq, snakemake-testing-data/genome.fa.amb,␣
↪snakemake-testing-data/genome.fa.ann, snakemake-testing-data/genome.fa.bwt,␣
↪snakemake-testing-data/genome.fa.pac, snakemake-testing-data/genome.fa.sa,␣
↪snakemake-testing-data/mapped_reads/B.bam
        Shutting down, this might take some time.
        Exiting because a job execution failed. Look above for error message
```

(continues on next page)

(continued from previous page)

```
        Complete log: /workdir/.snakemake/log/2020-04-17T220254.129519.snakemake.log
    description: |-
      Stopped running "snakejob-bwa_map-8": exit status 1: me.fa.bwt
      Finished download.
      /bin/bash: bwa: command not found
      /bin/bash: samtools: command not found
      [Fri Apr 17 22:02:57 2020]
      Error in rule bwa_map:
          jobid: 0
          output: snakemake-testing-data/mapped_reads/B.bam
          shell:
              bwa mem snakemake-testing-data/genome.fa snakemake-testing-data/samples/
↪B.fastq | samtools view -Sb - > snakemake-testing-data/mapped_reads/B.bam
              (one of the commands exited with non-zero exit code; note that␣
↪snakemake uses bash strict mode!)

      Removing output files of failed job bwa_map since they might be corrupted:
      snakemake-testing-data/samples/B.fastq, snakemake-testing-data/genome.fa.amb,␣
↪snakemake-testing-data/genome.fa.ann, snakemake-testing-data/genome.fa.bwt,␣
↪snakemake-testing-data/genome.fa.pac, snakemake-testing-data/genome.fa.sa,␣
↪snakemake-testing-data/mapped_reads/B.bam
      Shutting down, this might take some time.
      Exiting because a job execution failed. Look above for error message
      Complete log: /workdir/.snakemake/log/2020-04-17T220254.129519.snakemake.log
    timestamp: '2020-04-17T22:02:57.842442588Z'
  - containerStarted:
      actionId: 1
    description: Started running "snakejob-bwa_map-8"
    timestamp: '2020-04-17T22:02:51.724433437Z'
  - description: Stopped pulling "snakemake/snakemake:v5.10.0"
    pullStopped:
      imageUri: snakemake/snakemake:v5.10.0
    timestamp: '2020-04-17T22:02:43.696978950Z'
  - description: Started pulling "snakemake/snakemake:v5.10.0"
    pullStarted:
      imageUri: snakemake/snakemake:v5.10.0
    timestamp: '2020-04-17T22:02:10.339950219Z'
  - description: Worker "google-pipelines-worker-b1cdd36c743c3b477af8114d2511e37e"
      assigned in "us-west1-c"
    timestamp: '2020-04-17T22:01:43.232858222Z'
    workerAssigned:
      instance: google-pipelines-worker-b1cdd36c743c3b477af8114d2511e37e
      machineType: n2-highmem-2
      zone: us-west1-c
  labels:
    app: snakemake
    name: snakejob-b346c449-9fd6-4f1e-8043-17c300cc9c0d-bwa_map-8
  pipeline:
    actions:
    - commands:
      - /bin/bash
      - -c
      - 'mkdir -p /workdir && cd /workdir && wget -O /download.py https://gist.
↪githubusercontent.com/vsoch/84886ef6469bedeeb9a79a4eb7aec0d1/raw/
↪181499f8f17163dcb2f89822079938cbfbd258cc/download.py
        && chmod +x /download.py && source activate snakemake || true && pip install
        crc32c && python /download.py download snakemake-testing-data source/cache/
```

(continues on next page)

(continued from previous page)

```
↪snakeworkdir-5f4f325b9ddb188d5da8bfab49d915f023509c0b1986eb72cb4a2540d7991c12.tar.gz
       /tmp/workdir.tar.gz && tar -xzvf /tmp/workdir.tar.gz && snakemake snakemake-
↪testing-data/mapped_reads/B.bam
       --snakefile Snakefile --force -j --keep-target-files --keep-remote --latency-
↪wait
       0 --attempt 1 --force-use-threads  --allowed-rules bwa_map --nocolor --notemp
       --no-hooks --nolock  --use-conda  --default-remote-provider GS --default-
↪remote-prefix
       snakemake-testing-data  --default-resources "mem_mb=15360" "disk_mb=128000" '
     containerName: snakejob-bwa_map-8
     imageUri: snakemake/snakemake:v5.10.0
     labels:
       app: snakemake
       name: snakejob-b346c449-9fd6-4f1e-8043-17c300cc9c0d-bwa_map-8
   resources:
     regions:
     - us-west1
     virtualMachine:
       bootDiskSizeGb: 135
       bootImage: projects/cos-cloud/global/images/family/cos-stable
       labels:
         app: snakemake
         goog-pipelines-worker: 'true'
       machineType: n2-highmem-2
       serviceAccount:
         email: default
         scopes:
         - https://www.googleapis.com/auth/cloud-platform
   timeout: 604800s
 startTime: '2020-04-17T22:01:43.232858222Z'
name: projects/411393320858/locations/us/operations/11698975339184312706
```

The log is hefty, so let's break it into pieces to talk about. Firstly, it's intended to be read from the bottom up if you want to see things in chronological order. The very bottom line is the unique id of the operation, and this is what you used (with the project identifier string, the number after projects, replaced with your project name) to query for the log. Let's look at the next section, *pipeline*. This was the specification built up by Snakemake and sent to the Google Life Sciences API as a request:

```
pipeline:
  actions:
  - commands:
    - /bin/bash
    - -c
    - 'mkdir -p /workdir && cd /workdir && wget -O /download.py https://gist.
↪githubusercontent.com/vsoch/84886ef6469bedeeb9a79a4eb7aec0d1/raw/
↪181499f8f17163dcb2f89822079938cbfbd258cc/download.py
      && chmod +x /download.py && source activate snakemake || true && pip install
      crc32c && python /download.py download snakemake-testing-data source/cache/
↪snakeworkdir-5f4f325b9ddb188d5da8bfab49d915f023509c0b1986eb72cb4a2540d7991c12.tar.gz
      /tmp/workdir.tar.gz && tar -xzvf /tmp/workdir.tar.gz && snakemake snakemake-
↪testing-data/mapped_reads/B.bam
      --snakefile Snakefile --force -j --keep-target-files --keep-remote --latency-
↪wait
      0 --attempt 1 --force-use-threads  --allowed-rules bwa_map --nocolor --notemp
      --no-hooks --nolock  --use-conda  --default-remote-provider GS --default-remote-
↪prefix
```

(continues on next page)

(continued from previous page)

```
      snakemake-testing-data  --default-resources "mem_mb=15360" "disk_mb=128000" '
    containerName: snakejob-bwa_map-8
    imageUri: snakemake/snakemake:v5.10.0
    labels:
      app: snakemake
      name: snakejob-b346c449-9fd6-4f1e-8043-17c300cc9c0d-bwa_map-8
  resources:
    regions:
    - us-west1
    virtualMachine:
      bootDiskSizeGb: 135
      bootImage: projects/cos-cloud/global/images/family/cos-stable
      labels:
        app: snakemake
        goog-pipelines-worker: 'true'
      machineType: n2-highmem-2
      serviceAccount:
        email: default
        scopes:
        - https://www.googleapis.com/auth/cloud-platform
  timeout: 604800s
startTime: '2020-04-17T22:01:43.232858222Z'
```

There is a lot of useful information here. Under *resources*:

- **virtualMachine** shows the **machineType** that should correspond to the instance type. You can specify a full name or prefix with *–machine-type-prefix* or "machine_type" defined under resources for a step. Since we didn't set any requirements, it chose a reasonable choice for us. This section also shows the size of the boot disk (in GB) and if you added hardware accelerators (GPU) they should show up here too.
- **regions** is the region that the instance was deployed in, which is important to know if you need to specify to run from a particular region. This parameter defalts to regions in the US, and can be modified with the *–google-lifesciences-regions* parameter.

Under *actions* you'll find a few important fields:

- **imageUri** is important to know to see the version of Snakemake (or another container base) that was used. You can customize this with *–container-image*, and it will default to the latest snakemake.
- **commands** are the commands run to execute the container (also known as the entrypoint). For example, if you wanted to bring up your own instance manually and pull the container defined by *imageUri*, you could execute the commands to the container (or shell inside and run them interactively) to interactively debug. Notice that the commands ends with a call to snakemake, and shows the arguments that are used. Make sure that this matches your expectation.

The next set of steps pertain to assigning the worker, pulling the container, and starting it. That looks something like this, and it's fairly straight forward. You can again see that earlier timestamps are on the bottom.

```
- containerStarted:
    actionId: 1
  description: Started running "snakejob-bwa_map-8"
  timestamp: '2020-04-17T22:02:51.724433437Z'
- description: Stopped pulling "snakemake/snakemake:v5.10.0"
  pullStopped:
    imageUri: snakemake/snakemake:v5.10.0
  timestamp: '2020-04-17T22:02:43.696978950Z'
- description: Started pulling "snakemake/snakemake:v5.10.0"
  pullStarted:
```

(continues on next page)

(continued from previous page)

```
    imageUri: snakemake/snakemake:v5.10.0
  timestamp: '2020-04-17T22:02:10.339950219Z'
- description: Worker "google-pipelines-worker-b1cdd36c743c3b477af8114d2511e37e"
    assigned in "us-west1-c"
  timestamp: '2020-04-17T22:01:43.232858222Z'
  workerAssigned:
    instance: google-pipelines-worker-b1cdd36c743c3b477af8114d2511e37e
    machineType: n2-highmem-2
    zone: us-west1-c
```

The next section, when the container is stopped, have the meat of the information that we need to debug! This is the step where there was a non-zero exit code.

```
- containerStopped:
    actionId: 1
    exitStatus: 1
    stderr: |
      me.fa.bwt
      Finished download.
      /bin/bash: bwa: command not found
      /bin/bash: samtools: command not found
      [Fri Apr 17 22:02:57 2020]
      Error in rule bwa_map:
          jobid: 0
          output: snakemake-testing-data/mapped_reads/B.bam
          shell:
              bwa mem snakemake-testing-data/genome.fa snakemake-testing-data/samples/
↪B.fastq | samtools view -Sb - > snakemake-testing-data/mapped_reads/B.bam
              (one of the commands exited with non-zero exit code; note that␣
↪snakemake uses bash strict mode!)

      Removing output files of failed job bwa_map since they might be corrupted:
      snakemake-testing-data/samples/B.fastq, snakemake-testing-data/genome.fa.amb,␣
↪snakemake-testing-data/genome.fa.ann, snakemake-testing-data/genome.fa.bwt,␣
↪snakemake-testing-data/genome.fa.pac, snakemake-testing-data/genome.fa.sa,␣
↪snakemake-testing-data/mapped_reads/B.bam
      Shutting down, this might take some time.
      Exiting because a job execution failed. Look above for error message
      Complete log: /workdir/.snakemake/log/2020-04-17T220254.129519.snakemake.log
  description: |-
    Stopped running "snakejob-bwa_map-8": exit status 1: me.fa.bwt
    Finished download.
    /bin/bash: bwa: command not found
    /bin/bash: samtools: command not found
    [Fri Apr 17 22:02:57 2020]
    Error in rule bwa_map:
        jobid: 0
        output: snakemake-testing-data/mapped_reads/B.bam
        shell:
            bwa mem snakemake-testing-data/genome.fa snakemake-testing-data/samples/B.
↪fastq | samtools view -Sb - > snakemake-testing-data/mapped_reads/B.bam
            (one of the commands exited with non-zero exit code; note that snakemake␣
↪uses bash strict mode!)

    Removing output files of failed job bwa_map since they might be corrupted:
    snakemake-testing-data/samples/B.fastq, snakemake-testing-data/genome.fa.amb,␣
```

(continues on next page)

(continued from previous page)

```
↪snakemake-testing-data/genome.fa.ann, snakemake-testing-data/genome.fa.bwt,␣
↪snakemake-testing-data/genome.fa.pac, snakemake-testing-data/genome.fa.sa,␣
↪snakemake-testing-data/mapped_reads/B.bam
    Shutting down, this might take some time.
    Exiting because a job execution failed. Look above for error message
    Complete log: /workdir/.snakemake/log/2020-04-17T220254.129519.snakemake.log
  timestamp: '2020-04-17T22:02:57.842442588Z'
```

Along with seeing the error in *stderr*, the description key holds the same error. We see what we would have seen if we were running the bwa mem command on our own command line, that the executables weren't found:

```
stderr: |
  me.fa.bwt
  Finished download.
  /bin/bash: bwa: command not found
  /bin/bash: samtools: command not found
```

But we shouldn't be surprised, we on purpose removed the environment file to install them! This is where you would read the error, and respond by updating your Snakefile with a fix.

Step 6: Adding a Log File

How might we do better at debugging in the future? The answer is to add a log file for each step, which is where any stderr will be written in the case of failure. For the same step above, we would update the rule to look like this:

```
rule bwa_map:
    input:
        fastq="samples/{sample}.fastq",
        idx=multiext("genome.fa", ".amb", ".ann", ".bwt", ".pac", ".sa")
    output:
        "mapped_reads/{sample}.bam"
    params:
        idx=lambda w, input: os.path.splitext(input.idx[0])[0]
    shell:
        "bwa mem {params.idx} {input.fastq} | samtools view -Sb - > {output}"
    log:
        "logs/bwa_map/{sample}.log"
```

In the above, we would write a log file to storage in a "subfolder" of the snakemake prefix located at "logs/bwa_map." The log file will be named according to the sample. You could also imagine a flatted structure with a path like *logs/bwa_map-{sample}.log*. It's up to you how you want to organize your output. This means that when you see the error appear in your terminal, you can quickly look at this log file instead of resorting to using the gcloud tool. It's generally good to remember when debugging that:

- You should not make assumptions about anything's existence. Use print statements to verify.
- The biggest errors tend to be syntax and/or path errors
- If you want to test a different snakemake container, you can use the *–container* flag.
- If the error is especially challenging, set up a small toy example that implements the most basic functionality that you want to achieve.
- If you need help, reach out to ask for it! If there is an issue with the Google Life Sciences workflow executor, please open an issue.
- It also sometimes helps to take a break from working on something, and coming back with fresh eyes.

4.4.2 Auto-scaling Azure Kubernetes cluster without shared filesystem

In this tutorial we will show how to execute a Snakemake workflow on an auto-scaling Azure Kubernetes cluster without a shared file-system. While Kubernetes is mainly known as microservice orchestration system with self-healing properties, we will use it here simply as auto-scaling compute orchestrator. One could use persistent volumes in Kubernetes as shared file system, but this adds an unnecessary level of complexity and most importantly costs. Instead we use cheap Azure Blob storage, which is used by Snakemake to automatically stage data in and out for every job.

Following the steps below you will

1. set up Azure Blob storage, download the Snakemake tutorial data and upload to Azure
2. then create an Azure Kubernetes (AKS) cluster
3. and finally run the analysis with Snakemake on the cluster

Setup

To go through this tutorial, you need the following software installed:

- Python ≥3.5
- Snakemake ≥5.17

You should install conda as outlined in the *tutorial*, and then install full snakemake with:

```
conda create -c bioconda -c conda-forge -n snakemake snakemake
```

Make sure that the `kubernetes` and `azure-storage-blob` modules are installed in this environment. Should they be missing install with:

```
pip install kubernetes
pip install azure-storage-blob
```

In addition you will need the Azure CLI command installed.

Create an Azure storage account and upload data

We will be starting from scratch, i.e. we will create a new resource group and storage account. You can obviously reuse existing resources instead.

```
# change the following names as required
# azure region where to run:
region=southeastasia
# name of the resource group to create:
resgroup=snakemaks-rg
# name of storage account to create (all lowercase, no hyphens etc.):
stgacct=snakemaksstg

# create a resource group with name and in region as defined above
az group create --name $resgroup --location $region
# create a general purpose storage account with cheapest SKU
az storage account create -n $stgacct -g $resgroup --sku Standard_LRS -l $region
```

Get a key for that account and save it as `stgkey` for later use:

```
stgkey=$(az storage account keys list -g $resgroup -n $storageacct | head -n1 | cut -
↪f 3)
```

Next, you will create a storage container (think: bucket) to upload the Snakemake tutorial data to:

```
az storage container create --resource-group $resgroup --account-name $stgacct \
    --account-key $stgkey --name snakemake-tutorial
cd /tmp
git clone https://github.com/snakemake/snakemake-tutorial-data.git
cd snakemake-tutorial-data
az storage blob upload-batch -d snakemake-tutorial --account-name $stgacct \
    --account-key $stgkey -s data/ --destination-path data
```

We are using *az storage blob* for uploading, because that *az* is already installed. A more efficient way of uploading would be to use azcopy.

Create an auto-scaling Kubernetes cluster

```
# change the cluster name as you like
clustername=snakemaks-aks
az aks create --resource-group $resgroup --name $clustername \
    --vm-set-type VirtualMachineScaleSets --load-balancer-sku standard --enable-
↪cluster-autoscaler \
    --node-count 1 --min-count 1 --max-count 3 --node-vm-size Standard_D3_v2
```

There is a lot going on here, so let's unpack it: this creates an auto-scaling Kubernetes cluster (`--enable-cluster-autoscaler`) called `$clustername` (i.e. `snakemaks-aks`), which starts out with one node (`--node-count 1`) and has a maximum of three nodes (`--min-count 1 --max-count 3`). For real world applications you will want to increase the maximum count and also increase the VM size. You could for example choose a large instance from the DSv2 series and add a larger disk with (`--node-osdisk-size`) if needed. See here for more info on Linux VM sizes.

Note, if you are creating the cluster in the Azure portal, click on the ellipsis under node-pools to find the auto-scaling option.

Next, let's fetch the credentials for this cluster, so that we can actually interact with it.

```
az aks get-credentials --resource-group $resgroup --name $clustername
# print basic cluster info
kubectl cluster-info
```

Running the workflow

Below we will task Snakemake to install software on the fly with conda. For this we need a Snakefile with corresponding conda environment yaml files. You can download the package containing all those files here. After downloading, unzip it and cd into the newly created directory.

```
$ cd /tmp
$ unzip ~/Downloads/snakedir.zip
$ cd snakedir
$ find .
.
./Snakefile
./envs
./envs/calling.yaml
./envs/mapping.yaml
```

Now, we will need to setup the credentials that allow the Kubernetes nodes to read and write from blob storage. For the AzBlob storage provider in Snakemake this is done through the environment variables `AZ_BLOB_ACCOUNT_URL` and optionally `AZ_BLOB_CREDENTIAL`. See the documentation for more info. `AZ_BLOB_ACCOUNT_URL` takes the form `https://<accountname>.blob.core.windows.net` and may also contain a shared access signature (SAS), which is a powerful way to define fine grained and even time controlled access to storage on Azure. The SAS can be part of the URL, but if it's missing, then you can set it with `AZ_BLOB_CREDENTIAL` or alternatively use the storage account key. To keep things simple we'll use the storage key here, since we already have it available, but a SAS is generally more powerful. We'll pass those variables on to the Kubernetes with `--envvars` (see below).

Now you are ready to run the analysis:

```
export AZ_BLOB_ACCOUNT_URL="https://${stgacct}.blob.core.windows.net"
export AZ_BLOB_CREDENTIAL="$stgkey"
snakemake --kubernetes \
    --default-remote-prefix snakemake-tutorial --default-remote-provider AzBlob \
    --envvars AZ_BLOB_ACCOUNT_URL AZ_BLOB_CREDENTIAL --use-conda --jobs 3
```

This will use the default Snakemake image from Dockerhub. If you would like to use your own, make sure that the image contains the same Snakemake version as installed locally and also supports Azure Blob storage. If you plan to use your own image hosted on

> Azure Container Registries (ACR), make sure to attach the ACR to your Kubernetes cluster. See here for more info.

While Snakemake is running the workflow, it prints handy debug statements per job, e.g.:

```
kubectl describe pod snakejob-c4d9bf9e-9076-576b-a1f9-736ec82afc64
kubectl logs snakejob-c4d9bf9e-9076-576b-a1f9-736ec82afc64
```

With these you can also follow the scale-up of the cluster:

```
Events:
Type     Reason             Age                From                Message
----     ------             ----               ----                -------
Warning  FailedScheduling   60s (x3 over 62s)  default-scheduler   0/1 nodes are␣
↪available: 1 Insufficient cpu.
Normal   TriggeredScaleUp   50s                cluster-autoscaler  pod triggered␣
↪scale-up: [{aks-nodepool1-17839284-vmss 1->3 (max: 3)}]
```

After a while you will see three nodes (each running one BWA job), which was defined as the maximum above while creating your Kubernetes cluster:

```
$ kubectl get nodes
NAME                                STATUS   ROLES   AGE   VERSION
aks-nodepool1-17839284-vmss000000   Ready    agent   74m   v1.15.11
aks-nodepool1-17839284-vmss000001   Ready    agent   11s   v1.15.11
aks-nodepool1-17839284-vmss000002   Ready    agent   62s   v1.15.11
```

To get detailed information including historical data about used resources, check Insights in the Azure portal under your AKS cluster Monitoring/Insights. The alternative is an instant snapshot on the command line:

```
$ kubectl top node
NAME                                CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
aks-nodepool1-17839284-vmss000000   217m         5%     1796Mi          16%
aks-nodepool1-17839284-vmss000001   1973m        51%    529Mi           4%
aks-nodepool1-17839284-vmss000002   698m         18%    1485Mi          13%
```

After completion all results including logs can be found in the blob container. You will also find results listed in the first Snakefile target downloaded to the working directoy.

```
$ find snakemake-tutorial/
snakemake-tutorial/
snakemake-tutorial/calls
snakemake-tutorial/calls/all.vcf


$ az storage blob list  --container-name snakemake-tutorial --account-name $stgacct --
↪account-key $stgkey -o table
Name                      Blob Type    Blob Tier    Length    Content Type            ␣
↪          Last Modified               Snapshot
------------------------  -----------  -----------  --------  ------------------------
↪--------  -------------------------  ----------
calls/all.vcf             BlockBlob    Hot          90986     application/octet-stream␣
↪          2020-06-08T05:11:31+00:00
data/genome.fa            BlockBlob    Hot          234112    application/octet-stream␣
↪          2020-06-08T03:26:54+00:00
# etc.
logs/mapped_reads/A.log   BlockBlob    Hot          346       application/octet-stream␣
↪          2020-06-08T04:59:50+00:00
mapped_reads/A.bam        BlockBlob    Hot          2258058   application/octet-stream␣
↪          2020-06-08T04:59:50+00:00
sorted_reads/A.bam        BlockBlob    Hot          2244660   application/octet-stream␣
↪          2020-06-08T05:03:41+00:00
sorted_reads/A.bam.bai    BlockBlob    Hot          344       application/octet-stream␣
↪          2020-06-08T05:06:25+00:00
# same for samples B and C
```

Now that the execution is complete, the AKS cluster will scale down automatically. If you are not planning to run anything else, it makes sense to shut down it down entirely:

```
az aks delete --name akscluster --resource-group $resgroup
```

4.4.3 Flux Tutorial

Setup

To go through this tutorial, you need the following software installed:

- Docker

Flux-framework is a flexible resource scheduler that can work on both high performance computing systems and cloud (e.g., Kubernetes). Since it is more modern (e.g., has an official Python API) we define it under a cloud resource. For this example, we will show you how to set up a "single node" local Flux container to interact with snakemake. You can use the Dockerfile in examples/flux that will provide a container with Flux and snakemake Note that we install from source and bind to `/home/fluxuser/snakemake` with the intention of being able to develop (if desired). First, build the container:

```
$ docker build -t flux-snake .
```

And then you can run the container with or without any such bind:

```
$ docker run -it --rm flux-snake
```

Once you shelled into the container, you can view and start a Flux instance:

```
$ flux getattr size
$ flux start --test-size=4
$ flux getattr size
```

And see resources available:

```
$ flux resource status
STATUS NNODES NODELIST
 avail      4 5a74dc238d[98,98,98,98]
```

Resources

Flux currently has support for `runtime`, which should be set to a number (seconds) and defaults to 0, meaning unlimited runtime. Flux currently does not support `mem_mb` or `disk_mb`.

Run Snakemake

Now let's run Snakemake with the Flux executor. There is an example `Snakefile` in the flux examples folder that will show running a "Hello World!" example, and this file should be in your fluxuser home:

```
$ ls
Snakefile  snakemake
```

Here is how to run the workflow:

```
$ snakemake --flux --jobs=1
```

The flags above refer to:

- `--flux`: tell Snakemake to use the flux executor

Once you submit the job, you'll immediately see the familiar Snakemake console output. The jobs happen very quickly, but the default wait time between checks is 10 seconds so it will take a bit longer.

```
Building DAG of jobs...
Using shell: /usr/bin/bash
Provided cores: 1 (use --cores to define parallelism)
Rules claiming more threads will be scaled down.
Job stats:
job                         count    min threads    max threads
------------------------  -------  -------------  -------------
all                             1              1              1
multilingual_hello_world        2              1              1
total                           3              1              1

Select jobs to execute...

[Fri Aug 12 21:09:32 2022]
rule multilingual_hello_world:
    output: hello/world.txt
    jobid: 1
    reason: Missing output files: hello/world.txt
    wildcards: greeting=hello
    resources: tmpdir=/tmp
```

(continues on next page)

(continued from previous page)

```
Checking status for job f3sWJLhD
[Fri Aug 12 21:09:42 2022]
Finished job 1.
1 of 3 steps (33%) done
Select jobs to execute...

[Fri Aug 12 21:09:42 2022]
rule multilingual_hello_world:
    output: hola/world.txt
    jobid: 2
    reason: Missing output files: hola/world.txt
    wildcards: greeting=hola
    resources: tmpdir=/tmp

Checking status for job f8JAY1Kd
[Fri Aug 12 21:09:52 2022]
Finished job 2.
2 of 3 steps (67%) done
Select jobs to execute...

[Fri Aug 12 21:09:52 2022]
localrule all:
    input: hello/world.txt, hola/world.txt
    jobid: 0
    reason: Input files updated by another job: hola/world.txt, hello/world.txt
    resources: tmpdir=/tmp

[Fri Aug 12 21:09:52 2022]
Finished job 0.
3 of 3 steps (100%) done
Complete log: .snakemake/log/2022-08-12T210932.564786.snakemake.log
```

At this point you can inspect the local directory to see your job output!

```
$ ls
Snakefile  hello  hola
$ cat hello/world.txt
hello, World!
```

Flux Without Shared Filesystem

By default, the Flux executor assumes a shared filesystem. If this isn't the case, you can add the `--no-shared-fs` flag, which will tell Snakemake that Flux is running without a shared filesystem.

```
$ snakemake --flux --jobs=1 --no-shared-fs
```

See the flux documentation for more detail. For now, let's try interacting with flux via snakemake via the Flux Python Bindings.

The code for this example is provided in (examples/flux)

4.5 Best practices

- Snakemake (>=5.11) comes with a code quality checker (a so called linter), that analyzes your workflow and highlights issues that should be solved in order to follow best practices, achieve maximum readability, and reproducibility. The linter can be invoked with

```
snakemake --lint
```

  given that a `Snakefile` or `workflow/Snakefile` is accessible from your working directory. It is **highly recommended** to run the linter before publishing any workflow, asking questions on Stack Overflow or filing issues on Github.

- There is an automatic formatter for Snakemake workflows, called Snakefmt, which should be applied to any Snakemake workflow before publishing it.
- When publishing your workflow in a Github repository, it is a good idea to add some minimal test data and configure Github Actions for continuously testing the workflow on each new commit. For this purpose, we provide predefined Github actions for both running tests and linting here, as well as formatting here.
- For publishing and distributing a Snakemake workflow, it is a good idea to stick to a *standardized structure* that is expected by frequent users of Snakemake. The Snakemake workflow catalog automatically lists Snakemake workflows hosted on Github if they follow certain rules. By complying to these rules you can make your workflow more discoverable and even automate its usage documentation (see "Standardized usage").
- Configuration of a workflow should be handled via *config files* and, if needed, tabular configuration like sample sheets (either via Pandas or PEPs). Use such configuration for metadata and experiement information, **not for runtime specific configuration** like threads, resources and output folders. For those, just rely on Snakemake's CLI arguments like `--set-threads`, `--set-resources`, `--set-default-resources`, and `--directory`. This makes workflows more readable, scalable, and portable.
- Try to keep filenames short (thus easier on the eye), but informative. Avoid mixing of too many special characters (e.g. decide whether to use _ or − as a separator and do that consistently throughout the workflow).
- Try to keep Python code like helper functions separate from rules (e.g. in a `workflow/rules/common.smk` file). This way, you help non-experts to read the workflow without needing to parse internals that are irrelevant for them. The helper function names should be chosen in a way that makes them sufficiently informative without looking at their content. Also avoid `lambda` expressions inside of rules.
- Make use of Snakemake wrappers whenever possible. Consider contributing to the wrapper repo whenever you have a rule that reoccurs in at least two of your workflows.

4.6 Command line interface

This part of the documentation describes the `snakemake` executable. Snakemake is primarily a command-line tool, so the `snakemake` executable is the primary way to execute, debug, and visualize workflows.

4.6.1 Important environment variables

Snakemake caches source files for performance and reproducibility. The location of this cache is determined by the appdirs package. If you want to change the location on a unix/linux system, you can define an override path via the environment variable `XDG_CACHE_HOME`.

4.6.2 Useful Command Line Arguments

If called with the number of cores to use, i.e.

```
$ snakemake --cores 1
```

Snakemake tries to execute the workflow specified in a file called `Snakefile` in the same directory (the Snakefile can be given via the parameter `-s`).

By issuing

```
$ snakemake -n
```

a dry-run can be performed. This is useful to test if the workflow is defined properly and to estimate the amount of needed computation. Further, the reason for each rule execution can be printed via

```
$ snakemake -n -r
```

Importantly, Snakemake can automatically determine which parts of the workflow can be run in parallel. By specifying more than one available core, i.e.

```
$ snakemake --cores 4
```

one can tell Snakemake to use up to 4 cores and solve a binary knapsack problem to optimize the scheduling of jobs. If the number is omitted (i.e., only `--cores` is given), the number of used cores is determined as the number of available CPU cores in the machine.

Snakemake workflows usually define the number of used threads of certain rules. Sometimes, it makes sense to overwrite the defaults given in the workflow definition. This can be done by using the `--set-threads` argument, e.g.,

```
$ snakemake --cores 4 --set-threads myrule=2
```

would overwrite whatever number of threads has been defined for the rule `myrule` and use `2` instead. Similarly, it is possible to overwrite other resource definitions in rules, via

```
$ snakemake --cores 4 --set-resources myrule:partition="foo"
```

Both mechanisms can be particularly handy when used in combination with *cluster execution*.

Dealing with very large workflows

If your workflow has a lot of jobs, Snakemake might need some time to infer the dependencies (the job DAG) and which jobs are actually required to run. The major bottleneck involved is the filesystem, which has to be queried for existence and modification dates of files. To overcome this issue, Snakemake allows to run large workflows in batches. This way, fewer files have to be evaluated at once, and therefore the job DAG can be inferred faster. By running

```
$ snakemake --cores 4 --batch myrule=1/3
```

you instruct to only compute the first of three batches of the inputs of the rule `myrule`. To generate the second batch, run

```
$ snakemake --cores 4 --batch myrule=2/3
```

Finally, when running

```
$ snakemake --cores 4 --batch myrule=3/3
```

Snakemake will process beyond the rule `myrule`, because all of its input files have been generated, and complete the workflow. Obviously, a good choice of the rule to perform the batching is a rule that has a lot of input files and upstream jobs, for example a central aggregation step within your workflow. We advice all workflow developers to inform potential users of the best suited batching rule.

4.6.3 Profiles

Adapting Snakemake to a particular environment can entail many flags and options. Therefore, since Snakemake 4.1, it is possible to specify a configuration profile to be used to obtain default options:

```
$ snakemake --profile myprofile
```

Here, a folder `myprofile` is searched in per-user and global configuration directories (on Linux, this will be `$HOME/.config/snakemake` and `/etc/xdg/snakemake`, you can find the answer for your system via `snakemake --help`). Alternatively, an absolute or relative path to the folder can be given. The default profile to use when no `--profile` argument is specified can also be set via the environment variable `SNAKEMAKE_PROFILE`, e.g. by specifying `export SNAKEMAKE_PROFILE=myprofile` in your `~/.bashrc` or the system wide shell defaults.

The profile folder is expected to contain a file `config.yaml` that defines default values for the Snakemake command line arguments. For example, the file

```
cluster: qsub
jobs: 100
```

would setup Snakemake to always submit to the cluster via the `qsub` command, and never use more than 100 parallel jobs in total. The profile can be used to set a default for each option of the Snakemake command line interface. For this, option `--someoption` becomes `someoption:` in the profile. If options accept multiple arguments these must be given as YAML list in the profile. Under https://github.com/snakemake-profiles/doc, you can find publicly available profiles. Feel free to contribute your own.

The profile folder can additionally contain auxilliary files, e.g., jobscripts, or any kind of wrappers. See https://github.com/snakemake-profiles/doc for examples.

4.6.4 Visualization

To visualize the workflow, one can use the option `--dag`. This creates a representation of the DAG in the graphviz dot language which has to be postprocessed by the graphviz tool `dot`. E.g. to visualize the DAG that would be executed, you can issue:

```
$ snakemake --dag | dot | display
```

For saving this to a file, you can specify the desired format:

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

To visualize the whole DAG regardless of the eventual presence of files, the `forceall` option can be used:

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

Of course the visual appearance can be modified by providing further command line arguments to `dot`.

Note: The DAG is printed in DOT format straight to the standard output, along with other `print` statements you may have in your Snakefile. Make sure to comment these other `print` statements so that `dot` can build a visual representation of your DAG.

4.6.5 All Options

All command line options can be printed by calling `snakemake -h`.

Snakemake is a Python based language and execution environment for GNU Make-like workflows.

```
usage: snakemake [-h] [--dry-run] [--profile PROFILE] [--cache [RULE ...]]
                 [--snakefile FILE] [--cores [N]] [--jobs [N]]
                 [--local-cores N] [--resources [NAME=INT ...]]
                 [--set-threads RULE=THREADS [RULE=THREADS ...]]
                 [--max-threads MAX_THREADS]
                 [--set-resources RULE:RESOURCE=VALUE [RULE:RESOURCE=VALUE ...]]
                 [--set-scatter NAME=SCATTERITEMS [NAME=SCATTERITEMS ...]]
                 [--set-resource-scopes RESOURCE=[global|local]
                 [RESOURCE=[global|local] ...]]
                 [--default-resources [NAME=INT ...]]
                 [--preemption-default PREEMPTION_DEFAULT]
                 [--preemptible-rules PREEMPTIBLE_RULES [PREEMPTIBLE_RULES ...]]
                 [--config [KEY=VALUE ...]] [--configfile FILE [FILE ...]]
                 [--envvars VARNAME [VARNAME ...]] [--directory DIR] [--touch]
                 [--keep-going]
                 [--rerun-triggers {mtime,params,input,software-env,code} [{mtime,
↪params,input,software-env,code} ...]]
                 [--force] [--forceall] [--forcerun [TARGET ...]]
                 [--prioritize TARGET [TARGET ...]]
                 [--batch RULE=BATCH/BATCHES] [--until TARGET [TARGET ...]]
                 [--omit-from TARGET [TARGET ...]] [--rerun-incomplete]
                 [--shadow-prefix DIR] [--scheduler [{ilp,greedy}]]
                 [--wms-monitor [WMS_MONITOR]]
                 [--wms-monitor-arg [NAME=VALUE ...]]
                 [--scheduler-ilp-solver {}]
                 [--scheduler-solver-path SCHEDULER_SOLVER_PATH]
                 [--conda-base-path CONDA_BASE_PATH] [--no-subworkflows]
```

(continues on next page)

(continued from previous page)

```
                [--groups GROUPS [GROUPS ...]]
                [--group-components GROUP_COMPONENTS [GROUP_COMPONENTS ...]]
                [--report [FILE]] [--report-stylesheet CSSFILE]
                [--draft-notebook TARGET] [--edit-notebook TARGET]
                [--notebook-listen IP:PORT] [--lint [{text,json}]]
                [--generate-unit-tests [TESTPATH]] [--containerize]
                [--export-cwl FILE] [--list] [--list-target-rules] [--dag]
                [--rulegraph] [--filegraph] [--d3dag] [--summary]
                [--detailed-summary] [--archive FILE]
                [--cleanup-metadata FILE [FILE ...]] [--cleanup-shadow]
                [--skip-script-cleanup] [--unlock] [--list-version-changes]
                [--list-code-changes] [--list-input-changes]
                [--list-params-changes] [--list-untracked]
                [--delete-all-output] [--delete-temp-output]
                [--bash-completion] [--keep-incomplete] [--drop-metadata]
                [--version] [--reason] [--gui [PORT]] [--printshellcmds]
                [--debug-dag] [--stats FILE] [--nocolor]
                [--quiet [{progress,rules,all} ...]] [--print-compilation]
                [--verbose] [--force-use-threads] [--allow-ambiguity]
                [--nolock] [--ignore-incomplete]
                [--max-inventory-time SECONDS] [--latency-wait SECONDS]
                [--wait-for-files [FILE ...]] [--wait-for-files-file FILE]
                [--notemp] [--all-temp] [--keep-remote] [--keep-target-files]
                [--allowed-rules ALLOWED_RULES [ALLOWED_RULES ...]]
                [--target-jobs TARGET_JOBS [TARGET_JOBS ...]]
                [--local-groupid LOCAL_GROUPID]
                [--max-jobs-per-second MAX_JOBS_PER_SECOND]
                [--max-status-checks-per-second MAX_STATUS_CHECKS_PER_SECOND]
                [-T RETRIES] [--attempt ATTEMPT]
                [--wrapper-prefix WRAPPER_PREFIX]
                [--default-remote-provider {S3,GS,FTP,SFTP,S3Mocked,gfal,gridftp,
↪iRODS,AzBlob,XRootD}]
                [--default-remote-prefix DEFAULT_REMOTE_PREFIX]
                [--no-shared-fs] [--greediness GREEDINESS] [--no-hooks]
                [--overwrite-shellcmd OVERWRITE_SHELLCMD] [--debug]
                [--runtime-profile FILE] [--mode {0,1,2}]
                [--show-failed-logs] [--log-handler-script FILE]
                [--log-service {none,slack,wms}] [--slurm]
                [--cluster CMD | --cluster-sync CMD | --drmaa [ARGS]]
                [--cluster-config FILE] [--immediate-submit]
                [--jobscript SCRIPT] [--jobname NAME]
                [--cluster-status CLUSTER_STATUS]
                [--cluster-cancel CLUSTER_CANCEL]
                [--cluster-cancel-nargs CLUSTER_CANCEL_NARGS]
                [--cluster-sidecar CLUSTER_SIDECAR] [--drmaa-log-dir DIR]
                [--kubernetes [NAMESPACE]] [--container-image IMAGE]
                [--k8s-cpu-scalar FLOAT] [--tibanna]
                [--tibanna-sfn TIBANNA_SFN] [--precommand PRECOMMAND]
                [--tibanna-config TIBANNA_CONFIG [TIBANNA_CONFIG ...]]
                [--google-lifesciences]
                [--google-lifesciences-regions GOOGLE_LIFESCIENCES_REGIONS [GOOGLE_
↪LIFESCIENCES_REGIONS ...]]
                [--google-lifesciences-location GOOGLE_LIFESCIENCES_LOCATION]
                [--google-lifesciences-keep-cache] [--flux] [--tes URL]
                [--use-conda] [--conda-not-block-search-path-envvars]
                [--list-conda-envs] [--conda-prefix DIR]
                [--conda-cleanup-envs]
```

(continues on next page)

(continued from previous page)

```
[--conda-cleanup-pkgs [{tarballs,cache}]]
[--conda-create-envs-only] [--conda-frontend {conda,mamba}]
[--use-singularity] [--singularity-prefix DIR]
[--singularity-args ARGS] [--cleanup-containers]
[--use-envmodules]
[target ...]
```

EXECUTION

target Targets to build. May be rules or files.

--dry-run, --dryrun, -n Do not execute anything, and display what would be done. If you have a very large workflow, use –dry-run –quiet to just print a summary of the DAG of jobs.

Default: False

--profile Name of profile to use for configuring Snakemake. Snakemake will search for a corresponding folder in /etc/xdg/snakemake and /build-dir/.config/snakemake. Alternatively, this can be an absolute or relative path. The profile folder has to contain a file 'config.yaml'. This file can be used to set default values for command line options in YAML format. For example, '–cluster qsub' becomes 'cluster: qsub' in the YAML file. Profiles can be obtained from https://github.com/snakemake-profiles. The profile can also be set via the environment variable $SNAKEMAKE_PROFILE.

--cache Store output files of given rules in a central cache given by the environment variable $SNAKEMAKE_OUTPUT_CACHE. Likewise, retrieve output files of the given rules from this cache if they have been created before (by anybody writing to the same cache), instead of actually executing the rules. Output files are identified by hashing all steps, parameters and software stack (conda envs or containers) needed to create them.

--snakefile, -s The workflow definition in form of a snakefile.Usually, you should not need to specify this. By default, Snakemake will search for 'Snakefile', 'snakefile', 'workflow/Snakefile', 'workflow/snakefile' beneath the current working directory, in this order. Only if you definitely want a different layout, you need to use this parameter.

--cores, -c Use at most N CPU cores/jobs in parallel. If N is omitted or 'all', the limit is set to the number of available CPU cores. In case of cluster/cloud execution, this argument sets the maximum number of cores requested from the cluster or cloud scheduler. (See https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#resources-remote-execution for more info)This number is available to rules via workflow.cores.

--jobs, -j Use at most N CPU cluster/cloud jobs in parallel. For local execution this is an alias for –cores. Note: Set to 'unlimited' in case, this does not play a role.

--local-cores In cluster/cloud mode, use at most N cores of the host machine in parallel (default: number of CPU cores of the host). The cores are used to execute local rules. This option is ignored when not in cluster/cloud mode.

Default: 2

--resources, --res Define additional resources that shall constrain the scheduling analogously to –cores (see above). A resource is defined as a name and an integer value. E.g. –resources mem_mb=1000. Rules can use resources by defining the resource

keyword, e.g. resources: mem_mb=600. If now two rules require 600 of the resource 'mem_mb' they won't be run in parallel by the scheduler. In cluster/cloud mode, this argument will also constrain the amount of resources requested from the server. (See https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#resources-remote-execution for more info)

--set-threads Overwrite thread usage of rules. This allows to fine-tune workflow parallelization. In particular, this is helpful to target certain cluster nodes by e.g. shifting a rule to use more, or less threads than defined in the workflow. Thereby, THREADS has to be a positive integer, and RULE has to be the name of the rule.

--max-threads Define a global maximum number of threads available to any rule. Rules requesting more threads (via the threads keyword) will have their values reduced to the maximum. This can be useful when you want to restrict the maximum number of threads without modifying the workflow definition or overwriting rules individually with –set-threads.

--set-resources Overwrite resource usage of rules. This allows to fine-tune workflow resources. In particular, this is helpful to target certain cluster nodes by e.g. defining a certain partition for a rule, or overriding a temporary directory. Thereby, VALUE has to be a positive integer or a string, RULE has to be the name of the rule, and RESOURCE has to be the name of the resource.

--set-scatter Overwrite number of scatter items of scattergather processes. This allows to fine-tune workflow parallelization. Thereby, SCATTERITEMS has to be a positive integer, and NAME has to be the name of the scattergather process defined via a scattergather directive in the workflow.

--set-resource-scopes Overwrite resource scopes. A scope determines how a constraint is reckoned in cluster execution. With RESOURCE=local, a constraint applied to RESOURCE using –resources will be considered the limit for each group submission. With RESOURCE=global, the constraint will apply across all groups cumulatively. By default, only *mem_mb* and *disk_mb* are considered local, all other resources are global. This may be modified in the snakefile using the *resource_scopes:* directive. Note that number of threads, specified via –cores, is always considered local. (See https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#resources-remote-execution for more info)

--default-resources, --default-res Define default values of resources for rules that do not define their own values. In addition to plain integers, python expressions over inputsize are allowed (e.g. '2*input.size_mb'). The inputsize is the sum of the sizes of all input files of a rule. By default, Snakemake assumes a default for mem_mb, disk_mb, and tmpdir (see below). This option allows to add further defaults (e.g. account and partition for slurm) or to overwrite these default values. The defaults are 'mem_mb=max(2*input.size_mb, 1000)', 'disk_mb=max(2*input.size_mb, 1000)' (i.e., default disk and mem usage is twice the input file size but at least 1GB), and the system temporary directory (as given by $TMPDIR, $TEMP, or $TMP) is used for the tmpdir resource. The tmpdir resource is automatically used by shell commands, scripts and wrappers to store temporary data (as it is mirrored into $TMPDIR, $TEMP, and $TMP for the executed subprocesses). If this argument is not specified at all, Snakemake just uses the tmpdir resource as outlined above.

--preemption-default A preemptible instance can be requested when using the Google Life Sciences API. If you set a –preemption-default,all rules will be subject to the default. Specifically, this integer is the number of restart attempts that will be made given that the instance is killed unexpectedly. Note that preemptible instances have a maximum running time of 24 hours. If you want to set preemptible instances for only a subset

of rules, use –preemptible-rules instead.

--preemptible-rules A preemptible instance can be requested when using the Google Life Sciences API. If you want to use these instances for a subset of your rules, you can use –preemptible-rules and then specify a list of rule and integer pairs, where each integer indicates the number of restarts to use for the rule's instance in the case that the instance is terminated unexpectedly. –preemptible-rules can be used in combination with –preemption-default, and will take priority. Note that preemptible instances have a maximum running time of 24. If you want to apply a consistent number of retries across all your rules, use –premption-default instead. Example: snakemake –preemption-default 10 –preemptible-rules map_reads=3 call_variants=0

--config, -C Set or overwrite values in the workflow config object. The workflow config object is accessible as variable config inside the workflow. Default values can be set by providing a JSON file (see Documentation).

--configfile, --configfiles Specify or overwrite the config file of the workflow (see the docs). Values specified in JSON or YAML format are available in the global config dictionary inside the workflow. Multiple files overwrite each other in the given order. Thereby missing keys in previous config files are extended by following configfiles. Note that this order also includes a config file defined in the workflow definition itself (which will come first).

--envvars Environment variables to pass to cloud jobs.

--directory, -d Specify working directory (relative paths in the snakefile will use this as their origin).

--touch, -t Touch output files (mark them up to date without really changing them) instead of running their commands. This is used to pretend that the rules were executed, in order to fool future invocations of snakemake. Fails if a file does not yet exist. Note that this will only touch files that would otherwise be recreated by Snakemake (e.g. because their input files are newer). For enforcing a touch, combine this with –force, –forceall, or –forcerun. Note however that you loose the provenance information when the files have been created in realitiy. Hence, this should be used only as a last resort.

Default: False

--keep-going, -k Go on with independent jobs if a job fails.

Default: False

--rerun-triggers Possible choices: mtime, params, input, software-env, code

Define what triggers the rerunning of a job. By default, all triggers are used, which guarantees that results are consistent with the workflow code and configuration. If you rather prefer the traditional way of just considering file modification dates, use '–rerun-trigger mtime'.

Default: ['mtime', 'params', 'input', 'software-env', 'code']

--force, -f Force the execution of the selected target or the first rule regardless of already created output.

Default: False

--forceall, -F Force the execution of the selected (or the first) rule and all rules it is dependent on regardless of already created output.

Default: False

--forcerun, -R Force the re-execution or creation of the given rules or files. Use this option if you changed a rule and want to have all its output in your workflow updated.

--prioritize, -P Tell the scheduler to assign creation of given targets (and all their dependencies) highest priority. (EXPERIMENTAL)

--batch Only create the given BATCH of the input files of the given RULE. This can be used to iteratively run parts of very large workflows. Only the execution plan of the relevant part of the workflow has to be calculated, thereby speeding up DAG computation. It is recommended to provide the most suitable rule for batching when documenting a workflow. It should be some aggregating rule that would be executed only once, and has a large number of input files. For example, it can be a rule that aggregates over samples.

--until, -U Runs the pipeline until it reaches the specified rules or files. Only runs jobs that are dependencies of the specified rule or files, does not run sibling DAGs.

--omit-from, -O Prevent the execution or creation of the given rules or files as well as any rules or files that are downstream of these targets in the DAG. Also runs jobs in sibling DAGs that are independent of the rules or files specified here.

--rerun-incomplete, --ri Re-run all jobs the output of which is recognized as incomplete.

Default: False

--shadow-prefix Specify a directory in which the 'shadow' directory is created. If not supplied, the value is set to the '.snakemake' directory relative to the working directory.

--scheduler Possible choices: ilp, greedy

Specifies if jobs are selected by a greedy algorithm or by solving an ilp. The ilp scheduler aims to reduce runtime and hdd usage by best possible use of resources.

Default: "greedy"

--wms-monitor IP and port of workflow management system to monitor the execution of snakemake (e.g. http://127.0.0.1:5000) Note that if your service requires an authorization token, you must export WMS_MONITOR_TOKEN in the environment.

--wms-monitor-arg If the workflow management service accepts extra arguments, provide. them in key value pairs with –wms-monitor-arg. For example, to run an existing workflow using a wms monitor, you can provide the pair id=12345 and the arguments will be provided to the endpoint to first interact with the workflow

--scheduler-ilp-solver Specifies solver to be utilized when selecting ilp-scheduler.

Default: "COIN_CMD"

--scheduler-solver-path Set the PATH to search for scheduler solver binaries (internal use only).

--conda-base-path Path of conda base installation (home of conda, mamba, activate) (internal use only).

--no-subworkflows, --nosw Do not evaluate or execute subworkflows.

Default: False

GROUPING

--groups Assign rules to groups (this overwrites any group definitions from the workflow).

--group-components Set the number of connected components a group is allowed to span. By default, this is 1, but this flag allows to extend this. This can be used to run e.g. 3 jobs of the same rule in the same group, although they are not connected. It can be helpful for putting together many small jobs or benefitting of shared memory setups.

REPORTS

--report Create an HTML report with results and statistics. This can be either a .html file or a .zip file. In the former case, all results are embedded into the .html (this only works for small data). In the latter case, results are stored along with a file report.html in the zip archive. If no filename is given, an embedded report.html is the default.

--report-stylesheet Custom stylesheet to use for report. In particular, this can be used for branding the report with e.g. a custom logo, see docs.

NOTEBOOKS

--draft-notebook Draft a skeleton notebook for the rule used to generate the given target file. This notebook can then be opened in a jupyter server, exeucted and implemented until ready. After saving, it will automatically be reused in non-interactive mode by Snakemake for subsequent jobs.

--edit-notebook Interactively edit the notebook associated with the rule used to generate the given target file. This will start a local jupyter notebook server. Any changes to the notebook should be saved, and the server has to be stopped by closing the notebook and hitting the 'Quit' button on the jupyter dashboard. Afterwards, the updated notebook will be automatically stored in the path defined in the rule. If the notebook is not yet present, this will create an empty draft.

--notebook-listen The IP address and PORT the notebook server used for editing the notebook (–edit-notebook) will listen on.

Default: "localhost:8888"

UTILITIES

--lint Possible choices: text, json

Perform linting on the given workflow. This will print snakemake specific suggestions to improve code quality (work in progress, more lints to be added in the future). If no argument is provided, plain text output is used.

--generate-unit-tests Automatically generate unit tests for each workflow rule. This assumes that all input files of each job are already present. Rules without a job with present input files will be skipped (a warning will be issued). For each rule, one test case will be created in the specified test folder (.tests/unit by default). After successfull execution, tests can be run with 'pytest TESTPATH'.

--containerize Print a Dockerfile that provides an execution environment for the workflow, including all conda environments.

Default: False

--export-cwl Compile workflow to CWL and store it in given FILE.

--list, -l Show available rules in given Snakefile.

Default: False

--list-target-rules, --lt Show available target rules in given Snakefile.

Default: False

--dag Do not execute anything and print the directed acyclic graph of jobs in the dot language. Recommended use on Unix systems: snakemake –dag | dot | displayNote print statements in your Snakefile may interfere with visualization.

Default: False

--rulegraph Do not execute anything and print the dependency graph of rules in the dot language. This will be less crowded than above DAG of jobs, but also show less information. Note that each rule is displayed once, hence the displayed graph will be cyclic if a rule appears in several steps of the workflow. Use this if above option leads to a DAG that is too large. Recommended use on Unix systems: snakemake –rulegraph | dot | displayNote print statements in your Snakefile may interfere with visualization.

Default: False

--filegraph Do not execute anything and print the dependency graph of rules with their input and output files in the dot language. This is an intermediate solution between above DAG of jobs and the rule graph. Note that each rule is displayed once, hence the displayed graph will be cyclic if a rule appears in several steps of the workflow. Use this if above option leads to a DAG that is too large. Recommended use on Unix systems: snakemake –filegraph | dot | displayNote print statements in your Snakefile may interfere with visualization.

Default: False

--d3dag Print the DAG in D3.js compatible JSON format.

Default: False

--summary, -S Print a summary of all files created by the workflow. The has the following columns: filename, modification time, rule version, status, plan. Thereby rule version contains the versionthe file was created with (see the version keyword of rules), and status denotes whether the file is missing, its input files are newer or if version or implementation of the rule changed since file creation. Finally the last column denotes whether the file will be updated or created during the next workflow execution.

Default: False

--detailed-summary, -D Print a summary of all files created by the workflow. The has the following columns: filename, modification time, rule version, input file(s), shell command, status, plan. Thereby rule version contains the version the file was created with (see the version keyword of rules), and status denotes whether the file is missing, its input files are newer or if version or implementation of the rule changed since file creation. The input file and shell command columns are self explanatory. Finally the last column denotes whether the file will be updated or created during the next workflow execution.

Default: False

--archive Archive the workflow into the given tar archive FILE. The archive will be created such that the workflow can be re-executed on a vanilla system. The function needs conda and git to be installed. It will archive every file that is under git version control.

Note that it is best practice to have the Snakefile, config files, and scripts under version control. Hence, they will be included in the archive. Further, it will add input files that are not generated by by the workflow itself and conda environments. Note that symlinks are dereferenced. Supported formats are .tar, .tar.gz, .tar.bz2 and .tar.xz.

--cleanup-metadata, --cm Cleanup the metadata of given files. That means that snakemake removes any tracked version info, and any marks that files are incomplete.

--cleanup-shadow Cleanup old shadow directories which have not been deleted due to failures or power loss.

Default: False

--skip-script-cleanup Don't delete wrapper scripts used for execution

Default: False

--unlock Remove a lock on the working directory.

Default: False

--list-version-changes, --lv List all output files that have been created with a different version (as determined by the version keyword).

Default: False

--list-code-changes, --lc List all output files for which the rule body (run or shell) have changed in the Snakefile.

Default: False

--list-input-changes, --li List all output files for which the defined input files have changed in the Snakefile (e.g. new input files were added in the rule definition or files were renamed). For listing input file modification in the filesystem, use –summary.

Default: False

--list-params-changes, --lp List all output files for which the defined params have changed in the Snakefile.

Default: False

--list-untracked, --lu List all files in the working directory that are not used in the workflow. This can be used e.g. for identifying leftover files. Hidden files and directories are ignored.

Default: False

--delete-all-output Remove all files generated by the workflow. Use together with –dry-run to list files without actually deleting anything. Note that this will not recurse into subworkflows. Write-protected files are not removed. Nevertheless, use with care!

Default: False

--delete-temp-output Remove all temporary files generated by the workflow. Use together with –dry-run to list files without actually deleting anything. Note that this will not recurse into subworkflows.

Default: False

--bash-completion Output code to register bash completion for snakemake. Put the following in your .bashrc (including the accents): *snakemake –bash-completion* or issue it in an open terminal session.

Default: False

--keep-incomplete Do not remove incomplete output files by failed jobs.

Default: False

--drop-metadata Drop metadata file tracking information after job finishes. Provenance-information based reports (e.g. –report and the –list_x_changes functions) will be empty or incomplete.

Default: False

--version, -v show program's version number and exit

OUTPUT

--reason, -r Print the reason for each executed rule (deprecated, always true now).

Default: False

--gui Serve an HTML based user interface to the given network and port e.g. 168.129.10.15:8000. By default Snakemake is only available in the local network (default port: 8000). To make Snakemake listen to all ip addresses add the special host address 0.0.0.0 to the url (0.0.0.0:8000). This is important if Snakemake is used in a virtualised environment like Docker. If possible, a browser window is opened.

--printshellcmds, -p Print out the shell commands that will be executed.

Default: False

--debug-dag Print candidate and selected jobs (including their wildcards) while inferring DAG. This can help to debug unexpected DAG topology or errors.

Default: False

--stats Write stats about Snakefile execution in JSON format to the given file.

--nocolor Do not use a colored output.

Default: False

--quiet, -q Possible choices: progress, rules, all

Do not output certain information. If used without arguments, do not output any progress or rule information. Defining 'all' results in no information being printed at all.

--print-compilation Print the python representation of the workflow.

Default: False

--verbose Print debugging output.

Default: False

BEHAVIOR

--force-use-threads Force threads rather than processes. Helpful if shared memory (/dev/shm) is full or unavailable.

Default: False

--allow-ambiguity, -a Don't check for ambiguous rules and simply use the first if several can produce the same file. This allows the user to prioritize rules by their order in the snakefile.

Default: False

--nolock Do not lock the working directory

Default: False

--ignore-incomplete, --ii Do not check for incomplete output files.

Default: False

--max-inventory-time Spend at most SECONDS seconds to create a file inventory for the working directory. The inventory vastly speeds up file modification and existence checks when computing which jobs need to be executed. However, creating the inventory itself can be slow, e.g. on network file systems. Hence, we do not spend more than a given amount of time and fall back to individual checks for the rest.

Default: 20

--latency-wait, --output-wait, -w Wait given seconds if an output file of a job is not present after the job finished. This helps if your filesystem suffers from latency (default 5).

Default: 5

--wait-for-files Wait –latency-wait seconds for these files to be present before executing the workflow. This option is used internally to handle filesystem latency in cluster environments.

--wait-for-files-file Same behaviour as –wait-for-files, but file list is stored in file instead of being passed on the commandline. This is useful when the list of files is too long to be passed on the commandline.

--notemp, --nt Ignore temp() declarations. This is useful when running only a part of the workflow, since temp() would lead to deletion of probably needed files by other parts of the workflow.

Default: False

--all-temp Mark all output files as temp files. This can be useful for CI testing, in order to save space.

Default: False

--keep-remote Keep local copies of remote input files.

Default: False

--keep-target-files Do not adjust the paths of given target files relative to the working directory.

Default: False

--allowed-rules Only consider given rules. If omitted, all rules in Snakefile are used. Note that this is intended primarily for internal use and may lead to unexpected results otherwise.

--target-jobs Target particular jobs by RULE:WILDCARD1=VALUE,WILDCARD2=VALUE,… This is meant for internal use by Snakemake itself only.

--local-groupid Name for local groupid, meant for internal use only.

Default: "local"

--max-jobs-per-second Maximal number of cluster/drmaa jobs per second, default is 10, fractions allowed.

Default: 10

--max-status-checks-per-second Maximal number of job status checks per second, default is 10, fractions allowed.

Default: 10

-T, --retries, --restart-times Number of times to restart failing jobs (defaults to 0).

Default: 0

--attempt Internal use only: define the initial value of the attempt parameter (default: 1).

Default: 1

--wrapper-prefix Prefix for URL created from wrapper directive (default: https://github.com/snakemake/snakemake-wrappers/raw/). Set this to a different URL to use your fork or a local clone of the repository, e.g., use a git URL like 'git+file://path/to/your/local/clone@'.

Default: "https://github.com/snakemake/snakemake-wrappers/raw/"

--default-remote-provider Possible choices: S3, GS, FTP, SFTP, S3Mocked, gfal, gridftp, iRODS, AzBlob, XRootD

Specify default remote provider to be used for all input and output files that don't yet specify one.

--default-remote-prefix Specify prefix for default remote provider. E.g. a bucket name.

Default: ""

--no-shared-fs Do not assume that jobs share a common file system. When this flag is activated, Snakemake will assume that the filesystem on a cluster node is not shared with other nodes. For example, this will lead to downloading remote files on each cluster node separately. Further, it won't take special measures to deal with filesystem latency issues. This option will in most cases only make sense in combination with –default-remote-provider. Further, when using –cluster you will have to also provide –cluster-status. Only activate this if you know what you are doing.

Default: False

--greediness Set the greediness of scheduling. This value between 0 and 1 determines how careful jobs are selected for execution. The default value (1.0) provides the best speed and still acceptable scheduling quality.

--no-hooks Do not invoke onstart, onsuccess or onerror hooks after execution.

Default: False

--overwrite-shellcmd Provide a shell command that shall be executed instead of those given in the workflow. This is for debugging purposes only.

--debug Allow to debug rules with e.g. PDB. This flag allows to set breakpoints in run blocks.

Default: False

--runtime-profile Profile Snakemake and write the output to FILE. This requires yappi to be installed.

--mode Possible choices: 0, 1, 2

Set execution mode of Snakemake (internal use only).

Default: 0

--show-failed-logs Automatically display logs of failed jobs.

Default: False

--log-handler-script Provide a custom script containing a function 'def log_handler(msg):'. Snakemake will call this function for every logging output (given as a dictionary msg)allowing to e.g. send notifications in the form of e.g. slack messages or emails.

--log-service Possible choices: none, slack, wms

Set a specific messaging service for logging output.Snakemake will notify the service on errors and completed execution.Currently slack and workflow management system (wms) are supported.

SLURM

--slurm Execute snakemake rules as SLURM batch jobs according to their 'resources' definition. SLURM resources as 'partition', 'ntasks', 'cpus', etc. need to be defined per rule within the 'resources' definition. Note, that memory can only be defined as 'mem_mb' or 'mem_mb_per_cpu' as analoguous to the SLURM 'mem' and 'mem-per-cpu' flags to sbatch, respectively. Here, the unit is always 'MiB'. In addition '–default_resources' should contain the SLURM account.

Default: False

CLUSTER

--cluster Execute snakemake rules with the given submit command, e.g. qsub. Snakemake compiles jobs into scripts that are submitted to the cluster with the given command, once all input files for a particular job are present. The submit command can be decorated to make it aware of certain job properties (name, rulename, input, output, params, wildcards, log, threads and dependencies (see the argument below)), e.g.: $ snakemake –cluster 'qsub -pe threaded {threads}'.

--cluster-sync cluster submission command will block, returning the remote exitstatus upon remote termination (for example, this should be usedif the cluster command is 'qsub -sync y' (SGE)

--drmaa Execute snakemake on a cluster accessed via DRMAA, Snakemake compiles jobs into scripts that are submitted to the cluster with the given command, once all input files for a particular job are present. ARGS can be used to specify options of the underlying cluster system, thereby using the job properties name, rulename, input, output, params, wildcards, log, threads and dependencies, e.g.: –drmaa ' -pe threaded {threads}'. Note that ARGS must be given in quotes and with a leading whitespace.

--cluster-config, -u A JSON or YAML file that defines the wildcards used in 'cluster'for specific rules, instead of having them specified in the Snakefile. For example, for rule 'job' you may define: { 'job' : { 'time' : '24:00:00' } } to specify the time for rule 'job'. You can specify more than one file. The configuration files are merged with later values overriding earlier ones. This option is deprecated in favor of using –profile, see docs.

Default: []

--immediate-submit, --is Immediately submit all jobs to the cluster instead of waiting for present input files. This will fail, unless you make the cluster aware of job dependencies, e.g. via: $ snakemake –cluster 'sbatch –dependency {dependencies}. Assuming that your submit script (here sbatch) outputs the generated job id to the first stdout line, {dependencies} will be filled with space separated job ids this job depends on. Does not work for workflows that contain checkpoint rules.

Default: False

--jobscript, --js Provide a custom job script for submission to the cluster. The default script resides as 'jobscript.sh' in the installation directory.

--jobname, --jn Provide a custom name for the jobscript that is submitted to the cluster (see –cluster). NAME is "snakejob.{name}.{jobid}.sh" per default. The wildcard {jobid} has to be present in the name.

Default: "snakejob.{name}.{jobid}.sh"

--cluster-status Status command for cluster execution. This is only considered in combination with the –cluster flag. If provided, Snakemake will use the status command to determine if a job has finished successfully or failed. For this it is necessary that the submit command provided to –cluster returns the cluster job id. Then, the status command will be invoked with the job id. Snakemake expects it to return 'success' if the job was successfull, 'failed' if the job failed and 'running' if the job still runs.

--cluster-cancel Specify a command that allows to stop currently running jobs. The command will be passed a single argument, the job id.

--cluster-cancel-nargs Specify maximal number of job ids to pass to –cluster-cancel command, defaults to 1000.

Default: 1000

--cluster-sidecar Optional command to start a sidecar process during cluster execution. Only active when –cluster is given as well.

--drmaa-log-dir Specify a directory in which stdout and stderr files of DRMAA jobs will be written. The value may be given as a relative path, in which case Snakemake will use the current invocation directory as the origin. If given, this will override any given '-o' and/or '-e' native specification. If not given, all DRMAA stdout and stderr files are written to the current working directory.

FLUX

--flux Execute your workflow on a flux cluster. Flux can work with both a shared network filesystem (like NFS) or without. If you don't have a shared filesystem, additionally specify –no-shared-fs.

Default: False

KUBERNETES

--kubernetes Execute workflow in a kubernetes cluster (in the cloud). NAMESPACE is the namespace you want to use for your job (if nothing specified: 'default'). Usually, this requires –default-remote-provider and –default-remote-prefix to be set to a S3 or GS bucket where your . data shall be stored. It is further advisable to activate conda integration via –use-conda.

--container-image Docker image to use, e.g., when submitting jobs to kubernetes Defaults to 'https://hub.docker.com/r/snakemake/snakemake', tagged with the same version as the currently running Snakemake instance. Note that overwriting this value is up to your responsibility. Any used image has to contain a working snakemake installation that is compatible with (or ideally the same as) the currently running version.

--k8s-cpu-scalar K8s reserves some proportion of available CPUs for its own use. So, where an underlying node may have 8 CPUs, only e.g. 7600 milliCPUs are allocatable to k8s pods (i.e. snakemake jobs). As 8 > 7.6, k8s can't find a node with enough CPU resource to run such jobs. This argument acts as a global scalar on each job's CPU request, so that e.g. a job whose rule definition asks for 8 CPUs will request 7600m CPUs from k8s, allowing it to utilise one entire node. N.B: the job itself would still see the original value, i.e. as the value substituted in {threads}.

Default: 0.95

TIBANNA

--tibanna Execute workflow on AWS cloud using Tibanna. This requires –default-remote-prefix to be set to S3 bucket name and prefix (e.g. 'bucketname/subdirectory') where input is already stored and output will be sent to. Using –tibanna implies –default-resources is set as default. Optionally, use –precommand to specify any preparation command to run before snakemake command on the cloud (inside snakemake container on Tibanna VM). Also, –use-conda, –use-singularity, –config, –configfile are supported and will be carried over.

Default: False

--tibanna-sfn Name of Tibanna Unicorn step function (e.g. tibanna_unicorn_monty).This works as serverless scheduler/resource allocator and must be deployed first using tibanna cli. (e.g. tibanna deploy_unicorn –usergroup=monty –buckets=bucketname)

--precommand Any command to execute before snakemake command on AWS cloud such as wget, git clone, unzip, etc. This is used with –tibanna.Do not include input/output download/upload commands - file transfer between S3 bucket and the run environment (container) is automatically handled by Tibanna.

--tibanna-config Additional tibanna config e.g. –tibanna-config spot_instance=true subnet=<subnet_id> security group=<security_group_id>

GOOGLE_LIFE_SCIENCE

--google-lifesciences Execute workflow on Google Cloud cloud using the Google Life. Science API. This requires default application credentials (json) to be created and export to the environment to use Google Cloud Storage, Compute Engine, and Life Sciences. The credential file should be exported as GOOGLE_APPLICATION_CREDENTIALS for snakemake to discover. Also, –use-conda, –use-singularity, –config, –configfile are supported and will be carried over.

Default: False

--google-lifesciences-regions Specify one or more valid instance regions (defaults to US)

Default: ['us-east1', 'us-west1', 'us-central1']

--google-lifesciences-location The Life Sciences API service used to schedule the jobs. E.g., us-centra1 (Iowa) and europe-west2 (London) Watch the terminal output to see all options found to be available. If not specified, defaults to the first found with a matching prefix from regions specified with –google-lifesciences-regions.

--google-lifesciences-keep-cache Cache workflows in your Google Cloud Storage Bucket specified by –default-remote-prefix/{source}/{cache}. Each workflow working directory is compressed to a .tar.gz, named by the hash of the contents, and kept in Google Cloud Storage. By default, the caches are deleted at the shutdown step of the workflow.

Default: False

TES

--tes Send workflow tasks to GA4GH TES server specified by url.

CONDA

--use-conda If defined in the rule, run job in a conda environment. If this flag is not set, the conda directive is ignored.

Default: False

--conda-not-block-search-path-envvars Do not block environment variables that modify the search path (R_LIBS, PYTHONPATH, PERL5LIB, PERLLIB) when using conda environments.

Default: False

--list-conda-envs List all conda environments and their location on disk.

Default: False

--conda-prefix Specify a directory in which the 'conda' and 'conda-archive' directories are created. These are used to store conda environments and their archives, respectively. If not supplied, the value is set to the '.snakemake' directory relative to the invocation directory. If supplied, the *–use-conda* flag must also be set. The value may be given as a relative path, which will be extrapolated to the invocation directory, or as an absolute path. The value can also be provided via the environment variable $SNAKEMAKE_CONDA_PREFIX.

--conda-cleanup-envs Cleanup unused conda environments.

Default: False

--conda-cleanup-pkgs Possible choices: tarballs, cache

Cleanup conda packages after creating environments. In case of 'tarballs' mode, will clean up all downloaded package tarballs. In case of 'cache' mode, will additionally clean up unused package caches. If mode is omitted, will default to only cleaning up the tarballs.

--conda-create-envs-only If specified, only creates the job-specific conda environments then exits. The *–use-conda* flag must also be set.

Default: False

--conda-frontend Possible choices: conda, mamba

Choose the conda frontend for installing environments. Mamba is much faster and highly recommended.

Default: "mamba"

SINGULARITY

--use-singularity If defined in the rule, run job within a singularity container. If this flag is not set, the singularity directive is ignored.

Default: False

--singularity-prefix Specify a directory in which singularity images will be stored.If not supplied, the value is set to the '.snakemake' directory relative to the invocation directory. If supplied, the *–use-singularity* flag must also be set. The value may be given as a relative path, which will be extrapolated to the invocation directory, or as an absolute path.

--singularity-args Pass additional args to singularity.

Default: ""

--cleanup-containers Remove unused (singularity) containers

Default: False

ENVIRONMENT MODULES

--use-envmodules If defined in the rule, run job within the given environment modules, loaded in the given order. This can be combined with –use-conda and –use-singularity, which will then be only used as a fallback for rules which don't define environment modules.

Default: False

4.6.6 Bash Completion

Snakemake supports bash completion for filenames, rulenames and arguments. To enable it globally, just append

```
`snakemake --bash-completion`
```

including the backticks to your `.bashrc`. This only works if the `snakemake` command is in your path.

4.7 Cluster Execution

Snakemake can make use of cluster engines that support shell scripts and have access to a common filesystem, (e.g. Slurm or PBS). There exists a generic cluster support which works with any such engine (see *Generic cluster support*), and a specific support for Slurm (see *Executing on SLURM clusters*). When executing on a cluster, Snakemake implicitly assumes some default resources for all rules (see *Default Resources*).

4.7.1 Executing on SLURM clusters

SLURM is a widely used batch system for performance compute clusters. In order to use Snakemake with slurm, simply append `--slurm` to your Snakemake invocation.

Specifying Account and Partition

Most SLURM clusters have two mandatory resource indicators for accounting and scheduling, *Account* and *Partition*, respectivily. These resources are usually omitted from Snakemake workflows in order to keep the workflow definition independent from the platform. However, it is also possible to specify them inside of the workflow as resources in the rule definition (see *Resources*).

To specify them at the command line, define them as default resources:

```
$ snakemake --slurm --default-resources slurm_account=<your SLURM account> slurm_
↪partition=<your SLURM partition>
```

If individual rules require e.g. a different partition, you can override the default per rule:

```
$ snakemake --slurm --default-resources slurm_account=<your SLURM account> slurm_
↪partition=<your SLURM partition> --set-resources <somerule>:slurm_partition=<some␣
↪other partition>
```

Usually, it is advisable to persist such settings via a *configuration profile*, which can be provided system-wide or per user.

Ordinary SMP jobs

Most jobs will be carried out by programs which are either single core scripts or threaded programs, hence SMP (shared memory programs) in nature. Any given threads and `mem_mb` requirements will be passed to SLURM:

```
rule a:
    input: ...
    output: ...
    threads: 8
    resources:
        mem_mb: 14000
```

This will give jobs from this rule 14GB of memory and 8 CPU cores. It is advisable to use resonable default resources, such that you don't need to specify them for every rule. Snakemake already has reasonable defaults built in, which are automatically activated when using the `--default-resources` flag (see above, and also `snakemake --help`).

MPI jobs

Snakemake's Slurm backend also supports MPI jobs, see snakefiles-mpi for details. When using MPI with slurm, it is advisable to use `srun` as MPI starter.

```
rule calc_pi:
  output:
      "pi.calc",
  log:
      "logs/calc_pi.log",
  resources:
      tasks=10,
      mpi="srun",
  shell:
      "{resources.mpi} -n {resources.tasks} calc-pi-mpi > {output} 2> {log}"
```

Note that the `-n {resources.tasks}` is not necessary in case of SLURM, but it should be kept in order to allow execution of the workflow on other systems, e.g. by replacing `srun` with `mpiexec`:

```
$ snakemake --set-resources calc_pi:mpi="mpiexec" ...
```

Advanced Resource Specifications

A workflow rule may support a number of *resource* specification. For a SLURM cluster, a mapping between Snakemake and SLURM needs to be performed.

You can use the following specifications:

SLURM Resource	Snakemake resource	Background Information
`-p/--partition`	`slurm_partition`	the partition a rule/job is to use
`-t/--time`	`runtime`	the walltime per job in minutes
`-C/`*`–constraint`*	`constraint`	may hold features on some clusters
`--mem`	`mem_mb`	memory in MB a cluster node must provide
`--mem-per-cpu`	`mem_mb_per_cpu`	memory per reserved CPU
`-n/--ntasks`	`tasks`	number of concurrent tasks / ranks
`-c/--cpus-per-task`	`cpus_per_task`	number of cpus per task (in case of SMP, rather use `threads`)
`-N/--nodes`	`nodes`	number of nodes

Each of these can be part of a rule, e.g.:

```
rule:
    input: ...
    output: ...
    resources:
        partition: <partition name>
        runtime: <some number>
```

Please note: as `--mem` and `--mem-per-cpu` are mutually exclusive on SLURM clusters, there corresponding resource flags `mem_mb` and `mem_mb_per_cpu` are mutually exclusive, too. You can only reserve memory a compute node has to provide or the memory required per CPU (SLURM does not make any distintion between real CPU cores and those provided by hyperthreads). SLURM will try to sastify a combination of `mem_mb_per_cpu` and `cpus_per_task` and `nodes`, if `nodes` is not given.

Note that it is usually advisable to avoid specifying SLURM (and compute infrastructure) specific resources (like `constraint`) inside of your workflow because that can limit the reproducibility on other systems. Consider using the `--default-resources` and `--set-resources` flags to define such resources on the command line.

Additional custom job configuration

SLURM installations can support custom plugins, which may add support for additional flags to `sbatch`. In addition, there are various `sbatch` options not directly supported via the resource definitions shown above. You may use the `slurm_extra` resource to specify additional flags to `sbatch`:

```
rule:
    input: ...
    output: ...
    resources:
        slurm_extra="--qos=long --mail-type=ALL --mail-user=<your email>"
```

4.7.2 Generic cluster support

To use the generic cluster support, Snakemake simply needs to be given a submit command that accepts a shell script as first positional argument:

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

Here, `--jobs` denotes the number of jobs submitted to the cluster at the same time (here 32). The cluster command can be decorated with job specific information, e.g.

Note

Consider to *group jobs* in order to minimize overhead, in particular for short-running jobs.

```
$ snakemake --cluster "qsub {threads}"
```

Thereby, all keywords of a rule are allowed (e.g. rulename, params, input, output, threads, priority, resources, …). For example, you could encode the expected running time in minutes into a *resource* `runtime_min`:

```
rule:
    input:
        ...
    output:
        ...
    resources:
        runtime_min=240
    shell:
        ...
```

and forward it to the cluster scheduler:

```
$ snakemake --cluster "qsub --runtime {resources.runtime}"
```

In order to avoid specifying `runtime_min` for each rule, you can make use of the `--default-resources` flag, see `snakemake --help`.

If your cluster system supports DRMAA, Snakemake can make use of that to control jobs. With DRMAA, no `qsub` command needs to be provided, but system specific arguments can still be given as a string, e.g.

```
$ snakemake --drmaa " -q username" -j 32
```

Note that the string has to contain a leading whitespace. Else, the arguments will be interpreted as part of the normal Snakemake arguments, and execution will fail.

Adapting to a specific cluster can involve quite a lot of options. It is therefore a good idea to setup a *a profile*.

Job Properties

When executing a workflow on a cluster using the `--cluster` parameter (see below), Snakemake creates a job script for each job to execute. This script is then invoked using the provided cluster submission command (e.g. `qsub`). Sometimes you want to provide a custom wrapper for the cluster submission command that decides about additional parameters. As this might be based on properties of the job, Snakemake stores the job properties (e.g. name, rulename, threads, input, output, params etc.) as JSON inside the job script (for group jobs, the rulename will be "GROUP", otherwise it will be the same as the job name). For convenience, there exists a parser function *snakemake.utils.read_job_properties* that can be used to access the properties. The following shows an example job submission wrapper:

```
#!python

#!/usr/bin/env python3
import os
import sys

from snakemake.utils import read_job_properties

jobscript = sys.argv[1]
job_properties = read_job_properties(jobscript)

# do something useful with the threads
threads = job_properties[threads]

# access property defined in the cluster configuration file (Snakemake >=3.6.0)
job_properties["cluster"]["time"]

os.system("qsub -t {threads} {script}".format(threads=threads, script=jobscript))
```

4.8 Cloud execution

When executing on a cluster, Snakemake implicitly assumes some default resources for all rules (see *Default Resources*).

4.8.1 Generic cloud support via Kubernetes

Snakemake 4.0 and later supports execution in the cloud via Kubernetes. This is independent of the cloud provider, but we provide the setup steps for GCE below.

Setup Kubernetes on Google cloud engine

First, install the Google Cloud SDK. Then, run

```
$ gcloud init
```

to setup your access. Then, you can create a new kubernetes cluster via

```
$ gcloud container clusters create $CLUSTER_NAME --num-nodes=$NODES --scopes storage-
↪rw
```

with `$CLUSTER_NAME` being the cluster name and `$NODES` being the number of cluster nodes. If you intend to use google storage, make sure that `--scopes storage-rw` is set. This enables Snakemake to write to the google storage from within the cloud nodes. Next, you configure Kubernetes to use the new cluster via

```
$ gcloud container clusters get-credentials $CLUSTER_NAME
```

If you are having issues with authentication, please refer to the help text:

```
$ gcloud container clusters get-credentials --help
```

You likely also want to use google storage for reading and writing files. For this, you will additionally need to authenticate with your google cloud account via

```
$ gcloud auth application-default login
```

This enables Snakemake to access google storage in order to check existence and modification dates of files. Now, Snakemake is ready to use your cluster.

Important: After finishing your work, do not forget to delete the cluster with

```
$ gcloud container clusters delete $CLUSTER_NAME
```

in order to avoid unnecessary charges.

Executing a Snakemake workflow via kubernetes

Assuming that kubernetes has been properly configured (see above), you can execute a workflow via:

```
snakemake --kubernetes --use-conda --default-remote-provider $REMOTE --default-remote-
↪prefix $PREFIX
```

In this mode, Snakemake will assume all input and output files to be stored in a given remote location, configured by setting `$REMOTE` to your provider of choice (e.g. `GS` for Google cloud storage or `S3` for Amazon S3) and `$PREFIX` to a bucket name or subfolder within that remote storage. After successful execution, you find your results in the specified remote storage. Of course, if any input or output already defines a different remote location, the latter will be used instead. Importantly, this means that Snakemake does **not** require a shared network filesystem to work in the cloud.

Note

Consider to *group jobs* in order to minimize overhead, in particular for short-running jobs.

Currently, this mode requires that the Snakemake workflow is stored in a git repository. Snakemake uses git to query necessary source files (the Snakefile, scripts, config, …) for workflow execution and encodes them into the kubernetes job. Importantly, this also means that you should not put large non-source files into the git repo, since Snakemake will try to upload them to kubernetes with every job. With large files in the git repo, this can lead to performance issues or even random SSL errors from kubernetes.

It is further possible to forward arbitrary environment variables to the kubernetes jobs via the flag `--envvars` (see `snakemake --help`) or the `envvars` directive in the Snakefile. The former should be used e.g. for platform specific variables (e.g. secrets that are only needed for your kubernetes setup), whereas the latter should be used for variables that are needed for the workflow itself, regardless of whether it is executed on kubernetes or with a different backend.

When executing, Snakemake will make use of the defined resources and threads to schedule jobs to the correct nodes. In particular, it will forward memory requirements defined as `mem_mb` to kubernetes. Further, it will propagate the number of threads a job intends to use, such that kubernetes can allocate it to the correct cloud computing node.

Machine Types

To specify an exact machine type or a prefix to filter down to and then select based on other resource needs, you can set a default resource on the command line, either for a prefix or a full machine type:

```
--default-resources "machine_type=n1-standard"
```

For individual jobs, the default machine type can also be overwritten via

```
--set-resources "somerule:machine_type=n1-standard"
```

If you want to specify the machine type as a resource in the workflow definition, you can do that too (although it is not recommended in general because it ties your workflow to the used platform):

```
rule somerule:
    output:
        "test.txt"
    resources:
        machine_type="n1-standard-8"
    shell:
        "somecommand ..."
```

4.8.2 Executing a Snakemake workflow via Google Cloud Life Sciences

The Google Cloud Life Sciences provides a rich application programming interface to design pipelines. You'll first need to follow instructions here to create a Google Cloud Project and enable Life Sciences, Storage, and Compute Engine APIs, and continue with the prompts to create credentials. You'll want to create a service account for your host (it's easiest to give project Owner permissions), and save the json credentials. You'll want to export the full path to this file to `GOOGLE_APPLICATION_CREDENTIALS`:

```
$ export GOOGLE_APPLICATION_CREDENTIALS=$HOME/path/snakemake-credentials.json
```

If you lose the link to the credentials interface, you can find it here.

Optionally, you can export `GOOGLE_CLOUD_PROJECT` as the name of your Google Cloud Project. By default, the project associated with your application credentials will be used.

```
$ export GOOGLE_CLOUD_PROJECT=my-project-name
```

Data in Google Storage

Using this executor typically requires you to start with large data files already in Google Storage, and then interact with them via the Google Storage remote executor. An easy way to do this is to use the gsutil command line client. For example, here is how we might upload a file to storage using it:

```
$ gsutil -m cp mydata.txt gs://snakemake-bucket/1/mydata.txt
```

The `-m` parameter enables multipart uploads for large files, so you can remove it if you are uploading one or more smaller files. And note that you'll need to modify the file and bucket names. Note that you can also easily use the Google Cloud Console interface, if a graphical interface is preferable to you.

Environment Variables

Important: Google Cloud Life Sciences uses Google Compute, and does **not** encrypt environment variables. If you specify environment variables with the envvars directive or `--envvars` they will **not** be secrets.

Container Bases

By default, Google Life Sciences uses the latest stable version of snakemake/snakemake on Docker Hub. You can choose to modify the container base with the `--container-image` (or `container_image` from within Python), however if you do so, your container must meet the following requirements:

- have an entrypoint that can execute a `/bin/bash` command
- have snakemake installed, either via `conda activate snakemake` or already on the path
- also include snakemake Python dependencies for google.cloud

If you use any Snakemake container as a base, you should be good to go. If you'd like to get a reference for requirements, it's helpful to look at the Dockerfile for Snakemake.

Requesting GPUs

The Google Life Sciences API currently has support for NVIDIA GPUs, meaning that you can request a number of NVIDIA GPUs explicitly by adding `nvidia_gpu` or `gpu` to your Snakefile resources for a step:

```
rule a:
    output:
        "test.txt"
    resources:
        nvidia_gpu=1
    shell:
        "somecommand ..."
```

A specific gpu model can be requested using `gpu_model` and lowercase identifiers like `nvidia-tesla-p100` or `nvidia-tesla-p4`, for example: `gpu_model="nvidia-tesla-p100"`. If you don't specify `gpu` or `nvidia_gpu` with a count, but you do specify a `gpu_model`, the count will default to 1.

In addition to GPU for the Google Lifesciences Executor, you can request a Google Cloud preemptible virtual machine for one or more steps. See the rules documentation for how to add one or more preemptible arguments.

Machine Types

To specify an exact machine type or a prefix to filter down to and then select based on other resource needs, you can set a default resource on the command line, either for a prefix or a full machine type:

```
--default-resources "machine_type=n1-standard"
```

If you want to specify the machine type as a resource, you can do that too:

```
rule a:
    output:
        "test.txt"
    resources:
        machine_type="n1-standard-8"
    shell:
        "somecommand ..."
```

If you request a gpu, this requires the "n1" prefix and your preference from the file or command line will be overridden. Note that the default resources for Google Life Sciences (memory and disk) are the same as for Tibanna.

Running the Life Sciences Executor

When your Snakefile is ready, you can run snakemake to specify the life sciences executor. Notice that we are also providing a remote prefix for our storage path, along with a region.

```
$ snakemake --google-lifesciences --default-remote-prefix snakemake-testing-data --
↪use-conda --google-lifesciences-region us-west1
```

For more details and examples, we recommend you reference the Google Life Sciences Executor Tutorial.

4.8.3 Executing a Snakemake workflow via Tibanna on Amazon Web Services

First, install Tibanna.

```
$ pip install -U tibanna
```

Set up aws configuration either by creating files `~/.aws/credentials` and `~/.aws/config` or by setting up environment variables as below (see Tibanna or AWS documentation for more details):

```
$ export AWS_ACCESS_KEY_ID=<AWS_ACCESS_KEY>
$ export AWS_SECRET_ACCESS_KEY=<AWS_SECRET_ACCESS_KEY>
$ export AWS_DEFAULT_REGION=<AWS_DEFAULT_REGION>
```

As an AWS admin, deploy Tibanna Unicorn to Cloud with permissions to a specific S3 bucket. Name the Unicorn / Unicorn usergroup with the `--usergroup` option. Unicorn is a serverless scheduler, and keeping unicorn on the cloud does not incur extra cost. One may have many different unicorns with different names and different bucket permissions. Then, add other (IAM) users to the user group that has permission to use this unicorn / buckets.

```
$ tibanna deploy_unicorn -g <name> -b <bucket>
$ tibanna add_user -u <username> -g <name>
```

As a user that has been added to the group (or as an admin), set up the default unicorn.

```
$ export TIBANNA_DEFAULT_STEP_FUNCTION_NAME=tibanna_unicorn_<name>
```

Then, you can run as many snakemake runs as you wish as below, inside a directory that contains Snakefile and other necessary components (e.g. `env.yml`, `config.json`, ...).

```
$ snakemake --tibanna --default-remote-prefix=<bucketname>/<subdir> [<other options>]
```

In this mode, Snakemake will assume all input and output files to be stored in the specified remote location (a subdirectory inside a given S3 bucket.) After successful execution, you find your results in the specified remote storage. Of course, if any input or output already defines a different remote location, the latter will be used instead. In that case, Tibanna Unicorn must be deployed with all the relevant buckets (`-b bucket1,bucket2,bucket3,...`) to allow access to the Unicorn serverless components. Snakemake will assign 3x of the total input size as the allocated space for each execution. The execution may fail if the total input + output + temp file sizes exceed this estimate.

In addition to regular snakemake options, `--precommand=<command>` option allows sending a command to execute before executing each step on an isolated environment. This kind of command could involve downloading or installing necessary files that cannot be handled using conda (e.g. the command may begin with `wget`, `git clone`, etc.)

To check Tibanna execution logs, first use `tibanna stat` to see the list of all the individual runs.

```
$ tibanna stat -n <number_of_executions_to_view> -l
```

Then, check the detailed log for each job using the Tibanna job id that can be obtained from the first column of the output of `tibanna stat`.

```
$ tibanna log -j <jobid>
```

Note

Consider to *group jobs* in order to minimize overhead, in particular for short-running jobs.

When executing, Snakemake will make use of the defined resources and threads to schedule jobs to the correct nodes. In particular, it will forward memory requirements defined as *mem_mb* to Tibanna. Further, it will propagate the number of threads a job intends to use, such that Tibanna can allocate it to the most cost-effective cloud compute instance available.

4.8.4 Executing a Snakemake workflow via GA4GH TES

The task execution service (TES) is an application programming interface developed by the Global Alliance for Genomics and Health (GA4GH). It is used to process workflow tasks in a cloud environment. A TES server can be easily implemented in a public cloud or at a commercial cloud provider. Here, the TES standard provides an additional abstraction layer between the execution of a workflow (e.g. on your local machine) and technologies for execution of single tasks (e.g. based Kubernetes or HPC). We recommend using either Funnel or TESK to install a TES server. The guide here is based on Funnel (0.10.0). To install and configure Funnel follow its official documentation.

Configuration

Three steps are required to make a Snakemake workflow TES ready:

Attach conda to rules: Execution of Snakemake tasks via TES means, Snakemake is running in a container in the cloud and it executes a specific rule (or a group of rules) with defined input/output data. By default, the TES module uses the latest Snakemake container. Running Snakemake within a container requires having all external tools installed within this container. This can be done by providing a custom container image having installed Snakemake and other all required tools (e.g. BWA). Or it can be done by attaching a conda environment to each rule, such that those tools will be installed within the running container. For simplicity, this guide recommends to attach a specific conda environment to each rule, although it is more efficient in the long term to provide custom container images.

Use remote files: The TES module requires using a remote file storage system for input/output files such that all files are available on the cloud machines and within their running container. There are several options available in Snakemake to use remote files. This guide recommends to use S3 (or SWIFT) object storage. Please be aware to download final result files from S3 to your local machine by defining a rule that downloads files and gets executed locally (e.g. by setting *localrules: all, download*).

Install py-tes module: TES backend requires py-tes to be installed. Please install py-tes, e.g. via Conda or Pip.

```
$ pip install py-tes
```

Execution

Funnel starts container in read only mode, which is good practice. Anyhow, using the default Snakemake container image will likely require installing additional software within the running container. Therefore, we need to set two conda specific variables such that new environments will be installed at */tmp* which will be mounted as a writable volume in the container.

```
$ export CONDA_PKGS_DIRS=/tmp/conda
$ export CONDA_ENVS_PATH=/tmp/conda
```

Next, using S3 or SWIFT storage, we also need to set credentials.

```
$ export AWS_ACCESS_KEY_ID=YOUR_ACCESS_KEY
$ export AWS_SECRET_ACCESS_KEY=YOUR_SECRET_ACCESS_KEY
```

Now we can run Snakemake using:

```
$ snakemake \
    --tes $TES_URL \
    --use-conda \
    --envvars CONDA_PKGS_DIRS CONDA_ENVS_PATH AWS_ACCESS_KEY_ID AWS_SECRET_ACCESS_KEY␣
↪\
    --conda-prefix $CONDA_ENVS_PATH \
    all
```

If your TES instance requires authentication via OIDC tokens, you can forward your token by setting the *TES_TOKEN* environmental variable.

```
$ export TES_TOKEN=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
↪SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

4.9 Job Grouping

The graph of jobs that Snakemake determines before execution can be partitioned into groups. Such groups will be executed together in **cluster** or **cloud mode**, as a so-called **group job**, i.e., all jobs of a particular group will be submitted at once, to the same computing node. When executing locally, group definitions are ignored.

Groups can be defined along with the workflow definition via the `group` keyword, see *Defining groups for execution*. This way, queueing and execution time can be saved, in particular by attaching short-running downstream jobs to long running upstream jobs.

From Snakemake 7.11 on, Snakemake will request resources for groups by summing across jobs that can be run in parallel, and taking the max of jobs run in series. The only exception is `runtime`, where the max will be taken over parallel jobs, and the sum over series. If resource contraints are provided (via `--resources` or `--cores`), parallel job layers that exceed the constraints will be stacked in series. For example, if 6 instances of `somerule` are being run, each instance requires `1000MB` of memory and `30 min` runtime, and only `3000MB` are available, Snakemake will request `3000MB` and `60 min` runtime, enough to run 3 instances of `somerule`, then another 3 instances of `somerule` in series.

Often, the ideal group will be dependent on the specifics of the underlying computing platform. Hence, it is possible to assign groups via the command line. For example, with

```
snakemake --groups somerule=group0 someotherrule=group0
```

we assign the two rules `somerule` and `someotherrule` to the same group `group0`.

By default, groups do not span disconnected parts of the DAG. This means that, for example, jobs of `somerule` and `someotherrule` only end in the same group if they are directly connected. It is, however, possible to configure the number of connected DAG components that are spanned by a group via the flag `--group-components`. This makes it possible to define batches of jobs of the same kind that shall be executed within one group. For instance:

```
snakemake --groups somerule=group0 --group-components group0=5
```

means that given `n` jobs spawned from rule `somerule`, Snakemake will create `n / 5` groups which each execute 5 jobs of `somerule` together. For example, with 10 jobs from `somerule` you would end up with 2 groups of 5 jobs that are submitted as one piece each.

4.10 Between workflow caching

Within certain data analysis fields, there are certain intermediate results that reoccur in exactly the same way in many analysis. For example, in bioinformatics, reference genomes or annotations are downloaded, and read mapping indexes are built. Since such steps are independent of the actual data or measurements that are analyzed, but still computationally or timely expensive to conduct, it has been common practice to externalize their computation and assume the presence of the resulting files before execution of a workflow.

From version 5.8.0 on, Snakemake offers a way to keep those steps inside the actual analysis without requiring from redundant computations. By hashing all steps, parameters, software stacks (in terms of conda environments or containers), and raw input required up to a certain step in a blockchain, Snakemake is able to recognize **before** the computation whether a certain result is already available in a central cache on the same system. **Note that this is explicitly intended for caching results between workflows! There is no need to use this feature to avoid redundant computations within a workflow. Snakemake does this already out of the box.**

Such caching has to be explicitly activated per rule, which can be done via the command line interface. For example,

```
$ export SNAKEMAKE_OUTPUT_CACHE=/mnt/snakemake-cache/
$ snakemake --cache download_data create_index
```

would instruct Snakemake to cache and reuse the results of the rules `download_data` and `create_index`. The environment variable definition that happens in the first line (defining the location of the cache) should of course be done only once and system wide in reality. When Snakemake is executed without a shared filesystem (e.g., in the cloud, see *Cloud execution*), the environment variable has to point to a location compatible with the given remote provider (e.g. an S3 or Google Storage bucket). In any case, the provided location should be shared between all workflows of your group, institute or computing environment, in order to benefit from the reuse of previously obtained intermediate results.

Alternatively, rules can be marked as eligible for caching via the `cache` directive:

```
rule download_data:
    output:
        "results/data/worldcitiespop.csv"
    cache: True # allowed values: "all", "omit-software", True
    shell:
        "curl -L https://burntsushi.net/stuff/worldcitiespop.csv > {output}"
```

Here, the given value defines what information shall be considered for calculating the hash value. With `"all"` or `True`, all relevant rule information is used as outlined above (this is the recommended default). With `"omit-software"`, the software stack is not considered, which is useful if the software stack is not relevant for the result (e.g., if the rule is only a data download).

For workflows defining cache rules like this, it is enough to invoke Snakemake with

```
$ snakemake --cache
```

without explicit rulenames listed.

Note that only rules with just a single output file (or directory) or with *multiext output files* are eligible for caching. The reason is that for other rules it would be impossible to unambiguously assign the output files to cache entries while being agnostic of the actual file names. Also note that the rules need to retrieve all their parameters via the `params` directive (except input files). It is not allowed to directly use `wildcards`, `config` or any global variable in the shell command or script, because these are not captured in the hash (otherwise, reuse would be unnecessarily limited).

Also note that Snakemake will store everything in the cache as readable and writeable for **all users** on the system (except in the remote case, where permissions are not enforced and depend on your storage configuration). Hence, caching is not intended for private data, just for steps that deal with publicly available resources.

Finally, be aware that the implementation should be considered **experimental** until this note is removed.

4.11 Interoperability

4.11.1 CWL export

Snakemake workflows can be exported to CWL, such that they can be executed in any CWL-enabled workflow engine. Since, CWL is less powerful for expressing workflows than Snakemake (most importantly Snakemake offers more flexible scatter-gather patterns, since full Python can be used), export works such that every Snakemake job is encoded into a single step in the CWL workflow. Moreover, every step of that workflow calls Snakemake again to execute the job. The latter enables advanced Snakemake features like scripts, benchmarks and remote files to work inside CWL. So, when exporting keep in mind that the resulting CWL file can become huge, depending on the number of jobs in your workflow. To export a Snakemake workflow to CWL, simply run

```
$ snakemake --export-cwl workflow.cwl
```

The resulting workflow will by default use the Snakemake docker image for every step, but this behavior can be overwritten via the CWL execution environment. Then, the workflow can be executed in the same working directory with, e.g.,

```
$ cwltool workflow.cwl
```

Note that due to limitations in CWL, it seems currently impossible to avoid that all target files (output files of target jobs), are written directly to the workdir, regardless of their relative paths in the Snakefile.

Note that export is impossible in case the workflow contains *checkpoints* or output files with absolute paths.

4.12 Monitoring

Snakemake supports panoptes a server (under development) that lets you monitor the execution of snakemake workflows. Snakemake communicates with panoptes via the `--wms-monitor` flag. The flag specifies the ip and port where panoptes is running (e.g. `--wms-monitor http://127.0.0.1:5000`).

For panoptes versions 0.1.1 and lower, Snakemake sends the following requests to wms monitor:

API	Method	Data	Description
`/api/service-info`	GET	json	Snakemake gets the status of panoptes. Snakemake continues to run if the status (`json['status']`) is `'running'`. In all other cases snakemake exits with an error message.
`/create_workflow`	GET	json	Snakemake gets a unique id/name `str(uuid.uuid4())` for each workflow triggered.
`/update_workflow_status`	POST	dictionary	Snakemake posts updates for workflows/jobs. The dictionary sent contains the log message dictionary , the current timestamp and the unique id/name of the workflow. `{ 'msg': repr(msg), 'timestamp': time.asctime(), 'id': id }`

For future versions, Panoptes will implement`a more structured schema <https://github.com/panoptes-organization/monitor-schema>`_ to interact with the server. This means that for Snakemake 3.30.1 and lower, you should use Panoptes 0.1.1 and lower. The documentation here will be updated when a new version of Panoptes with the Monitor Schema is released.

4.13 Writing Workflows

In Snakemake, workflows are specified as Snakefiles. Inspired by GNU Make, a Snakefile contains rules that denote how to create output files from input files. Dependencies between rules are handled implicitly, by matching filenames of input files against output files. Thereby wildcards can be used to write general rules.

4.13.1 Grammar

The Snakefile syntax obeys the following grammar, given in extended Backus-Naur form (EBNF)

```
snakemake    = statement | rule | include | workdir | module | configfile | container
rule         = "rule" (identifier | "") ":" ruleparams
include      = "include:" stringliteral
workdir      = "workdir:" stringliteral
module       = "module" identifier ":" moduleparams
configfile   = "configfile" ":" stringliteral
userule      = "use" "rule" (identifier | "*") "from" identifier ["as" identifier] [
↪"with" ":" norunparams]
ni           = NEWLINE INDENT
norunparams  = [ni input] [ni output] [ni params] [ni message] [ni threads] [ni␣
↪resources] [ni log] [ni conda] [ni container] [ni benchmark] [ni cache]
ruleparams   = norunparams [ni (run | shell | script | notebook)] NEWLINE snakemake
input        = "input" ":" parameter_list
output       = "output" ":" parameter_list
params       = "params" ":" parameter_list
log          = "log" ":" parameter_list
benchmark    = "benchmark" ":" statement
cache        = "cache" ":" bool
message      = "message" ":" stringliteral
threads      = "threads" ":" integer
resources    = "resources" ":" parameter_list
version      = "version" ":" statement
conda        = "conda" ":" stringliteral
container    = "container" ":" stringliteral
run          = "run" ":" ni statement
shell        = "shell" ":" stringliteral
script       = "script" ":" stringliteral
notebook     = "notebook" ":" stringliteral
moduleparams = [ni snakefile] [ni metawrapper] [ni config] [ni skipval]
snakefile    = "snakefile" ":" stringliteral
metawrapper  = "meta_wrapper" ":" stringliteral
config       = "config" ":" stringliteral
skipval      = "skip_validation" ":" stringliteral
```

while all not defined non-terminals map to their Python equivalents.

Depend on a Minimum Snakemake Version

From Snakemake 3.2 on, if your workflow depends on a minimum Snakemake version, you can easily ensure that at least this version is installed via

```
from snakemake.utils import min_version

min_version("3.2")
```

given that your minimum required version of Snakemake is 3.2. The statement will raise a WorkflowError (and therefore abort the workflow execution) if the version is not met.

4.14 Snakefiles and Rules

A Snakemake workflow defines a data analysis in terms of rules that are specified in the Snakefile. Most commonly, rules consist of a name, input files, output files, and a shell command to generate the output from the input:

```
rule NAME:
    input: "path/to/inputfile", "path/to/other/inputfile"
    output: "path/to/outputfile", "path/to/another/outputfile"
    shell: "somecommand {input} {output}"
```

The name is optional and can be left out, creating an anonymous rule. It can also be overridden by setting a rule's `name` attribute.

Note

Note that any placeholders in the shell command (like `{input}`) are always evaluated and replaced when the corresponding job is executed, even if they are occuring inside a comment. To avoid evaluation and replacement, you have to mask the braces by doubling them, i.e. `{{input}}`.

Inside the shell command, all local and global variables, especially input and output files can be accessed via their names in the python format minilanguage. Here, input and output (and in general any list or tuple) automatically evaluate to a space-separated list of files (i.e. `path/to/inputfile path/to/other/inputfile`). From Snakemake 3.8.0 on, adding the special formatting instruction `:q` (e.g. `"somecommand {input:q} {output:q}"`) will let Snakemake quote each of the list or tuple elements that contains whitespace.

By default shell commands will be invoked with `bash` shell (unless the workflow specifies a different default shell via `shell.executable(...)`).

Instead of a shell command, a rule can run some python code to generate the output:

```
rule NAME:
    input: "path/to/inputfile", "path/to/other/inputfile"
    output: "path/to/outputfile", somename = "path/to/another/outputfile"
    run:
        for f in input:
            ...
            with open(output[0], "w") as out:
                out.write(...)
        with open(output.somename, "w") as out:
            out.write(...)
```

As can be seen, instead of accessing input and output as a whole, we can also access by index (`output[0]`) or by keyword (`output.somename`). Note that, when adding keywords or names for input or output files, their order won't be preserved when accessing them as a whole via e.g. `{output}` in a shell command.

Shell commands like above can also be invoked inside a python based rule, via the function `shell` that takes a string with the command and allows the same formatting like in the rule above, e.g.:

```
shell("somecommand {output.somename}")
```

Further, this combination of python and shell commands allows us to iterate over the output of the shell command, e.g.:

```
for line in shell("somecommand {output.somename}", iterable=True):
    ... # do something in python
```

Note that shell commands in Snakemake use the bash shell in strict mode by default.

4.14.1 Wildcards

Usually, it is useful to generalize a rule to be applicable to a number of e.g. datasets. For this purpose, wildcards can be used. Automatically resolved multiple named wildcards are a key feature and strength of Snakemake in comparison to other systems. Consider the following example.

```
rule complex_conversion:
    input:
        "{dataset}/inputfile"
    output:
        "{dataset}/file.{group}.txt"
    shell:
        "somecommand --group {wildcards.group} < {input} > {output}"
```

Here, we define two wildcards, `dataset` and `group`. By this, the rule can produce all files that follow the regular expression pattern `.+/file\..+\.txt`, i.e. the wildcards are replaced by the regular expression `.+`. If the rule's output matches a requested file, the substrings matched by the wildcards are propagated to the input files and to the variable wildcards, that is here also used in the shell command. The wildcards object can be accessed in the same way as input and output, which is described above.

For example, if another rule in the workflow requires the file `101/file.A.txt`, Snakemake recognizes that this rule is able to produce it by setting `dataset=101` and `group=A`. Thus, it requests file `101/inputfile` as input and executes the command `somecommand --group A < 101/inputfile > 101/file.A.txt`. Of course, the input file might have to be generated by another rule with different wildcards.

Importantly, the wildcard names in input and output must be named identically. Most typically, the same wildcard is present in both input and output, but it is of course also possible to have wildcards only in the output but not the input section.

Multiple wildcards in one filename can cause ambiguity. Consider the pattern `{dataset}.{group}.txt` and assume that a file `101.B.normal.txt` is available. It is not clear whether `dataset=101.B` and `group=normal` or `dataset=101` and `group=B.normal` in this case.

Hence wildcards can be constrained to given regular expressions. Here we could restrict the wildcard `dataset` to consist of digits only using `\d+` as the corresponding regular expression. With Snakemake 3.8.0, there are three ways to constrain wildcards. First, a wildcard can be constrained within the file pattern, by appending a regular expression separated by a comma:

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

Second, a wildcard can be constrained within the rule via the keyword `wildcard_constraints`:

```
rule complex_conversion:
    input:
        "{dataset}/inputfile"
    output:
        "{dataset}/file.{group}.txt"
    wildcard_constraints:
        dataset="\d+"
    shell:
        "somecommand --group {wildcards.group}  < {input}  > {output}"
```

Finally, you can also define global wildcard constraints that apply for all rules:

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

rule a:
```

(continues on next page)

(continued from previous page)

```
    ...

rule b:
    ...
```

See the Python documentation on regular expressions for detailed information on regular expression syntax.

4.14.2 Aggregation

Input files can be Python lists, allowing to easily aggregate over parameters or samples:

```
rule aggregate:
    input:
        ["{dataset}/a.txt".format(dataset=dataset) for dataset in DATASETS]
    output:
        "aggregated.txt"
    shell:
        ...
```

The above expression can be simplified in two ways.

The expand function

```
rule aggregate:
    input:
        expand("{dataset}/a.txt", dataset=DATASETS)
    output:
        "aggregated.txt"
    shell:
        ...
```

Note that *dataset* is NOT a wildcard here because it is resolved by Snakemake due to the `expand` statement. The `expand` function also allows us to combine different variables, e.g.

```
rule aggregate:
    input:
        expand("{dataset}/a.{ext}", dataset=DATASETS, ext=FORMATS)
    output:
        "aggregated.txt"
    shell:
        ...
```

If `FORMATS=["txt", "csv"]` contains a list of desired output formats then expand will automatically combine any dataset with any of these extensions.

Furthermore, the first argument can also be a list of strings. In that case, the transformation is applied to all elements of the list. E.g.

```
expand(["{dataset}/a.{ext}", "{dataset}/b.{ext}"], dataset=DATASETS, ext=FORMATS)
```

leads to

```
["ds1/a.txt", "ds1/b.txt", "ds2/a.txt", "ds2/b.txt", "ds1/a.csv", "ds1/b.csv", "ds2/a.
↪csv", "ds2/b.csv"]
```

Per default, `expand` uses the python itertools function `product` that yields all combinations of the provided wildcard values. However by inserting a second positional argument this can be replaced by any combinatoric function, e.g. `zip`:

```
expand(["{dataset}/a.{ext}", "{dataset}/b.{ext}"], zip, dataset=DATASETS, ext=FORMATS)
```

leads to

```
["ds1/a.txt", "ds1/b.txt", "ds2/a.csv", "ds2/b.csv"]
```

You can also mask a wildcard expression in `expand` such that it will be kept, e.g.

```
expand("{{dataset}}/a.{ext}", ext=FORMATS)
```

will create strings with all values for ext but starting with the wildcard `"{dataset}"`.

The multiext function

`multiext` provides a simplified variant of `expand` that allows us to define a set of output or input files that just differ by their extension:

```
rule plot:
    input:
        ...
    output:
        multiext("some/plot", ".pdf", ".svg", ".png")
    shell:
        ...
```

The effect is the same as if you would write `expand("some/plot{ext}", ext=[".pdf", ".svg", ".png"])`, however, using a simpler syntax. Moreover, defining output with `multiext` is the only way to use *between workflow caching* for rules with multiple output files.

4.14.3 Targets and aggregation

By default snakemake executes the first rule in the snakefile. This gives rise to pseudo-rules at the beginning of the file that can be used to define build-targets similar to GNU Make:

```
rule all:
  input:
    expand("{dataset}/file.A.txt", dataset=DATASETS)
```

Here, for each dataset in a python list `DATASETS` defined before, the file `{dataset}/file.A.txt` is requested. In this example, Snakemake recognizes automatically that these can be created by multiple applications of the rule `complex_conversion` shown above.

It is possible to overwrite this behavior to use the first rule as a default target, by explicitly marking a rule as being the default target via the `default_target` directive:

```
rule xy:
    input:
        expand("{dataset}/file.A.txt", dataset=DATASETS)
    default_target: True
```

Regardless of where this rule appears in the Snakefile, it will be the default target. Usually, it is still recommended to keep the default target rule (and in fact all other rules that could act as optional targets) at the top of the file, such that it

can be easily found. The `default_target` directive becomes particularly useful when *combining several pre-existing workflows*.

4.14.4 Threads

Further, a rule can be given a number of threads to use, i.e.

```
rule NAME:
    input: "path/to/inputfile", "path/to/other/inputfile"
    output: "path/to/outputfile", "path/to/another/outputfile"
    threads: 8
    shell: "somecommand --threads {threads} {input} {output}"
```

Note

On a cluster node, Snakemake uses as many cores as available on that node. Hence, the number of threads used by a rule never exceeds the number of physically available cores on the node. Note: This behavior is not affected by `--local-cores`, which only applies to jobs running on the main node.

Snakemake can alter the number of cores available based on command line options. Therefore it is useful to propagate it via the built in variable `threads` rather than hardcoding it into the shell command. In particular, it should be noted that the specified threads have to be seen as a maximum. When Snakemake is executed with fewer cores, the number of threads will be adjusted, i.e. `threads = min(threads, cores)` with `cores` being the number of cores specified at the command line (option `--cores`).

Hardcoding a particular maximum number of threads like above is useful when a certain tool has a natural maximum beyond which parallelization won't help to further speed it up. This is often the case, and should be evaluated carefully for production workflows. Also, setting a `threads:` maximum is required to achieve parallelism in tools that (often implicitly and without the user knowing) rely on an environment variable for the maximum of cores to use. For example, this is the case for many linear algebra libraries and for OpenMP. Snakemake limits the respective environment variables to one core by default, to avoid unexpected and unlimited core-grabbing, but will override this with the `threads:` you specify in a rule (the parameters set to `threads:`, or defaulting to 1, are: `OMP_NUM_THREADS`, `GOTO_NUM_THREADS`, `OPENBLAS_NUM_THREADS`, `MKL_NUM_THREADS`, `VECLIB_MAXIMUM_THREADS`, `NUMEXPR_NUM_THREADS`).

If it is certain that no maximum for efficient parallelism exists for a tool, one can instead define threads as a function of the number of cores given to Snakemake:

```
rule NAME:
    input: "path/to/inputfile", "path/to/other/inputfile"
    output: "path/to/outputfile", "path/to/another/outputfile"
    threads: workflow.cores * 0.75
    shell: "somecommand --threads {threads} {input} {output}"
```

The number of given cores is globally available in the Snakefile as an attribute of the workflow object: `workflow.cores`. Any arithmetic operation can be performed to derive a number of threads from this. E.g., in the above example, we reserve 75% of the given cores for the rule. Snakemake will always round the calculated value down (while enforcing a minimum of 1 thread).

Starting from version 3.7, threads can also be a callable that returns an `int` value. The signature of the callable should be `callable(wildcards[, input])` (input is an optional parameter). It is also possible to refer to a predefined variable (e.g, `threads: threads_max`) so that the number of cores for a set of rules can be changed with one change only by altering the value of the variable `threads_max`.

4.14.5 Resources

In addition to threads, a rule can use arbitrary user-defined resources by specifying them with the resources-keyword:

```
rule a:
    input:     ...
    output:    ...
    resources:
        mem_mb=100
    shell:
        "..."
```

If limits for the resources are given via the command line, e.g.

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

the scheduler will ensure that the given resources are not exceeded by running jobs. Resources are always meant to be specified as total per job, not by thread (i.e. above `mem_mb=100` in rule `a` means that any job from rule `a` will require `100` megabytes of memory in total, and not per thread).

Importantly, there are some *standard resources* that should be considered before making up your own.

In general, resources are just names to the Snakemake scheduler, i.e., Snakemake does not check on the resource consumption of jobs in real time. Instead, resources are used to determine which jobs can be executed at the same time without exceeding the limits specified at the command line. Apart from making Snakemake aware of hybrid-computing architectures (e.g. with a limited number of additional devices like GPUs) this allows us to control scheduling in various ways, e.g. to limit IO-heavy jobs by assigning an artificial IO-resource to them and limiting it via the `--resources` flag. If no limits are given, the resources are ignored in local execution.

Resources can have any arbitrary name, and must be assigned `int` or `str` values. They can also be callables that return `int`, `str` or `None` values. In case of `None`, the resource is considered to be unset (i.e. ignored) in the rule. The signature of the callable must be `callable(wildcards [, input] [, threads] [, attempt])` (`input`, `threads`, and `attempt` are optional parameters).

The parameter `attempt` allows us to adjust resources based on how often the job has been restarted (see *All Options*, option `--retries`). This is handy when executing a Snakemake workflow in a cluster environment, where jobs can e.g. fail because of too limited resources. When Snakemake is executed with `--retries 3`, it will try to restart a failed job 3 times before it gives up. Thereby, the parameter `attempt` will contain the current attempt number (starting from `1`). This can be used to adjust the required memory as follows

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

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

Here, the first attempt will require 100 MB memory, the second attempt will require 200 MB memory and so on. When passing memory requirements to the cluster engine, you can by this automatically try out larger nodes if it turns out to be necessary.

Another application of callables as resources is when memory usage depends on the number of threads:

```
def get_mem_mb(wildcards, threads):
    return threads * 150

rule b:
    input:      ...
    output:     ...
    threads: 8
    resources:
        mem_mb=get_mem_mb
    shell:
        "..."
```

Here, the value the function `get_mem_mb` returns grows linearly with the number of threads. Of course, any other arithmetic could be performed in that function.

Both threads and resources can be overwritten upon invocation via *–set-threads* and *–set-resources*, see user_manual-snakemake_options.

Standard Resources

There are several **standard resources**, for total memory, disk usage, runtime, and the temporary directory of a job: `mem`, `disk`, `runtime`, and `tmpdir`. All of these resources have specific meanings understood by snakemake and are treated in varying unique ways:

- The `tmpdir` resource automatically leads to setting the `$TMPDIR` variable for shell commands, scripts, wrappers and notebooks. In cluster or cloud setups, its evaluation is delayed until the actual execution of the job. This way, it can dynamically react on the context of the node of execution.
- The `runtime` resource indicates the amount of wall clock time a job needs to run. It can be given as string defining a time span or as integer defining **minutes**. In the former case, the time span can be defined as a string with a number followed by a unit (`ms`, `s`, `m`, `h`, `d`, `w`, `y` for seconds, minutes, hours, days, and years, respectively). The interpretation happens via the humanfriendly package. Cluster or cloud backends may use this to constrain the allowed execution time of the submitted job. See the section below for more information.
- `disk` and `mem` define the amount of memory and disk space needed by the job. They are given as strings with a number followed by a unit (`B`, `KB`, `MB`, `GB`, `TB`, `PB`, `KiB`, `MiB`, `GiB`, `TiB`, `PiB`). The interpretation of the definition happens via the humanfriendly package. Alternatively, the two can be directly defined as integers via the resources `mem_mb` and `disk_mb` (to which `disk` and `mem` are also automatically translated internally). They are both locally scoped by default, a fact important for cluster and compute execution. See below for more info. They are usually passed to execution backends, e.g. to allow the selection of appropriate compute nodes for the job execution.

Because of these special meanings, the above names should always be used instead of possible synonyms (e.g. `tmp`, `time`, `temp`, etc).

Default Resources

Since it could be cumbersome to define these standard resources for every rule, you can set default values at the terminal or in a *profile*. This works via the command line flag `--default-resources`, see `snakemake --help` for more information. If those resource definitions are mandatory for a certain execution mode, Snakemake will fail with a hint if they are missing. Any resource definitions inside a rule override what has been defined with `--default-resources`. If `--default-resources` are not specified, Snakemake uses `'mem_mb=max(2*input.size_mb, 1000)'`, `'disk_mb=max(2*input.size_mb, 1000)'`, and `'tmpdir=system_tmpdir'`. The latter points to whatever is the default of the operating system or specified by any of the environment variables `$TMPDIR`, `$TEMP`, or `$TMP` as outlined here. If `--default-resources` is specified with some definitions, but any of the above defaults (e.g.

mem_mb) is omitted, these are still used. In order to explicitly unset these defaults, assign them a value of None, e.g. --default-resources mem_mb=None.

Resources and Remote Execution

New to Snakemake 7.11. In cluster or cloud execution, resources may represent either a global constraint across all submissions (e.g. number of API calls per second), or a constraint local to each specific job sumbmission (e.g. the amount of memory available on a node). Snakemake distinguishes between these two types of constraints using **resource scopes**. By default, mem_mb, disk_mb, and threads are all considered "local" resources, meaning specific to individual submissions. So if a constraint of 16G of memory is given to snakemake (e.g. snakemake --resources mem_mb=16000), each group job will be allowed 16G of memory. All other resources are considered "global", meaning they are tracked across all jobs across all submissions. For example, if api_calls was limited to 5 and each job scheduled used 1 api call, only 5 jobs would be scheduled at a time, even if more job submissions were available.

These resource scopes may be modified both in the Snakefile and via the CLI parameter --set-resource-scopes. The CLI parameter takes priority. Modification in the Snakefile uses the following syntax:

```
resource_scopes:
    gpus="local",
    foo="local",
    disk_mb="global"
```

Here, we set both gpus and foo as local resources, and we changed disk_mb from its default to be a global resource. These options could be overriden at the command line using:

```
$ snakemake --set-resource-scopes gpus=global disk_mb=local
```

Resources and Group Jobs

New to Snakemake 7.11. When submitting *group jobs* to the cluster, Snakemake calculates how many resources to request by first determining which component jobs can be run in parallel, and which must be run in series. For most resources, such as mem_mb or threads, a sum will be taken across each parallel layer. The layer requiring the most resource (i.e. max()) will determine the final amount requested. The only exception is runtime. For it, max() will be used within each layer, then the total amount of time across all layers will be summed. If resource constraints are provided (via --resources or --cores) Snakemake will prevent group jobs from requesting more than the constraint. Jobs that could otherwise be run in parallel will be run in series to prevent the violation of resource constraints.

Preemptible Jobs

You can specify parameters preemptible-rules and preemption-default to request a Google Cloud preemptible virtual machine for use with the Google Life Sciences Executor. There are several ways to go about doing this. This first example will use preemptible instances for all rules, with 10 repeats (restarts of the instance if it stops unexpectedly).

```
snakemake --preemption-default 10
```

If your preference is to set a default but then overwrite some rules with a custom value, this is where you can use --preemtible-rules:

```
snakemake --preemption-default 10 --preemptible-rules map_reads=3 call_variants=0
```

The above statement says that we want to use preemtible instances for all steps, defaulting to 10 retries, but for the steps "map_reads" and "call_variants" we want to apply 3 and 0 retries, respectively. The final option is to not use preemptible instances by default, but only for a particular rule:

```
snakemake --preemptible-rules map_reads=10
```

Note that this is currently implemented for the Google Life Sciences API.

GPU Resources

The Google Life Sciences API currently has support for NVIDIA GPUs, meaning that you can request a number of NVIDIA GPUs explicitly by adding `nvidia_gpu` or `gpu` to your Snakefile resources for a step:

```
rule a:
    output:
        "test.txt"
    resources:
        nvidia_gpu=1
    shell:
        "somecommand ..."
```

A specific gpu model can be requested using `gpu_model` and lowercase identifiers like `nvidia-tesla-p100` or `nvidia-tesla-p4`, for example: `gpu_model="nvidia-tesla-p100"`. If you don't specify `gpu` or `nvidia_gpu` with a count, but you do specify a `gpu_model`, the count will default to 1.

4.14.6 Messages

When executing snakemake, a short summary for each running rule is given to the console. This can be overridden by specifying a message for a rule:

```
rule NAME:
    input: "path/to/inputfile", "path/to/other/inputfile"
    output: "path/to/outputfile", "path/to/another/outputfile"
    threads: 8
    message: "Executing somecommand with {threads} threads on the following files
↪{input}."
    shell: "somecommand --threads {threads} {input} {output}"
```

Note that access to wildcards is also possible via the variable `wildcards` (e.g, `{wildcards.sample}`), which is the same as with shell commands. It is important to have a namespace around wildcards in order to avoid clashes with other variable names.

4.14.7 Priorities

Snakemake allows for rules that specify numeric priorities:

```
rule:
  input: ...
  output: ...
  priority: 50
  shell: ...
```

Per default, each rule has a priority of 0. Any rule that specifies a higher priority, will be preferred by the scheduler over all rules that are ready to execute at the same time without having at least the same priority.

Furthermore, the `--prioritize` or `-P` command line flag allows to specify files (or rules) that shall be created with highest priority during the workflow execution. This means that the scheduler will assign the specified target and all its dependencies highest priority, such that the target is finished as soon as possible. The `--dry-run` (equivalently `--dryrun`) or `-n` option allows you to see the scheduling plan including the assigned priorities.

4.14.8 Log-Files

Each rule can specify a log file where information about the execution is written to:

```
rule abc:
    input: "input.txt"
    output: "output.txt"
    log: "logs/abc.log"
    shell: "somecommand --log {log} {input} {output}"
```

Log files can be used as input for other rules, just like any other output file. However, unlike output files, log files are not deleted upon error. This is obviously necessary in order to discover causes of errors which might become visible in the log file.

The variable `log` can be used inside a shell command to tell the used tool to which file to write the logging information. The log file has to use the same wildcards as output files, e.g.

```
log: "logs/abc.{dataset}.log"
```

For programs that do not have an explicit `log` parameter, you may always use `2> {log}` to redirect standard output to a file (here, the `log` file) in Linux-based systems. Note that it is also supported to have multiple (named) log files being specified:

```
rule abc:
    input: "input.txt"
    output: "output.txt"
    log: log1="logs/abc.log", log2="logs/xyz.log"
    shell: "somecommand --log {log.log1} METRICS_FILE={log.log2} {input} {output}"
```

4.14.9 Non-file parameters for rules

Sometimes you may want to define certain parameters separately from the rule body. Snakemake provides the `params` keyword for this purpose:

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

The `params` keyword allows you to specify additional parameters depending on the wildcards values. This allows you to circumvent the need to use `run:` and python code for non-standard commands like in the above case. Here, the command `somecommand` expects the prefix of the output file instead of the actual one. The `params` keyword helps here since you cannot simply add the prefix as an output file (as the file won't be created, Snakemake would throw an error after execution of the rule).

Furthermore, for enhanced readability and clarity, the `params` section is also an excellent place to name and assign parameters and variables for your subsequent command.

Similar to `input`, `params` can take functions as well (see *Input functions*), e.g. you can write

```
rule:
    input:
        ...
    params:
        prefix=lambda wildcards, output: output[0][:-4]
    output:
        "somedir/{sample}.csv"
    shell:
        "somecommand -o {params.prefix}"
```

Note

When accessing auxiliary source files (i.e. files that are located relative to the current Snakefile, e.g. some additional configuration) it is crucial to not manually build their path but rather rely on Snakemake's special registration for these files, see *Accessing auxiliary source files*.

to get the same effect as above. Note that in contrast to the `input` directive, the `params` directive can optionally take more arguments than only `wildcards`, namely `input`, `output`, `threads`, and `resources`. From the Python perspective, they can be seen as optional keyword arguments without a default value. Their order does not matter, apart from the fact that `wildcards` has to be the first argument. In the example above, this allows you to derive the prefix name from the output file.

4.14.10 External scripts

A rule can also point to an external script instead of a shell command or inline Python code, e.g.

Python

```
rule NAME:
    input:
        "path/to/inputfile",
        "path/to/other/inputfile"
    output:
        "path/to/outputfile",
        "path/to/another/outputfile"
    script:
        "scripts/script.py"
```

Note

It is possible to refer to wildcards and params in the script path, e.g. by specifying `"scripts/{params.scriptname}.py"` or `"scripts/{wildcards.scriptname}.py"`.

The script path is always relative to the Snakefile containing the directive (in contrast to the input and output file paths, which are relative to the working directory). It is recommended to put all scripts into a subfolder `scripts` as above. Inside the script, you have access to an object `snakemake` that provides access to the same objects that are available

in the `run` and `shell` directives (input, output, params, wildcards, log, threads, resources, config), e.g. you can use `snakemake.input[0]` to access the first input file of above rule.

An example external Python script could look like this:

```
def do_something(data_path, out_path, threads, myparam):
    # python code

do_something(snakemake.input[0], snakemake.output[0], snakemake.threads, snakemake.
↪config["myparam"])
```

You can use the Python debugger from within the script if you invoke Snakemake with `--debug`.

R and R Markdown

Apart from Python scripts, this mechanism also allows you to integrate R and R Markdown scripts with Snakemake, e.g.

```
rule NAME:
    input:
        "path/to/inputfile",
        "path/to/other/inputfile"
    output:
        "path/to/outputfile",
        "path/to/another/outputfile"
    script:
        "scripts/script.R"
```

In the R script, an S4 object named `snakemake` analogous to the Python case above is available and allows access to input and output files and other parameters. Here the syntax follows that of S4 classes with attributes that are R lists, e.g. we can access the first input file with `snakemake@input[[1]]` (note that the first file does not have index `0` here, because R starts counting from `1`). Named input and output files can be accessed in the same way, by just providing the name instead of an index, e.g. `snakemake@input[["myfile"]]`.

An equivalent script (*to the Python one above*) written in R would look like this:

```
do_something <- function(data_path, out_path, threads, myparam) {
    # R code
}

do_something(snakemake@input[[1]], snakemake@output[[1]], snakemake@threads,␣
↪snakemake@config[["myparam"]])
```

To debug R scripts, you can save the workspace with `save.image()`, and invoke R after Snakemake has terminated. Then you can use the usual R debugging facilities while having access to the `snakemake` variable. It is best practice to wrap the actual code into a separate function. This increases the portability if the code shall be invoked outside of Snakemake or from a different rule. A convenience method, `snakemake@source()`, acts as a wrapper for the normal R `source()` function, and can be used to source files relative to the original script directory.

An R Markdown file can be integrated in the same way as R and Python scripts, but only a single output (html) file can be used:

```
rule NAME:
    input:
        "path/to/inputfile",
        "path/to/other/inputfile"
    output:
        "path/to/report.html",
```

(continues on next page)

(continued from previous page)

```
    script:
        "path/to/report.Rmd"
```

In the R Markdown file you can insert output from a R command, and access variables stored in the S4 object named `snakemake`

```
---
title: "Test Report"
author:
    - "Your Name"
date: "`r format(Sys.time(), '%d %B, %Y')`"
params:
   rmd: "report.Rmd"
output:
  html_document:
  highlight: tango
  number_sections: no
  theme: default
  toc: yes
  toc_depth: 3
  toc_float:
    collapsed: no
    smooth_scroll: yes
---

## R Markdown

This is an R Markdown document.

Test include from snakemake `r snakemake@input`.

## Source
<a download="report.Rmd" href="`r base64enc::dataURI(file = params$rmd, mime = 'text/
↪rmd', encoding = 'base64')`">R Markdown source file (to produce this document)</a>
```

A link to the R Markdown document with the snakemake object can be inserted. Therefore a variable called `rmd` needs to be added to the `params` section in the header of the `report.Rmd` file. The generated R Markdown file with snakemake object will be saved in the file specified in this `rmd` variable. This file can be embedded into the HTML document using base64 encoding and a link can be inserted as shown in the example above. Also other input and output files can be embedded in this way to make a portable report. Note that the above method with a data URI only works for small files. An experimental technology to embed larger files is using Javascript Blob object.

Julia

```
rule NAME:
    input:
        "path/to/inputfile",
        "path/to/other/inputfile"
    output:
        "path/to/outputfile",
        "path/to/another/outputfile"
    script:
        "path/to/script.jl"
```

In the Julia script, a `snakemake` object is available, which can be accessed similar to the *Python case*, with the only

difference that you have to index from 1 instead of 0.

Rust

```
rule NAME:
    input:
        "path/to/inputfile",
        "path/to/other/inputfile",
        named_input="path/to/named/inputfile",
    output:
        "path/to/outputfile",
        "path/to/another/outputfile"
    params:
        seed=4
    conda:
        "rust.yaml"
    log:
        stdout="path/to/stdout.log",
        stderr="path/to/stderr.log",
    script:
        "path/to/script.rs"
```

The ability to execute Rust scripts is facilitated by `rust-script`. As such, the script must be a valid `rust-script` script and `rust-script` (plus OpenSSL and a C compiler toolchain, provided by Conda packages `openssl`, `c-compiler`, `pkg-config`) must be available in the environment the rule is run in. The minimum required `rust-script` version is 1.15.0, so in the example above, the contents of `rust.yaml` might look like this:

```
channels:
  - conda-forge
  - bioconda
dependencies:
  - rust-script>=0.15.0
  - openssl
  - c-compiler
  - pkg-config
```

Some example scripts can be found in the tests directory.

In the Rust script, a `snakemake` instance is available, which is automatically generated from the python snakemake object using `json_typegen`. It usually looks like this:

```
pub struct Snakemake {
    input: Input,
    output: Ouput,
    params: Params,
    wildcards: Wildcards,
    threads: u64,
    log: Log,
    resources: Resources,
    config: Config,
    rulename: String,
    bench_iteration: Option<usize>,
    scriptdir: String,
}
```

Any named parameter is translated to a corresponding `field_name: Type`, such that `params.seed` from the example above can be accessed just like in python, i.e.:

```
let seed = snakemake.params.seed;
assert_eq!(seed, 4);
```

Positional arguments for `input`, `output`, `log` and `wildcards` can be accessed by index and iterated over:

```
let input = &snakemake.input;

// Input implements Index<usize>
let inputfile = input[0];
assert_eq!(inputfile, "path/to/inputfile");

// Input implements IntoIterator
//
// prints
// > 'path/to/inputfile'
// > 'path/to/other/inputfile'
for f in input {
    println!("> '{}'", &f);
}
```

It is also possible to redirect `stdout` and `stderr`:

```
println!("This will NOT be written to path/to/stdout.log");
// redirect stdout to "path/to/stdout.log"
let _stdout_redirect = snakemake.redirect_stdout(snakemake.log.stdout)?;
println!("This will be written to path/to/stdout.log");

// redirect stderr to "path/to/stderr.log"
let _stderr_redirect = snakemake.redirect_stderr(snakemake.log.stderr)?;
eprintln!("This will be written to path/to/stderr.log");
drop(_stderr_redirect);
eprintln!("This will NOT be written to path/to/stderr.log");
```

Redirection of stdout/stderr is only "active" as long as the returned `Redirect` instance is alive; in order to stop redirecting, drop the respective instance.

In order to work, rust-script support for snakemake has some dependencies enabled by default:

1. `anyhow=1`, for its `Result` type
2. `gag=1`, to enable stdout/stderr redirects
3. `json_typegen=0.6`, for generating rust structs from a json representation of the snakemake object
4. `lazy_static=1.4`, to make a `snakemake` instance easily accessible
5. `serde=1`, explicit dependency of `json_typegen`
6. `serde_derive=1`, explicit dependency of `json_typegen`
7. `serde_json=1`, explicit dependency of `json_typegen`

If your script uses any of these packages, you do not need to `use` them in your script. Trying to `use` them will cause a compilation error.

Bash

Bash scripts work much the same as the other script languages above, but with some important differences. Access to the rule's directives is provided through the use of associative arrays - **requiring Bash version 4.0 or greater**. One "limitation" of associative arrays is they cannot be nested. As such, the following rule directives are found in a separate variable, named as `snakemake_<directive>`:

- `input`
- `output`
- `log`
- `wildcards`
- `resources`
- `params`
- `config`

Access to the `input` directive is faciliated through the bash associative array named `snakemake_input`. The remaining directives can be found in the variable `snakemake`.

Note

As arrays cannot be nested in Bash, use of python's `dict` in directives is not supported. So, adding a `params` key of `data={"foo": "bar"}` will not be reflected - `${snakemake_params[data]}` actually only returns `"foo"`.

Bash Example 1

```
rule align:
    input:
        "{sample}.fq",
        reference="ref.fa",
    output:
        "{sample}.sam"
    params:
        opts="-a -x map-ont",
    threads: 4
    log:
        "align/{sample}.log"
    conda:
        "envs/align.yaml"
    script:
        "scripts/align.sh"
```

`align.sh`

```
#!/usr/bin/env bash

echo "Aligning sample ${snakemake_wildcards[sample]} with minimap2" 2> "${snakemake_
↪log[0]}"

minimap2 ${snakemake_params[opts]} -t ${snakemake[threads]} "${snakemake_
```

(continues on next page)

(continued from previous page)

```
↪input[reference]}" \
    "${snakemake_input[0]}" > "${snakemake_output[0]}" 2>> "${snakemake_log[0]}"
```

If you don't add a shebang, the default `#!/usr/bin/env bash` will be inserted for you. A tutorial on how to use associative arrays can be found here.

You may also have noticed the mixed use of double-quotes when accessing some variables. It is generally good practice in Bash to double-quote variables for which you want to prevent word splitting; generally, you will want to double-quote any variable that could contain a file name. However, in some cases, word splitting *is* desired, such as `${snakemake_params[opts]}` in the above example.

Bash Example 2

```
rule align:
    input:
        reads=["{sample}_R1.fq", "{sample}_R2.fq]"],
        reference="ref.fa",
    output:
        "{sample}.sam"
    params:
        opts="-M",
    threads: 4
    log:
        "align/{sample}.log"
    conda:
        "envs/align.yaml"
    script:
        "scripts/align.sh"
```

In this example, the `input` variable `reads`, which is a python list, actually gets stored as a space-separated string in Bash because, you guessed it, you can't nest arrays in Bash! So in order to access the individual members, we turn the string into an array; allowing us to access individual elements of the list/array. See this stackoverflow question for other solutions.

`align.sh`

```
#!/usr/bin/env bash

exec 2> "${snakemake_log[0]}"  # send all stderr from this script to the log file

reads=(${snakemake_input[reads]})  # don't double-quote this - we want word splitting

r1="${reads[0]}"
r2="${reads[1]}"

bwa index "${snakemake_input[reference]}"
bwa mem ${snakemake_params[opts]} -t ${snakemake[threads]} \
    "${snakemake_input[reference]}" "$r1" "$r2" > "${snakemake_output[0]}"
```

If, in the above example, the fastq reads were not in a named variable, but were instead just a list, they would be available as `"${snakemake_input[0]}"` and `"${snakemake_input[1]}"`.

For technical reasons, scripts are executed in `.snakemake/scripts`. The original script directory is available as `scriptdir` in the `snakemake` object.

4.14.11 Jupyter notebook integration

Instead of plain scripts (see above), one can integrate Jupyter Notebooks. This enables the interactive development of data analysis components (e.g. for plotting). Integration works as follows (note the use of *notebook:* instead of *script:*):

```
rule hello:
    output:
        "test.txt"
    log:
        # optional path to the processed notebook
        notebook="logs/notebooks/processed_notebook.ipynb"
    notebook:
        "notebooks/hello.py.ipynb"
```

Note

Consider Jupyter notebook integration as a way to get the best of both worlds. A modular, readable workflow definition with Snakemake, and the ability to quickly explore and plot data with Jupyter. The benefit will be maximal when integrating many small notebooks that each do a particular job, hence allowing to get away from large monolithic, and therefore unreadable notebooks.

It is recommended to prefix the `.ipynb` suffix with either `.py` or `.r` to indicate the notebook language. In the notebook, a snakemake object is available, which can be accessed in the same way as the with script integration. In other words, you have access to input files via `snakemake.input` (in the Python case) and `snakemake@input` (in the R case) etc.. Optionally it is possible to automatically store the processed notebook. This can be achieved by adding a named logfile `notebook=...` to the `log` directive.

Note

It is possible to refer to wildcards and params in the notebook path, e.g. by specifying `"notebook/{params.name}.py"` or `"notebook/{wildcards.name}.py"`.

In order to simplify the coding of notebooks given the automatically inserted `snakemake` object, Snakemake provides an interactive edit mode for notebook rules. Let us assume you have written above rule, but the notebook does not yet exist. By running

```
snakemake --cores 1 --edit-notebook test.txt
```

you instruct Snakemake to allow interactive editing of the notebook needed to create the file `test.txt`. Snakemake will run all dependencies of the notebook rule, such that all input files are present. Then, it will start a jupyter notebook server with an empty draft of the notebook, in which you can interactively program everything needed for this particular step. Once done, you should save the notebook from the jupyter web interface, go to the jupyter dashboard and hit the `Quit` button on the top right in order to shut down the jupyter server. Snakemake will detect that the server is closed and automatically store the drafted notebook into the path given in the rule (here `hello.py.ipynb`). If the notebook already exists, above procedure can be used to easily modify it. Note that Snakemake requires local execution for the notebook edit mode. On a cluster or the cloud, you can generate all dependencies of the notebook rule via

```
snakemake --cluster ... --jobs 100 --until test.txt
```

Then, the notebook rule can easily be executed locally.

Finally, it is advisable to combine the `notebook` directive with the `conda` directive (see *Integrated Package Management*) in order to define a software stack to use. At least, this software stack should contain jupyter and the language to use (e.g. Python or R). For the above case, this means

```
rule hello:
    output:
        "test.txt"
    conda:
        "envs/hello.yaml"
    notebook:
        "notebooks/hello.py.ipynb"
```

with

```
channels:
  - conda-forge
dependencies:
  - python =3.8
  - jupyter =1.0
  - jupyterlab_code_formatter =1.4
```

The last dependency is advisable in order to enable autoformatting of notebook cells when editing. When using other languages than Python in the notebook, one needs to additionally add the respective kernel, e.g. `r-irkernel` for R support.

When using an IDE with built-in Jupyter support, an alternative to `--edit-notebook` is `--draft-notebook`. Instead of firing up a notebook server, `--draft-notebook` just creates a skeleton notebook for editing within the IDE. In addition, it prints instructions for configuring the IDE's notebook environment to use the interpreter from the Conda environment defined in the corresponding rule. For example, running

```
snakemake --cores 1 --draft-notebook test.txt --use-conda
```

will generate skeleton code in `notebooks/hello.py.ipynb` and additionally print instructions on how to open and execute the notebook in VSCode.

4.14.12 Protected and Temporary Files

A particular output file may require a huge amount of computation time. Hence one might want to protect it against accidental deletion or overwriting. Snakemake allows this by marking such a file as `protected`:

```
rule NAME:
    input:
        "path/to/inputfile"
    output:
        protected("path/to/outputfile")
    shell:
        "somecommand {input} {output}"
```

A protected file will be write-protected after the rule that produces it is completed.

Further, an output file marked as `temp` is deleted after all rules that use it as an input are completed:

```
rule NAME:
    input:
        "path/to/inputfile"
    output:
        temp("path/to/outputfile")
    shell:
        "somecommand {input} {output}"
```

4.14.13 Directories as outputs

Sometimes it can be convenient to have directories, rather than files, as outputs of a rule. As of version 5.2.0, directories as outputs have to be explicitly marked with `directory`. This is primarily for safety reasons; since all outputs are deleted before a job is executed, we don't want to risk deleting important directories if the user makes some mistake. Marking the output as `directory` makes the intent clear, and the output can be safely removed. Another reason comes down to how modification time for directories work. The modification time on a directory changes when a file or a subdirectory is added, removed or renamed. This can easily happen in not-quite-intended ways, such as when Apple macOS or MS Windows add `.DS_Store` or `thumbs.db` files to store parameters for how the directory contents should be displayed. When the `directory` flag is used a hidden file called `.snakemake_timestamp` is created in the output directory, and the modification time of that file is used when determining whether the rule output is up to date or if it needs to be rerun. Always consider if you can't formulate your workflow using normal files before resorting to using `directory()`.

```
rule NAME:
    input:
        "path/to/inputfile"
    output:
        directory("path/to/outputdir")
    shell:
        "somecommand {input} {output}"
```

4.14.14 Ignoring timestamps

For determining whether output files have to be re-created, Snakemake checks whether the file modification date (i.e. the timestamp) of any input file of the same job is newer than the timestamp of the output file. This behavior can be overridden by marking an input file as `ancient`. The timestamp of such files is ignored and always assumed to be older than any of the output files:

```
rule NAME:
    input:
        ancient("path/to/inputfile")
    output:
        "path/to/outputfile"
    shell:
        "somecommand {input} {output}"
```

Here, this means that the file `path/to/outputfile` will not be triggered for re-creation after it has been generated once, even when the input file is modified in the future. Note that any flag that forces re-creation of files still also applies to files marked as `ancient`.

4.14.15 Ensuring output file properties like non-emptyness or checksum compliance

It is possible to annotate certain additional criteria for output files to be ensured after they have been generated successfully. For example, this can be used to check for output files to be non-empty, or to compare them against a given sha256 checksum. If this functionality is used, Snakemake will check such annotated files before considering a job to be successfull. Non-emptyness can be checked as follows:

```
rule NAME:
    output:
        ensure("test.txt", non_empty=True)
    shell:
        "somecommand {output}"
```

Above, the output file `test.txt` is marked as non-empty. If the command `somecommand` happens to generate an empty output, the job will fail with an error listing the unexpected empty file.

A sha256 checksum can be compared as follows:

```
my_checksum = "u98a9cjsd98saud090923ßkpoasköf9ß32"

rule NAME:
    output:
        ensure("test.txt", sha256=my_checksum)
    shell:
        "somecommand {output}"
```

In addition to providing the checksum as plain string, it is possible to provide a pointer to a function (similar to input functions). The function has to accept a single argument that will be the wildcards object generated from the application of the rule to create some requested output files:

```
def get_checksum(wildcards):
    # e.g., look up the checksum with the value of the wildcard sample
    # in some dictionary
    return my_checksums[wildcards.sample]

rule NAME:
    output:
        ensure("test/{sample}.txt", sha256=get_checksum)
    shell:
        "somecommand {output}"
```

Note that you can also use lambda expressions instead of full function definitions.

Often, it is a good idea to combine `ensure` annotations with *retry definitions*, e.g. for retrying upon invalid checksums or empty files.

4.14.16 Shadow rules

Shadow rules result in each execution of the rule to be run in isolated temporary directories. This "shadow" directory contains symlinks to files and directories in the current workdir. This is useful for running programs that generate lots of unused files which you don't want to manually cleanup in your snakemake workflow. It can also be useful if you want to keep your workdir clean while the program executes, or simplify your workflow by not having to worry about unique filenames for all outputs of all rules.

By setting `shadow: "shallow"`, the top level files and directories are symlinked, so that any relative paths in a subdirectory will be real paths in the filesystem. The setting `shadow: "full"` fully shadows the entire subdirectory structure of the current workdir. The setting `shadow: "minimal"` only symlinks the inputs to the rule, and `shadow: "copy-minimal"` copies the inputs instead of just creating symlinks. Once the rule successfully executes, the output file will be moved if necessary to the real path as indicated by `output`.

Typically, you will not need to modify your rule for compatibility with `shadow`, unless you reference parent directories relative to your workdir in a rule.

```
rule NAME:
    input: "path/to/inputfile"
    output: "path/to/outputfile"
    shadow: "shallow"
    shell: "somecommand --other_outputs other.txt {input} {output}"
```

Shadow directories are stored one per rule execution in `.snakemake/shadow/`, and are cleared on successful execution. Consider running with the `--cleanup-shadow` argument every now and then to remove any remaining shadow

directories from aborted jobs. The base shadow directory can be changed with the `--shadow-prefix` command line argument.

4.14.17 Defining retries for fallible rules

Sometimes, rules may be expected to fail occasionally. For example, this can happen when a rule downloads some online resources. For such cases, it is possible to defined a number of automatic retries for each job from that particular rule via the `retries` directive:

```
rule a:
    output:
        "test.txt"
    retries: 3
    shell:
        "curl https://some.unreliable.server/test.txt > {output}"
```

Often, it is a good idea to combine retry functionality with ensure annotations, e.g. for retrying upon invalid checksums or empty files.

Note that it is also possible to define retries globally (via the `--retries` command line option, see *All Options*). The local definition of the rule thereby overwrites the global definition.

Importantly the `retries` directive is meant to be used for defining platform independent behavior (like adding robustness to above download command). For dealing with unreliable cluster or cloud systems, you should use the `--retries` command line option.

4.14.18 Flag files

Sometimes it is necessary to enforce some rule execution order without real file dependencies. This can be achieved by "touching" empty files that denote that a certain task was completed. Snakemake supports this via the *touch* flag:

```
rule all:
    input: "mytask.done"

rule mytask:
    output: touch("mytask.done")
    shell: "mycommand ..."
```

With the `touch` flag, Snakemake touches (i.e. creates or updates) the file `mytask.done` after `mycommand` has finished successfully.

4.14.19 Job Properties

When executing a workflow on a cluster using the `--cluster` parameter (see below), Snakemake creates a job script for each job to execute. This script is then invoked using the provided cluster submission command (e.g. `qsub`). Sometimes you want to provide a custom wrapper for the cluster submission command that decides about additional parameters. As this might be based on properties of the job, Snakemake stores the job properties (e.g. rule name, threads, input files, params etc.) as JSON inside the job script. For convenience, there exists a parser function `snakemake.utils.read_job_properties` that can be used to access the properties. The following shows an example job submission wrapper:

```
#!/usr/bin/env python3
import os
import sys

from snakemake.utils import read_job_properties

jobscript = sys.argv[1]
job_properties = read_job_properties(jobscript)

# do something useful with the threads
threads = job_properties[threads]

# access property defined in the cluster configuration file (Snakemake >=3.6.0)
job_properties["cluster"]["time"]

os.system("qsub -t {threads} {script}".format(threads=threads, script=jobscript))
```

4.14.20 Input functions

Instead of specifying strings or lists of strings as input files, snakemake can also make use of functions that return single **or** lists of input files:

```
def myfunc(wildcards):
    return [... a list of input files depending on given wildcards ...]

rule:
    input:
        myfunc
    output:
        "someoutput.{somewildcard}.txt"
    shell:
        "..."
```

The function has to accept a single argument that will be the wildcards object generated from the application of the rule to create some requested output files. Note that you can also use lambda expressions instead of full function definitions. By this, rules can have entirely different input files (both in form and number) depending on the inferred wildcards. E.g. you can assign input files that appear in entirely different parts of your filesystem based on some wildcard value and a dictionary that maps the wildcard value to file paths.

In additon to a single wildcards argument, input functions can optionally take a `groupid` (with exactly that name) as second argument, see *Group-local jobs* for details.

Finally, when implementing the input function, it is best practice to make sure that it can properly handle all possible wildcard values your rule can have. In particular, input files should not be combined with very general rules that can be applied to create almost any file: Snakemake will try to apply the rule, and will report the exceptions of your input function as errors.

For a practical example, see the *Snakemake Tutorial* (*Step 3: Input functions*).

4.14.21 Input Functions and `unpack()`

In some cases, you might want to have your input functions return named input files. This can be done by having them return `dict()` objects with the names as the dict keys and the file names as the dict values and using the `unpack()` keyword.

```
def myfunc(wildcards):
    return {'foo': '{wildcards.token}.txt'.format(wildcards=wildcards)}

rule:
    input:
        unpack(myfunc)
    output:
        "someoutput.{token}.txt"
    shell:
        "..."
```

Note that `unpack()` is only necessary for input functions returning `dict`. While it also works for `list`, remember that lists (and nested lists) of strings are automatically flattened.

Also note that if you do not pass in a *function* into the input list but you directly *call a function* then you shouldn't use `unpack()`. Here, you can simply use Python's double-star (`**`) operator for unpacking the parameters.

Note that as Snakefiles are translated into Python for execution, the same rules as for using the star and double-star unpacking Python operators apply. These restrictions do not apply when using `unpack()`.

```
def myfunc1():
    return ['foo.txt']

def myfunc2():
    return {'foo': 'nowildcards.txt'}

rule:
    input:
        *myfunc1(),
        **myfunc2(),
    output:
        "..."
    shell:
        "..."
```

4.14.22 Code Tracking

Snakemake tracks the code that was used to create your files. In combination with `--summary` or `--list-code-changes` this can be used to see what files may need a re-run because the implementation changed. Re-run can be automated by invoking Snakemake as follows:

```
$ snakemake -R `snakemake --list-code-changes`
```

4.14.23 Onstart, onsuccess and onerror handlers

Sometimes, it is necessary to specify code that shall be executed when the workflow execution is finished (e.g. cleanup, or notification of the user). With Snakemake 3.2.1, this is possible via the `onsuccess` and `onerror` keywords:

```
onsuccess:
    print("Workflow finished, no error")

onerror:
    print("An error occurred")
    shell("mail -s "an error occurred" youremail@provider.com < {log}")
```

The `onsuccess` handler is executed if the workflow finished without error. Otherwise, the `onerror` handler is executed. In both handlers, you have access to the variable `log`, which contains the path to a logfile with the complete Snakemake output. Snakemake 3.6.0 adds an `onstart` handler, that will be executed before the workflow starts. Note that dry-runs do not trigger any of the handlers.

4.14.24 Rule dependencies

From version 2.4.8 on, rules can also refer to the output of other rules in the Snakefile, e.g.:

```
rule a:
    input:  "path/to/input"
    output: "path/to/output"
    shell:  ...

rule b:
    input:  rules.a.output
    output: "path/to/output/of/b"
    shell:  ...
```

Importantly, be aware that referring to rule `a` here requires that rule `a` was defined above rule `b` in the file, since the object has to be known already. This feature also allows us to resolve dependencies that are ambiguous when using filenames.

Note that when the rule you refer to defines multiple output files but you want to require only a subset of those as input for another rule, you should name the output files and refer to them specifically:

```
rule a:
    input:  "path/to/input"
    output: a = "path/to/output", b = "path/to/output2"
    shell:  ...

rule b:
    input:  rules.a.output.a
    output: "path/to/output/of/b"
    shell:  ...
```

4.14.25 Handling Ambiguous Rules

When two rules can produce the same output file, snakemake cannot decide which one to use without additional guidance. Hence an `AmbiguousRuleException` is thrown. Note: ruleorder is not intended to bring rules in the correct execution order (this is solely guided by the names of input and output files you use), it only helps snakemake to decide which rule to use when multiple ones can create the same output file! To deal with such ambiguity, provide a `ruleorder` for the conflicting rules, e.g.

```
ruleorder: rule1 > rule2 > rule3
```

Here, `rule1` is preferred over `rule2` and `rule3`, and `rule2` is preferred over `rule3`. Only if rule1 and rule2 cannot be applied (e.g. due to missing input files), rule3 is used to produce the desired output file.

Alternatively, rule dependencies (see above) can also resolve ambiguities.

Another (quick and dirty) possiblity is to tell snakemake to allow ambiguity via a command line option

```
$ snakemake --allow-ambiguity
```

such that similar to GNU Make always the first matching rule is used. Here, a warning that summarizes the decision of snakemake is provided at the terminal.

4.14.26 Local Rules

When working in a cluster environment, not all rules need to become a job that has to be submitted (e.g. downloading some file, or a target-rule like *all*, see *Targets and aggregation*). The keyword *localrules* allows to mark a rule as local, so that it is not submitted to the cluster and instead executed on the host node:

```
localrules: all, foo

rule all:
    input: ...

rule foo:
    ...

rule bar:
    ...
```

Here, only jobs from the rule `bar` will be submitted to the cluster, whereas all and foo will be run locally. Note that you can use the localrules directive **multiple times**. The result will be the union of all declarations.

4.14.27 Benchmark Rules

Since version 3.1, Snakemake provides support for benchmarking the run times of rules. This can be used to create complex performance analysis pipelines. With the *benchmark* keyword, a rule can be declared to store a benchmark of its code into the specified location. E.g. the rule

```
rule benchmark_command:
    input:
        "path/to/input.{sample}.txt"
    output:
        "path/to/output.{sample}.txt"
    benchmark:
        "benchmarks/somecommand/{sample}.tsv"
```

(continues on next page)

(continued from previous page)

```
    shell:
        "somecommand {input} {output}"
```

benchmarks the

- CPU time (in seconds),
- wall clock time,
- memory usage (RSS, VMS, USS, PSS in megabytes),
- CPU load (CPU time divided by wall clock time),
- I/O (in bytes)

of the command `somecommand` for the given output and input files.

For this, the shell or run body of the rule is executed on that data, and all run times are stored into the given benchmark tsv file (which will contain a tab-separated table of run times and memory usage in MiB). Per default, Snakemake executes the job once, generating one run time. However, the benchmark file can be annotated with the desired number of repeats, e.g.,

```
rule benchmark_command:
    input:
        "path/to/input.{sample}.txt"
    output:
        "path/to/output.{sample}.txt"
    benchmark:
        repeat("benchmarks/somecommand/{sample}.tsv", 3)
    shell:
        "somecommand {input} {output}"
```

will instruct Snakemake to run each job of this rule three times and store all measurements in the benchmark file. The resulting tsv file can be used as input for other rules, just like any other output file.

Note

Note that benchmarking is only possible in a reliable fashion for subprocesses (thus for tasks run through the `shell`, `script`, and `wrapper` directive). In the `run` block, the variable `bench_record` is available that you can pass to `shell()` as `bench_record=bench_record`. When using `shell(..., bench_record=bench_record)`, the maximum of all measurements of all `shell()` calls will be used but the running time of the rule execution including any Python code.

4.14.28 Defining scatter-gather processes

Via Snakemake's powerful and abitrary Python based aggregation abilities (via the `expand` function and arbitrary Python code, see *here*), scatter-gather workflows are well supported. Nevertheless, it can sometimes be handy to use Snakemake's specific scatter-gather support, which allows to avoid boilerplate and offers additional configuration options. Scatter-gather processes can be defined via a global `scattergather` directive:

```
scattergather:
    split=8
```

Each process thereby defines a name (here e.g. `split`) and a default number of scatter items. Then, scattering and gathering can be implemented by using globally available `scatter` and `gather` objects:

```
rule all:
    input:
        "gathered/all.txt"


rule split:
    output:
        scatter.split("splitted/{scatteritem}.txt")
    shell:
        "touch {output}"


rule intermediate:
    input:
        "splitted/{scatteritem}.txt"
    output:
        "splitted/{scatteritem}.post.txt"
    shell:
        "cp {input} {output}"


rule gather:
    input:
        gather.split("splitted/{scatteritem}.post.txt")
    output:
        "gathered/all.txt"
    shell:
        "cat {input} > {output}"
```

Thereby, `scatter.split("splitted/{scatteritem}.txt")` yields a list of paths `"splitted/1-of-n.txt"`, `"splitted/2-of-n.txt"`, ..., depending on the number `n` of scatter items defined. Analogously, `gather.split("splitted/{scatteritem}.post.txt")`, yields a list of paths `"splitted/0.post.txt"`, `"splitted/1.post.txt"`, ..., which request the application of the rule `intermediate` to each scatter item.

The default number of scatter items can be overwritten via the command line interface. For example

```
snakemake --set-scatter split=2
```

would set the number of scatter items for the split process defined above to 2 instead of 8. This allows to adapt parallelization according to the needs of the underlying computing platform and the analysis at hand.

For more complex workflows it's possible to define multiple processes, for example:

```
scattergather:
    split_a=8,
    split_b=3,
```

The calls to `scatter` and `gather` would need to reference the appropriate process name, e.g. `scatter.split_a` and `gather.split_a` to use the `split_a` settings.

4.14.29 Defining groups for execution

From Snakemake 5.0 on, it is possible to assign rules to groups. Such groups will be executed together in **cluster** or **cloud mode**, as a so-called **group job**, i.e., all jobs of a particular group will be submitted at once, to the same computing node. When executing locally, group definitions are ignored.

Groups can be defined via the `group` keyword. This way, queueing and execution time can be saved, in particular if one or several short-running rules are involved.

```
samples = [1,2,3,4,5]


rule all:
    input:
        "test.out"


rule a:
    output:
        "a/{sample}.out"
    group: "mygroup"
    shell:
        "touch {output}"


rule b:
    input:
        "a/{sample}.out"
    output:
        "b/{sample}.out"
    group: "mygroup"
    shell:
        "touch {output}"


rule c:
    input:
        expand("b/{sample}.out", sample=samples)
    output:
        "test.out"
    shell:
        "touch {output}"
```

Here, jobs from rule `a` and `b` end up in one group `mygroup`, whereas jobs from rule `c` are executed separately. Note that Snakemake always determines a **connected subgraph** with the same group id to be a **group job**. Here, this means that, e.g., the jobs creating `a/1.out` and `b/1.out` will be in one group, and the jobs creating `a/2.out` and `b/2.out` will be in a separate group. However, if we would add `group: "mygroup"` to rule `c`, all jobs would end up in a single group, including the one spawned from rule `c`, because `c` connects all the other jobs.

Alternatively, groups can be defined via the command line interface. This enables to almost arbitrarily partition the DAG, e.g. in order to safe network traffic, see *here*.

For execution on the cloud using Google Life Science API and preemptible instances, we expect all rules in the group to be homogenously set as preemptible instances (e.g., with command-line option `--preemptible-rules`), such that a preemptible VM is requested for the execution of the group job.

Group-local jobs

From Snakemake 7.0 on, it is further possible to ensure that jobs from a certain rule are executed separately within each *job group*. For this purpose we use *input functions*, which, in addition to the `wildcards` argument can expect a `groupid` argument. In such a case, Snakemake passes the ID of the corresponding group job to the input function. Consider the following example

```
rule all:
    input:
        expand("bar{i}.txt", i=range(3))


rule grouplocal:
    output:
        "foo.{groupid}.txt"
    group:
        "foo"
    shell:
        "echo test > {output}"


def get_input(wildcards, groupid):
    return f"foo.{groupid}.txt"


rule consumer:
    input:
        get_input
    output:
        "bar{i}.txt"
    group:
        "foo"
    shell:
        "cp {input} {output}"
```

Here, the value of `groupid` that is passed by Snakemake to the input function is a UUID that uniquely identifies the group job in which each instance of the rule `consumer` is contained. In the input function `get_input` we use this ID to request the desired input file from the rule `grouplocal`. Since the value of the corresponding wildcard `groupid` is now always a group specific unique ID, it is ensured that the rule `grouplocal` will run for every group job spawned from the group `foo` (remember that group jobs by default only span one connected component, and that this can be configured via the command line, see *Job Grouping*). Of course, above example would also work if the groups are not specified via the rule definition but entirely via the *command line*.

4.14.30 Piped output

From Snakemake 5.0 on, it is possible to mark output files as pipes, via the `pipe` flag, e.g.:

```
rule all:
    input:
        expand("test.{i}.out", i=range(2))


rule a:
    output:
        pipe("test.{i}.txt")
    shell:
```

(continues on next page)

(continued from previous page)

```
        "for i in {{0..2}}; do echo {wildcards.i} >> {output}; done"


rule b:
    input:
        "test.{i}.txt"
    output:
        "test.{i}.out"
    shell:
        "grep {wildcards.i} < {input} > {output}"
```

If an output file is marked to be a pipe, then Snakemake will first create a named pipe with the given name and then execute the creating job simultaneously with the consuming job, inside a **group job** (see above). This works in all execution modes, local, cluster, and cloud. Naturally, a pipe output may only have a single consumer. It is possible to combine explicit group definition as above with pipe outputs. Thereby, pipe jobs can live within, or (automatically) extend existing groups. However, the two jobs connected by a pipe may not exist in conflicting groups.

As with other groups, Snakemake will automatically calculate the required resources for the group job (see *resources*.

4.14.31 Service rules/jobs

From Snakemake 7.0 on, it is possible to define so-called service rules. Jobs spawned from such rules provide at least one special output file that is marked as `service`, which means that it is considered to provide a resource that shall be kept available until all consuming jobs are finished. This can for example be the socket of a database, a shared memory device, a ramdisk, and so on. It can even just be a dummy file, and access to the service might happen via a different channel (e.g. a local http port). Service jobs are expected to not exit after creating that resource, but instead wait until Snakemake terminates them (e.g. via SIGTERM on Unixoid systems).

Consider the following example:

```
rule the_service:
    output:
        service("foo.socket")
    shell:
        # here we simulate some kind of server process that provides data via a socket
        "ln -s /dev/random {output}; sleep 10000"


rule consumer1:
    input:
        "foo.socket"
    output:
        "test.txt"
    shell:
        "head -n1 {input} > {output}"


rule consumer2:
    input:
        "foo.socket"
    output:
        "test2.txt"
    shell:
        "head -n1 {input} > {output}"
```

Snakemake will schedule the service with all consumers to the same physical node (in the future we might provide further controls and other modes of operation). Once all consumer jobs are finished, the service job will be terminated automatically by Snakemake, and the service output will be removed.

Group-local service jobs

Since Snakemake supports arbitrary partitioning of the DAG into so-called job groups, one should consider what this implies for service jobs when running a workflow in a cluster of cloud context: since each group job spans at least one connected component (see job groups and *the Snakemake paper <https://doi.org/10.12688/f1000research.29032.2>*), this means that the service job will automatically connect all consumers into one big group. This can be undesired, because depending on the number of consumers that group job can become too big for efficient execution on the underlying architecture. In case of local execution, this is not a problem because here DAG partitioning has no effect.

However, to make a workflow portable across different backends, this behavior should always be considered. In order to circumvent it, it is possible to model service jobs as group-local, i.e. ensuring that each group job gets its own instance of the service rule. This works by combining the service job pattern from above with the *group-local pattern* as follows:

```
rule the_service:
    output:
        service("foo.{groupid}.socket")
    shell:
        # here we simulate some kind of server process that provides data via a socket
        "ln -s /dev/random {output}; sleep 10000"


def get_socket(wildcards, groupid):
    return f"foo.{groupid}.socket"


rule consumer1:
    input:
        get_socket
    output:
        "test.txt"
    shell:
        "head -n1 {input} > {output}"


rule consumer2:
    input:
        get_socket
    output:
        "test2.txt"
    shell:
        "head -n1 {input} > {output}"
```

4.14.32 Parameter space exploration

The basic Snakemake functionality already provides everything to handle parameter spaces in any way (sub-spacing for certain rules and even depending on wildcard values, the ability to read or generate spaces on the fly or from files via pandas, etc.). However, it usually would require some boilerplate code for translating a parameter space into wildcard patterns, and translate it back into concrete parameters for scripts and commands. From Snakemake 5.31 on (inspired by JUDI), this is solved via the Paramspace helper, which can be used as follows:

```
from snakemake.utils import Paramspace
import pandas as pd

# declare a dataframe to be a paramspace
paramspace = Paramspace(pd.read_csv("params.tsv", sep="\t"))


rule all:
    input:
        # Aggregate over entire parameter space (or a subset thereof if needed)
        # of course, something like this can happen anywhere in the workflow (not
        # only at the end).
        expand("results/plots/{params}.pdf", params=paramspace.instance_patterns)


rule simulate:
    output:
        # format a wildcard pattern like "alpha~{alpha}/beta~{beta}/gamma~{gamma}"
        # into a file path, with alpha, beta, gamma being the columns of the data␣
↪frame
        f"results/simulations/{paramspace.wildcard_pattern}.tsv"
    params:
        # automatically translate the wildcard values into an instance of the param␣
↪space
        # in the form of a dict (here: {"alpha": ..., "beta": ..., "gamma": ...})
        simulation=paramspace.instance
    script:
        "scripts/simulate.py"


rule plot:
    input:
        f"results/simulations/{paramspace.wildcard_pattern}.tsv"
    output:
        f"results/plots/{paramspace.wildcard_pattern}.pdf"
    shell:
        "touch {output}"
```

In above example, **please note** the Python `f`-string formatting (the `f` before the initial quotes) applied to the input and output file strings that contain `paramspace.wildcard_pattern`. This means that the file that is registered as input or output file by Snakemake does not contain a wildcard `{paramspace.wildcard_pattern}`, but instead this item is replaced by a pattern of multiple wildcards derived from the columns of the paramter space dataframe. This is done by the Python `f`-string formatting before the string is registered in the rule. Given that *params.tsv* contains:

```
alpha       beta    gamma
1.0 0.1     0.99
2.0 0.0     3.9
```

This workflow will run as follows:

```
[Fri Nov 27 20:57:27 2020]
rule simulate:
    output: results/simulations/alpha~2.0/beta~0.0/gamma~3.9.tsv
    jobid: 4
    wildcards: alpha=2.0, beta=0.0, gamma=3.9

[Fri Nov 27 20:57:27 2020]
rule simulate:
    output: results/simulations/alpha~1.0/beta~0.1/gamma~0.99.tsv
    jobid: 2
    wildcards: alpha=1.0, beta=0.1, gamma=0.99

[Fri Nov 27 20:57:27 2020]
rule plot:
    input: results/simulations/alpha~2.0/beta~0.0/gamma~3.9.tsv
    output: results/plots/alpha~2.0/beta~0.0/gamma~3.9.pdf
    jobid: 3
    wildcards: alpha=2.0, beta=0.0, gamma=3.9


[Fri Nov 27 20:57:27 2020]
rule plot:
    input: results/simulations/alpha~1.0/beta~0.1/gamma~0.99.tsv
    output: results/plots/alpha~1.0/beta~0.1/gamma~0.99.pdf
    jobid: 1
    wildcards: alpha=1.0, beta=0.1, gamma=0.99


[Fri Nov 27 20:57:27 2020]
localrule all:
    input: results/plots/alpha~1.0/beta~0.1/gamma~0.99.pdf, results/plots/alpha~2.0/
↪beta~0.0/gamma~3.9.pdf
    jobid: 0
```

Naturally, it is possible to create sub-spaces from `Paramspace` objects, simply by applying all the usual methods and attributes that Pandas data frames provide (e.g. `.loc[...]`, `.filter()` etc.). Further, the form of the created `wildcard_pattern` can be controlled via additional arguments of the `Paramspace` constructor. In particular, using the argument `single_wildcard` the default behavior of encoding each column as a wildcard can be replaced with a single given wildcard name. This can be handy in case a rule shall serve multiple param spaces with different sets of columns.

4.14.33 Data-dependent conditional execution

From Snakemake 5.4 on, conditional reevaluation of the DAG of jobs based on the content outputs is possible. The key idea is that rules can be declared as checkpoints, e.g.,

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

Snakemake allows to re-evaluate the DAG after the successful execution of every job spawned from a checkpoint. For this, every checkpoint is registered by its name in a globally available `checkpoints` object. The `checkpoints`

object can be accessed by *input functions*. Assuming that the checkpoint is named `somestep` as above, the output files for a particular job can be retrieved with

```
checkpoints.somestep.get(sample="a").output
```

Note

Note that output files of checkpoints that are accessed via this mechanism should not be marked as temporary. Otherwise, they would require to trigger reruns of the checkpoint whenever the DAG shall be reevaluated (because they are already missing at that point).

Thereby, the `get` method throws `snakemake.exceptions.IncompleteCheckpointException` if the checkpoint has not yet been executed for these particular wildcard value(s). Inside an input function, the exception will be automatically handled by Snakemake, and leads to a re-evaluation after the checkpoint has been successfully passed.

To illustrate the possibilities of this mechanism, consider the following complete example:

```
# a target rule to define the desired final output
rule all:
    input:
        "aggregated/a.txt",
        "aggregated/b.txt"


# the checkpoint that shall trigger re-evaluation of the DAG
checkpoint somestep:
    input:
        "samples/{sample}.txt"
    output:
        "somestep/{sample}.txt"
    shell:
        # simulate some output value
        "echo {wildcards.sample} > somestep/{wildcards.sample}.txt"


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


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

(continues on next page)

(continued from previous page)

```
# input function for the rule aggregate
def aggregate_input(wildcards):
    # decision based on content of output file
    # Important: use the method open() of the returned file!
    # This way, Snakemake is able to automatically download the file if it is␣
↪generated in
    # a cloud environment without a shared filesystem.
    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}"
```

As can be seen, the rule aggregate uses an input function. Inside the function, we first retrieve the output files of the checkpoint `somestep` with the wildcards, passing through the value of the wildcard sample. Upon execution, if the checkpoint is not yet complete, Snakemake will record `somestep` as a direct dependency of the rule `aggregate`. Once `somestep` has finished for a given sample, the input function will automatically be re-evaluated and the method `get` will no longer return an exception. Instead, the output file will be opened, and depending on its contents either `"post/{sample}.txt"` or `"alt/{sample}.txt"` will be returned by the input function. This way, the DAG becomes conditional on some produced data.

It is also possible to use checkpoints for cases where the output files are unknown before execution. Consider the following example where an arbitrary number of files is generated by a rule before being aggregated:

```
# a target rule to define the desired final output
  rule all:
      input:
          "aggregated.txt"


  # the checkpoint that shall trigger re-evaluation of the DAG
  # an number of file is created in a defined directory
  checkpoint somestep:
      output:
          directory("my_directory/")
      shell:'''
      mkdir my_directory/
      cd my_directory
      for i in 1 2 3; do touch $i.txt; done
      '''



  # input function for rule aggregate, return paths to all files produced by the␣
↪checkpoint 'somestep'
  def aggregate_input(wildcards):
      checkpoint_output = checkpoints.somestep.get(**wildcards).output[0]
```

(continues on next page)

(continued from previous page)

```
    return expand("my_directory/{i}.txt",
                  i=glob_wildcards(os.path.join(checkpoint_output, "{i}.txt")).i)


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

Because the number of output files is unknown beforehand, the checkpoint only defines an output *directory*. This time, instead of explicitly writing

```
checkpoints.somestep.get(sample=wildcards.sample).output[0]
```

we use the shorthand

```
checkpoints.somestep.get(**wildcards).output[0]
```

which automatically unpacks the wildcards as keyword arguments (this is standard python argument unpacking). If the checkpoint has not yet been executed, accessing `checkpoints.somestep.get(**wildcards)` ensures that Snakemake records the checkpoint as a direct dependency of the rule `aggregate`. Upon completion of the checkpoint, the input function is re-evaluated, and the code beyond its first line is executed. Here, we retrieve the values of the wildcard `i` based on all files named `{i}.txt` in the output directory of the checkpoint. Because the wildcard `i` is evaluated only after completion of the checkpoint, it is nescessay to use `directory` to declare its output, instead of using the full wildcard patterns as output.

A more practical example building on the previous one is a clustering process with an unknown number of clusters for different samples, where each cluster shall be saved into a separate file. In this example the clusters are being processed by an intermediate rule before being aggregated:

```
# a target rule to define the desired final output
rule all:
    input:
        "aggregated/a.txt",
        "aggregated/b.txt"


# the checkpoint that shall trigger re-evaluation of the DAG
checkpoint clustering:
    input:
        "samples/{sample}.txt"
    output:
        clusters=directory("clustering/{sample}")
    shell:
        "mkdir clustering/{wildcards.sample}; "
        "for i in 1 2 3; do echo $i > clustering/{wildcards.sample}/$i.txt; done"


# an intermediate rule
rule intermediate:
    input:
        "clustering/{sample}/{i}.txt"
    output:
```

(continues on next page)

(continued from previous page)

```
        "post/{sample}/{i}.txt"
    shell:
        "cp {input} {output}"


def aggregate_input(wildcards):
    checkpoint_output = checkpoints.clustering.get(**wildcards).output[0]
    return expand("post/{sample}/{i}.txt",
           sample=wildcards.sample,
           i=glob_wildcards(os.path.join(checkpoint_output, "{i}.txt")).i)


# an aggregation over all produced clusters
rule aggregate:
    input:
        aggregate_input
    output:
        "aggregated/{sample}.txt"
    shell:
        "cat {input} > {output}"
```

Here a new directory will be created for each sample by the checkpoint. After completion of the checkpoint, the `aggregate_input` function is re-evaluated as previously. The values of the wildcard `i` is this time used to expand the pattern `"post/{sample}/{i}.txt"`, such that the rule `intermediate` is executed for each of the determined clusters.

4.14.34 Rule inheritance

With Snakemake 6.0 and later, it is possible to inherit from previously defined rules, or in other words, reuse an existing rule in a modified way. This works via the `use rule` statement that also allows to declare the usage of rules from external modules (see *Modules*). Consider the following example:

```
rule a:
    output:
        "test.out"
    shell:
        "echo test > {output}"


use rule a as b with:
    output:
        "test2.out"
```

As can be seen, we first declare a rule a, and then we reuse the rule a as rule b, while changing only the output file and keeping everything else the same. In reality, one will often change more. Analogously to the `use rule` from external modules, any properties of the rule (`input`, `output`, `log`, `params`, `benchmark`, `threads`, `resources`, etc.) can be modified, except the actual execution step (`shell`, `notebook`, `script`, `cwl`, or `run`). All unmodified properties are inherited from the parent rule.

4.14.35 Accessing auxiliary source files

Snakemake workflows can refer to various other source files via paths relative to the current Snakefile. This happens for example with the *script directive* or the *conda directive*. Sometimes, it is necessary to access further source files that are in a directory relative to the current Snakefile. Since workflows can be imported from remote locations (e.g. when using *modules*), it is important to not do this manually, so that Snakemake has the chance to cache these files locally before they are accessed. This can be achieved by accessing their path via the `workflow.source_path`, which (a) computes the correct path relative to the current Snakefile such that the file can be accessed from any working directory, and (b) downloads remote files to a local cache:

```
rule a:
    output:
        "test.out"
    params:
        json=workflow.source_path("../resources/test.json")
    shell:
        "somecommand {params.json} > {output}"
```

4.14.36 Template rendering integration

Sometimes, data analyses entail the dynamic rendering of internal configuration files that are required for certain steps. From Snakemake 7 on, such template rendering is directly integrated such that it can happen with minimal code and maximum performance. Consider the following example:

```
rule render_jinja2_template:
    input:
        "some-jinja2-template.txt"
    output:
        "results/{sample}.rendered-version.txt"
    params:
        foo=0.1
    template_engine:
        "jinja2"
```

Here, Snakemake will automatically use the specified template engine Jinja2 to render the template given as input file into the given output file. The template_engine instruction has to be specified at the end of the rule. Template rendering rules may only have a single output file. If the rule needs more than one input file, there has to be one input file called `template`, pointing to the main template to be used for the rendering:

```
rule render_jinja2_template:
    input:
        template="some-jinja2-template.txt",
        other_file="some-other-input-file-used-by-the-template.txt"
    output:
        "results/{sample}.rendered-version.txt"
    params:
        foo=0.1
    template_engine:
        "jinja2"
```

The template itself has access to `input`, `params`, `wildcards`, and `config`, which are the same objects you can use for example in the `shell` or `run` directive, and the same objects as can be accessed from `script` or `notebook` directives (but in the latter two cases they are stored behind the `snakemake` object which serves as a dedicated namespace to avoid name clashes).

An example Jinja2 template could look like this:

```
This is some text and now we access {{ params.foo }}.
```

Apart from Jinja2, Snakemake supports YTE (YAML template engine), which is particularly designed to support templating of the ubiquitious YAML file format:

```
rule render_jinja2_template:
    input:
        "some-yte-template.yaml"
    output:
        "results/{sample}.rendered-version.yaml"
    params:
        foo=0.1
    template_engine:
        "yte"
```

Analogously to the jinja2 case YTE has access to `params`, `wildcards`, and `config`:

```
?if params.foo < 0.5:
  x:
    - 1
    - 2
    - 3
?else:
  y:
    - a
    - b
    - ?config["threshold"]
```

Template rendering rules are always executed locally, without submission to cluster or cloud processes (since templating is usually not resource intensive).

4.14.37 MPI support

Highly parallel programs may use the MPI (:ref: message passing interface<https://en.wikipedia.org/wiki/Message_Passing_Interface>) to enable a programm to span work across an invidual compute node's boundary. The command to run the MPI program (in below example we assume there exists a program `calc-pi-mpi`) has to be specified in the `mpi`-resource, e.g.:

```
rule calc_pi:
  output:
      "pi.calc",
  log:
      "logs/calc_pi.log",
  resources:
      tasks=10,
      mpi="mpiexec",
  shell:
      "{resources.mpi} -n {resources.tasks} calc-pi-mpi 10 > {output} 2> {log}"
```

Thereby, additional parameters may be passed to the MPI-starter, e.g.:

```
rule calc_pi:
  output:
      "pi.calc",
  log:
```

(continues on next page)

(continued from previous page)

```
        "logs/calc_pi.log",
    resources:
        tasks=10,
        mpi="mpiexec -arch x86",
    shell:
        "{resources.mpi} -n {resources.tasks} calc-pi-mpi 10 > {output} 2> {log}"
```

As any other resource, the *mpi*-resource can be overwritten via the command line e.g. in order to adapt to a specific platform (see *Resources*):

```
$ snakemake --set-resources calc_pi:mpi="srun --hint nomultithread" ...
```

Note that in case of distributed, remote execution (cluster, cloud), MPI support might not be available. So far, explicit MPI support is implemented in the *SLURM backend*.

4.15 Configuration

Snakemake allows you to use configuration files for making your workflows more flexible and also for abstracting away direct dependencies to a fixed HPC cluster scheduler.

4.15.1 Standard Configuration

Snakemake directly supports the configuration of your workflow. A configuration is provided as a JSON or YAML file and can be loaded with:

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

The config file can be used to define a dictionary of configuration parameters and their values. In the workflow, the configuration is accessible via the global variable *config*, e.g.

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

If the *configfile* statement is not used, the config variable provides an empty array. In addition to the *configfile* statement, config values can be overwritten via the command line or the api_reference_snakemake, e.g.:

```
$ snakemake --config yourparam=1.5
```

Further, you can manually alter the config dictionary using any Python code **outside** of your rules. Changes made from within a rule won't be seen from other rules. Finally, you can use the `--configfile` command line argument to overwrite values from the *configfile* statement. Note that any values parsed into the `config` dictionary with any of above mechanisms are merged, i.e., all keys defined via a `configfile` statement, or the `--configfile` and `--config` command line arguments will end up in the final *config* dictionary, but if two methods define the same key, command line overwrites the `configfile` statement.

For adding config placeholders into a shell command, Python string formatting syntax requires you to leave out the quotes around the key name, like so:

```
shell:
    "mycommand {config[foo]} ..."
```

4.15.2 Tabular configuration

It is usually advisable to complement YAML based configuration (see above) by a sheet based approach for meta-data that is of tabular form. For example, such a sheet can contain per-sample information. With the Pandas library such data can be read and used with minimal overhead, e.g.,

```
import pandas as pd

samples = pd.read_table("samples.tsv").set_index("samples", drop=False)
```

reads in a table `samples.tsv` in TSV format and makes every record accessible by the sample name. For details, see the Pandas documentation. A fully working real-world example containing both types of configuration can be found here.

4.15.3 Environment variables

Sometimes, it is not desirable to put configuration information into text files. For example, this holds for secrets like access tokens or passwords. Here, environment variables are the method of choice. Snakemake allows to assert the existence of environment variables by adding a statement like:

```
envvars:
    "SOME_VARIABLE",
    "SOME_OTHER_VARIABLE"
```

When executing, Snakemake will fail with a reasonable error message if the variables `SOME_VARIABLE` and `SOME_OTHER_VARIABLE` are undefined. Otherwise, it will take care of passing them to cluster and cloud environments. However, note that this does **not** mean that Snakemake makes them available e.g. in the jobs shell command. Instead, for data provenance and reproducibility reasons, you are required to pass them explicitly to your job via the params directive, e.g. like this:

```
envvars:
    "SOME_VARIABLE"

rule do_something:
    output:
        "test.txt"
    params:
        x=os.environ["SOME_VARIABLE"]
    shell:
        "echo {params.x} > {output}"
```

4.15.4 Validation

With Snakemake 5.1, it is possible to validate both types of configuration via JSON schemas. The function `snakemake.utils.validate` takes a loaded configuration (a config dictionary or a Pandas data frame) and validates it with a given JSON schema. Thereby, the schema can be provided in JSON or YAML format. Also, by using the defaults property it is possible to populate entries with default values. See jsonschema FAQ on setting default values for details. In case of the data frame, the schema should model the record that is expected in each row of the data frame. In the following example,

```
import pandas as pd
from snakemake.utils import validate

configfile: "config.yaml"
validate(config, "config.schema.yaml")
```

(continues on next page)

(continued from previous page)

```
samples = pd.read_table(config["samples"]).set_index("sample", drop=False)
validate(samples, "samples.schema.yaml")


rule all:
    input:
        expand("test.{sample}.txt", sample=samples.index)


rule a:
    output:
        "test.{sample}.txt"
    shell:
        "touch {output}"
```

the schema for validating the samples data frame looks like this:

```
$schema: "https://json-schema.org/draft-06/schema#"
description: an entry in the sample sheet
properties:
  sample:
    type: string
    description: sample name/identifier
  condition:
    type: string
    description: sample condition that will be compared during differential␣
↪expression analysis (e.g. a treatment, a tissue time, a disease)
  case:
    type: boolean
    default: true
    description: boolean that indicates if sample is case or control

required:
  - sample
  - condition
```

Here, in case the case column is missing, the validate function will populate it with True for all entries.

4.15.5 Configuring scientific experiments via PEPs

Often scientific experiments consist of a set of samples (with optional subsamples), for which raw data and metainformation is known. Instead of writing custom sample sheets as shown above, Snakemake allows to use portable encapsulated project (PEP) definitions to configure a workflow. This is done via a special directive *pepfile*, that can optionally complemented by a schema for validation (which is recommended for production workflows):

```
pepfile: "pep/config.yaml"
pepschema: "schemas/pep.yaml"

rule all:
    input:
        expand("{sample}.txt", sample=pep.sample_table["sample_name"])

rule a:
    output:
```

(continues on next page)

(continued from previous page)

```
        "{sample}.txt"
    shell:
        "touch {output}"
```

Using the `pepfile` directive leads to parsing of the provided PEP with peppy. The resulting project object is made globally available under the name `pep`. Here, we use it to aggregate over the set of sample names that is defined in the corresponding PEP.

Importantly, note that PEPs are meant to contain sample metadata and any global information about a project or experiment. They should **not** be used to encode workflow specific configuration options. For those, one should always complement the pepfile with an ordinary *config file*. The rationale is that PEPs should be portable between different data analysis workflows (that could be applied to the same data) and even between workflow management systems. In other words, a PEP should describe everything needed about the data, while a workflow and its configuration should describe everything needed about the analysis that is applied to it.

Validating PEPs

Using the `pepschema` directive leads to an automatic parsing of the provided schema *and* PEP validation with the PEP validation tool – eido. Eido schemas extend JSON Schema vocabulary to accommodate the powerful PEP features. Follow the How to write a PEP schema guide to learn more.

4.15.6 Cluster Configuration (deprecated)

While still being possible, **cluster configuration has been deprecated** by the introduction of *Profiles*.

Snakemake supports a separate configuration file for execution on a cluster. A cluster config file allows you to specify cluster submission parameters outside the Snakefile. The cluster config is a JSON- or YAML-formatted file that contains objects that match names of rules in the Snakefile. The parameters in the cluster config are then accessed by the `cluster.*` wildcard when you are submitting jobs. Note that a workflow shall never depend on a cluster configuration, because this would limit its portability. Therefore, it is also not intended to access the cluster configuration from **within** the workflow.

For example, say that you have the following Snakefile:

```
rule all:
    input: "input1.txt", "input2.txt"

rule compute1:
    output: "input1.txt"
    shell: "touch input1.txt"

rule compute2:
    output: "input2.txt"
    shell: "touch input2.txt"
```

This Snakefile can then be configured by a corresponding cluster config, say "cluster.json":

```
{
    "__default__" :
    {
        "account" : "my account",
        "time" : "00:15:00",
        "n" : 1,
        "partition" : "core"
```

(continues on next page)

(continued from previous page)

```
    },
    "compute1" :
    {
        "time" : "00:20:00"
    }
}
```

Any string in the cluster configuration can be formatted in the same way as shell commands, e.g. `{rule}.{wildcards.sample}` is formatted to `a.xy` if the rulename is `a` and the wildcard value is `xy`. Here `__default__` is a special object that specifies default parameters, these will be inherited by the other configuration objects. The `compute1` object here changes the `time` parameter, but keeps the other parameters from `__default__`. The rule `compute2` does not have any configuration, and will therefore use the default configuration. You can then run the Snakefile with the following command on a SLURM system.

```
$ snakemake -j 999 --cluster-config cluster.json --cluster "sbatch -A {cluster.
↪account} -p {cluster.partition} -n {cluster.n}  -t {cluster.time}"
```

For cluster systems using LSF/BSUB, a cluster config may look like this:

```
{
    "__default__" :
    {
        "queue"     : "medium_priority",
        "nCPUs"     : "16",
        "memory"    : 20000,
        "resources" : "\"select[mem>20000] rusage[mem=20000] span[hosts=1]\"",
        "name"      : "JOBNAME.{rule}.{wildcards}",
        "output"    : "logs/cluster/{rule}.{wildcards}.out",
        "error"     : "logs/cluster/{rule}.{wildcards}.err"
    },


    "trimming_PE" :
    {
        "memory"    : 30000,
        "resources" : "\"select[mem>30000] rusage[mem=30000] span[hosts=1]\"",
    }
}
```

The advantage of this setup is that it is already pretty general by exploiting the wildcard possibilities that Snakemake provides via `{rule}` and `{wildcards}`. So job names, output and error files all have reasonable and trackable default names, only the directies (`logs/cluster`) and job names (`JOBNAME`) have to adjusted accordingly. If a rule named `bamCoverage` is executed with the wildcard `basename = sample1`, for example, the output and error files will be `bamCoverage.basename=sample1.out` and `bamCoverage.basename=sample1.err`, respectively.

4.15.7 Configure Working Directory

All paths in the snakefile are interpreted relative to the directory snakemake is executed in. This behaviour can be overridden by specifying a workdir in the snakefile:

```
workdir: "path/to/workdir"
```

Usually, it is preferred to only set the working directory via the command line, because above directive limits the portability of Snakemake workflows.

4.16 Modularization

Modularization in Snakemake comes at four different levels.

1. The most fine-grained level are wrappers. They are available and can be published at the Snakemake Wrapper Repository. These wrappers can then be composed and customized according to your needs, by copying skeleton rules into your workflow. In combination with conda integration, wrappers also automatically deploy the needed software dependencies into isolated environments.
2. For larger, reusable parts that shall be integrated into a common workflow, it is recommended to write small Snakefiles and include them into a main Snakefile via the include statement. In such a setup, all rules share a common config file.
3. The third level is provided via the *module statement*, which enables arbitrary combination and reuse of rules.
4. Finally, Snakemake provides a syntax for defining *subworkflows*, which is however deprecated in favor of the module statement.

4.16.1 Wrappers

The wrapper directive allows to have re-usable wrapper scripts around e.g. command line tools. In contrast to modularization strategies like `include` or subworkflows, the wrapper directive allows to re-wire the DAG of jobs. For example

```
rule samtools_sort:
    input:
        "mapped/{sample}.bam"
    output:
        "mapped/{sample}.sorted.bam"
    params:
        "-m 4G"
    threads: 8
    wrapper:
        "0.0.8/bio/samtools/sort"
```

Note: It is possible to refer to wildcards and params in the wrapper identifier, e.g. by specifying `"0.0.8/bio/{params.wrapper}"` or `"0.0.8/bio/{wildcards.wrapper}"`.

Refers to the wrapper `"0.0.8/bio/samtools/sort"` to create the output from the input. Snakemake will automatically download the wrapper from the Snakemake Wrapper Repository. Thereby, `0.0.8` can be replaced with the git version tag you want to use, or a commit id. This ensures reproducibility since changes in the wrapper implementation will only be propagated to your workflow once you update the version tag. Examples for each wrapper can be found in the READMEs located in the wrapper subdirectories at the Snakemake Wrapper Repository.

Alternatively, for example during development, the wrapper directive can also point to full URLs, including URLs to local files with absolute paths `file://` or relative paths `file:`. Such a URL will have to point to the folder containing the `wrapper.*` and `environment.yaml` files. In the above example, the full GitHub URL could for example be provided with `wrapper: https://github.com/snakemake/snakemake-wrappers/raw/0.0.8/bio/samtools/sort`. Note that it needs to point to the `/raw/` version of the folder, not the rendered HTML version.

In addition, the Snakemake Wrapper Repository offers so-called meta-wrappers, which can be used as modules, see *Meta-Wrappers*.

The Snakemake Wrapper Repository is meant as a collaborative project and pull requests are very welcome.

4.16.2 Common-Workflow-Language (CWL) support

With Snakemake 4.8.0, it is possible to refer to CWL tool definitions in rules instead of specifying a wrapper or a plain shell command. A CWL tool definition can be used as follows.

```
rule samtools_sort:
    input:
        input="mapped/{sample}.bam"
    output:
        output_name="mapped/{sample}.sorted.bam"
    params:
        threads=lambda wildcards, threads: threads,
        memory="4G"
    threads: 8
    cwl:
        "https://github.com/common-workflow-language/workflows/blob/"
        "fb406c95/tools/samtools-sort.cwl"
```

Note: It is possible to refer to wildcards and params in the tool definition URL, e.g. by specifying something like `"https://.../tools/{params.tool}.cwl"` or `"https://.../tools/{wildcards.tool}.cwl"`.

It is advisable to use a github URL that includes the commit as above instead of a branch name, in order to ensure reproducible results. Snakemake will execute the rule by invoking *cwltool*, which has to be available via your *$PATH* variable, and can be, e.g., installed via *conda* or *pip*. When using in combination with *–use-singularity*, Snakemake will instruct *cwltool* to execute the command via Singularity in user space. Otherwise, *cwltool* will in most cases use a Docker container, which requires Docker to be set up properly.

The advantage is that predefined tools available via any repository of CWL tool definitions can be used in any supporting workflow management system. In contrast to a *Snakemake wrapper*, CWL tool definitions are in general not suited to alter the behavior of a tool, e.g., by normalizing output names or special input handling. As you can see in comparison to the analog *wrapper declaration* above, the rule becomes slightly more verbose, because input, output, and params have to be dispatched to the specific expectations of the CWL tool definition.

4.16.3 Includes

Another Snakefile with all its rules can be included into the current:

```
include: "path/to/other/snakefile"
```

The default target rule (often called the `all`-rule), won't be affected by the include. I.e. it will always be the first rule in your Snakefile, no matter how many includes you have above your first rule. Includes are relative to the directory of the Snakefile in which they occur. For example, if above Snakefile resides in the directory `my/dir`, then Snakemake will search for the include at `my/dir/path/to/other/snakefile`, regardless of the working directory.

4.16.4 Modules

With Snakemake 6.0 and later, it is possible to define external workflows as modules, from which rules can be used by explicitly "importing" them.

```
from snakemake.utils import min_version
min_version("6.0")

module other_workflow:
    snakefile:
        # here, plain paths, URLs and the special markers for code hosting providers␣
↪(see below) are possible.
        "other_workflow/Snakefile"

use rule * from other_workflow exclude ruleC as other_*
```

The `module other_workflow:` statement registers the external workflow as a module, by defining the path to the main snakefile of `other_workflow`. Here, plain paths, HTTP/HTTPS URLs and special markers for code hosting providers like Github or Gitlab are possible (see *Code hosting providers*). The second statement, `use rule * from other_workflow exclude ruleC as other_*`, declares all rules of that module to be used in the current one, except for ruleC. Thereby, the `as other_*` at the end renames all those rules with a common prefix. This can be handy to avoid rule name conflicts (note that rules from modules can otherwise overwrite rules from your current workflow or other modules).

The module is evaluated in a separate namespace, and only the selected rules are added to the current workflow. Non-rule Python statements inside the module are also evaluated in that separate namespace. They are available in the module-defining workflow under the name of the module (e.g. here `other_workflow.myfunction()` would call the function `myfunction` that has been defined in the model, e.g. in `other_workflow/Snakefile`). Also note that this means that any Python variables and functions available in the module-defining namespace will **not** be visible from inside the module. However, it is possible to pass information to the module using the `config` mechanism described in the following.

It is possible to overwrite the global config dictionary for the module, which is usually filled by the `configfile` statement (see *Standard Configuration*):

```
from snakemake.utils import min_version
min_version("6.0")

configfile: "config/config.yaml"

module other_workflow:
    # here, plain paths, URLs and the special markers for code hosting providers (see␣
↪below) are possible.
    snakefile: "other_workflow/Snakefile"
```

(continues on next page)

(continued from previous page)

```
    config: config["other-workflow"]

use rule * from other_workflow as other_*
```

In this case, any `configfile` statements inside the module are ignored. In addition, it is possible to skip any *validation* statements in the module, by specifying `skip_validation: True` in the module statment. Moreover, one can automatically move all relative input and output files of a module into a dedicated folder: by specifying `prefix: "foo"` in the module definition, e.g. any output file `path/to/output.txt` in the module would be stored under `foo/path/to/output.txt` instead. This becomes particularly usefull when combining multiple modules, see *Using and combining pre-exising workflows*.

Instead of using all rules, it is possible to import specific rules. Specific rules may even be modified before using them, via a final `with:` followed by a block that lists items to overwrite. This modification can be performed after a general import, and will overwrite any unmodified import of the same rule.

```
from snakemake.utils import min_version
min_version("6.0")

module other_workflow:
    # here, plain paths, URLs and the special markers for code hosting providers (see␣
↪below) are possible.
    snakefile: "other_workflow/Snakefile"
    config: config["other-workflow"]

use rule * from other_workflow as other_*

use rule some_task from other_workflow as other_some_task with:
    output:
        "results/some-result.txt"
```

By such a modifying use statement, any properties of the rule (`input`, `output`, `log`, `params`, `benchmark`, `threads`, `resources`, etc.) can be overwritten, except the actual execution step (`shell`, `notebook`, `script`, `cwl`, or `run`).

Note that the second use statement has to use the original rule name, not the one that has been prefixed with `other_` via the first use statement (there is no rule `other_some_task` in the module `other_workflow`). In order to overwrite the rule `some_task` that has been imported with the first `use rule` statement, it is crucial to ensure that the rule is used with the same name in the second statement, by adding an equivalent `as` clause (here `other_some_task`). Otherwise, you will have two versions of the same rule, which might be unintended (a common symptom of such unintended repeated uses would be ambiguous rule exceptions thrown by Snakemake).

Of course, it is possible to combine the use of rules from multiple modules (see *Using and combining pre-exising workflows*), and via modifying statements they can be rewired and reconfigured in an arbitrary way.

Meta-Wrappers

Snakemake wrappers offer a simple way to include commonly used tools in Snakemake workflows. In addition the Snakemake Wrapper Repository offers so-called meta-wrappers, which are combinations of wrappers, meant to perform common tasks. Both wrappers and meta-wrappers are continously tested. The module statement also allows to easily use meta-wrappers, for example:

```
from snakemake.utils import min_version
min_version("6.0")

configfile: "config.yaml"
```

(continues on next page)

(continued from previous page)

```
module bwa_mapping:
    meta_wrapper: "0.72.0/meta/bio/bwa_mapping"


use rule * from bwa_mapping


def get_input(wildcards):
    return config["samples"][wildcards.sample]


use rule bwa_mem from bwa_mapping with:
    input:
        get_input
```

First, we define the meta-wrapper as a module. Next, we declare all rules from the module to be used. And finally, we overwrite the input directive of the rule `bwa_mem` such that the raw data is taken from the place where our workflow configures it via it's config file.

4.16.5 Sub-Workflows

Snakemake allows to depend on the output of other workflows as sub-workflows. However, note that sub-workflows are deprecated in favor of *modules*. A sub-workflow is executed independently before the current workflow is executed. Thereby, Snakemake ensures that all files the current workflow depends on are created or updated if necessary. This allows to create links between otherwise separate data analyses.

```
subworkflow otherworkflow:
    workdir:
        "../path/to/otherworkflow"
    snakefile:
        "../path/to/otherworkflow/Snakefile"
    configfile:
        "path/to/custom_configfile.yaml"

rule a:
    input:
        otherworkflow("test.txt")
    output: ...
    shell:  ...
```

Here, the subworkflow is named "otherworkflow" and it is located in the working directory `../path/to/otherworkflow`. The snakefile is in the same directory and called `Snakefile`. If `snakefile` is not defined for the subworkflow, it is assumed be located in the workdir location and called `Snakefile`, hence, above we could have left the `snakefile` keyword out as well. If `workdir` is not specified, it is assumed to be the same as the current one. The (optional) definition of a `configfile` allows to parameterize the subworkflow as needed. Files that are output from the subworkflow that we depend on are marked with the `otherworkflow` function (see the input of rule a). This function automatically determines the absolute path to the file (here `../path/to/otherworkflow/test.txt`).

When executing, snakemake first tries to create (or update, if necessary) `test.txt` (and all other possibly mentioned dependencies) by executing the subworkflow. Then the current workflow is executed. This can also happen recursively, since the subworkflow may have its own subworkflows as well.

Note that subworkflow rules will not be displayed in a *Snakemake report* generated from the surrounding workflow.

4.16.6 Code hosting providers

To obtain the correct URL to an external source code resource (e.g. a snakefile, see *Modules*), Snakemake provides markers for code hosting providers. Currently, Github

```
github("owner/repo", path="workflow/Snakefile", tag="v1.0.0")
```

and Gitlab are supported:

```
gitlab("owner/repo", path="workflow/Snakefile", tag="v1.0.0")
```

For the latter, it is also possible to specify an alternative host, e.g.

```
gitlab("owner/repo", path="workflow/Snakefile", tag="v1.0.0", host="somecustomgitlab.
↪org")
```

While specifying a tag is highly encouraged, it is alternatively possible to specify a *commit* or a *branch* via respective keyword arguments. Note that only when specifying a tag or a commit, Snakemake is able to persistently cache the source, thereby avoiding to repeatedly query it in case of multiple executions.

Private repositories

To access source code resources located in private repositories you can define an access token in the `GITHUB_TOKEN` and/or `GITLAB_TOKEN` environment variables.

4.17 Remote files

In versions `snakemake>=3.5`.

The `Snakefile` supports a wrapper function, `remote()`, indicating a file is on a remote storage provider (this is similar to `temp()` or `protected()`). In order to use all types of remote files, the Python packages `boto`, `moto`, `filechunkio`, `pysftp`, `dropbox`, `requests`, `ftputil`, `XRootD`, and `biopython` must be installed.

During rule execution, a remote file (or object) specified is downloaded to the local `cwd`, within a sub-directory bearing the same name as the remote provider. This sub-directory naming lets you have multiple remote origins with reduced likelihood of name collisions, and allows Snakemake to easily translate remote objects to local file paths. You can think of each local remote sub-directory as a local mirror of the remote system. The `remote()` wrapper is mutually-exclusive with the `temp()` and `protected()` wrappers.

Snakemake includes the following remote providers, supported by the corresponding classes:

- Amazon Simple Storage Service (AWS S3): `snakemake.remote.S3`
- Google Cloud Storage (GS): `snakemake.remote.GS`
- Microsoft Azure Blob Storage: `snakemake.remote.AzBlob`
- File transfer over SSH (SFTP): `snakemake.remote.SFTP`
- Read-only web (HTTP[S]): `snakemake.remote.HTTP`
- File transfer protocol (FTP): `snakemake.remote.FTP`
- Dropbox: `snakemake.remote.dropbox`
- XRootD: `snakemake.remote.XRootD`
- GenBank / NCBI Entrez: `snakemake.remote.NCBI`

- WebDAV: `snakemake.remote.webdav`
- GFAL: `snakemake.remote.gfal`
- GridFTP: `snakemake.remote.gridftp`
- iRODS: `snakemake.remote.iRODS`
- EGA: `snakemake.remote.EGA`
- Zenodo: `snakemake.remote.zenodo`
- AUTO: an automated remote selector

4.17.1 Amazon Simple Storage Service (S3)

This section describes usage of the S3 RemoteProvider, and also provides an intro to remote files and their usage.

It is important to note that you must have credentials (`access_key_id` and `secret_access_key`) which permit read/write access. If a file only serves as input to a Snakemake rule, read access is sufficient. You may specify credentials as environment variables or in the file `~/.aws/credentials`, prefixed with `AWS_*`, as with a standard boto config. Credentials may also be explicitly listed in the `Snakefile`, as shown below:

For the Amazon S3 and Google Cloud Storage providers, the sub-directory used must be the bucket name.

Using remote files is easy (AWS S3 shown):

```
from snakemake.remote.S3 import RemoteProvider as S3RemoteProvider
S3 = S3RemoteProvider(access_key_id="MYACCESSKEY", secret_access_key="MYSECRET")

rule all:
    input:
        S3.remote("bucket-name/file.txt")
```

Expand still works as expected, just wrap the expansion:

```
from snakemake.remote.S3 import RemoteProvider as S3RemoteProvider
S3 = S3RemoteProvider()

rule all:
    input:
        S3.remote(expand("bucket-name/{letter}-2.txt", letter=["A", "B", "C"]))
```

Only remote files needed to satisfy the DAG build are downloaded for the workflow. By default, remote files are downloaded prior to rule execution and are removed locally as soon as no rules depend on them. Remote files can be explicitly kept by setting the `keep_local=True` keyword argument:

```
from snakemake.remote.S3 import RemoteProvider as S3RemoteProvider
S3 = S3RemoteProvider(access_key_id="MYACCESSKEY", secret_access_key="MYSECRET")

rule all:
    input: S3.remote('bucket-name/prefix{split_id}.txt', keep_local=True)
```

If you wish to have a rule to simply download a file to a local copy, you can do so by declaring the same file path locally as is used by the remote file:

```
from snakemake.remote.S3 import RemoteProvider as S3RemoteProvider
S3 = S3RemoteProvider(access_key_id="MYACCESSKEY", secret_access_key="MYSECRET")

```

(continues on next page)

(continued from previous page)

```
rule all:
    input:
        S3.remote("bucket-name/out.txt")
    output:
        "bucket-name/out.txt"
    run:
        shell("cp {output[0]} ./")
```

In some cases the rule can use the data directly on the remote provider, in these cases `stay_on_remote=True` can be set to avoid downloading/uploading data unnecessarily. Additionally, if the backend supports it, any potentially corrupt output files will be removed from the remote. The default for `stay_on_remote` and `keep_local` can be configured by setting these properties on the remote provider object:

```
from snakemake.remote.S3 import RemoteProvider as S3RemoteProvider
S3 = S3RemoteProvider(access_key_id="MYACCESSKEY", secret_access_key="MYSECRET", keep_
↪local=True, stay_on_remote=True)
```

The remote provider also supports a new `glob_wildcards()` (see *How do I run my rule on all files of a certain directory?*) which acts the same as the local version of `glob_wildcards()`, but for remote files:

```
from snakemake.remote.S3 import RemoteProvider as S3RemoteProvider
S3 = S3RemoteProvider(access_key_id="MYACCESSKEY", secret_access_key="MYSECRET")
S3.glob_wildcards("bucket-name/{file_prefix}.txt")

# (result looks just like as if the local glob_wildcards() function were used on a␣
↪locally with a folder called "bucket-name")
```

If the AWS CLI is installed it is possible to configure your keys globally. This removes the necessity of hardcoding the keys in the Snakefile. The interactive AWS credentials setup can be done using the following command:

```
aws configure
```

S3 then can be used without the keys.

```
from snakemake.remote.S3 import RemoteProvider as S3RemoteProvider
S3 = S3RemoteProvider()
```

Finally, it is also possible to overwrite the S3 host via adding a `host` argument (taking a URL string) to `S3RemoteProvider`.

4.17.2 Google Cloud Storage (GS)

Usage of the GS provider is the same as the S3 provider. For authentication, one simply needs to login via the `gcloud` tool before executing Snakemake, i.e.:

```
$ gcloud auth application-default login
```

In the Snakefile, no additional authentication information has to be provided:

```
from snakemake.remote.GS import RemoteProvider as GSRemoteProvider
GS = GSRemoteProvider()

rule all:
    input:
        GS.remote("bucket-name/file.txt")
```

4.17.3 Microsoft Azure Blob Storage

Usage of the Azure Blob Storage provider is similar to the S3 provider. For authentication, an account name and shared access signature (SAS) or key can be used. If these variables are not passed directly to AzureRemoteProvider (see [BlobServiceClient class](https://docs.microsoft.com/en-us/python/api/azure-storage-blob/azure.storage.blob.blobserviceclient?view=azure-python) for naming), they will be read from environment variables, named *AZ_BLOB_ACCOUNT_URL* and *AZ_BLOB_CREDENTIAL*. *AZ_BLOB_ACCOUNT_URL* takes the form *https://<accountname>.blob.core.windows.net* and may also contain a SAS. If a SAS is not part of the URL, *AZ_BLOB_CREDENTIAL* has to be set to the SAS or alternatively to the storage account key.

When using AzBlob as default remote provider you will almost always want to pass these environment variables on to the remote execution environment (e.g. Kubernetes) with *–envvars*, e.g *–envvars AZ_BLOB_ACCOUNT_URL AZ_BLOB_CREDENTIAL*.

```
from snakemake.remote.AzBlob import RemoteProvider as AzureRemoteProvider
AS = AzureRemoteProvider()# assumes env vars AZ_BLOB_ACCOUNT_URL and possibly AZ_BLOB_
↪CREDENTIAL are set

rule a:
    input:
        AS.remote("path/to/file.txt")
```

4.17.4 File transfer over SSH (SFTP)

Snakemake can use files on remove servers accessible via SFTP (i.e. most *nix servers). It uses pysftp for the underlying support of SFTP, so the same connection options exist. Assuming you have SSH keys already set up for the server you are using in the `Snakefile`, usage is simple:

```
from snakemake.remote.SFTP import RemoteProvider
SFTP = RemoteProvider()

rule all:
    input:
        SFTP.remote("example.com/path/to/file.bam")
```

If you need to create the output directories in the remote server, you can specify `mkdir_remote=True` in the `RemoteProvider` constructor.

```
from snakemake.remote.SFTP import RemoteProvider
SFTP = RemoteProvider(mkdir_remote=True)

rule all:
    input:
        "/home/foo/bar.txt"
    output:
        SFTP.remote('example.com/home/foo/create/dir/bar.txt')
    shell:
        "cp {input} {output}"
```

The remote file addresses used must be specified with the host (domain or IP address) and the absolute path to the file on the remote server. A port may be specified if the SSH daemon on the server is listening on a port other than 22, in either the `RemoteProvider` or in each instance of `remote()`:

```
from snakemake.remote.SFTP import RemoteProvider
SFTP = RemoteProvider(port=4040)
```

(continues on next page)

(continued from previous page)

```
rule all:
    input:
        SFTP.remote("example.com/path/to/file.bam")
```

```
from snakemake.remote.SFTP import RemoteProvider
SFTP = RemoteProvider()

rule all:
    input:
        SFTP.remote("example.com:4040/path/to/file.bam")
```

The standard keyword arguments used by pysftp may be provided to the RemoteProvider to specify credentials (either password or private key):

```
from snakemake.remote.SFTP import RemoteProvider
SFTP = RemoteProvider(username="myusername", private_key="/Users/myusername/.ssh/
↪particular_id_rsa")

rule all:
    input:
        SFTP.remote("example.com/path/to/file.bam")
```

```
from snakemake.remote.SFTP import RemoteProvider
SFTP = RemoteProvider(username="myusername", password="mypassword")

rule all:
    input:
        SFTP.remote("example.com/path/to/file.bam")
```

If you share credentials between servers but connect to one on a different port, the alternate port may be specified in the `remote()` wrapper:

```
from snakemake.remote.SFTP import RemoteProvider
SFTP = RemoteProvider(username="myusername", password="mypassword")

rule all:
    input:
        SFTP.remote("some-example-server-1.com/path/to/file.bam"),
        SFTP.remote("some-example-server-2.com:2222/path/to/file.bam")
```

There is a `glob_wildcards()` function:

```
from snakemake.remote.SFTP import RemoteProvider
SFTP = RemoteProvider()
SFTP.glob_wildcards("example.com/path/to/{sample}.bam")
```

4.17.5 Read-only web (HTTP[s])

Snakemake can access web resources via a read-only HTTP(S) provider. This provider can be helpful for including public web data in a workflow.

Web addresses must be specified without protocol, so if your URI looks like this:

```
https://server3.example.com/path/to/myfile.tar.gz
```

The URI used in the `Snakefile` must look like this:

```
server3.example.com/path/to/myfile.tar.gz
```

It is straightforward to use the HTTP provider to download a file to the *cwd*:

```
import os
from snakemake.remote.HTTP import RemoteProvider as HTTPRemoteProvider

HTTP = HTTPRemoteProvider()

rule all:
    input:
        HTTP.remote("www.example.com/path/to/document.pdf", keep_local=True)
    run:
        outputName = os.path.basename(input[0])
        shell("mv {input} {outputName}")
```

To connect on a different port, specify the port as part of the URI string:

```
from snakemake.remote.HTTP import RemoteProvider as HTTPRemoteProvider
HTTP = HTTPRemoteProvider()

rule all:
    input:
        HTTP.remote("www.example.com:8080/path/to/document.pdf", keep_local=True)
```

By default, the HTTP provider always uses HTTPS (TLS). If you need to connect to a resource with regular HTTP (no TLS), you must explicitly include `insecure` as a `kwarg` to `remote()`:

```
from snakemake.remote.HTTP import RemoteProvider as HTTPRemoteProvider
HTTP = HTTPRemoteProvider()

rule all:
    input:
        HTTP.remote("www.example.com/path/to/document.pdf", insecure=True, keep_
↪local=True)
```

If the URI used includes characters not permitted in a local file path, you may include them as part of the `additional_request_string` in the `kwargs` for `remote()`. This may also be useful for including additional parameters you don not want to be part of the local filename (since the URI string becomes the local file name).

```
from snakemake.remote.HTTP import RemoteProvider as HTTPRemoteProvider
HTTP = HTTPRemoteProvider()

rule all:
    input:
        HTTP.remote("example.com/query.php", additional_request_string="?range=2;3")
```

If the file requires authentication, you can specify a username and password for HTTP Basic Auth with the Remote Provider, or with each instance of *remote()*. For different types of authentication, you can pass in a Python `requests.auth` object (see here) the *auth* `kwarg`.

```
from snakemake.remote.HTTP import RemoteProvider as HTTPRemoteProvider
HTTP = HTTPRemoteProvider(username="myusername", password="mypassword")

rule all:
    input:
        HTTP.remote("example.com/interactive.php", keep_local=True)
```

```
from snakemake.remote.HTTP import RemoteProvider as HTTPRemoteProvider
HTTP = HTTPRemoteProvider()

rule all:
    input:
        HTTP.remote("example.com/interactive.php", username="myusername", password=
↪"mypassword", keep_local=True)
```

```
from snakemake.remote.HTTP import RemoteProvider as HTTPRemoteProvider
HTTP = HTTPRemoteProvider()

rule all:
    input:
        HTTP.remote("example.com/interactive.php", auth=requests.auth.HTTPDigestAuth(
↪"myusername", "mypassword"), keep_local=True)
```

Since remote servers do not present directory contents uniformly, `glob_wildcards()` is __not__ supported by the HTTP provider.

Note: Snakemake automatically decompresses http remote files if they are marked as *Content-Encoding: gzip* by the server and **not** end with `.gz`. The reason is that for those files the rule obviously expects the uncompressed version. If in contrast the file ends on `.gz` the compressed version is expected and therefore no automatic decompression happens.

4.17.6 File Transfer Protocol (FTP)

Snakemake can work with files stored on regular FTP. Currently supported are authenticated FTP and anonymous FTP, excluding FTP via TLS.

Usage is similar to the SFTP provider, however the paths specified are relative to the FTP home directory (since this is typically a chroot):

```
from snakemake.remote.FTP import RemoteProvider as FTPRemoteProvider

FTP = FTPRemoteProvider(username="myusername", password="mypassword")

rule all:
    input:
        FTP.remote("example.com/rel/path/to/file.tar.gz")
```

The port may be specified in either the provider, or in each instance of *remote()*:

```
from snakemake.remote.FTP import RemoteProvider as FTPRemoteProvider

```

(continues on next page)

(continued from previous page)

```
FTP = FTPRemoteProvider(username="myusername", password="mypassword", port=2121)

rule all:
    input:
        FTP.remote("example.com/rel/path/to/file.tar.gz")
```

```
from snakemake.remote.FTP import RemoteProvider as FTPRemoteProvider

FTP = FTPRemoteProvider(username="myusername", password="mypassword")

rule all:
    input:
        FTP.remote("example.com:2121/rel/path/to/file.tar.gz")
```

Anonymous download of FTP resources is possible:

```
from snakemake.remote.FTP import RemoteProvider as FTPRemoteProvider
FTP = FTPRemoteProvider()

rule all:
    input:
        # only keeping the file so we can move it out to the cwd
        FTP.remote("example.com/rel/path/to/file.tar.gz", keep_local=True)
    run:
        shell("mv {input} ./")
```

`glob_wildcards()`:

```
from snakemake.remote.FTP import RemoteProvider as FTPRemoteProvider
FTP = FTPRemoteProvider(username="myusername", password="mypassword")

print(FTP.glob_wildcards("example.com/somedir/{file}.txt"))
```

Setting *immediate_close=True* allows the use of a large number of remote FTP input files in a job where the endpoint server limits the number of concurrent connections. When *immediate_close=True*, Snakemake will terminate FTP connections after each remote file action (*exists()*, *size()*, *download()*, *mtime()*, etc.). This is in contrast to the default behavior which caches FTP details and leaves the connection open across actions to improve performance (closing the connection upon job termination). :

```
from snakemake.remote.FTP import RemoteProvider as FTPRemoteProvider
FTP = FTPRemoteProvider()

rule all:
    input:
        # only keep the file so we can move it out to the cwd
        # This server limits the number of concurrent connections so we need to have␣
↪Snakemake close each after each FTP action.
        FTP.remote(expand("ftp.example.com/rel/path/to/{file}", file=large_list),␣
↪keep_local=True, immediate_close=True)
    run:
        shell("mv {input} ./")
```

`glob_wildcards()`:

```
from snakemake.remote.FTP import RemoteProvider as FTPRemoteProvider
FTP = FTPRemoteProvider(username="myusername", password="mypassword")
```

(continues on next page)

(continued from previous page)

```
print(FTP.glob_wildcards("example.com/somedir/{file}.txt"))
```

4.17.7 Dropbox

The Dropbox remote provider allows you to upload and download from your Dropbox account without having the client installed on your machine. In order to use the provider you first need to register an "app" on the Dropbox developer website, with access to the Full Dropbox. After registering, generate an OAuth2 access token. You will need the token to use the Snakemake Dropbox remote provider.

Using the Dropbox provider is straightforward:

```
from snakemake.remote.dropbox import RemoteProvider as DropboxRemoteProvider
DBox = DropboxRemoteProvider(oauth2_access_token="mytoken")

rule all:
    input:
        DBox.remote("path/to/input.txt")
```

`glob_wildcards()` is supported:

```
from snakemake.remote.dropbox import RemoteProvider as DropboxRemoteProvider
DBox = DropboxRemoteProvider(oauth2_access_token="mytoken")

DBox.glob_wildcards("path/to/{title}.txt")
```

Note that Dropbox paths are case-insensitive.

4.17.8 XRootD

Snakemake can be used with XRootD backed storage provided the python bindings are installed. This is typically most useful when combined with the `stay_on_remote` flag to minimise local storage requirements. This flag can be overridden on a file by file basis as described in the S3 remote. Additionally `glob_wildcards()` is supported:

```
from snakemake.remote.XRootD import RemoteProvider as XRootDRemoteProvider

XRootD = XRootDRemoteProvider(stay_on_remote=True)
file_numbers = XRootD.glob_wildcards("root://eospublic.cern.ch//eos/opendata/lhcb/
↪MasterclassDatasets/D0lifetime/2014/mclasseventv2_D0_{n}.root").n

rule all:
    input:
        expand("local_data/mclasseventv2_D0_{n}.root", n=file_numbers)

rule make_data:
    input:
        XRootD.remote("root://eospublic.cern.ch//eos/opendata/lhcb/
↪MasterclassDatasets/D0lifetime/2014/mclasseventv2_D0_{n}.root")
    output:
        'local_data/mclasseventv2_D0_{n}.root'
    shell:
        'xrdcp {input[0]} {output[0]}'
```

4.17.9 GenBank / NCBI Entrez

Snakemake can directly source input files from GenBank and other NCBI Entrez databases if the Biopython library is installed.

```
from snakemake.remote.NCBI import RemoteProvider as NCBIRemoteProvider
NCBI = NCBIRemoteProvider(email="someone@example.com") # email required by NCBI to␣
↪prevent abuse

rule all:
    input:
        "size.txt"

rule download_and_count:
    input:
        NCBI.remote("KY785484.1.fasta", db="nuccore")
    output:
        "size.txt"
    run:
        shell("wc -c {input} > {output}")
```

The output format and source database of a record retrieved from GenBank is inferred from the file extension specified. For example, `NCBI.RemoteProvider().remote("KY785484.1.fasta", db="nuccore")` will download a FASTA file while `NCBI.RemoteProvider().remote("KY785484.1.gb", db="nuccore")` will download a GenBank-format file. If the options are ambiguous, Snakemake will raise an exception and inform the user of possible format choices. To see available formats, consult the Entrez EFetch documentation. To view the valid file extensions for these formats, access `NCBI.RemoteProvider()._gb.valid_extensions`, or instantiate an `NCBI.NCBIHelper()` and access `NCBI.NCBIHelper().valid_extensions` (this is a property).

When used in conjunction with `NCBI.RemoteProvider().search()`, Snakemake and `NCBI.RemoteProvider().remote()` can be used to find accessions by query and download them:

```
from snakemake.remote.NCBI import RemoteProvider as NCBIRemoteProvider
NCBI = NCBIRemoteProvider(email="someone@example.com") # email required by NCBI to␣
↪prevent abuse

# get accessions for the first 3 results in a search for full-length Zika virus␣
↪genomes
# the query parameter accepts standard GenBank search syntax
query = '"Zika virus"[Organism] AND (("9000"[SLEN] : "20000"[SLEN]) AND ("2017/03/20
↪"[PDAT] : "2017/03/24"[PDAT])) '
accessions = NCBI.search(query, retmax=3)

# give the accessions a file extension to help the RemoteProvider determine the
# proper output type.
input_files = expand("{acc}.fasta", acc=accessions)

rule all:
    input:
        "sizes.txt"

rule download_and_count:
    input:
        # Since *.fasta files could come from several different databases, specify␣
↪the database here.
        # if the input files are ambiguous, the provider will alert the user with␣
↪possible options
```

(continues on next page)

(continued from previous page)

```
        # standard options like "seq_start" are supported
        NCBI.remote(input_files, db="nuccore", seq_start=5000)

    output:
        "sizes.txt"
    run:
        shell("wc -c {input} > sizes.txt")
```

Normally, all accessions for a query are returned from `NCBI.RemoteProvider.search()`. To truncate the results, specify `retmax=<desired_number>`. Standard Entrez fetch query options are supported as kwargs, and may be passed in to `NCBI.RemoteProvider.remote()` and `NCBI.RemoteProvider.search()`.

4.17.10 WebDAV

WebDAV support is currently `experimental` and available in Snakemake 4.0 and later.

Snakemake supports reading and writing WebDAV remote files. The protocol defaults to `https://`, but insecure connections can be used by specifying `protocol="http://"`. Similarly, the port defaults to 443, and can be overridden by specifying `port=##` or by including the port as part of the file address.

```
from snakemake.remote import webdav

webdav = webdav.RemoteProvider(username="test", password="test", protocol="http://")

rule a:
    input:
        webdav.remote("example.com:8888/path/to/input_file.csv"),
    shell:
        # do something
```

4.17.11 GFAL

GFAL support is available in Snakemake 4.1 and later.

Snakemake supports reading and writing remote files via the GFAL command line client (gfal-* commands). By this, it supports various grid storage protocols like GridFTP. In general, if you are able to use the *gfal-** commands directly, Snakemake support for GFAL will work as well.

```
from snakemake.remote import gfal

gfal = gfal.RemoteProvider()

rule a:
    input:
        gfal.remote("gridftp.grid.sara.nl:2811/path/to/infile.txt")
    output:
        gfal.remote("gridftp.grid.sara.nl:2811/path/to/outfile.txt")
    shell:
        # do something
```

Authentication has to be setup in the system, e.g. via certificates in the `.globus` directory. Usually, this is already the case and no action has to be taken.

Note that GFAL support used together with the flags `--no-shared-fs` and `--default-remote-provider` enables you to transparently use Snakemake in a grid computing environment without a shared network filesystem. For an example see the surfsara-grid configuration profile.

4.17.12 GridFTP

GridFTP support is available in Snakemake 4.3.0 and later.

As a more specialized alternative to the GFAL remote provider, Snakemake provides a GridFTP remote provider. This provider only supports the GridFTP protocol. Internally, it uses the globus-url-copy command for downloads and uploads, while all other tasks are delegated to the GFAL remote provider.

```
from snakemake.remote import gridftp

gridftp = gridftp.RemoteProvider(streams=4)

rule a:
    input:
        gridftp.remote("gridftp.grid.sara.nl:2811/path/to/infile.txt")
    output:
        gridftp.remote("gridftp.grid.sara.nl:2811/path/to/outfile.txt")
    shell:
        # do something
```

Authentication has to be setup in the system, e.g. via certificates in the `.globus` directory. Usually, this is already the case and no action has to be taken. The keyword argument to the remote provider allows to set the number of parallel streams used for file tranfers(4 per default). When ``streams``is set to 1 or smaller, the files are trasfered in a serial way. Paralell stream may be unsupported depending on the system configuration.

Note that GridFTP support used together with the flags `--no-shared-fs` and `--default-remote-provider` enables you to transparently use Snakemake in a grid computing environment without a shared network filesystem. For an example see the surfsara-grid configuration profile.

4.17.13 Remote cross-provider transfers

It is possible to use Snakemake to transfer files between remote providers (using the local machine as an intermediary), as long as the sub-directory (bucket) names differ:

```
from snakemake.remote.GS import RemoteProvider as GSRemoteProvider
from snakemake.remote.S3 import RemoteProvider as S3RemoteProvider

GS = GSRemoteProvider(access_key_id="MYACCESSKEYID", secret_access_key=
↪"MYSECRETACCESSKEY")
S3 = S3RemoteProvider(access_key_id="MYACCESSKEYID", secret_access_key=
↪"MYSECRETACCESSKEY")

fileList, = S3.glob_wildcards("source-bucket/{file}.bam")
rule all:
    input:
        GS.remote( expand("destination-bucket/{file}.bam", file=fileList) )
rule transfer_S3_to_GS:
    input:
        S3.remote( expand("source-bucket/{file}.bam", file=fileList) )
    output:
        GS.remote( expand("destination-bucket/{file}.bam", file=fileList) )
```

(continues on next page)

(continued from previous page)

```
    run:
        shell("cp {input} {output}")
```

4.17.14 iRODS

You can access an iRODS server to retrieve data from and upload data to it. If your iRODS server is not set to a certain timezone, it is using UTC. It is advised to shift the modification time provided by iRODS (`modify_time`) then to your timezone by providing the `timezone` parameter such that timestamps coming from iRODS are converted to the correct time.

iRODS actually does not save the timestamp from your original file but creates its own timestamp of the upload time. When iRODS downloads the file for processing, it does not take the timestamp from the remote file. Instead, the file will have the timestamp when it was downloaded. To get around this, we create a metadata entry to store the original file stamp from your system and alter the timestamp of the downloaded file accordingly. While uploading, the metadata entries `atime`, `ctime` and `mtime` are added. When this entry does not exist (because this module didn't upload the file), we fall back to the timestamp provided by iRODS with the above mentioned strategy.

To access the iRODS server you need to have an iRODS environment configuration file available and in this file the authentication needs to be configured. The iRODS configuration file can be created by following the official instructions).

The default location for the configuration file is `~/.irods/irods_environment.json`. The `RemoteProvider()` class accepts the parameter `irods_env_file` where an alternative path to the `irods_environment.json` file can be specified. Another way is to export the environment variable `IRODS_ENVIRONMENT_FILE` in your shell to specify the location.

There are several ways to configure the authentication against the iRODS server, depending on what your iRODS server offers. If you are using the authentication via password, the default location of the authentication file is `~/.irods/.irodsA`. Usually this file is generated with the `iinit` command from the `iCommands` program suite. Inside the `irods_environment.json` file, the parameter `"irods_authentication_file"` can be set to specifiy an alternative location for the `.irodsA` file. Another possibility to change the location is to export the environment variable `IRODS_AUTHENTICATION_FILE`.

The `glob_wildcards()` function is supported.

```
from snakemake.remote.iRODS import RemoteProvider

irods = RemoteProvider(irods_env_file='setup-data/irods_environment.json',
                       timezone="Europe/Berlin") # all parameters are optional

# please note the comma after the variable name!
# access: irods.remote(expand('home/rods/{f}), f=files))
files, = irods.glob_wildcards('home/rods/{files}')

rule all:
    input:
        irods.remote('home/rods/testfile.out'),

rule gen:
    input:
        irods.remote('home/rods/testfile.in')
    output:
        irods.remote('home/rods/testfile.out')
    shell:
        r"""
```

(continues on next page)

(continued from previous page)

```
        touch {output}
        """
```

An example for the iRODS configuration file (`irods_environment.json`):

```
{
    "irods_host": "localhost",
    "irods_port": 1247,
    "irods_user_name": "rods",
    "irods_zone_name": "tempZone",
    "irods_authentication_file": "setup-data/.irodsA"
}
```

Please note that the `zone` folder is not included in the path as it will be taken from the configuration file. The path also must not start with a `/`.

By default, temporarily stored local files are removed. You can specify anyway the parameter `overwrite` to tell iRODS to overwrite existing files that are downloaded, because iRODS complains if a local file already exists when a download attempt is issued (uploading is not a problem, though).

In the Snakemake source directory in `snakemake/tests/test_remote_irods` you can find a working example.

4.17.15 EGA

The European Genome-phenome Archive (EGA) is a service for permanent archiving and sharing of all types of personally identifiable genetic and phenotypic data resulting from biomedical research projects.

From version 5.2 on, Snakemake provides experimental support to use EGA as a remote provider, such that EGA hosted files can be transparently used as input. For this to work, you need to define your username and password as environment variables `EGA_USERNAME` and `EGA_PASSWORD`.

Files in a dataset are addressed via the pattern `ega/<dataset_id>/<filename>`. Note that the filename should not include the `.cip` ending that is sometimes displayed in EGA listings:

```
import snakemake.remote.EGA as EGA

ega = EGA.RemoteProvider()


rule a:
    input:
        ega.remote("ega/EGAD00001002142/COLO_829_EPleasance_TGENPipe.bam.bai")
    output:
        "data/COLO_829BL_BCGSC_IlluminaPipe.bam.bai"
    shell:
        "cp {input} {output}"
```

Upon download, Snakemake will automatically decrypt the file and check the MD5 hash.

4.17.16 Zenodo

Zenodo is a catch-all open data and software repository. Snakemake allows file upload and download from Zenodo. To access your Zenodo files you need to set up Zenodo account and create a personal access token with at least write scope. Personal access token must be supplied as `access_token` argument. You need to supply deposition id as `deposition` to upload or download files from your deposition. If no deposition id is supplied, Snakemake creates a new deposition for upload. Zenodo UI and REST API responses were designed with having in mind uploads of a total of 20-30 files. Avoid creating uploads with too many files, and instead group and zip them to make it easier their distribution to end-users.

```
from snakemake.remote.zenodo import RemoteProvider
import os

# let Snakemake assert the presence of the required environment variable
envvars:
    "ZENODO_ACCESS_TOKEN"

zenodo = RemoteProvider(deposition="your deposition id", access_token=os.environ[
↪"ZENODO_ACCESS_TOKEN"])

rule upload:
    input:
        "output/results.csv"
    output:
        zenodo.remote("results.csv")
    shell:
        "cp {input} {output}"
```

It is possible to use Zenodo sandbox environment for testing by setting `sandbox=True` argument. Using sandbox environment requires setting up sandbox account with its personal access token.

Restricted access

If you need to access a deposition with restricted access, you have to additionally provide a `restricted_access_token`. This can be obtained from the restricted access URL that Zenodo usually sends you via email once restricted access to a deposition (requested via the web interface) has been granted by the owner. Let `` https://zenodo.org/record/000000000?token=dlksajdlkjaslnflkndlfnjnn`` be the URL provided by Zenodo. Then, the `restricted_access_token` is `dlksajdlkjaslnflkndlfnjnn`, and it can be used as follows:

```
from snakemake.remote.zenodo import RemoteProvider
import os

# let Snakemake assert the presence of the required environment variable
envvars:
    "ZENODO_ACCESS_TOKEN",
    "ZENODO_RESTRICTED_ACCESS_TOKEN"

zenodo = RemoteProvider(
    deposition="your deposition id",
    access_token=os.environ["ZENODO_ACCESS_TOKEN"],
    restricted_access_token=os.environ["ZENODO_RESTRICTED_ACCESS_TOKEN"]
)

rule upload:
    input:
        "output/results.csv"
```

(continues on next page)

(continued from previous page)

```
    output:
        zenodo.remote("results.csv")
    shell:
        "cp {input} {output}"
```

4.17.17 Auto remote provider

A wrapper which automatically selects an appropriate remote provider based on the url's scheme. It removes some of the boilerplate code required to download remote files from various providers. The auto remote provider only works for those which do not require the passing of keyword arguments to the `RemoteProvider` object.

```
from snakemake.remote import AUTO


rule all:
    input:
        'foo'


rule download:
    input:
        ftp_file_list=AUTO.remote([
            'ftp://ftp.ncbi.nlm.nih.gov/pub/taxonomy/taxcat.tar.gz',
            'ftp://ftp.ncbi.nlm.nih.gov/pub/taxonomy/taxdump.tar.gz'
        ], keep_local=True),
        http_file=AUTO.remote(
            'https://github.com/hetio/hetionet/raw/master/hetnet/tsv/hetionet-v1.0-
↪nodes.tsv'
        )
    output:
        touch('foo')
    shell:
        """
        head {input.http_file}
        """
```

4.18 Utils

The module `snakemake.utils` provides a collection of helper functions for common tasks in Snakemake workflows. Details can be found in utils-api.

4.19 Distribution and Reproducibility

It is recommended to store each workflow in a dedicated git repository of the following structure:

```
├── .gitignore
├── README.md
├── LICENSE.md
├── workflow
|   ├── rules
|   |   ├── module1.smk
|   |   └── module2.smk
|   ├── envs
|   |   ├── tool1.yaml
|   |   └── tool2.yaml
|   ├── scripts
|   |   ├── script1.py
|   |   └── script2.R
|   ├── notebooks
|   |   ├── notebook1.py.ipynb
|   |   └── notebook2.r.ipynb
|   ├── report
|   |   ├── plot1.rst
|   |   └── plot2.rst
|   └── Snakefile
├── config
|   ├── config.yaml
|   └── some-sheet.tsv
├── results
└── resources
```

In other words, the workflow code goes into a subfolder `workflow`, while the configuration is stored in a subfolder `config`. Inside of the `workflow` subfolder, the central `Snakefile` marks the entrypoint of the workflow (it will be automatically discovered when running snakemake from the root of above structure. This main structure and the recommendations below are implemented in this Snakemake workflow template that you can use to create your own workflow repository with a single click on "Use this template". In addition to the central `Snakefile`, rules can be stored in a modular way, using the optional subfolder `workflow/rules`. Such modules should end with `.smk`, the recommended file extension of Snakemake. Further, *scripts* should be stored in a subfolder `workflow/scripts` and notebooks in a subfolder `workflow/notebooks`. Conda environments (see *Integrated Package Management*) should be stored in a subfolder `workflow/envs` (make sure to keep them as finegrained as possible to improve transparency and maintainability). Finally, *report caption files* should be stored in `workflow/report`. All output files generated in the workflow should be stored under `results`, unless they are rather retrieved resources, in which case they should be stored under `resources`. The latter subfolder may also contain small resources that shall be delivered along with the workflow via git (although it might be tempting, please refrain from trying to generate output file paths with string concatenation of a central `outdir` variable or so, as this hampers readability).

Workflows set up in above structure can be easily used and combined via *the Snakemake module system*. Such deployment can even be automated via Snakedeploy. Moreover, by publishing a workflow on Github and following a set of additional rules the workflow will be automatically included in the Snakemake workflow catalog, thereby easing discovery and even automating its usage documentation. For an example of such automated documentation, see here.

Visit the Snakemake Workflows Project for more best-practice workflows.

4.19.1 Using and combining pre-exising workflows

Via the *module/use* system introduced with Snakemake 6.0, it is very easy to deploy existing workflows for new projects. This ranges from the simple application to new data to the complex combination of several complementary workflows in order to perfom an integrated analysis over multiple data types.

Consider the following example:

```
from snakemake.utils import min_version
min_version("6.0")

configfile: "config/config.yaml"

module dna_seq:
    snakefile:
        # here, it is also possible to provide a plain raw URL like "https://github.
↪com/snakemake-workflows/dna-seq-gatk-variant-calling/raw/v2.0.1/workflow/Snakefile"
        github("snakemake-workflows/dna-seq-gatk-variant-calling", path="workflow/
↪Snakefile", tag="v2.0.1")
    config:
        config

use rule * from dna_seq
```

First, we load a local configuration file. Next, we define the module `dna_seq` to be loaded from the URL `https://github.com/snakemake-workflows/dna-seq-gatk-variant-calling/raw/v2.0.1/workflow/Snakefile`, while using the contents of the local configuration file. Note that it is possible to either specify the full URL pointing to the raw Snakefile as a string or to use the github marker as done here. With the latter, Snakemake can however cache the used source files persistently (if a tag is given), such that they don't have to be downloaded on each invocation. Finally we declare all rules of the dna_seq module to be used.

This kind of deployment is equivalent to just cloning the original repository and modifying the configuration in it. However, the advantage here is that we are (a) able to easily extend of modify the workflow, while making the changes transparent, and (b) we can store this workflow in a separate (e.g. private) git repository, along with for example configuration and meta data, without the need to duplicate the workflow code. Finally, we are always able to later combine another module into the current workflow, e.g. when further kinds of analyses are needed. The ability to modify rules upon using them (see *Modules*) allows for arbitrary rewiring and configuration of the combined modules.

For example, we can easily add another rule to extend the given workflow:

```
from snakemake.utils import min_version
min_version("6.0")

configfile: "config/config.yaml"

module dna_seq:
    snakefile:
        # here, it is also possible to provide a plain raw URL like "https://github.
↪com/snakemake-workflows/dna-seq-gatk-variant-calling/raw/v2.0.1/workflow/Snakefile"
        github("snakemake-workflows/dna-seq-gatk-variant-calling", path="workflow/
↪Snakefile", tag="v2.0.1")
    config: config

use rule * from dna_seq as dna_seq_*

# easily extend the workflow
rule plot_vafs:
    input:
```

(continues on next page)

(continued from previous page)

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

# Define a new default target that collects both the targets from the dna_seq module␣
↪as well as
# the new plot.
rule all:
    input:
        rules.dna_seq_all.input,
        "results/plots/vafs.svg",
    default_target: True
```

Above, we have added a prefix to all rule names of the dna_seq module, such that there is no name clash with the added rules (`as dna_seq_*` in the `use rule` statement). In addition, we have added a new rule `all`, defining the default target in case the workflow is executed (as usually) without any specific target files or rule. The new target rule collects both all input files of the rule `all` from the dna_seq workflow, as well as additionally collecting the new plot.

It is possible to further extend the workflow with other modules, thereby generating an integrative analysis. Here, let us assume that we want to conduct another kind of analysis, say RNA-seq, using a different external workflow. We can extend above example in the following way:

```
from snakemake.utils import min_version
min_version("6.0")

configfile: "config/config.yaml"

module dna_seq:
    snakefile:
        github("snakemake-workflows/dna-seq-gatk-variant-calling", path="workflow/
↪Snakefile", tag="v2.0.1")
    config: config["dna-seq"]
    prefix: "dna-seq"

use rule * from dna_seq as dna_seq_*

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

module rna_seq:
    snakefile:
        github("snakemake-workflows/rna-seq-kallisto-sleuth", path="workflow/Snakefile
↪", tag="v2.0.1")
    config: config["rna-seq"]
    prefix: "rna-seq"

use rule * from rna_seq as rna_seq_*
```

(continues on next page)

(continued from previous page)

```
# Define a new default target that collects all the targets from the dna_seq and rna_
↪seq module.
rule all:
    input:
        rules.dna_seq_all.input,
        rules.rna_seq_all.input,
    default_target: True
```

Above, several things have changed.

- First, we have added another module `rna_seq`.
- Second, we have added a prefix to all non-absolute input and output file names of both modules (`prefix: "dna-seq"` and `prefix: "rna-seq"`) in order to avoid file name clashes.
- Third, we have added a default target rule that collects both the default targets from the module `dna_seq` as well as the module `rna_seq`.
- Finally, we provide the config of the two modules via two separate sections in the common config file (`config["dna-seq"]` and `config["rna-seq"]`).

4.19.2 Uploading workflows to WorkflowHub

In order to share a workflow with the scientific community it is advised to upload the repository to WorkflowHub, where each submission will be automatically parsed and encapsulated into a Research Object Crate. That way a *snakemake* workflow is annotated with proper metadata and thus complies with the FAIR principles of scientific data.

To adhere to the high WorkflowHub standards of scientific workflows the recommended *snakemake* repository structure presented above needs to be extended by the following elements:

- Code of Conduct
- Contribution instructions
- Workflow rule graph
- Workflow documentation
- Test directory

A code of conduct for the repository developers as well as instruction on how to contribute to the project should be placed in the top-level files: `CODE_OF_CONDUCT.md` and `CONTRIBUTING.md`, respectively. Each *snakemake* workflow repository needs to contain an SVG-formatted rule graph placed in a subdirectory `images/rulegraph.svg`. Additionally, the workflow should be annotated with a technical documentation of all of its subsequent steps, described in `workflow/documentation.md`. Finally, the repository should contain a `.tests` directory with two subdirectories: `.tests/integration` and `.tests/unit`. The former has to contain all the input data, configuration specifications and shell commands required to run an integration test of the whole workflow. The latter shall contain subdirectories dedicated to testing each of the separate workflow steps independently. To simplify the testing procedure *snakemake* can automatically generate unit tests from a successful workflow execution (see *Automatically generating unit tests*).

Therefore, the repository structure should comply with:

```
├── .gitignore
├── README.md
├── LICENSE.md
├── CODE_OF_CONDUCT.md
├── CONTRIBUTING.md
```

(continues on next page)

(continued from previous page)

```
├── .tests
│   ├── integration
│   └── unit
├── images
│   └── rulegraph.svg
├── workflow
│   ├── rules
│   │   ├── module1.smk
│   │   └── module2.smk
│   ├── envs
│   │   ├── tool1.yaml
│   │   └── tool2.yaml
│   ├── scripts
│   │   ├── script1.py
│   │   └── script2.R
│   ├── notebooks
│   │   ├── notebook1.py.ipynb
│   │   └── notebook2.r.ipynb
│   ├── report
│   │   ├── plot1.rst
│   │   └── plot2.rst
│   ├── Snakefile
│   └── documentation.md
├── config
│   ├── config.yaml
│   └── some-sheet.tsv
├── results
└── resources
```

4.19.3 Integrated Package Management

With Snakemake 3.9.0 it is possible to define isolated software environments per rule. Upon execution of a workflow, the Conda package manager is used to obtain and deploy the defined software packages in the specified versions. Packages will be installed into your working directory, without requiring any admin/root priviledges. Given that conda is available on your system (see Miniconda), to use the Conda integration, add the `--use-conda` flag to your workflow execution command, e.g. `snakemake --cores 8 --use-conda`. When `--use-conda` is activated, Snakemake will automatically create software environments for any used wrapper (see *Wrappers*). Further, you can manually define environments via the `conda` directive, e.g.:

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

with the following environment definition:

```
channels:
  - r
dependencies:
```

(continues on next page)

(continued from previous page)

```
- r=3.3.1
- r-ggplot2=2.1.0
```

The path to the environment definition is interpreted as **relative to the Snakefile that contains the rule** (unless it is an absolute path, which is discouraged).

Instead of using a concrete path, it is also possible to provide a path containing wildcards (which must also occur in the output files of the rule), analogous to the specification of input files.

In addition, it is possible to use a callable which returns a `str` value. The signature of the callable has to be `callable(wildcards [, params] [, input])` (`params` and `input` are optional parameters).

Note that the use of distinct conda environments for different jobs from the same rule is currently not properly displayed in the generated reports. At the moment, only a single, random conda environment is shown.

Note

Note that conda environments are only used with `shell`, `script`, `notebook` and the `wrapper` directive, not the `run` directive. The reason is that the `run` directive has access to the rest of the Snakefile (e.g. globally defined variables) and therefore must be executed in the same process as Snakemake itself. If used with `notebook` directive, the associated conda environment should have package `jupyter` installed (this package contains dependencies required to execute the notebook).

Further, note that search path modifying environment variables like `R_LIBS` and `PYTHONPATH` can interfere with your conda environments. Therefore, Snakemake automatically deactivates them for a job when a conda environment definition is used. If you know what you are doing, in order to deactivate this behavior, you can use the flag `--conda-not-block-search-path-envvars`.

Snakemake will store the environment persistently in `.snakemake/conda/$hash` with `$hash` being the MD5 hash of the environment definition file content. This way, updates to the environment definition are automatically detected. Note that you need to clean up environments manually for now. However, in many cases they are lightweight and consist of symlinks to your central conda installation.

Conda deployment also works well for offline or air-gapped environments. Running `snakemake --use-conda --conda-create-envs-only` will only install the required conda environments without running the full workflow. Subsequent runs with `--use-conda` will make use of the local environments without requiring internet access.

Freezing environments to exactly pinned packages

If Snakemake finds a special file ending on `<platform>.pin.txt` next to a conda environment file (with `<platform>` being the current platform, e.g. `linux-64`), it will try to use the contents of that file to determine the conda packages to deploy. The file is expected to contain conda's explicit specification file format. Snakemake will first try to deploy the environment using that file, and only if that fails it will use the regular enviroment file.

This enables to freeze an environment to a certain state, and will ensure that people using a workflow will get exactly the same environments down to the individual package builds, which is in fact very similar to providing the environment encapsulated in a container image. Generating such pin files for conda environments can be automatically done using Snakedeploy. Let `envs/ggplot.yaml` be the conda environment file used in the example above. Then, the pinning can be generated with

```
snakedeploy pin-conda-envs envs/ggplot.yaml
```

Multiple paths to environments can be provided at the same time; also see `snakedeploy pin-conda-envs --help`.

Of course, it is **important to update the pinnings** whenever the original environment is modified, such that they do not diverge.

Updating environments

When a workflow contains many conda environments, it can be helpful to automatically update them to the latest versions of all packages. This can be done automatically via Snakedeploy:

```
snakedeploy update-conda-envs envs/ggplot.yaml
```

Multiple paths to environments can be provided at the same time; also see `snakedeploy update-conda-envs --help`.

Providing post-deployment scripts

From Snakemake 6.14 onwards post-deployment shell-scripts can be provided to perform additional adjustments of a conda environment. This might be helpful in case a conda package is missing components or requires further configuration for execution. Post-deployment scripts must be placed next to their corresponding environment-file and require the suffix `.post-deploy.sh`, e.g.:

```
rule NAME:
    input:
        "seqs.fastq"
    output:
        "results.tsv"
    conda:
        "envs/interproscan.yaml"
    shell:
        "interproscan.sh -i {input} -f tsv -o {output}"
```

```
├── Snakefile
└── envs
    ├── interproscan.yaml
    └── interproscan.post-deploy.sh
```

The path of the conda environment can be accessed within the script via `$CONDA_PREFIX`. Importantly, if the script relies on certain shell specific syntax, (e.g. *set -o pipefail* for bash), make sure to add a matching shebang to the script, e.g.:

```
#!env bash
set -o pipefail
# ...
```

If no shebang line like above (`#!env bash`) is provided, the script will be executed with the `sh` command.

4.19.4 Using already existing named conda environments

Sometimes it can be handy to refer to an already existing named conda environment from a rule, instead of defining a new one from scratch. Importantly, one should be aware that this can **hamper reproducibility**, because the workflow then relies on this environment to be present **in exactly the same way** on any new system where the workflow is executed. Essentially, you will have to take care of this manually in such a case. Therefore, the approach using environment definition files described above is highly recommended and preferred.

Nevertheless, in case you are still sure that you want to use an existing named environment, it can simply be put into the conda directive, e.g.

```
rule NAME:
    input:
        "table.txt"
    output:
        "plots/myplot.pdf"
    conda:
        "some-env-name"
    script:
        "scripts/plot-stuff.R"
```

For such a rule, Snakemake will just activate the given environment, instead of automatically deploying anything. Instead of using a concrete name, it is also possible to provide a name containing wildcards (which must also occur in the output files of the rule), analogous to the specification of input files.

Note that Snakemake distinguishes file based environments from named ones as follows: if the given specification ends on `.yaml` or `.yml`, Snakemake assumes it to be a path to an environment definition file; otherwise, it assumes the given specification to be the name of an existing environment.

4.19.5 Running jobs in containers

As an alternative to using Conda (see above), it is possible to define, for each rule, a (docker) container to use, e.g.,

```
rule NAME:
    input:
        "table.txt"
    output:
        "plots/myplot.pdf"
    container:
        "docker://joseespinosa/docker-r-ggplot2"
    script:
        "scripts/plot-stuff.R"
```

When executing Snakemake with

```
snakemake --use-singularity
```

it will execute the job within a container that is spawned from the given image. Allowed image urls entail everything supported by singularity (e.g., `shub://` and `docker://`). However, `docker://` is preferred, as other container runtimes will be supported in the future (e.g. podman).

Note

Note that singularity integration is only used with `shell`, `script` and the `wrapper` directive, not the `run` directive. The reason is that the `run` directive has access to the rest of the Snakefile (e.g. globally defined variables)

and therefore must be executed in the same process as Snakemake itself.

When `--use-singularity` is combined with `--kubernetes` (see *Executing a Snakemake workflow via kubernetes*), cloud jobs will be automatically configured to run in priviledged mode, because this is a current requirement of the singularity executable. Importantly, those privileges won't be shared by the actual code that is executed in the singularity container though.

A global definition of a container image can be given:

```
container: "docker://joseespinosa/docker-r-ggplot2"

rule NAME:
    ...
```

In this case all jobs will be executed in a container. You can disable execution in container by setting the container directive of the rule to `None`.

```
container: "docker://joseespinosa/docker-r-ggplot2"

rule NAME:
    container: None
```

4.19.6 Containerization of Conda based workflows

While *Integrated Package Management* provides control over the used software in exactly the desired versions, it does not control the underlying operating system. However, given a workflow with conda environments for each rule, Snakemake can automatically generate a container image specification (in the form of a `Dockerfile`) that contains all required environments via the flag –containerize:

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

The container image specification generated by Snakemake aims to be transparent and readable, e.g. by displaying each contained environment in a human readable way. Via the special directive `containerized` this container image can be used in the workflow (both globally or per rule) such that no further conda package downloads are necessary, for example:

```
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"
```

Using the containerization of Snakemake has three advantages over manually crafting a container image for a workflow:

1. A workflow with conda environment definitions is much more transparent to the reader than a black box container image, as each rule directly shows which software stack is used. Containerization just persistently projects those environments into a container image.
2. It remains possible to run the workflow without containers, just via the conda environments.

3. During development, testing can first happen without the container and just on the conda environments. When releasing a production version of the workflow the image can be uploaded just once and for future stable releases, thereby limiting the overhead created in container registries.

4.19.7 Ad-hoc combination of Conda package management with containers

While *Integrated Package Management* provides control over the used software in exactly the desired versions, it does not control the underlying operating system. Here, it becomes handy that Snakemake >=4.8.0 allows to combine Conda-based package management with *Running jobs in containers*. For example, you can write

```
container: "docker://continuumio/miniconda3:4.4.10"

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

in other words, a global definition of a container image can be combined with a per-rule conda directive. Then, upon invocation with

```
snakemake --use-conda --use-singularity
```

Snakemake will first pull the defined container image, and then create the requested conda environment from within the container. The conda environments will still be stored in your working environment, such that they don't have to be recreated unless they have changed. The hash under which the environments are stored includes the used container image url, such that changes to the container image also lead to new environments to be created. When a job is executed, Snakemake will first enter the container and then activate the conda environment.

By this, both packages and OS can be easily controlled without the overhead of creating and distributing specialized container images. Of course, it is also possible (though less common) to define a container image per rule in this scenario.

The user can, upon execution, freely choose the desired level of reproducibility:

- no package management (use whatever is on the system)
- Conda based package management (use versions defined by the workflow developer)
- Conda based package management in containerized OS (use versions and OS defined by the workflow developer)

4.19.8 Using environment modules

In high performace cluster systems (HPC), it can be preferable to use environment modules for deployment of optimized versions of certain standard tools. Snakemake allows to define environment modules per rule:

```
rule bwa:
    input:
        "genome.fa"
        "reads.fq"
    output:
        "mapped.bam"
    conda:
```

(continues on next page)

(continued from previous page)

```
        "envs/bwa.yaml"
    envmodules:
        "bio/bwa/0.7.9",
        "bio/samtools/1.9"
    shell:
        "bwa mem {input} | samtools view -Sbh - > {output}"
```

Here, when Snakemake is executed with `snakemake --use-envmodules`, it will load the defined modules in the given order, instead of using the also defined conda environment. Note that although not mandatory, one should always provide either a conda environment or a container (see above), along with environment module definitions. The reason is that environment modules are often highly platform specific, and cannot be assumed to be available somewhere else, thereby limiting reproducibility. By definition an equivalent conda environment or container as a fallback, people outside of the HPC system where the workflow has been designed can still execute it, e.g. by running `snakemake --use-conda` instead of `snakemake --use-envmodules`.

4.19.9 Sustainable and reproducible archiving

With Snakemake 3.10.0 it is possible to archive a workflow into a tarball (*.tar*, *.tar.gz*, *.tar.bz2*, *.tar.xz*), via

```
snakemake --archive my-workflow.tar.gz
```

If above layout is followed, this will archive any code and config files that is under git version control. Further, all input files will be included into the archive. Finally, the software packages of each defined conda environment are included. This results in a self-contained workflow archive that can be re-executed on a vanilla machine that only has Conda and Snakemake installed via

```
tar -xf my-workflow.tar.gz
snakemake -n
```

Note that the archive is platform specific. For example, if created on Linux, it will run on any Linux newer than the minimum version that has been supported by the used Conda packages at the time of archiving (e.g. CentOS 6).

A useful pattern when publishing data analyses is to create such an archive, upload it to Zenodo and thereby obtain a DOI. Then, the DOI can be cited in manuscripts, and readers are able to download and reproduce the data analysis at any time in the future.

4.20 Reports

From Snakemake 5.1 on, it is possible to automatically generate detailed self-contained HTML reports that encompass runtime statistics, provenance information, workflow topology and results. **As an example, the report of the Snakemake rolling paper can be found** here.

For including results into the report, the Snakefile has to be annotated with additional information. Each output file that shall be part of the report has to be marked with the `report` flag, which optionally points to a caption in restructured text format and allows to define a `category` for grouping purposes. Moreover, a global workflow description can be defined via the `report` directive. Consider the following example:

```
report: "report/workflow.rst"


rule all:
    input:
```

(continues on next page)

(continued from previous page)

```
        ["fig1.svg", "fig2.png", "testdir"]


rule c:
    output:
        "test.{i}.out"
    singularity:
        "docker://continuumio/miniconda3:4.4.10"
    conda:
        "envs/test.yaml"
    shell:
        "sleep `shuf -i 1-3 -n 1`; touch {output}"


rule a:
    input:
        expand("test.{i}.out", i=range(10))
    output:
        report("fig1.svg", caption="report/fig1.rst", category="Step 1")
    shell:
        "sleep `shuf -i 1-3 -n 1`; cp data/fig1.svg {output}"


rule b:
    input:
        expand("{model}.{i}.out", i=range(10))
    output:
        report("fig2.png", caption="report/fig2.rst", category="Step 2", subcategory="
↪{model}")
    shell:
        "sleep `shuf -i 1-3 -n 1`; cp data/fig2.png {output}"

rule d:
    output:
        report(
            directory("testdir"),
            patterns=["{name}.txt"],
            caption="report/somedata.rst",
            category="Step 3")
    shell:
        "mkdir {output}; for i in 1 2 3; do echo $i > {output}/$i.txt; done"
```

As can be seen, we define a global description which is contained in the file `report/workflow.rst`. In addition, we mark `fig1.svg` and `fig2.png` for inclusion into the report, while in both cases specifying a caption text via again referring to a restructured text file. Note the paths to the `.rst`-files are interpreted relative to the current Snakefile.

Inside the `.rst`-files you can use Jinja2 templating to access context information. In case of the global description, you can access the config dictionary via `{{ snakemake.config }}`, (e.g., use `{{ snakemake.config["mykey"] }}` to access the key `mykey`). In case of output files, you can access the same values as available with the *script directive* (e.g., `snakemake.wildcards`).

When marking files for inclusion in the report, a `category` and a `subcategory` can be given, allowing to group results in of the report. For both, wildcards (like `{model}` see rule b in the example), are automatically replaced with the respective values from the corresponding job.

The last rule `d` creates a directory with several files, here mimicing the case that it is impossible to specify exactly which files will be created while writing the workflow (e.g. it might depend on the data). Nevertheless, it is still possible to include those files one by one into the report by defining inclusion patterns (here `patterns=["{name}.txt"]`) along with

the report flag. When creating the report, Snakemake will scan the directory for files matching the given patterns and include all of them in the report. Wildcards in those patterns are made available in the jinja-templated caption document along with the rules wildcards in the `snakemake.wildcards` object.

If the output of a rule is a directory with an HTML file hierarchy, it is also possible to specify an entry-point HTML file for inclusion into the report, instead of the `patterns` approach from above. This works as follows:

```
rule generate_html_hierarchy:
    output:
        report(directory("test"), caption="report/caption.rst", htmlindex="test.html")
    shell:
        """
        # mimic writing of an HTML hierarchy
        mkdir test
        cp template.html test/test.html
        mkdir test/js
        echo \"alert('test')\" > test/js/test.js
        """
```

4.20.1 Defining file labels

In addition to category, and subcategory, it is possible to define a dictionary of labels for each report item. By that, the actual filename will be hidden in the report and instead a table with the label keys as columns and the values in the respective row for the file will be displayed. This can lead to less technical reports that abstract away the fact that the results of the analysis are actually files. Consider the following modification of rule `b` from above:

```
rule b:
  input:
      expand("{model}.{i}.out", i=range(10))
  output:
      report(
          "fig2.png",
          caption="report/fig2.rst",
          category="Step 2",
          subcategory="{model}",
          labels={
              "model": "{model}",
              "figure": "some plot"
          }
      )
  shell:
      "sleep `shuf -i 1-3 -n 1`; cp data/fig2.png {output}"
```

4.20.2 Determining category, subcategory, and labels dynamically via functions

Similar to e.g. with input file and parameter definition (see *Input functions*), `category` and a `subcategory` and `labels` can be specified by pointing to a function that takes `wildcards` as the first argument (and optionally in addition `input`, `output`, `params` in any order). The function is expected to return a string or number (int, float, numpy types), or, in case of labels, a dict with strings as keys and strings or numbers as values.

4.20.3 Linking between items

In every `.rst` document, you can link to

- the **Workflow** panel (with `Rules_`),
- the **Statistics** panel (with `Statistics_`),
- any **category** panel (with `Mycategory_`, while `Mycategory` is the name given for the category argument of the report flag). E.g., with above example, you could write `see `Step 2`_` in order to link to the section with the results that have been assigned to the category `Step 2`.
- any **file** marked with the report flag (with `myfile.txt_`, while `myfile.txt` is the basename of the file, without any leading directories). E.g., with above example, you could write `see fig2.png_` in order to link to the result in the report document.

For details about the hyperlink mechanism of restructured text see here.

4.20.4 Rendering reports

To create the report simply run

```
snakemake --report report.html
```

after your workflow has finished. All other information contained in the report (e.g. runtime statistics) is automatically collected during creation. These statistics are obtained from the metadata that is stored in the `.snakemake` directory inside your working directory.

You can define an institute specific stylesheet with:

```
snakemake --report report.html --report-stylesheet custom-stylesheet.css
```

In particular, this allows you to e.g. set a logo at the top (by using CSS to inject a background for the placeholder `<div id="brand">`, or overwrite colors. For an example custom stylesheet defining the logo, see `here`. The report for above example can be found `here` (with a custom branding for the University of Duisburg-Essen). The full example source code can be found here.

Note that the report can be restricted to particular jobs and results by specifying targets at the command line, analog to normal Snakemake execution. For example, with

```
snakemake fig1.svg --report report-short.html
```

the report contains only `fig1.svg`.

4.21 Automatically generating unit tests

Snakemake can automatically generate unit tests from a workflow that has already been successfully executed. By running

```
snakemake --generate-unit-tests
```

Snakemake is instructed to take one representative job for each rule and copy its input files to a hidden folder `.tests/unit`, along with generating test cases for Pytest.

Importantly, note that such unit tests shall not be generated from big data, as they should usually be finished in a few seconds. Further, it makes sense to store the generated unit tests in version control (e.g. git), such that huge files are not recommended. Instead, we suggest to first execute the workflow that shall be tested with some kind of small dummy

datasets, and then use the results thereof to generate the unit tests. The small dummy datasets can in addition be used to generate an integration test, that could e.g. be stored under `.tests/integration`, next to the unit tests.

Each auto-generated unit test is stored in a file `.tests/unit/test_<rulename>.py`, and executes just the one representative job of the respective rule. After successfull execution of the job, it will compare the obtained results with those that have been present when running `snakemake --generate-unit-tests`. By default, the comparison happens byte by byte (using `cmp`). This behavior can be overwritten by modifying the test file.

4.22 Integrating foreign workflow management systems

Snakemake 6.2 and later allows to hand over execution steps to other workflow management systems. By this, it is possible to make use of workflows written for other systems, while performing any further pre- or postprocessing within Snakemake. Such a handover is indicated with the `handover` directive. Consider the following example:

```
rule chipseq_pipeline:
    input:
        input="design.csv",
        fasta="data/genome.fasta",
        gtf="data/genome.gtf",
    output:
        "multiqc/broadPeaks/multiqc_report.html",
    params:
        pipeline="nf-core/chipseq",
        revision="1.2.1",
        profile=["conda"],
    handover: True
    wrapper:
        "0.74.0/utils/nextflow"
```

Here, the workflow is executed as usual until this rule is reached. Then, Snakemake passes all resources to the nextflow workflow management system, which generates certain files. The rule is executed as a local rule, meaning that it would not be submitted to a cluster or cloud system by Snakemake. Instead, the invoked other workflow management system is responsible for that. E.g., in case of Nextflow, submission behavior can be configured via a `nextflow.conf` file or environment variables. After the step is done, Snakemake continues execution with the output files produced by the foreign workflow.

4.23 Citing and Citations

This section gives instructions on how to cite Snakemake and lists citing articles.

4.23.1 Citing Snakemake

When using Snakemake for a publication, **please cite the following article** in you paper:

Mölder, F., Jablonski, K.P., Letcher, B., Hall, M.B., Tomkins-Tinch, C.H., Sochat, V., Forster, J., Lee, S., Twardziok, S.O., Kanitz, A., Wilm, A., Holtgrewe, M., Rahmann, S., Nahnsen, S., Köster, J., 2021. Sustainable data analysis with Snakemake. F1000Res 10, 33.

This "rolling" paper will be regularly updated when Snakemake receives new features.

More References

The initial Snakemake publication was:

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

Another publication describing more of Snakemake internals:

Köster, Johannes and Rahmann, Sven. "Building and Documenting Bioinformatics Workflows with Python-based Snakemake". Proceedings of the GCB 2012.

And my PhD thesis which describes all algorithmic details as of 2015:

Johannes Köster, "Parallelization, Scalability, and Reproducibility in Next-Generation Sequencing Analysis", TU Dortmund 2014

The most comprehensive publication is our "rolling" paper (see above):

Mölder, F., Jablonski, K.P., Letcher, B., Hall, M.B., Tomkins-Tinch, C.H., Sochat, V., Forster, J., Lee, S., Twardziok, S.O., Kanitz, A., Wilm, A., Holtgrewe, M., Rahmann, S., Nahnsen, S., Köster, J., 2021. Sustainable data analysis with Snakemake. F1000Res 10, 33.

Project Pages

If you publish a Snakemake workflow, consider to add this badge to your project page:

The markdown syntax is

```
[![Snakemake](https://img.shields.io/badge/snakemake-≥5.6.0-brightgreen.svg?
↪style=flat)](https://snakemake.readthedocs.io)
```

Replace the `5.6.0` with the minimum required Snakemake version. You can also change the style.

4.24 More Resources

4.24.1 Talks and Posters

- Poster at ECCB 2016, The Hague, Netherlands.
- Invited talk by Johannes Köster at the Broad Institute, Boston 2015.
- Introduction to Snakemake. Tutorial Slides presented by Johannes Köster at the GCB 2015, Dortmund, Germany.
- Invited talk by Johannes Köster at the DTL Focus Meeting: "NGS Production Pipelines", Dutch Techcentre for Life Sciences, Utrecht 2014.
- Taming Snakemake by Jeremy Leipzig, Bioinformatics software developer at Children's Hospital of Philadelphia, 2014.
- "Snakemake makes … snakes?" - An Introduction by Marcel Martin from SciLifeLab, Stockholm 2015
- "Workflow Management with Snakemake" by Johannes Köster, 2015. Held at the Department of Biostatistics and Computational Biology, Dana-Farber Cancer Institute

4.24.2 External Resources

These resources are not part of the official documentation.

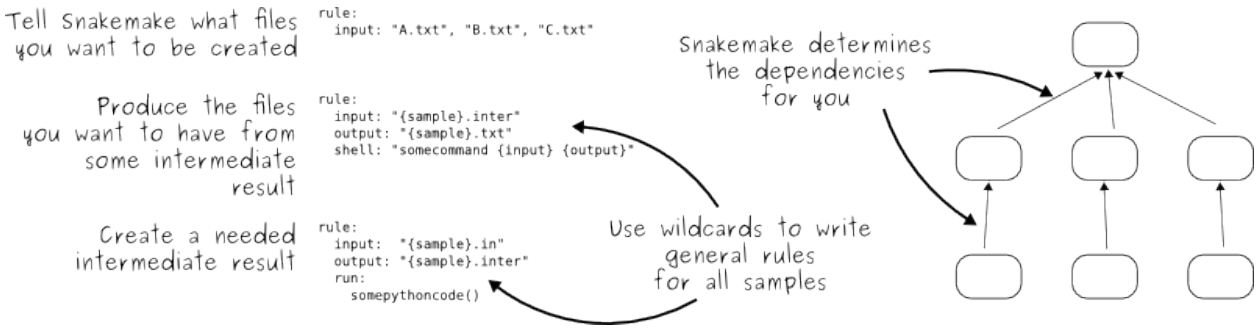
- A number of tutorials on the subject "Tools for reproducible research"
- Snakemake workflow used for the Kallisto paper
- An alternative tutorial for Snakemake
- An Emacs mode for Snakemake
- Flexible bioinformatics pipelines with Snakemake
- Sandwiches with Snakemake
- A visualization of the past years of Snakemake development
- Japanese version of the Snakemake tutorial
- Basic and advanced french Snakemake tutorial.
- Mini tutorial on Snakemake and Bioconda
- Snakeparse: a utility to expose Snakemake workflow configuation via a command line interface

4.25 Frequently Asked Questions

Contents

4.25.1 What is the key idea of Snakemake workflows?

The key idea is very similar to GNU Make. The workflow is determined automatically from top (the files you want) to bottom (the files you have), by applying very general rules with wildcards you give to Snakemake:



When you start using Snakemake, please make sure to walk through the *official tutorial*. It is crucial to understand how to properly use the system.

4.25.2 How does Snakemake interpret relative paths?

Relative paths in Snakemake are interpreted depending on their context.

- Input, output, log, and benchmark files are considered to be relative to the working directory (either the directory in which you have invoked Snakemake or whatever was specified for `--directory` or the `workdir:` directive).
- Any other directives (e.g. `conda:`, `include:`, `script:`, `notebook:`) consider paths to be relative to the Snakefile they are defined in.

If you have to manually specify a file that has to be relative to the currently evaluated Snakefile, you can use `workflow.source_path(filepath)`.

```
rule read_a_file_relative_to_snakefile:
    input:
        workflow.source_path("resources/some-file.txt")
    output:
        "results/some-output.txt"
    shell:
        "somecommand {input} {output}"
```

This will in particular also work in combination with *modules*.

4.25.3 Snakemake does not connect my rules as I have expected, what can I do to debug my dependency structure?

Since dependencies are inferred implicitly, results can sometimes be suprising when little errors are made in filenames or when input functions raise unexpected errors. For debugging such cases, Snakemake provides the command line flag `--debug-dag` that leads to printing details each decision that is taken while determining the dependencies.

In addition, it is advisable to check whether certain intermediate files would be created by targetting them individually via the command line.

Finally, it is possible to constrain the rules that are considered for DAG creating via `--allowed-rules`. This way, you can easily check rule by rule if it does what you expect. However, note that `--allowed-rules` is only meant for debugging. A workflow should always work fine without it.

4.25.4 My shell command fails with with errors about an "unbound variable", what's wrong?

This happens often when calling virtual environments from within Snakemake. Snakemake is using bash strict mode, to ensure e.g. proper error behavior of shell scripts. Unfortunately, virtualenv and some other tools violate bash strict mode. The quick fix for virtualenv is to temporarily deactivate the check for unbound variables

```
set +u; source /path/to/venv/bin/activate; set -u
```

For more details on bash strict mode, see the here.

4.25.5 My shell command fails with exit code != 0 from within a pipe, what's wrong?

Snakemake is using bash strict mode to ensure best practice error reporting in shell commands. This entails the pipefail option, which reports errors from within a pipe to outside. If you don't want this, e.g., to handle empty output in the pipe, you can disable pipefail via prepending

```
set +o pipefail;
```

to your shell command in the problematic rule.

4.25.6 I don't want Snakemake to detect an error if my shell command exits with an exitcode > 1. What can I do?

Sometimes, tools encode information in exit codes bigger than 1. Snakemake by default treats anything > 0 as an error. Special cases have to be added by yourself. For example, you can write

```
shell:
    """
    set +e
    somecommand ...
    exitcode=$?
    if [ $exitcode -eq 1 ]
    then
        exit 1
    else
        exit 0
    fi
    """
```

This way, Snakemake only treats exit code 1 as an error, and thinks that everything else is fine. Note that such tools are an excellent use case for contributing a wrapper.

4.25.7 How do I run my rule on all files of a certain directory?

In Snakemake, similar to GNU Make, the workflow is determined from the top, i.e. from the target files. Imagine you have a directory with files `1.fastq, 2.fastq, 3.fastq, ...`, and you want to produce files `1.bam, 2.bam, 3.bam, ...` you should specify these as target files, using the ids `1,2,3,...`. You could end up with at least two rules like this (or any number of intermediate steps):

```
IDS = "1 2 3 ...".split() # the list of desired ids

# a pseudo-rule that collects the target files
rule all:
    input:  expand("otherdir/{id}.bam", id=IDS)

# a general rule using wildcards that does the work
rule:
    input:  "thedir/{id}.fastq"
    output: "otherdir/{id}.bam"
    shell:  "..."
```

Snakemake will then go down the line and determine which files it needs from your initial directory.

In order to infer the IDs from present files, Snakemake provides the `glob_wildcards` function, e.g.

```
IDS, = glob_wildcards("thedir/{id}.fastq")
```

The function matches the given pattern against the files present in the filesystem and thereby infers the values for all wildcards in the pattern. A named tuple that contains a list of values for each wildcard is returned. Here, this named tuple has only one item, that is the list of values for the wildcard `{id}`.

4.25.8 I don't want expand to use the product of every wildcard, what can I do?

By default the expand function uses `itertools.product` to create every combination of the supplied wildcards. Expand takes an optional, second positional argument which can customize how wildcards are combined. To create the list `["a_1.txt", "b_2.txt", "c_3.txt"]`, invoke expand as: `expand("{sample}_{id}.txt", zip, sample=["a", "b", "c"], id=["1", "2", "3"])`

4.25.9 I don't want expand to use every wildcard, what can I do?

Sometimes partially expanding wildcards is useful to define inputs which still depend on some wildcards. Expand takes an optional keyword argument, allow_missing=True, that will format only wildcards which are supplied, leaving others as is. To create the list `["{sample}_1.txt", "{sample}_2.txt"]`, invoke expand as: `expand("{sample}_{id}.txt", id=["1", "2"], allow_missing=True)` If the filename contains the wildcard `allow_missing`, it will be formatted normally: `expand("{allow_missing}.txt", allow_missing=True)` returns `["True.txt"]`.

4.25.10 Snakemake complains about a cyclic dependency or a PeriodicWildcardError. What can I do?

One limitation of Snakemake is that graphs of jobs have to be acyclic (similar to GNU Make). This means, that no path in the graph may be a cycle. Although you might have considered this when designing your workflow, Snakemake sometimes runs into situations where a cyclic dependency cannot be avoided without further information, although the solution seems obvious for the developer. Consider the following example:

```
rule all:
    input:
        "a"

rule unzip:
    input:
        "{sample}.tar.gz"
    output:
        "{sample}"
    shell:
        "tar -xf {input}"
```

If this workflow is executed with

```
snakemake -n
```

two things may happen.

1. If the file `a.tar.gz` is present in the filesystem, Snakemake will propose the following (expected and correct) plan:

```
rule a:
        input: a.tar.gz
    output: a
    wildcards: sample=a
localrule all:
        input: a
Job counts:
        count   jobs
        1       a
        1       all
        2
```

2. If the file `a.tar.gz` is not present and cannot be created by any other rule than rule `a`, Snakemake will try to run rule `a` again, with `{sample}=a.tar.gz`. This would infinitely go on recursively. Snakemake detects this case and produces a `PeriodicWildcardError`.

In summary, `PeriodicWildcardErrors` hint to a problem where a rule or a set of rules can be applied to create its own input. If you are lucky, Snakemake can be smart and avoid the error by stopping the recursion if a file exists in the filesystem. Importantly, however, bugs upstream of that rule can manifest as `PeriodicWildcardError`, although in reality just a file is missing or named differently. In such cases, it is best to restrict the wildcard of the output file(s), or follow the general rule of putting output files of different rules into unique subfolders of your working directory. This way, you can discover the true source of your error.

4.25.11 Is it possible to pass variable values to the workflow via the command line?

Yes, this is possible. Have a look at *Configuration*. Previously it was necessary to use environment variables like so: E.g. write

```
$ SAMPLES="1 2 3 4 5" snakemake
```

and have in the Snakefile some Python code that reads this environment variable, i.e.

```
SAMPLES = os.environ.get("SAMPLES", "10 20").split()
```

4.25.12 I get a NameError with my shell command. Are braces unsupported?

You can use the entire Python format minilanguage in shell commands. Braces in shell commands that are not intended to insert variable values thus have to be escaped by doubling them:

This:

```
...
shell: "awk '{print $1}' {input}"
```

becomes:

```
...
shell: "awk '{{print $1}}' {input}"
```

Here the double braces are escapes, i.e. there will remain single braces in the final command. In contrast, `{input}` is replaced with an input filename.

In addition, if your shell command has literal backslashes, `\\`, you must escape them with a backslash, `\\\\`. For example:

This:

```
shell: """printf \">%s\"" {{input}}"""
```

becomes:

```
shell: """printf \\">%s\\"" {{input}}"""
```

4.25.13 How do I incorporate files that do not follow a consistent naming scheme?

The best solution is to have a dictionary that translates a sample id to the inconsistently named files and use a function (see *Input functions*) to provide an input file like this:

```
FILENAME = dict(...)  # map sample ids to the irregular filenames here

rule:
    # use a function as input to delegate to the correct filename
    input: lambda wildcards: FILENAME[wildcards.sample]
    output: "somefolder/{sample}.csv"
    shell: ...
```

4.25.14 How do I force Snakemake to rerun all jobs from the rule I just edited?

This can be done by invoking Snakemake with the `--forcerun` or `-R` flag, followed by the rules that should be re-executed:

```
$ snakemake -R somerule
```

This will cause Snakemake to re-run all jobs of that rule and everything downstream (i.e. directly or indirectly depending on the rules output).

4.25.15 How should Snakefiles be formatted?

To ensure readability and consistency, you can format Snakefiles with our tool snakefmt.

Python code gets formatted with black and Snakemake-specific blocks are formatted using similar principles (such as PEP8).

4.25.16 How do I enable syntax highlighting in Vim for Snakefiles?

Instructions for doing this are located here.

Note that you can also format Snakefiles in Vim using *snakefmt*, with instructions located here!

4.25.17 I want to import some helper functions from another python file. Is that possible?

Yes, from version 2.4.8 on, Snakemake allows to import python modules (and also simple python files) from the same directory where the Snakefile resides.

4.25.18 How can I run Snakemake on a cluster where its main process is not allowed to run on the head node?

This can be achived by submitting the main Snakemake invocation as a job to the cluster. If it is not allowed to submit a job from a non-head cluster node, you can provide a submit command that goes back to the head node before submitting:

```
qsub -N PIPE -cwd -j yes python snakemake --cluster "ssh user@headnode_address 'qsub -
↪N pipe_task -j yes -cwd -S /bin/sh ' " -j
```

This hint was provided by Inti Pedroso.

4.25.19 Can the output of a rule be a symlink?

Yes. As of Snakemake 3.8, output files are removed before running a rule and then touched after the rule completes to ensure they are newer than the input. Symlinks are treated just the same as normal files in this regard, and Snakemake ensures that it only modifies the link and not the target when doing this.

Here is an example where you want to merge N files together, but if N == 1 a symlink will do. This is easier than attempting to implement workflow logic that skips the step entirely. Note the **-r** flag, supported by modern versions of ln, is useful to achieve correct linking between files in subdirectories.

```
rule merge_files:
    output: "{foo}/all_merged.txt"
    input: my_input_func  # some function that yields 1 or more files to merge
    run:
        if len(input) > 1:
            shell("cat {input} | sort > {output}")
        else:
            shell("ln -sr {input} {output}")
```

Do be careful with symlinks in combination with *Step 6: Temporary and protected files*. When the original file is deleted, this can cause various errors once the symlink does not point to a valid file any more.

If you get a message like `Unable to set utime on symlink .... Your Python build does not support it.` this means that Snakemake is unable to properly adjust the modification time of the symlink. In this case, a workaround is to add the shell command *touch -h {output}* to the end of the rule.

4.25.20 Can the input of a rule be a symlink?

Yes. In this case, since Snakemake 3.8, one extra consideration is applied. If *either* the link itself or the target of the link is newer than the output files for the rule then it will trigger the rule to be re-run.

4.25.21 I would like to receive a mail upon snakemake exit. How can this be achieved?

On unix, you can make use of the commonly pre-installed *mail* command:

```
snakemake 2> snakemake.log
mail -s "snakemake finished" youremail@provider.com < snakemake.log
```

In case your administrator does not provide you with a proper configuration of the sendmail framework, you can configure *mail* to work e.g. via Gmail (see here).

4.25.22 I want to pass variables between rules. Is that possible?

Because of the cluster support and the ability to resume a workflow where you stopped last time, Snakemake in general should be used in a way that information is stored in the output files of your jobs. Sometimes it might though be handy to have a kind of persistent storage for simple values between jobs and rules. Using plain python objects like a global dict for this will not work as each job is run in a separate process by snakemake. What helps here is the *PersistentDict* from the pytools package. Here is an example of a Snakemake workflow using this facility:

```
from pytools.persistent_dict import PersistentDict

storage = PersistentDict("mystorage")

rule a:
    input: "test.in"
    output: "test.out"
    run:
        myvar = storage.fetch("myvar")
        # do stuff

rule b:
    output: temp("test.in")
    run:
        storage.store("myvar", 3.14)
```

Here, the output rule b has to be temp in order to ensure that `myvar` is stored in each run of the workflow as rule a relies on it. In other words, the *PersistentDict* is persistent between the job processes, but not between different runs of this workflow. If you need to conserve information between different runs, use output files for them.

4.25.23 Why do my global variables behave strangely when I run my job on a cluster?

This is closely related to the question above. Any Python code you put outside of a rule definition is normally run once before Snakemake starts to process rules, but on a cluster it is re-run again for each submitted job, because Snakemake implements jobs by re-running itself.

Consider the following…

```
from mydatabase import get_connection

dbh = get_connection()
latest_parameters = dbh.get_params().latest()

rule a:
```

(continues on next page)

(continued from previous page)

```
    input: "{foo}.in"
    output: "{foo}.out"
    shell: "do_op -params {latest_parameters}  {input} {output}"
```

When run a single machine, you will see a single connection to your database and get a single value for *latest_parameters* for the duration of the run. On a cluster you will see a connection attempt from the cluster node for each job submitted, regardless of whether it happens to involve rule a or not, and the parameters will be recalculated for each job.

4.25.24 I want to configure the behavior of my shell for all rules. How can that be achieved with Snakemake?

You can set a prefix that will prepended to all shell commands by adding e.g.

```
shell.prefix("set -o pipefail; ")
```

to the top of your Snakefile. Make sure that the prefix ends with a semicolon, such that it will not interfere with the subsequent commands. To simulate a bash login shell, you can do the following:

```
shell.executable("/bin/bash")
shell.prefix("source ~/.bashrc; ")
```

4.25.25 Some command line arguments like –config cannot be followed by rule or file targets. Is that intended behavior?

This is a limitation of the argparse module, which cannot distinguish between the perhaps next arg of `--config` and a target. As a solution, you can put the *–config* at the end of your invocation, or prepend the target with a single `--`, i.e.

```
$ snakemake --config foo=bar -- mytarget
$ snakemake mytarget --config foo=bar
```

4.25.26 How do I enforce config values given at the command line to be interpreted as strings?

When passing config values like this

```
$ snakemake --config version=2018_1
```

Snakemake will first try to interpret the given value as number. Only if that fails, it will interpret the value as string. Here, it does not fail, because the underscore _ is interpreted as thousand separator. In order to ensure that the value is interpreted as string, you have to pass it in quotes. Since bash otherwise automatically removes quotes, you have to also wrap the entire entry into quotes, e.g.:

```
$ snakemake --config 'version="2018_1"'
```

4.25.27 How do I make my rule fail if an output file is empty?

Snakemake expects shell commands to behave properly, meaning that failures should cause an exit status other than zero. If a command does not exit with a status other than zero, Snakemake assumes everything worked fine, even if output files are empty. This is because empty output files are also a reasonable tool to indicate progress where no real output was produced. However, sometimes you will have to deal with tools that do not properly report their failure with an exit status. Here, the recommended way is to use bash to check for non-empty output files, e.g.:

```
rule:
    input:  ...
    output: "my/output/file.txt"
    shell:  "somecommand {input} {output} && [[ -s {output} ]]"
```

4.25.28 How does Snakemake lock the working directory?

Per default, Snakemake will lock a working directory by output and input files. Two Snakemake instances that want to create the same output file are not possible. Two instances creating disjoint sets of output files are possible. With the command line option `--nolock`, you can disable this mechanism on your own risk. With `--unlock`, you can be remove a stale lock. Stale locks can appear if your machine is powered off with a running Snakemake instance.

4.25.29 How do I trigger re-runs for rules with updated code or parameters?

Similar to the solution above, you can use

```
$ snakemake -n -R `snakemake --list-params-changes`
```

and

```
$ snakemake -n -R `snakemake --list-code-changes`
```

Again, the list commands in backticks return the list of output files with changes, which are fed into `-R` to trigger a re-run.

4.25.30 How do I remove all files created by snakemake, i.e. like `make clean`

To remove all files created by snakemake as output files to start from scratch, you can use

```
$ snakemake some_target --delete-all-output
```

Only files that are output of snakemake rules will be removed, not those that serve as primary inputs to the workflow. Note that this will only affect the files involved in reaching the specified target(s). It is strongly advised to first run together with `--dry-run` to list the files that would be removed without actually deleting anything. The flag `--delete-temp-output` can be used in a similar manner to only delete files flagged as temporary.

4.25.31 Why can't I use the conda directive with a run block?

The run block of a rule (see *Snakefiles and Rules*) has access to anything defined in the Snakefile, outside of the rule. Hence, it has to share the conda environment with the main Snakemake process. To avoid confusion we therefore disallow the conda directive together with the run block. It is recommended to use the script directive instead (see *External scripts*).

4.25.32 My workflow is very large, how do I stop Snakemake from printing all this rule/job information in a dry-run?

Indeed, the information for each individual job can slow down a dry-run if there are tens of thousands of jobs. If you are just interested in the final summary, you can use the `--quiet` flag to suppress this.

```
$ snakemake -n --quiet
```

4.25.33 Git is messing up the modification times of my input files, what can I do?

When you checkout a git repository, the modification times of updated files are set to the time of the checkout. If you rely on these files as input **and** output files in your workflow, this can cause trouble. For example, Snakemake could think that a certain (git-tracked) output has to be re-executed, just because its input has been checked out a bit later. In such cases, it is advisable to set the file modification dates to the last commit date after an update has been pulled. One solution is to add the following lines to your `.bashrc` (or similar):

```
gitmtim(){
    local f
    for f; do
        touch -d @0`git log --pretty=%at -n1 -- "$f"` "$f"
    done
}
gitmodtimes(){
    for f in $(git ls-tree -r $(git rev-parse --abbrev-ref HEAD) --name-only); do
        gitmtim $f
    done
}
```

(inspired by the answer here). You can then run `gitmodtimes` to update the modification times of all tracked files on the current branch to their last commit time in git; BE CAREFUL–this does not account for local changes that have not been commited.

4.25.34 How do I exit a running Snakemake workflow?

There are two ways to exit a currently running workflow.

1. If you want to kill all running jobs, hit Ctrl+C. Note that when using `--cluster`, this will only cancel the main Snakemake process.
2. If you want to stop the scheduling of new jobs and wait for all running jobs to be finished, you can send a TERM signal, e.g., via

   ```
   killall -TERM snakemake
   ```

4.25.35 How can I make use of node-local storage when running cluster jobs?

When running jobs on a cluster you might want to make use of a node-local scratch directory in order to reduce cluster network traffic and/or get more efficient disk storage for temporary files. There is currently no way of doing this in Snakemake, but a possible workaround involves the `shadow` directive and setting the `--shadow-prefix` flag to e.g. `/scratch`.

```
rule:
    output:
        "some_summary_statistics.txt"
    shadow: "minimal"
    shell:
        """
        generate huge_file.csv
        summarize huge_file.csv > {output}
        """
```

The following would then lead to the job being executed in `/scratch/shadow/some_unique_hash/`, and the temporary file `huge_file.csv` could be kept at the compute node.

```
$ snakemake --shadow-prefix /scratch some_summary_statistics.txt --cluster ...
```

If you want the input files of your rule to be copied to the node-local scratch directory instead of just using symbolic links, you can use `copy-minimal` in the `shadow` directive. This is useful for example for benchmarking tools as a black-box.

```
rule:
    input:
        "input_file.txt"
    output:
        file = "output_file.txt",
        benchmark = "benchmark_results.txt",
    shadow: "copy-minimal"
    shell:
        """
        /usr/bin/time -v command "{input}" "{output.file}" > "{output.benchmark}"
        """
```

Executing snakemake as above then leads to the shell script accessing only node-local storage.

4.25.36 How do I access elements of input or output by a variable index?

Assuming you have something like the following rule

```
rule a:
    output:
        expand("test.{i}.out", i=range(20))
    run:
        for i in range(20):
            shell("echo test > {output[i]}")
```

Snakemake will fail upon execution with the error `'OutputFiles' object has no attribute 'i'`. The reason is that the shell command is using the Python format mini language, which only allows indexing via constants, e.g., `output[1]`, but not via variables. Variables are treated as attribute names instead. The solution is to write

```
rule a:
    output:
        expand("test.{i}.out", i=range(20))
    run:
        for i in range(20):
            f = output[i]
            shell("echo test > {f}")
```

or, more concise in this special case:

```
rule a:
    output:
        expand("test.{i}.out", i=range(20))
    run:
        for f in output:
            shell("echo test > {f}")
```

4.25.37 There is a compiler error when installing Snakemake with pip or easy_install, what shall I do?

Snakemake itself is plain Python, hence the compiler error must come from one of the dependencies, like e.g., datrie. You should have a look if maybe you are missing some library or a certain compiler package. If everything seems fine, please report to the upstream developers of the failing dependency.

Note that in general it is recommended to install Snakemake via Conda which gives you precompiled packages and the additional benefit of having *automatic software deployment* integrated into your workflow execution.

4.25.38 How to enable autocompletion for the zsh shell?

For users of the Z shell (zsh), just run the following (assuming an activated zsh) to activate autocompletion for snakemake:

```
compdef _gnu_generic snakemake
```

Example: Say you have forgotten how to use the various options starting `force`, just type the partial match i.e. `--force` which results in a list of all potential hits along with a description:

```
$snakemake --force**pressing tab**

--force              -- Force the execution of the selected target or the
--force-use-threads  -- Force threads rather than processes. Helpful if shared
--forceall           -- Force the execution of the selected (or the first)
--forcerun           -- (TARGET (TARGET ...)), -R (TARGET (TARGET ...))
```

To activate this autocompletion permanently, put this line in `~/.zshrc`.

Here is some further reading.

4.25.39 How can I avoid system /tmp to be used when combining singularity and conda?

When using both singularity and conda the idea is that inside the singularity container the conda environment is being installed. Some singularity instances are set to share the system /tmp with the containers. This can lead to unexpected behaviour where the system /tmp gets full. To stop this behaviour you'd have to run singularity with the `--contain` option.

4.26 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.26.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/snakemake/snakemake/issues

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the Github issues for bugs. If you want to start working on a bug then please write short message on the issue tracker to prevent duplicate work.

Implement Features

Look through the Github issues for features. If you want to start working on an issue then please write short message on the issue tracker to prevent duplicate work.

Contributing a new cluster or cloud execution backend

Execution backends are added by implementing a so-called `Executor`. All executors are located in snakemake/executors/. In order to implement a new executor, you have to inherit from the class `ClusterExecutor`. Below you find a skeleton

```
class SkeletonExecutor(ClusterExecutor):
    def __init__(self, workflow, dag, cores,
                 jobname="snakejob.{name}.{jobid}.sh",
                 printreason=False,
                 quiet=False,
                 printshellcmds=False,
                 latency_wait=3,
```

(continues on next page)

(continued from previous page)

```
            cluster_config=None,
            local_input=None,
            restart_times=None,
            exec_job=None,
            assume_shared_fs=True,
            max_status_checks_per_second=1):

        super().__init__(workflow, dag, None,
                         jobname=jobname,
                         printreason=printreason,
                         quiet=quiet,
                         printshellcmds=printshellcmds,
                         latency_wait=latency_wait,
                         cluster_config=cluster_config,
                         local_input=local_input,
                         restart_times=restart_times,
                         assume_shared_fs=False, # if your executor relies on a␣
↪shared file system, set this to True
                         max_status_checks_per_second=max_status_checks_per_second)
↪# set this to a reasonable default

        # add additional attributes

    def shutdown(self):
        # perform additional steps on shutdown if necessary
        super().shutdown()

    def cancel(self):
        for job in self.active_jobs:
            # cancel active jobs here
            pass
        self.shutdown()

    def run_jobs(self, jobs, callback=None, submit_callback=None, error_
↪callback=None):
        """Run a list of jobs that is ready at a given point in time.

        By default, this method just runs each job individually.
        This behavior is inherited and therefore this method can be removed from the␣
↪skeleton if the
        default behavior is intended.
        This method can be overwritten to submit many jobs in a more efficient way␣
↪than one-by-one.

        Note that in any case, for each job, the callback functions have to be called␣
↪individually!
        """
        for job in jobs:
            self.run(
                job,
                callback=callback,
                submit_callback=submit_callback,
                error_callback=error_callback,
            )

    def run(self, job,
            callback=None,
```

(continues on next page)

(continued from previous page)

```
            submit_callback=None,
            error_callback=None):
        """Run an individual job or a job group.
        """

        # Necessary: perform additional executor independent steps before running the␣
↪job
        super()._run(job)

        # obtain job execution command
        exec_job = self.format_job(
            self.exec_job, job, _quote_all=True,
            use_threads="--force-use-threads" if not job.is_group() else "")

        # submit job here, and obtain job ids from the backend

        # register job as active, using your own namedtuple.
        # The namedtuple must at least contain the attributes
        # job, jobid, callback, error_callback.
        self.active_jobs.append(MyJob(
            job, jobid, callback, error_callback))

    async def _wait_for_jobs(self):
        from snakemake.executors import sleep
        # busy wait on job completion
        # This is only needed if your backend does not allow to use callbacks
        # for obtaining job status.
        while True:
            # always use self.lock to avoid race conditions
            async with async_lock(self.lock):
                if not self.wait:
                    return
                active_jobs = self.active_jobs
                self.active_jobs = list()
                still_running = list()
            for j in active_jobs:
                # use self.status_rate_limiter to avoid too many API calls.
                async with self.status_rate_limiter:

                    # Retrieve status of job j from your backend via j.jobid
                    # Handle completion and errors, calling either j.callback(j.job)
                    # or j.error_callback(j.job)
                    # In case of error, add job j to still_running.
                    pass
            async with async_lock(self.lock):
                self.active_jobs.extend(still_running)
            await sleep()
```

Write Documentation

Snakemake could always use more documentation, whether as part of the official vcfpy docs, in docstrings, or even on the web in blog posts, articles, and such.

Snakemake uses Sphinx for the user manual (that you are currently reading). See *project_info-doc_guidelines* on how the documentation reStructuredText is used.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/snakemake/snakemake/issues

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.26.2 Pull Request Guidelines

To update the documentation, fix bugs or add new features you need to create a Pull Request . A PR is a change you make to your local copy of the code for us to review and potentially integrate into the code base.

To create a Pull Request you need to do these steps:

1. Create a Github account.
2. Fork the repository.
3. Clone your fork locally.
4. Go to the created snakemake folder with `cd snakemake`.
5. Create a new branch with `git checkout -b <descriptive_branch_name>`.
6. Make your changes to the code or documentation.
7. Run `git add .` to add all the changed files to the commit (to see what files will be added you can run `git add . --dry-run`).
8. To commit the added files use `git commit`. (This will open a command line editor to write a commit message. These should have a descriptive 80 line header, followed by an empty line, and then a description of what you did and why. To use your command line text editor of choice use (for example) `export GIT_EDITOR=vim` before running `git commit`).
9. Now you can push your changes to your Github copy of Snakemake by running `git push origin <descriptive_branch_name>`.
10. If you now go to the webpage for your Github copy of Snakemake you should see a link in the sidebar called "Create Pull Request".
11. Now you need to choose your PR from the menu and click the "Create pull request" button. Be sure to change the pull request target branch to <descriptive_branch_name>!

If you want to create more pull requests, first run `git checkout main` and then start at step 5. with a new branch name.

Feel free to ask questions about this if you want to contribute to Snakemake :)

4.26.3 Testing Guidelines

To ensure that you do not introduce bugs into Snakemake, you should test your code thouroughly.

To have integration tests run automatically when commiting code changes to Github, you need to sign up on wercker.com and register a user.

The easiest way to run your development version of Snakemake is perhaps to go to the folder containing your local copy of Snakemake and call:

```
$ conda env create -f test-environment.yml -n snakemake-testing
$ conda activate snakemake-testing
$ pip install -e .
```

This will make your development version of Snakemake the one called when running snakemake. You do not need to run this command after each time you make code changes.

From the base snakemake folder you call `nosetests` to run all the tests, or choose one specific test. For this to work, Nose (the testing framework we use) can be installed to the conda environment using pip:

```
$ pip install nose
$ nosetests
$ nosetests tests.tests:test_log_input
```

If you introduce a new feature you should add a new test to the tests directory. See the folder for examples.

4.26.4 Documentation Guidelines

For the documentation, please adhere to the following guidelines:

- Put each sentence on its own line, this makes tracking changes through Git SCM easier.
- Provide hyperlink targets, at least for the first two section levels. For this, use the format `<document_part>-<section_name>`, e.g., `project_info-doc_guidelines`.
- Use the section structure from below.

```
.. document_part-heading_1:

=========
Heading 1
=========


.. document_part-heading_2:

---------
Heading 2
---------


.. document_part-heading_3:

Heading 3
=========


.. document_part-heading_4:
```

(continues on next page)

(continued from previous page)

```
Heading 4
---------


.. document_part-heading_5:

Heading 5
~~~~~~~~~


.. document_part-heading_6:

Heading 6
:::::::::
```

4.26.5 Documentation Setup

For building the documentation, you have to install the Sphinx. If you have already installed Conda, all you need to do is to create a Snakemake development environment via

```
$ git clone git@github.com:snakemake/snakemake.git
$ cd snakemake
$ conda env create -f doc-environment.yml -n snakemake
```

Then, the docs can be built with

```
$ conda activate snakemake
$ cd docs
$ make html
$ make clean && make html  # force rebuild
```

Alternatively, you can use virtualenv. The following assumes you have a working Python 3 setup.

```
$ git clone git@github.org:snakemake/snakemake.git
$ cd snakemake/docs
$ virtualenv -p python3 .venv
$ source .venv/bin/activate
$ pip install --upgrade -r requirements.txt
```

Afterwards, the docs can be built with

```
$ source .venv/bin/activate
$ make html  # rebuild for changed files only
$ make clean && make html  # force rebuild
```

4.27 Credits

4.27.1 Development Lead

- Johannes Köster

4.27.2 Development Team

- Christopher Tomkins-Tinch
- David Koppstein
- Tim Booth
- Manuel Holtgrewe
- Christian Arnold
- Wibowo Arindrarto
- Rasmus Ågren
- Soohyun Lee
- Vanessa Sochat

4.27.3 Contributors

In alphabetical order

- Andreas Wilm
- Anthony Underwood
- Ryan Dale
- David Alexander
- Elias Kuthe
- Elmar Pruesse
- Hyeshik Chang
- Jay Hesselberth
- Jesper Foldager
- John Huddleston
- Joona Lehtomäki
- Karel Brinda
- Karl Gutwin
- Kemal Eren
- Kostis Anagnostopoulos
- Kyle A. Beauchamp
- Kyle Meyer

- Lance Parsons
- Manuel Holtgrewe
- Marcel Martin
- Matthew Shirley
- Mattias Franberg
- Matt Shirley
- Paul Moore
- Per Unneberg
- Ryan C. Thompson
- Ryan Dale
- Sean Davis
- Simon Ye
- Tobias Marschall
- Vanessa Sochat
- Willem Ligtenberg

4.28 Changelog

4.28.1 7.22.0 (2023-02-12)

Features

- add cleanup containers option (#2088) (053e3b3)

Bug Fixes

- assume shared filesystem by default when running with –flux (#2075) (4bec2fd)
- properly handle NA values for paramspaces (#2098) (6b6a880)

4.28.2 7.21.0 (2023-01-30)

Features

- ability to encode paramspaces into a single wildcard, via the newly introduced `single_wildcard` argument of `Paramspace`. (#2069) (728ab3c)
- allow input, output, and params to be used in functions passed to report mark arguments (#2081) (93ff8b6)

Bug Fixes

- more robust encoding of params in persistent metadata storage. This way, pandas parameters do not lead to spurious rerun triggers. (#2080) (106a4c3)
- more robust parsing of sacct output in slurm executor (#2036) (fe651f8)
- Postprocess job groups in toposorted order for correct touch times (#2073) (10b5849)

4.28.3 7.20.0 (2023-01-18)

Features

- add tes token (#1966) (59a8fa0)
- Add token auth to GitLab/GitHub hosting providers (#1761) (e03a3b4), closes #1301
- allow for human friendly resource definitions (e.g. mem="5GB", runtime="1d") (#1861) (24610ac)

Bug Fixes

- :bug: - fix hyperlink (#2046) (9519d31)
- Catch missing error stream in Slurm executor (#2063) (c21fc7e)
- correctly parse empty values in config cli (#2032) (1b0689d)
- Correctly parse UserDicts in executors (#2016) (e3926fa)
- Fix handling of –jobs in no-exec state (#2029) (e8e8222)
- make `--show-failed-logs` handle empty log files (#2039) (683c6f2), closes #2023
- make python version check more robust (#2058) (e685621)
- parsing error when last line is comment (#2054) (a928dd4)
- prevent overriding of retries when set to 0 (#2053) (a328f3e)
- propagate attempt count from group to subjobs (#2052) (da3f1c0)
- remove overflow from rulegraph div in report (9a0aaa7)
- skip type checks of missing dir in touch mode (#2051) (ae00c25)
- slurm default_resources quoting (#2043) (47d3fc3)
- Update list of python versions in classifiers (#2020) (7a98100)
- use short argument name for `--chdir` for compatibility with Slurm <=v17 (#2040) (a9ed3ec)

Documentation

- Fix typo in SLURM help text (#2049) (79b7025)
- mention XDG_CACHE_HOME (#2057) (ec2ef45)

4.28.4 7.19.1 (2022-12-13)

Bug Fixes

- improved default resources parsing (also allowing to deactivate a default resource via setting it to None) (#2006) (e6cdb32)

Documentation

- fix link (4889c93)
- fix typo (e1c3cc6)
- fix typos (e45b9e6)
- fix typos (151095d)
- format table (4180a1b)
- polished text and table display (413356c)

4.28.5 7.19.0 (2022-12-13)

Features

- add keyword to gridftp remote provide to specify the number or disable usage of multiple data stream (#1974) (3e6675d)
- provide information about temp, pipe, and service files in –summary (#1977) (c7c7776)
- native SLURM support (–slurm, see docs) (#1015) (c7ea059)

Bug Fixes

- avoid logfile writing in case of dryrun; better hints in case of incomplete checkpoints (#1994) (a022705)
- handle case where zenodo deposition does not return files (#2004) (b63c4a7)
- issue #1882 WorkflowError: Metadata can't be created as it already exists (Windows) (#1971) (d4484e6)
- json validation error with markdown cells (#1986) (6c26f75)

4.28.6 7.18.2 (2022-11-10)

Bug Fixes

- Change ratelimiter dependency to throttler (#1958) (50b8f16)
- fixed problem with leaked modifications when inheriting multiple times from the same rule (#1957) (2475cbc)
- forwarding –keep-incomplete to cluster executor (#1951) (2894c7d)
- show input files on job error (#1949) (ad21631)

4.28.7 7.18.1 (2022-11-03)

Bug Fixes

- regression ValueError introduced with 7.17.2 (#1947) (53a4fca)

4.28.8 7.18.0 (2022-10-31)

Features

- first try to match output files against input files while persisting wildcard values from the consuming job. This can dramatically reduce ambiuity problems. Thanks to @descostesn! (#1939) (d093907)

4.28.9 7.17.2 (2022-10-28)

Bug Fixes

- Consider source cache when setting search path for python scripts. This allows to import from Python modules next to scripts while deploying the workflow as a snakemake module, even from remote locations. (#1940) (27be1d4)

4.28.10 7.17.1 (2022-10-28)

Bug Fixes

- change source cache entries to keep the original name and folder structure, such that imports from e.g. scripts also work with remote modules (if specified as additional input files with workflow.source_path) (#1936) (c34f3f6)

4.28.11 7.17.0 (2022-10-27)

Features

- allow to define the cache mode per rule (this enables to exclude software envs from the caching hash value, which can be handy e.g. for download rules where the software version does not affect the result) (#1933) (715e618)

Performance Improvements

- cached os.pathconf() call in _record_path() (#1920) (551badb)

4.28.12 7.16.2 (2022-10-26)

Bug Fixes

- fix false rerun triggering downstream of checkpoints due to spurious parameter, code or software env changes (638ea86)
- remove redundant dot in expand call in multiext documentation (#1921) (278beaa)

4.28.13 7.16.1 (2022-10-18)

Bug Fixes

- conda create –no-shortcuts absent on Linux/MacOS (regression from #1046) (#1916) (8a86a1e)
- fix typo in line display of exceptions (#1912) (55e38a6)

4.28.14 7.16.0 (2022-10-14)

Features

- k8s: add –k8s-cpu-scalar (#1857) (a067a1b)

Bug Fixes

- allow report generation to handle pathlib objects (#1904) (7c34656)
- fix false reruns after checkpoints (#1907) (dc5af12)

4.28.15 7.15.2 (2022-10-08)

Bug Fixes

- Comparison of rules and non-rule instances (#1894) (bf01ece)
- delay evaluation of tmpdir to actual job execution, and not submission. This way, tmpdir can be dependent on the node context. (#1860) (4203556)
- ensure that rule name string instead of object is passed to tabulate package (#1898) (f9ff157)
- issue 1846 (#1888) (da2dfbd)
- lexicographically sorted rule display with –list, and trimmed rule docstrings (#1880) (32128ae)

Performance Improvements

- Average NamedList **getitem** performance improvement (#1825) (10451b7)

4.28.16 7.15.1 (2022-10-04)

Bug Fixes

- fix `--immediate-submit` (#1851) (e358372)
- Handle temp files for all jobs in a group. (#1779) (d28b893)

Documentation

- small tweaks to flux documentation (#1886) (f29b371)
- various little fixes (#1875) (b93f8e3)

4.28.17 7.15.0 (2022-10-04)

Features

- adding flux executor (#1810) (40d2bd0)

Bug Fixes

- Add back logging of run directives (#1883) (a65559c)

Documentation

- fix grammar in the intro (#1859) (774bc6a)
- fix typo (#1843) (6572ad9)

4.28.18 7.14.2 (2022-09-26)

Bug Fixes

- reduce resource requirements for kubernetes tests (#1876) (cb4b78a)

4.28.19 7.14.1 (2022-09-23)

Bug Fixes

- allocation of local ssds in k8s tests (#1870) (d0de4dc)
- allow script directive to take pathlib Path (#1869) (12cdc96)
- catch errors in remote.AUTO provider list (#1834) (c613ed2)
- consistently use text output in conda shell commands and various little fixes for failing test cases due to conda package changes (#1864) (4234fe7)
- declare associative arrays (#1844) (90ae449)
- fix falsely triggered reruns if input files are obtained via workflow.source_path() (#1862) (2dc2e6a)
- fixed typos (#1847) (a1e49b6)
- k8s container volume mounts as list (#1868) (5c54df3)
- None type error when invoking Workflow object manually (#1731) (dc45ccb)
- request disk_mb resource from k8s (#1858) (f68f166)
- respect shebang lines in post-deploy scripts (see deployment docs) (#1841) (c26c4b6)

4.28.20 7.14.0 (2022-08-27)

Features

- add support for bash scripts in the script directive (beyond small shell commands) (#1821) (c4cf8fd)

Documentation

- fix small typo in FAQ (#1832) (914172b)

4.28.21 7.13.0 (2022-08-25)

Features

- add gitfile option to make it possible to use local git repos when importing modules (#1376) (1a3b91f)

Bug Fixes

- allow to use {wildcards} for group jobs in cluster config (#1555) (f0ec73d)
- avoid “Admin” prompt when using conda on windows (#1046) (552fadf)
- handle benmark bug that arise with singularity (#1671) (10ef7c4)
- Open Snakefile for reading with explicit encoding specified (#1146) (ec1d859)
- remove superfluous comma causing TypeError in conda-frontend error message (#1804) (87b013c)

Documentation

- explain SNAKEMAKE_PROFILE environment variable (2b32bba)
- update contribution docs (09a5595)

4.28.22 7.12.1 (2022-08-09)

Bug Fixes

- Fix case of multiple scattergather processes (#1799) (417aad4)
- more comprehensive error reporting for RuleExceptions (#1802) (1cd9512)

4.28.23 7.12.0 (2022-07-29)

Features

- print reason summary in case of dryrun (#1778) (bd2a68b)

Bug Fixes

- Fix technical bugs in resource-scope documentation (#1784) (878420c)
- move max_status_checks_per_second attribute setting before the wait thread of cluster backends is started to avoid missing attribute errors (#1775) (a48e9d0)

4.28.24 7.11.0 (2022-07-27)

Features

- improved resource handling in groups and ability to define resource scopes (global or per node), see docs and –help (#1218) (a8014d0)

Bug Fixes

- fixed conda frontend detection and checking to also work with latest mambaforge (#1781) (225e68c)

4.28.25 7.10.0 (2022-07-26)

Features

- Support conda environment definitions to be passed as function pointers, similar to input, params, and resources (#1300) (6f582f1)

Bug Fixes

- fix regression in workflow source acquisition of google life science executor (#1773) (c07732e)
- limit filename length of temporary files generated by the persistence backend (metadata, incomplete markers, etc.) (#1780) (59053e7)

4.28.26 7.9.0 (2022-07-19)

Features

- make it possible to exclude rules that will be imported when using 'use rule' statement (#1717) (d9e0611)

Bug Fixes

- add lock free mechanism for avoiding race conditions when writing persistence information; consider corrupt metadata records as non-existent (#1745) (71fe952)
- conda python interpreter path on Windows (#1711) (155c9d6)
- ensures that REncoder also checks for numpy.bool_ in encode_value (#1749) (10a6e1d)
- Move quiet default after profile parsing (#1764) (6ade76d)

4.28.27 7.8.5 (2022-06-30)

Documentation

- fix long description type for pypi (set to markdown) (d8d9b8f)

4.28.28 7.8.4 (2022-06-30)

Bug Fixes

- only display a warning in case of non-strict channel priorities (#1752) (b84fa33)
- pass triggers and resources to subworkflow (#1733) (fa7fb75)
- add pyproject.toml to use setuptools features (#1725) (454bfd1)

Documentation

- add workflows.community metadata (#1736) (8a42afc)

7.8.3 (2022-06-20)

Bug Fixes

- allow apptainer as a successor to singularity. (#1706) (bcbdb0b)
- improved provenance trigger info (#1720) (29d959d)
- small changes to make docs checkpoint example functional (#1714) (1d4909e)

7.8.2 (2022-06-08)

Bug Fixes

- fixed bug in needrun computation of jobs downsteam of checkpoints (#1704) (c634b78)

7.8.1 (2022-05-31)

Bug Fixes

- handling of remaining jobs when using –keep-going (#1693) (87e4303)
- more robust calculation of number of jobs until ready for execution (#1691) (fdfc717)
- propagate rerun trigger info to cluster jobs; fix a bug leading to software stack trigger generating false positives in case of conda environments; fixed display of info message in case of provenance triggered reruns (#1686) (503c70c)
- set channel priority in container system wide (#1690) (41175b3)

4.28.29 7.8.0 (2022-05-24)

Features

- automatically rerun jobs if parameters, code, input file set, or software stack changed (thanks to @cclienty and @timtroendle). This also increases performance of DAG building by handling job "needrun" updates level wise, while avoiding to perform a full check for those jobs that are already downstream of a job that has been determined to require a rerun. (#1663) (4c11893)
- enable the definition of conda pin files in order to freeze an environment. This can drastically increase the robustness because it allows to freeze an environment at a working state. (#1667) (53972bf)

Bug Fixes

- fail with error if conda installation is not set to strict channel priorities (#1672) (f1ffbf2)
- fix errors occurring when refering to input func via rules..input (#1669) (28a4795)
- parsing error when combining single line directive with multi-line directive in use rule statements (#1662) (26e57d6)

4.28.30 7.7.0 (2022-05-16)

Features

- add flag `ensure` that allows to annotate that certain output files should be non-empty or agree with a given checksum (#1651) (76f69d9)
- for small files, compare checksums to determine if job needs to run if input file is newer than output file (#1568) (1ae85c6)
- LockException (#1276) (f5e6fa6)
- new directive "retries" for annotating the number of times a job shall be restarted after a failure (#1649) (c8d81d0)

Bug Fixes

- iRODS functionality - issue #1510 (#1611) (9c3767d)

Documentation

- singularity sometimes uses system /tmp explanation (#1588) (170c1d9)

7.6.2 (2022-05-06)

Bug Fixes

- fixed permission issues when using zenodo remote provider to access restricted depositions (#1634) (510f534)

7.6.1 (2022-05-04)

Bug Fixes

- check for skipped rules in case of local rule inheritance (#1631) (9083ac1)

4.28.31 7.6.0 (2022-05-03)

Features

- enable restricted access support in zenodo remote provider (#1623) (692caf9)

Bug Fixes

- avoid erroneous too early deletion of parent directories in case of failed jobs (thanks to @SichongP). (#1601) (b0917e6)
- ensure that rule inheritance considers the same globals and other settings as parent module (#1621) (104cab9)
- issue 1615 - Switch formatting condition for dictionary (#1617) (0771062)
- multiext prefix computation in case it is used within a module that defines an additional prefix (#1609) (fc6dfc6)
- remove redundant print (#1608) (cc7e0e3)

4.28.32 7.5.0 (2022-04-26)

Features

- vim syntax updates (#1584) (b8c77f6)

Bug Fixes

- properly use configfiles specified via CLI also if configfile specified via configfile directive is not present (1e0649a)

Documentation

- checkpoint documentation (#1562) (4cbfb47)

4.28.33 7.4.0 (2022-04-22)

Features

- Allow paramspace to separate filename params with custom separator (#1299) (8236e80)

Bug Fixes

- preserve dtypes across paramspace (#1578) (70ce6a0)
- use mambaforge for snakemake container image (#1595) (b7e6906)

7.3.8 (2022-04-06)

Bug Fixes

- support multiple input files for template_engine rules (#1571) (aee7cf2)

7.3.7 (2022-04-05)

Bug Fixes

- allow labels function to return None (#1565) (fef74d6)
- do not wrap whitespace in result info headers of reports (653d0d0)
- fixed detection of norun rules inside of modules (#1566) (d2223d4)
- properly use retry mechanism in source cache (#1564) (624a83d)

7.3.6 (2022-04-02)

Bug Fixes

- always recalculate job resources before job is scheduled as input might have changed or not have been present initially (#1552) (44aacdb)
- fixed handling of input functions and unpack when using the prefix setting of module definitions (#1553) (d561e04)
- fixed parsing of subsequent use rule statements directly beneath each other (#1548) (77d5a08)
- fix spurious missing file errors when using google storage (#1541) (1b3ede1)
- proper error message if resource types do not match (#1556) (1112321)
- quote workdir in job exec prefix to allow to spaces in the workdir (#1547) (c3a593e)
- report error and possible cause if metadata cleanup fails (#1554) (6866134)

7.3.5 (2022-03-31)

Bug Fixes

- do not remove existing temp files in case of dryrun (#1543) (e820f97)
- fixed bug in missing input file handling for cluster jobs (#1544) (40e2eb2)

Documentation

- explain automatic decompression strategy for http remote provider (e6826b6)

7.3.4 (2022-03-30)

Bug Fixes

- better error messages in case of missing files after latency period (#1528) (5b394c0)
- correct handling of exceptions in input functions that are generators (#1536) (d9a56aa)
- obtaining conda prefix when using in combination with singularity (#1535) (99b22d3)
- proper error message in case of missing git when checking for source files (#1534) (92887a3)
- throw error message in case of target rule that depends on a pipe. (#1532) (b9e9a7e)

Documentation

- display rust-script env. (950d8ba)
- zenodo example (76159ae)

7.3.3 (2022-03-28)

Bug Fixes

- better error message in case of failing to create conda env (#1526) (e7a461c)
- fix singularity logging messages causing conda fail (#1523) (7797595)
- more robust handling of incompletely evaluated parameters (any interaction with them will result in a string now). (#1525) (3d4c768)

Documentation

- details on benchmarked results (64fea09)

7.3.2 (2022-03-25)

Bug Fixes

- fixed code change detection (#1513) (67298c6)
- modify dag and workflow display in report to also work for big DAGs (#1517) (1364dfb)

Documentation

- Clarify the use of conda with notebook directive (#1515) (aefb1eb)

7.3.1 (2022-03-23)

Bug Fixes

- add about page to report, including embedded packages and licenses (#1511) (142a452)
- in google live science backend, save multiple logs per rule name and overwrite existing logs (#1504) (9e92d63)
- in rules from imported modules, exclude modified paths from module prefixing (#1494) (1e73db0)
- Replaced pathlib relative_to with os.relpath (#1505) (dc65e29)
- update for minimum of Python 3.7 (#1509) (62024e2)

4.28.34 7.3.0 (2022-03-21)

Features

- Support for machine_type for kubernetes executor (#1291) (12d6f67)

Bug Fixes

- always wait for input files before starting jobs, also upon local execution and within group jobs. This should add further robustness against NFS latency issues. (#1486) (cab2adb)
- cleaned up and rewritten execution backend structure, (fixing #1475, #860, #1007, #1008) (PR #1491) (e87cc97)
- do not skip local conda env creation per se when having no shared FS, because it is still needed for local jobs. Instead, decide for each env whether it is needed locally or not. (#1490) (3f03c5d)
- fixed temp file deletion for group jobs (#1487) (d030443)
- improve robustness when retrieving remote source files, fixed usage of local git repos as wrapper prefixes (in collaboration with @cokelaer and @Smeds) (#1495) (e16531d)
- mtime inventory for google storage was accidentally setting a float instead of a proper mtime object (#1484) (7c762c7)
- render empty caption if nothing defined in report flag (013a6e8)

Documentation

- clarify namespacing when using modules. (dbed4a3)
- separate api docs (ded7da9)
- separate api docs (#1499) (5cf275a)

7.2.1 (2022-03-14)

Bug Fixes

- add missing report.templates.components module to setup.py (cb4e3fe)

Documentation

- add install info of development (git) version to docs (#1477) (2a2d6cd)

4.28.35 7.2.0 (2022-03-13)

Features

- improved reports: more interactive and modern interface, ability to define a label based representation of files (#1470) (d09df0c)

Bug Fixes

- always deploy conda envs in main process when assuming a shared file system (fixes issue #1463) (#1472) (79788eb)
- do not wait for named or containerized conda envs (#1473) (6b1d09c)
- implement lock-free source file caching. This avoids hangs on network file systems like NFS. (#1464) (9520e98)

7.1.1 (2022-03-07)

Bug Fixes

- quote jobid passed to status script to support multi-cluster Slurm setup (#1459) (0232201)

4.28.36 7.1.0 (2022-03-04)

Features

- Zenodo remote provider for transparent storage on and retrieval from Zenodo (#1455) (4586ef7)

Bug Fixes

- disable mtime retrieval from github api for now. This quickly exceeds rate limits. (1858bb9)
- display change warnings only for jobs that won't be executed otherwise (086f60f)
- work around segfault with >100 jobs in google life sciences backend (#1451) (2c0fee2)

7.0.4 (2022-03-03)

Bug Fixes

- more details on input and output exceptions (missing input, protected output, etc.) (#1453) (8d64af2)

7.0.3 (2022-03-02)

Bug Fixes

- fix a bug leading to duplicate conda env initializations; fix display of jobs and output files with changes (994b151)
- preserve empty names input or output file lists in params or resource functions (0d19ab0)
- remove accidental pdb statement (9c935f1)
- remove deprecated and add missing arguments to internal functions (93a7e39)

7.0.2 (2022-03-01)

Bug Fixes

- add local marker for input files in cufflinks example. fixes issue #1362 (90bc88b)
- failure to properly apply default remote prefix in combination with the unpack marker (#1448) (82666f1)
- set mtime for cached source files [WIP] (#1443) (dd27209)
- small bug in snakemake.executors (#1440) (6e64292)

Documentation

- fix list display in docs (3724367)
- fix list display in docs (2dd0e91)
- Fix typo and grammar mistake in scatter-gather section. (#1441) (f218aaa)

7.0.1 (2022-02-26)

Bug Fixes

- avoid incomplete remote files in case of errors and automatically retry download and upload (#1432) (8fc23ed)
- do not apply module prefix in case of remote files (5645b3f)
- do not require –cores or –jobs to be set when –cleanup-metadata is used. (#1429) (9c73907)
- more robust place for runtime source file cache (#1436) (2681f6f)
- provide details on error when failing to evaluate default resources (#1430) (04f39a9)
- provide proper error when using immediate submit in combination with checkpoint jobs. (#1437) (865cf0f)

Documentation

- explain relative path interpretation (#1428) (add9a05)
- Fix problems with code blocks and broken internal link. (#1424) (5d4e7d8)
- temaplte rendering examples and available variables (#1431) (5995e9e)
- update copyright year (#1427) (6b9f5da)

4.28.37 7.0.0 (2022-02-23)

⚠ BREAKING CHANGES

- require at least Python 3.7 (fd5daae)

Features

- adding service jobs, i.e. the ability to define jobs that provide a resource for consumers (like a shared memory device or a database), and will be automatically terminated by Snakemake once all consumers are finished. (see docs, #1413) (a471adb)
- support for group local jobs by enabling optional groupid consideration in input functions (see docs, #1418) (5d45493)
- Adding –cluster-cancel and –cluster-cancel-nargs (#1395) (0593de1)
- cluster sidecar (#1397) (b992cd1)
- template rendering integration (yte and jinja2) (#1410) (e1cbde5)

Bug Fixes

- bug in pipe group handling that led to multiple assignments of the same group id to different groups; bug that accidentally added already running groups of the list of ready jobs (issue #1331) (#1332) (1a9b483)
- display wrapper or external script code in report #1393 (#1404) (a007bd1)
- do not pass SNAKEMAKE_PROFILE into cluster-submit (#1398) (#1407) (7189183)
- issue with duplicated prefix for checkpoints on cloud (#1294) (8ed0c8c)
- keep flags with apply_wildcards on cloned IOFile (#1416) (23c943f)
- remove raise that limits using –config with dicts (#1341) (bd65057)
- Repair MREs from #823 (#1203) (b007979)
- warn on non-file-modification-date changes like params, code, or input files (#1419) (b5f53f0)

6.15.5 (2022-02-09)

Bug Fixes

- convert conda env to string before checks (#1382) (7a8da9f)
- fix pepfile handling in case of module usage (#1387) (f097a76)

6.15.4 (2022-02-09)

Bug Fixes

- fix issue when generating unit tests for rules with directory output (#1385) (7db614f)

Documentation

- fix tutorial setup instructions for MacOS. (#1383) (b57b749)

6.15.3 (2022-02-07)

Bug Fixes

- skip global report caption when using a module (#1379) (a755cee)

6.15.2 (2022-02-05)

Bug Fixes

- avoid mutable default argument (#1330) (978cc93)
- don't raise WorkflowError when entry is empty (#1368) (1fc6f7b)
- fix assertion error in conda env file spec when applying wildcards (thanks @ddesvillechabrol) (#1377) (6200652)
- fix None type error when invoking Workflow object manually (#1366) (fca3895)
- XRootDHelper.exists supports non posix filesystem (object store) (#1348) (7a3ad2f)

Documentation

- add sentence about workflow template to docs (#1369) (5fabffb)
- fix typo in installation.rst (#1344) (c45d47a)

6.15.1 (2022-01-31)

Bug Fixes

- consider post-deploy script for env hashing (#1363) (d50efd9)

4.28.38 6.15.0 (2022-01-29)

Features

- adding default_target directive for declaring default target rules that are not the first rule in the workflow. (#1358) (638ec1a)

Bug Fixes

- Draft notebook filename with wildcards and params. (#1352) (11d4dc8)
- proper error message when defining cache eligibility for rules with multiple output files and no multiext declaration. (#1357) (47b5096)

Documentation

- Command line arguments for configuration files (#1343) (ad8aaa4)
- fix broken link in executor_tutorial/tutorial.rst (#1360) (c9be764)

4.28.39 6.14.0 (2022-01-26)

Features

- Added timestamp to each log message (#1304) (a5769f0)
- implement support for removing GFAL remote files (#1103) (25943e5)
- specify conda environments via their name (#1340) (735ab23)
- support for post deploy scripts (#1325) (e5dac4f)

Documentation

- link to list of dependencies from installation (#1336) (99d7bfe)
- update URL to emacs snakemake-mode (#1339) (dae7b8f)

6.13.1 (2022-01-11)

Bug Fixes

- –conda-frontend value not passed on to cluster jobs (#1317) (df46ddb)
- atomic job error display (#1326) (aa2c265)
- fix source cache handling for remote source files retrieved via github() or gitlab() tags. (#1322) (6e2ecd2)
- typos in code examples (#1324) (60010e4)

4.28.40 6.13.0 (2021-12-21)

Features

- allow prefix definition in module statements (#1310) (29e6540)

6.12.3 (2021-12-09)

Bug Fixes

- fixed display of any exceptions and errors from within a workflow definition (23d40d9)

6.12.2 (2021-12-07)

Bug Fixes

- rule inheritance within modules (did previously lead to key errors) (#1292) (603e0a8)

Documentation

- Fix typo in rules.rst (—draft-notebook) (#1290) (f5c42cf)

6.12.1 (2021-11-29)

Bug Fixes

- set default number of nodes to 1 in test cases (#1288) (f6e12b4)

4.28.41 6.12.0 (2021-11-29)

Features

- add flag –draft-notebook for generating a skeleton notebook for manual editing (e.g. in VSCode). (#1284) (d279322)

Bug Fixes

- issue #1257 (missing logfile failure when using shadow directory) (#1258) (426d92f)
- keep empty output and input dirs of –draft-notebook job (f1181bd)
- SameFileError #1153 (#1220) (ede313d)
- snakemake API using only 1 job as default (#1283) (e92ad48)

Documentation

- short tutorial updates (#1286) (b653a44)

6.11.1 (2021-11-26)

Bug Fixes

- provide temporary IPYTHONDIR for notebook execution in order to avoid race conditions in https://github.com/ipython/ipython/blob/master/IPython/paths.py#L20 upon execution of multiple notebooks at the same time. (#1280) (4d70da1)

Documentation

- move psutil import into benchmark methods to avoid needing it as a dependency for doc building (6ffe38d)
- require sphinx>=3 (1773875)
- skip lazy property (2883718)

4.28.42 6.11.0 (2021-11-25)

Features

- fail with an error if snakemake cannot write job metadata. (#1273) (cd968cd)

Bug Fixes

- Adds fixes for the first two MREs in #823 (#1215) (cfd2f89)
- env file usage after changes to source file handling (inspired by #1233 and #1211). (#1236) (3ac8e85)
- fixed code change detection when using modules (#1264) (b571e09)
- handle config file extension/overwriting more explicitly (#1251) (d0a7bf2)
- Issue #1253 (problems editing Jupyter Notebooks) (#1255) (3398ddf)
- more informative nothing to be done message (#1234) (368d265)
- only consider context of shell command for technical switches if called from snakemake rules. (#1213) (4816a58)
- R encoding of pathlib.Path objects (#1201) (bd516e9)
- Use 'snakemake.utils.update_config' instead of 'dict.update' (#1126) (2658027)

4.28.43 6.10.0 (2021-10-21)

Features

- Add more informative errors when evaluation of `--default-resources` fails (#1192) (b3c4e68)

Bug Fixes

- add quotes to each item of the wait_for_files list (#1160) (72856ed)
- caching process (#1225) (0825a29)
- enable usage of job grouping in GLS (#1054) (d243c22)
- Only –bind Snakemake when we're working with a Python script (#1206) (1d79f62)
- run dependencies with non-existent ancient files before the consuming job (#1202) (84d1f64), closes #946
- status cmd repeats until killed by 11 *different* signals (#1207) (8b28b57)
- typo in sourcecache use (#1229) (8b54bc5)
- wms monitor arg parsing now accepts any kind of value (#1181) (313de93)

Documentation

- Clarification of –cluster-stats docs & elaborating on the situation where job ids are not passed to the status script (#1221) (ed0e4a2)
- Combine CHANGELOG.rst with CHANGELOG.md (#1228) (19f5a43)
- Mention required openssl dep for rust-script (#1216) (fc8c5f6)
- Unpin docutils version (#1230) (15a82bf)

6.9.1 (2021-09-30)

Bug Fixes

- fix function call when creating report and hashes for between workflow caching (#1198) (a4f6836)

4.28.44 6.9.0 (2021-09-29)

Features

- autoconvert Path objects to str when passing to R or Julia scripts (80ec513)

Bug Fixes

- fix source retrieval during between workflow caching and report generation (2394ca4)

6.8.2 (2021-09-29)

Bug Fixes

- fix path returned by get_source() (ee05315)

6.8.1 (2021-09-24)

Bug Fixes

- async_run to allow nested event loops. (#1170) (5dc6bbd)
- merging of pipe groups when multiple rules are chained together via pipes (#1173) (de91d2c)
- potential memory corruption caused by Google storage objects accessed from different threads (#1174) (41a5071)

Performance Improvements

- more extensive caching of source files, including wrappers. (#1182) (bdb75f8)

Documentation

- move note (75a544b)
- polish (47a7b62)
- tutorial formatting (594f5fb)

4.28.45 6.8.0 (2021-09-06)

Features

- Add `shadow: "copy-minimal"` directive (#1155) (1803f0b)
- support XRootD as a default remote provider (#1017) (fe03157)

Bug Fixes

- AmbiguousRuleException bug caused by weak ordering of rules (#1124) (7f54c39)
- Bugfix tes add files (#1133) (8892bf2)
- Disable Persistence cache for snakemake jobs (#1159) (7110f9d)
- efficient job status checking when using DRMAA API (this should yield much better parallelization and performance when using –drmaa) (#1156) (ac004cb)
- improved error handling for cluster status scripts and smarter job selector choice in case of cluster submission (use greedy for single jobs). (#1142) (48d2dd9)
- Initialize assignments dictionary when setting rule-based resources (#1154) (68c13fd)
- key error when handling FileNotFoundError in input functions. (#1138) (d25f04d)
- linting of remote snakefiles (#1131) (2104e10)

Performance Improvements

- improve job selection performance in case of potential ambiguity that is resolved by comprehensive ruleorder statements. (#1147) (921f4f7)

4.28.46 6.7.0 (2021-08-12)

Features

- Add support for rust scripts (enabling directly integrated ad-hoc robust high performance scripting) (#1053) (f0e8fa2)

Bug Fixes

- Ga4gh tes bugfixes (#1127) (af21d6c)
- improved display of percentage of done jobs (1fee8c0)
- improved error message in case of target rule misspecification (83b1f5b)

Documentation

- fix contributing executors link (#1112) (4bb58d1)
- Fix typo in file path in remote files documentation (#1110) (9ce294f)

6.6.1 (2021-07-19)

Bug Fixes

- avoid superfluous calls of conda info that have slowed down Snakemake since 6.4.1. (#1099) (e990927)

4.28.47 6.6.0 (2021-07-16)

Features

- Allow to mark all output files as temp with –all-temp (#1097) (0ac3b38)

6.5.5 (2021-07-16)

Bug Fixes

- dummy release (e4dca50)

6.5.4 (2021-07-16)

Fixes

- Fixed –touch in combination with temp files (issue #1028) (@johanneskoester, @iromeo).

Documentation

- Fix syntax error in docs/conf.py and update sphinx.ext.napoleon import (#1084) (3e3fac2)
- Improved pepfile (pepschema) documentation (@stolarczyk).

[6.5.3] - 2021-07-06

- Fixed a bug occuring when using –resources in the command line interface (@johanneskoester).
- Minor improvements in the docs (@johanneskoester).

[6.5.2] - 2021-07-02

- Create directory pointed to by tmpdir resource if it does not yet exist (@johanneskoester).
- Use a single core again in dryrun if –cores is not specified (@johanneskoester).
- Bugfix for FTP remote provider (@jmeppley).
- Improved documentation (@corneliusroemer).

[6.5.1] - 2021-06-24

- Extended best practices document (@johanneskoester)
- Restore `-j all` behavior for local execution as a (deprecated) way of running Snakemake on all cores. Recommended now: `--cores all` (@johanneskoester).
- Improved handling and better error messages for checkpoints (@johanneskoester).

4.28.48 [6.5.0] - 2021-06-22

- Allow to set the default profile via the environment variable $SNAKEMAKE_PROFILE.
- There is a new default resource tmpdir (by default reflects the system setting), which is automatically used for temporary files by shell commands and scripts which properly consider the usual environment variables like $TMP, $TEMP, $TMPDIR (@johanneskoester).
- The CLI flags –jobs and –cores are now separated, with –cores being responsible for local cores and global cores in the cluster case, and –jobs being responsible for number of jobs. Still -j and –jobs works as a fallback for local execution (@johanneskoester).
- Added the ability to overwrite resources via –set-resources (@johanneskoester).
- Various fixes for Windows execution (@melund).
- Fixed a bug with fractional resources (@johanneskoester).
- Fixed timeouts and other issues in google life science backend (@johanneskoester).

- Fixed a bug with missing conda frontend definitions in subworkflows (@johanneskoester).
- Skip envvar checking during linting (@johanneskoester).
- Fixed a bug causing container images in modules to be ignored (@johanneskoester).

[6.4.1] - 2021-05-27

- Fixed bug in `workflow.source_path()` that occurred with modules included from remote locations (@johanneskoester).
- Inform cluster jobs about conda/mamba/activate path such that they don't need to determine this themselves (@johanneskoester).

4.28.49 [6.4.0] - 2021-05-20

- Improvements in the docs (resource usage, best practices, remote files) (@johanneskoester, @admorris).
- functions given to params can now safely open input files generated by previous rules. If they are not present, TBD will be displayed and function will be reevaluated immediately before the job is executed (i.e. when files are present) (@ASLeonard).
- Connection pool for SFTP and FTP remote files, increasing download performance (@jmeppley).
- Require correct minimum version of smart_open (@Redmar-van-den-Berg).
- Added workflow.source_path(path), allowing to get the correct path relative to the current Snakefile, even when Snakefile is included via URL (@johanneskoester).
- Fixed bugs in module system (@johanneskoester, @dlaehnemann).
- Fixed issue with checkpoints and ruleorder where phantom dependencies are not properly removed from the DAG (@jmeppley, @johanneskoester).
- Disable tibanna behavior that opens a browser window for each job (@nigiord).
- Allow `Paramspace(..., filename_params="*")`, meaning that all columns of the paramspace will be encoded into the filename (@kpj).
- Avoid PATH modification in cluster jobs (@johanneskoester).
- For large sets of input files, pass files to wait for (FS latency) as a file instead of command line args (@kpj, @epruesse).

4.28.50 [6.3.0] - 2021-04-29

- Changed behavior of `workflow.snakefile` to always point to the current file instead of the main Snakefile (also in case of includes and modules) (@johanneskoester).
- Fixed a typo in an error message (@nikostr).

4.28.51 [6.2.0] - 2021-04-22

- Support for integration of foreign workflow management systems by introducing a `handover` directive that passes on all resources to a particular rule (which can then invoke another workflow management system). See the docs ("Integrating foreign workflow management systems") (@johanneskoester).
- Behavior improvement for temp handling of checkpoint rules (@epruesse).
- Several improvements in the docs (@johanneskoester).

[6.2.1] - 2021-04-20

- Fixed a minor bug in the linter.

4.28.52 [6.2.0] - 2021-04-20

- Fixed several glitches in paramspace implementation (handling of bools, returning scalar values) (@kpj).
- Fixed bugs in module implementation (@dlaehnemann, @johanneskoester).
- Fall back to greedy scheduling solver if ILP solver needs more than 10 sec (@johanneskoester).

[6.1.1] - 2021-04-07

- Fixed several small bugs of the new module system (@johanneskoester, @dlaehnemann).
- Fixed archive based conda deployment (@johanneskoester).
- Better handling of download and target attributed in the interactive report (@johanneskoester).

4.28.53 [6.1.0] - 2021-04-01

- Snakemake now uses **mamba** as the default conda frontend (which can be overwritten by specifying to use conda via the –conda-frontend flag) (@johanneskoester).
- Profiles using –cluster option can now handle relative submit script paths in combination with arguments (@kdm9).
- New AutoRemoteProvider, which infers the type of remote file protocol from the given URL (@kpj).
- When using global container directive, container usage can be deactivated on a per rule base (@bilke).
- Bugfixes for checkpoint handling (@johanneskoester).
- Bugfixes for the module system (@johanneskoester, @dlaehnemann).
- Various improvements for the tutorial.

[6.0.5] - 2021-03-11

- Fix bug (introduced with 6.0) when handling of HTML directories in report (@johanneskoester).

[6.0.4] - 2021-03-11

- Various textual improvements in the tutorial (@dlaehnemann).

[6.0.3] - 2021-03-08

- No longer use a shortened hash for naming conda environments in .snakemake/conda (@johanneskoester).
- Various little updates to the docs (@johanneskoester).

[6.0.2] - 2021-03-03

- Fix race condition in conda checking code (@johanneskoester).

[6.0.1] - 2021-03-03

- Restored Python 3.5 compatibility by removing f-strings (@mbhall88)
- Fix rendering issue in the docs.
- Add gitpod dev environment and gitpod environment for the tutorial.

4.28.54 [6.0.0] - 2021-02-26

- Introduced a new module system, see https://snakemake.readthedocs.io/en/stable/snakefiles/modularization.html#modules (@johanneskoester).
- Introduced a rule inheritance mechanism, see https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#rule-inheritance (@johanneskoester).
- Automatically containerize a conda-based pipeline with `--containerize`, see https://snakemake.readthedocs.io/en/stable/snakefiles/deployment.html#containerization-of-conda-based-workflows (@johanneskoester).
- Use temporary files for long shell commands (@epruesse).
- Various fixes in the documentation (@ctb, @SilasK, @EthanHolleman).
- Fixed a bug in job grouping that led to non-deterministic behavior (@johanneskoester).

[5.32.2] - 2021-02-11

Changed

- Fixed infinite loading of results in Snakemake reports (@FelixMoelder)

[5.32.1] - 2021-02-08

Changed

- Improved warning on wildcard constraints (@jheuel)
- Improved logging from the new scheduler implementation (@johanneskoester)
- Restored Python 3.5 compatibility by removing f-strings (@mbhall88)
- Snakemake now automatically adds a global wildcard constraint for {scatteritem}, when scatter/gather support is used.
- The zip variant of Snakemake reports is now compressed (@FelixMoelder).
- Improved docs (@ctb).
- Make output file removal in cluster mode more robust (@sebschmi).

4.28.55 [5.32.0] - 2021-01-15

Changed

- Handle accidental use of GLS backend with singularity (@vsoch).
- Improved and extended WMS-monitor implementation (@vsoch).
- Display index and total count in `{scatteritem}` when using the scatter-gather helper (@johanneskoester).
- Fixed problems with jobid display when handling checkpoint updates (@johanneskoester, @jmeppley).
- Fixed bug when checking for directory containment of output files (@jmeppley).
- Implement –no-subworkflows treatment in combination with –cluster (@goi42).

[5.31.1] - 2020-12-21

Changed

- added wget again to the container image

4.28.56 [5.31.0] - 2020-12-21

Added

- The `Paramspace` helper for automatically exploring parameter spaces given as Pandas dataframes.
- A new directive `name:` for setting rule names from variables.

Changed

- Various small bug fixes for scheduling and checkpoint handling.
- Automatically block R_LIBS, PYTHONPATH, PERL5LIB, and PERLLIB when using conda with –use-conda. This behavior can be deactivated with –conda-not-block-envvars.
- Update container image to latest singularity.

[5.30.2] - 2020-12-16

Changed

- Fix permission issues with jobscripts on some systems (@Phhere).
- Added notes on WSL to the tutorial (@RomainFeron).
- Scheduler fixes (@johanneskoester).
- Fixed a bug in checkpoint handling that led to hanging workflow execution (@jmeppley).
- Pass cluster nodes to subworkflows (@votti).
- Fix start time recording in metadata (@lparsons).
- Fix time retrieval in reports (@johanneskoester).
- Fix error when returning a Path from an input function (@sappjw).
- Extending monitoring docs with some notes about future api changes (@vsoch).

4.28.57 [5.30.0] - 2020-11-23

Added

- Benchmarks now also report CPU time (@natir).

Changed

- Fixed a reauthentication bug in Kubernetes support (@haizi-zh).

4.28.58 [5.29.0] - 2020-11-19

Changed

- Fixed several bugs in reports and scheduler.
- Remove automatic (but buggy) encoding of csv/tsv files into HTML tables in the report (we will soon have a better alternative).
- Fixed bug in kubernetes executor occurring with large source files.

4.28.59 [5.28.0] - 2020-11-12

Added

- Execution backend for GA4GH TES (task execution scheduler) an abstraction layer for various cluster and cloud queuing systems (@svedziok, @uniqueg).
- script, notebook, wrapper and cwl directives now permit to use wildcards and params for composing paths (@johanneskoester).

Changed

- Restored compatibility with Python 3.5 and 3.6 (@cclienti).
- Various usability bug fixes (@goi43, @johanneskoester, @dcroote).
- Better and more secure parsing of values when using –config (@bingxiao).

[5.27.4] - 2020-11-03

Changed

- Further speed improvements for DAG computation.
- Fixed metadata migration errors occuring with long output file paths.
- Add WorkflowHub specifications to the docs.
- Fix group assignments.

[5.27.3] - 2020-10-30

Changed

- Added missing files to source distribution.

[5.27.2] - 2020-10-30

Changed

- DAG computation runtime has been improved by orders of magnitude, it is linear in the number of jobs now (@mhulsmann, @johanneskoester).
- Stat calls have been dramatically reduced and are now performed in parallel (@johanneskoester).
- Scheduler fixes (@FelixMoelder).
- Directory support and other fixes for Google Life Sciences backend (@vsoch, @millerdz).
- Support for panoptes monitor server (@fgypas).
- Extended pathlib support (@mbhall88).
- Vim plugin improvements (@troycomi).
- Prevent jobs being rerun when input files are marked as ancient and another job in the DAG creates them.

- Fixed –list-code-changes for included rules (@jbloom).

Added

- Syntax highlighting for nano (@baileythegreen).

[5.26.1] - 2020-10-01

Changed

- Use coin ILP solver for scheduling by default (GLPK has bugs that can cause it to fail in certain situations).
- If coin is not available, fall back to greedy scheduler.

4.28.60 [5.26.0] - 2020-09-30

Added

- Flag –max-inventory-time for setting maximum time spend on creating file inventory.
- Flag –scheduler-ilp-solver for defining which solver to use for the ILP scheduler.

Changed

- Fixed various bugs with the new scheduler (@FelixMoelder).
- Fixed bug causing certain parameters not to be passed to the cluster (–set-scatter, –scheduler, –set-threads).
- Updated docs and fixed of google backend (@vsoch).
- Display jupyter notebook code in reports.
- Improved scheduler behavior in order to directly remove temporary files if possible.

4.28.61 [5.25.0] - 2020-09-18

Added

- Simplified and more configurable support for scatter-gather processes (see docs).
- Fully configurable DAG partitioning by grouping jobs at the command line. This should provide a vast additional improvement to scalability in cluster and cloud settings.

Changed

- Depend on latest pulp, thereby enable Python >=3.8 compatibility again.
- Fixes for snakefile handling in google life sciences backend (@vsoch).

[5.24.2] - 2020-09-15

Changed

- Fixed a bug in the linter that caused a false warning when using resources in shell commands.

[5.24.1] - 2020-09-13

Changed

- Depend on pulp < 2.0, which includes the default coin cbc solver for all platforms.

4.28.62 [5.24.0] - 2020-09-09

Added

- Preemtion support for google cloud backend (@vsoch).

Changed

- Fixed compatibility issues in new scheduler code (@dtrodrigues and @johanneskoester).
- Improved error messages (@Sam-Tygier, @terrycojones)
- Various small bug fixes.
- Improved profile documentation (@johanneskoester).

4.28.63 [5.23.0] - 2020-08-24

Added

- Support for workflow configuration via portable encapsulated projects (PEPs, https://pep.databio.org).
- A new ILP based default scheduler now ensures that temporary files are deleted as fast as possible (@FelixMoelder, @johanneskoester).

Changed

- Fixed bug in modification date comparison for files in google storage (@vsoch).
- Various small documentation improvements (@dcroote, @erjel, @dlaehnemann, @goi42).

[5.22.1] - 2020-08-14

Changed

- Fixed a missing dependency for google storage in cloud execution.

4.28.64 [5.22.0] - 2020-08-13

Added

- Added short option `-T` for CLI parameter `--restart-times` (@mbhall88).

Changed

- Various small fixes for google storage and life sciences backends (@vsoch).

4.28.65 [5.21.0] - 2020-08-11

Changed

- Added default-remote-provider support for Azure storage (@andreas-wilm).
- Various small bug fixes and documentation improvements.

[5.20.1] - 2020-07-08

Changed

- Fixed a bug that caused singularity args to be not passed on correctly when using script or conda.

4.28.66 [5.20.0] - 2020-07-08

Changed

- Exceptions in input functions are now handled in a smarter way, by choosing alternative paths in the DAG if available.
- Debugging dag creation (–debug-dag) now gives more hints if alternative DAG paths are chosen.
- Fixes for XRootD remote file implementation.
- Improved CLI documentation.
- Improved docs.

- Various minor bug fixes.
- Restored Python 3.5 compatibility.
- Speed improvements for workdir cleanup.
- Allow Path objects to be passed to expand.

[5.19.3] - 2020-06-16

Changed

- Performance improvements for DAG generation (up to 7x in the google cloud, anything from a little to massive in a cluster, depending on the overall filesystem performance).
- Made harcoded bucket in google cloud executor configurable.
- Improved speed of –unlock command.

[5.19.2] - 2020-06-04

Changed

- Fixed a bug in script and wrapper directives. Tried to decode a str.

[5.19.1] - 2020-06-03

Changed

- Fixed an issue with the parameter linting code, that could cause an index out of bounds exception.

4.28.67 [5.19.0] - 2020-06-02

Added

- The multiext function now allows arbitrary file extensions (no longer required to start with a “.” (thanks to @jafors)
- The include directive can now also take a Pathlib Path object (thanks to @mbhall88).

Changed

- Jupyter notebook integration no longer automatically starts a browser.
- Empty directories are cleaned up after workflow execution.
- Fixed directory handling: no longer fail if the same job writes both a dir and a contained file.
- Linter now recommends using spaces only for indentation.
- Persistence dir “aux” has been renamed to “auxilliary” in order to make windows happy.
- Linter now distinguishes awk syntax from regular variable usage.
- Various bug fixes for Windows (thanks to @melund).

4.28.68 [5.18.0] - 2020-05-21

Added

- Native Google Cloud support via the (despite the name generic) lifesciences API.
- Ability to optionally exchange the conda frontend to mamba (faster and sometimes more correct) instead of conda.

Changed

- Improved notebook integration experience, with various removed bugs and pitfalls.
- Auto-retry google storage API calls on transient or checksum errors.

4.28.69 [5.17.0] - 2020-05-07

Added

- –envvars flag for passing secrets to cloud executors

Changed

- Wider thumbnail dialogs in report.
- Updated installation instructions.
- Various small kubernetes bug fixes.
- Bug fix for iRods remote files.

4.28.70 [5.16.0] - 2020-04-29

Added

- Interactive jupyter notebook editing. Notebooks defined by rules can be interactively drafted and updated using snakemake –edit-notebook (see docs).

Changed

- Fixed group resource usage to occupy one cluster/cloud node.
- Minor bug fixes.

4.28.71 [5.15.0] - 2020-04-21

Changed

- The resource directive can now take strings, e.g. for defining a GPU model (see docs). This will e.g. be used for upcoming updates to cloud executors.
- More extensive conda cleanup with –conda-cleanup-packages, meant for CI usage.
- Further polish for reports.

4.28.72 [5.14.0] - 2020-04-08

Changed

- Redesigned HTML reports, with improved interface and performance.
- For big data, HTML reports can now be stored as ZIP, where files are not anymore embedded but rather are stored in an auxilliary folder, such that they don't have to be in memory during report rendering.
- Added subcategories to report (see docs).
- Fixed a bug linter, leading to only one rule or snakefile to be linted.
- Breaking change in CLI: added flags –conda-cleanup-envs and –conda-cleanup-pkgs, removed flag –cleanup-conda.
- Fixed scheduling of pipe jobs, they are now always scheduled, fixing a hangup.
- Corrected quoting of shell command for cluster submission.

4.28.73 [5.13.0] - 2020-03-27

Added

- Allow to flag directories for inclusion in the report.

Changed

- Fixed hash computation for –cache in case of positional params arguments.
- Automatically restrict thread usage of linear algebra libraries to whatever is specified in the rule/job.

[5.12.3] - 2020-03-24

Changed

- Various minor bug fixes.

[5.12.2] - 2020-03-24

Changed

- Further improved linter output.

[5.12.1] - 2020-03-24

Changed

- Linter fixes

4.28.74 [5.12.0] - 2020-03-24

Changed

- Fixed the ability to supply functions for the thread directive.
- Improved error messages for caching.

Added

- A new "cache: true" directive that allows to annotate between workflow caching eligibility for rules in the workflow.

[5.11.2] - 2020-03-19

Changed

- Fixed a spurious error message complaining about missing singularity image if –use-singularity is not activated.

[5.11.1] - 2020-03-16

Changed

- Fixed a KeyError bug when executing a workflow that defines containers without –use-singularity.

4.28.75 [5.11.0] - 2020-03-16

Changed

- Fixes for environment modules and tibanna-based AWS execution.
- Fixes for –default-resources defaults.
- –cores is now a mandatory argument!
- Automatic checksum validation for google storage.

Added

- Azure storage authentication via SAS
- A generic container directive that will in the future allow for other backends than just singularity. This deprecates the singularity directive, which will however stay functional at least until the next major release.
- envvars directive for asserting environment variable existence. See docs.
- support for AWS spot instances via –tibanna-config spot=true.
- Automatic code quality linting via –lint.

4.28.76 [5.10.0] - 2020-01-20

Added

- Jupyter notebook integration, see docs. This enables interactive development of certain data analysis parts (e.g. for plotting).
- Ability to overwrite thread definitions at the command line (`--threads rulename=3`), thereby improving scalability.
- Requester pays configuration for google storage remote files.
- Add keyword `allow_missing` to expand function, thereby allowing partical expansion by skipping wildcards for which no keywords are defined.

Changed

- Various bug fixes, e.g. for between workflow caching and script execution.

[5.9.1] - 2019-12-20

Changed

- Added a missing module.

4.28.77 [5.9.0] - 2019-12-20

Added

- Support for per-rule environment module definitions to enable HPC specific software deployment (see docs).
- Allow custom log handler defitions via –log-handler-script (e.g. post errors and progress to a slack channel or send emails).
- Allow setting threads as a function of the given cores (see docs).

Changed

- Various minor fixes.

[5.8.2] - 2019-12-16

Added

- Implemented a `multiext` helper, allowing to define a set of output files that just differ by extension.

Changed

- Fixed a failure when caching jobs with conda environments.
- Fixed various minor bugs.
- Caching now allows to cache the output of rules using `multiext`.

[5.8.1] - 2019-11-15

Changed

- Fixed a bug by adding a missing module.

4.28.78 [5.8.0] - 2019-11-15

Added

- Blockchain based caching between workflows (in collaboration with Sven Nahnsen from QBiC), see the docs.
- New flag –skip-cleanup-scripts, that leads to temporary scripts (coming from script or wrapper directive) are not deleted (by Vanessa Sochat).

Changed

- Various bug fixes.

[5.7.4] - 2019-10-23

Changed

- Various fixes and adaptations in the docker container image and the test suite.

[5.7.1] - 2019-10-16

Added

- Ability to print log files of failed jobs with –show-failed-logs.

Changed

- Fixed bugs in tibanna executor.
- Fixed handling of symbolic links.
- Fixed typos in help texts.
- Fixed handling of default resources.
- Fixed bugs in azure storage backend.

4.28.79 [5.7.0] - 2019-10-07

Changed

- Fixed various corner case bugs. Many thanks to the community for pull requests and reporting!
- Container execution adapted to latest singularity.

Added

- First class support for Amazon cloud execution via a new Tibanna backend. Thanks to Soo Lee from Harvard Biomedical Informatics!
- Allow multiple config files to be passed via the command line.
- A new, more detailed way to visualize the DAG (–filegraph). Thanks to Henning Timm!
- Pathlib compatibility added. Input and output files can now also be Path objects. Thanks to Frederik Boulund!
- New azure storage remote provider. Transparently access input and output files on Microsoft Azure. Thanks to Sebastian Kurscheid!

4.28.80 [5.6.0] - 2019-09-06

Changed

- Fix compatibility with latest singularity versions.
- Various bug fixes (e.g. in cluster error handling, remote providers, kubernetes backend).

Added

- Add –default-resources flag, that allows to define default resources for jobs (e.g. mem_mb, disk_mb), see docs.
- Accept `--dry-run` as a synonym of `--dryrun`. Other Snakemake options are similarly hyphenated, so other documentation now refers to `--dry-run` but both (and also `-n`) will always be accepted equivalently.

[5.5.4] - 2019-07-21

Changed

- Reports now automatically include workflow code and configuration for improved transparency.

[5.5.3] - 2019-07-11

Changed

- Various bug fixes.
- Polished reports.

[5.5.2] - 2019-06-25

Changed

- Various minor bug fixes in reports.
- Speed improvements when using checkpoints.

[5.5.1] - 2019-06-18

Changed

- Improved report interface. In particular for large files.
- Small TSV tables are automatically rendered as HTML with datatables.
- Be more permissive with Snakefile choices: allow “Snakefile”, “snakefile”, “workflow/Snakefile”, “workflow/snakefile”.

4.28.81 [5.5.0] - 2019-05-31

Added

- Script directives now also support Julia.

Changed

- Various small bug fixes.

[5.4.5] - 2019-04-12

Changed

- Fixed a bug with pipe output.
- Cleaned up error output.

[5.4.4] - 2019-03-22

Changed

- Vastly improved performance of HTML reports generated with –report, via a more efficient encoding of dara-uri based download links.
- Tighter layout, plus thumbnails and a lightbox for graphical results in HTML reports.
- Bug fix for pipe groups.
- Updated docs.
- Better error handling in DRMAA executor.

[5.4.3] - 2019-03-11

Changed

- More robust handling of conda environment activation that should work with all setups where the conda is available when starting snakemake.
- Fixed bugs on windows.

[5.4.2] - 2019-02-15

Changed

- Fixed a bug where git module cannot be imported from wrapper.

[5.4.1] - 2019-02-14

Added

- Warning when R script is used in combination with conda and R_LIBS environment variable is set. This can cause unexpected results and should be avoided.

Changed

- Improved quoting of paths in conda commands.
- Fixed various issues with checkpoints.
- Improved error messages when combining groups with cluster config.
- Fixed bugs in group implementation.
- Fixed singularity in combination with shadow.

4.28.82 [5.4.0] - 2018-12-18

Added

- Snakemake now allows for data-dependent conditional re-evaluation of the job DAG via checkpoints. This feature also deprecates the `dynamic` flag. See the docs.

[5.3.1] - 2018-12-06

Changed

- Various fixed bugs and papercuts, e.g., in group handling, kubernetes execution, singularity support, wrapper and script usage, benchmarking, schema validation.

4.28.83 [5.3.0] - 2018-09-18

Added

- Snakemake workflows can now be exported to CWL via the flag –export-cwl, see the docs.

Changed

- Fixed bug in script and wrapper execution when using `--use-singularity --use-conda`.
- Add host argument to S3RemoteProvider.
- Various minor bug fixes.

[5.2.4] - 2018-09-10

Added

- New command line flag –shadow-prefix

Changed

- Fixed permission issue when using the script directive. This is a breaking change for scripts referring to files relative to the script directory (see the docs).
- Fixed various minor bugs and papercuts.
- Allow URL to local git repo with wrapper directive (`git+file:///path/to/your/repo/path_to_file@@version`)

[5.2.2] - 2018-08-01

Changed

- Always print timestamps, removed the –timestamps CLI option.
- more robust detection of conda command
- Fixed bug in RMarkdown script execution.
- Fixed a bug in detection of group jobs.

4.28.84 [5.2.0] - 2018-06-28

Changed

- Directory outputs have to marked with `directory`. This ensures proper handling of timestamps and cleanup. This is a breaking change. Implemented by Rasmus Ågren.
- Fixed kubernetes tests, fixed kubernetes volume handling. Implemented by Andrew Schriefer.
- jinja2 and networkx are not optional dependencies when installing via pip.
- When conda or singularity directives are used and the corresponding CLI flags are not specified, the user is notified at the beginning of the log output.
- Fixed numerous small bugs and papercuts and extended documentation.

[5.1.5] - 2018-06-24

Changed

- fixed missing version info in docker image.
- several minor fixes to EGA support.

[5.1.4] - 2018-05-28

Added

- Allow `category` to be set.

Changed

- Various cosmetic changes to reports.
- Fixed encoding issues in reports.

[5.1.3] - 2018-05-22

Changed

- Fixed various bugs in job groups, shadow directive, singularity directive, and more.

[5.1.2] - 2018-05-18

Changed

- Fixed a bug in the report stylesheet.

4.28.85 [5.1.0] - 2018-05-17

Added

- A new framework for self-contained HTML reports, including results, statistics and topology information. In future releases this will be further extended.
- A new utility snakemake.utils.validate() which allows to validate config and pandas data frames using JSON schemas.
- Two new flags –cleanup-shadow and –cleanup-conda to clean up old unused conda and shadow data.

Changed

- Benchmark repeats are now specified inside the workflow via a new flag repeat().
- Command line interface help has been refactored into groups for better readability.

4.28.86 [5.0.0] - 2018-05-11

Added

- Group jobs for reduced queuing and network overhead, in particular with short running jobs.
- Output files can be marked as pipes, such that producing and consuming job are executed simultaneously and interfomation is transferred directly without using disk.
- Command line flags to clean output files.
- Command line flag to list files in working directory that are not tracked by Snakemake.

Changed

- Fix of –default-remote-prefix in case of input functions returning lists or dicts.
- Scheduler no longer prefers jobs with many downstream jobs.

[4.8.1] - 2018-04-25

Added

- Allow URLs for the conda directive. # Changed
- Various minor updates in the docs.
- Several bug fixes with remote file handling.
- Fix ImportError occuring with script directive.
- Use latest singularity.
- Improved caching for file existence checks. We first check existence of parent directories and cache these results. By this, large parts of the generated FS tree can be pruned if files are not yet present. If files are present, the overhead is minimal, since the checks for the parents are cached.
- Various minor bug fixes.

4.28.87 [4.8.0] - 2018-03-13

Added

- Integration with CWL: the `cwl` directive allows to use CWL tool definitions in addition to shell commands or Snakemake wrappers.
- A global `singularity` directive allows to define a global singularity container to be used for all rules that don't specify their own.
- Singularity and Conda can now be combined. This can be used to specify the operating system (via singularity), and the software stack (via conda), without the overhead of creating specialized container images for workflows or tasks.

4.28.88 [4.7.0] - 2018-02-19

Changed

- Speedups when calculating dry-runs.
- Speedups for workflows with many rules when calculating the DAG.
- Accept SIGTERM to gracefully finish all running jobs and exit.
- Various minor bug fixes.

4.28.89 [4.6.0] - 2018-02-06

Changed

- Log files can now be used as input files for other rules.
- Adapted to changes in Kubernetes client API.
- Fixed minor issues in –archive option.
- Search path order in scripts was changed to fix a bug with leaked packages from root env when using script directive together with conda.

[4.5.1] - 2018-02-01

Added

- Input and output files can now tag pathlib objects. # ## Changed
- Various minor bug fixes.

4.28.90 [4.5.0] - 2018-01-18

Added

- iRODS remote provider # ## Changed
- Bug fix in shell usage of scripts and wrappers.
- Bug fixes for cluster execution, –immediate-submit and subworkflows.

4.28.91 [4.4.0] - 2017-12-21

Added

- A new shadow mode (minimal) that only symlinks input files has been added.

Changed

- The default shell is now bash on linux and macOS. If bash is not installed, we fall back to sh. Previously, Snakemake used the default shell of the user, which defeats the purpose of portability. If the developer decides so, the shell can be always overwritten using shell.executable().
- Snakemake now requires Singularity 2.4.1 at least (only when running with –use-singularity).
- HTTP remote provider no longer automatically unpacks gzipped files.
- Fixed various smaller bugs.

[4.3.1] - 2017-11-16

Added

- List all conda environments with their location on disk via –list-conda-envs.

Changed

- Do not clean up shadow on dry-run.
- Allow R wrappers.

4.28.92 [4.3.0] - 2017-10-27

Added

- GridFTP remote provider. This is a specialization of the GFAL remote provider that uses globus-url-copy to download or upload files. # ## Changed
- Scheduling and execution mechanisms have undergone a major revision that removes several potential (but rare) deadlocks.
- Several bugs and corner cases of the singularity support have been fixed.
- Snakemake now requires singularity 2.4 at least.

4.28.93 [4.2.0] - 2017-10-10

Added

- Support for executing jobs in per-rule singularity images. This is meant as an alternative to the conda directive (see docs), providing even more guarantees for reproducibility.

Changed

- In cluster mode, jobs that are still running after Snakemake has been killed are automatically resumed.
- Various fixes to GFAL remote provider.
- Fixed –summary and –list-code-changes.
- Many other small bug fixes.

4.28.94 [4.1.0] - 2017-09-26

Added

- Support for configuration profiles. Profiles allow to specify default options, e.g., a cluster submission command. They can be used via 'snakemake –profile myprofile'. See the docs for details.
- GFAL remote provider. This allows to use GridFTP, SRM and any other protocol supported by GFAL for remote input and output files.
- Added –cluster-status flag that allows to specify a command that returns jobs status. # ## Changed
- The scheduler now tries to get rid of the largest temp files first.
- The Docker image used for kubernetes support can now be configured at the command line.
- Rate-limiting for cluster interaction has been unified.
- S3 remote provider uses boto3.
- Resource functions can now use an additional `attempt` parameter, that contains the number of times this job has already been tried.
- Various minor fixes.

4.28.95 [4.0.0] - 2017-07-24

Added

- Cloud computing support via Kubernetes. Snakemake workflows can be executed transparently in the cloud, while storing input and output files within the cloud storage (e.g. S3 or Google Storage). I.e., this feature does not need a shared filesystem between the cloud notes, and thereby makes the setup really simple.
- WebDAV remote file support: Snakemake can now read and write from WebDAV. Hence, it can now, e.g., interact with Nextcloud or Owncloud.
- Support for default remote providers: define a remote provider to implicitly use for all input and output files.
- Added an option to only create conda environments instead of executing the workflow. # ## Changed
- The number of files used for the metadata tracking of Snakemake (e.g., code, params, input changes) in the .snakemake directory has been reduced by a factor of 10, which should help with NFS and IO bottlenecks. This is a breaking change in the sense that Snakemake 4.x won't see the metadata of workflows executed with Snakemake 3.x. However, old metadata won't be overwritten, so that you can always go back and check things by installing an older version of Snakemake again.
- The google storage (GS) remote provider has been changed to use the google SDK. This is a breaking change, since the remote provider invocation has been simplified (see docs).
- Due to WebDAV support (which uses asyncio), Snakemake now requires Python 3.5 at least.

- Various minor bug fixes (e.g. for dynamic output files).

[3.13.3] - 2017-06-23

Changed

- Fix a followup bug in Namedlist where a single item was not returned as string.

[3.13.2] - 2017-06-20

Changed

- The –wrapper-prefix flag now also affects where the corresponding environment definition is fetched from.
- Fix bug where empty output file list was recognized as containing duplicates (issue #574).

[3.13.1] - 2017-06-20

Changed

- Fix –conda-prefix to be passed to all jobs.
- Fix cleanup issue with scripts that fail to download.

4.28.96 [3.13.0] - 2017-06-12

Added

- An NCBI remote provider. By this, you can seamlessly integrate any NCBI resouce (reference genome, gene/protein sequences, …) as input file. # ## Changed
- Snakemake now detects if automatically generated conda environments have to be recreated because the workflow has been moved to a new path.
- Remote functionality has been made more robust, in particular to avoid race conditions.
- `--config` parameter evaluation has been fixed for non-string types.
- The Snakemake docker container is now based on the official debian image.

4.28.97 [3.12.0] - 2017-05-09

Added

- Support for RMarkdown (.Rmd) in script directives.
- New option –debug-dag that prints all decisions while building the DAG of jobs. This helps to debug problems like cycles or unexpected MissingInputExceptions.
- New option –conda-prefix to specify the place where conda environments are stored.

Changed

- Benchmark files now also include the maximal RSS and VMS size of the Snakemake process and all sub processes.
- Speedup conda environment creation.
- Allow specification of DRMAA log dir.
- Pass cluster config to subworkflow.

[3.11.2] - 2017-03-15

Changed

- Fixed fix handling of local URIs with the wrapper directive.

[3.11.1] - 2017-03-14

Changed

- –touch ignores missing files
- Fixed handling of local URIs with the wrapper directive.

4.28.98 [3.11.0] - 2017-03-08

Added

- Param functions can now also refer to threads. # ## Changed
- Improved tutorial and docs.
- Made conda integration more robust.
- None is converted to NULL in R scripts.

[3.10.2] - 2017-02-28

Changed

- Improved config file handling and merging.
- Output files can be referred in params functions (i.e. lambda wildcards, output: …)
- Improved conda-environment creation.
- Jobs are cached, leading to reduced memory footprint.
- Fixed subworkflow handling in input functions.

4.28.99 [3.10.0] - 2017-01-18

Added

- Workflows can now be archived to a tarball with `snakemake --archive my-workflow.tar.gz`. The archive contains all input files, source code versioned with git and all software packages that are defined via conda environments. Hence, the archive allows to fully reproduce a workflow on a different machine. Such an archive can be uploaded to Zenodo, such that your workflow is secured in a self-contained, executable way for the future. # ## Changed
- Improved logging.
- Reduced memory footprint.
- Added a flag to automatically unpack the output of input functions.
- Improved handling of HTTP redirects with remote files.
- Improved exception handling with DRMAA.
- Scripts referred by the script directive can now use locally defined external python modules.

[3.9.1] - 2016-12-23

Added

- Jobs can be restarted upon failure (–restart-times). # ## Changed
- The docs have been restructured and improved. Now available under snakemake.readthedocs.org.
- Changes in scripts show up with –list-code-changes.
- Duplicate output files now cause an error.
- Various bug fixes.

4.28.100 [3.9.0] - 2016-11-15

Added

- Ability to define isolated conda software environments (YAML) per rule. Environments will be deployed by Snakemake upon workflow execution.
- Command line argument –wrapper-prefix in order to overwrite the default URL for looking up wrapper scripts. # ## Changed
- –summary now displays the log files correspoding to each output file.
- Fixed hangups when using run directive and a large number of jobs
- Fixed pickling errors with anonymous rules and run directive.
- Various small bug fixes

[3.8.2] - 2016-09-23

Changed

- Add missing import in rules.py.
- Use threading only in cluster jobs.

[3.8.1] - 2016-09-14

Changed

- Snakemake now warns when using relative paths starting with "./".
- The option -R now also accepts an empty list of arguments.
- Bug fix when handling benchmark directive.
- Jobscripts exit with code 1 in case of failure. This should improve the error messages of cluster system.
- Fixed a bug in SFTP remote provider.

4.28.101 [3.8.0] - 2016-08-26

Added

- Wildcards can now be constrained by rule and globally via the new `wildcard_constraints` directive (see the docs).
- Subworkflows now allow to overwrite their config file via the configfile directive in the calling Snakefile.
- A method `log_fmt_shell` in the snakemake proxy object that is available in scripts and wrappers allows to obtain a formatted string to redirect logging output from STDOUT or STDERR.
- Functions given to resources can now optionally contain an additional argument `input` that refers to the input files.
- Functions given to params can now optionally contain additional arguments `input` (see above) and `resources`. The latter refers to the resources.
- It is now possible to let items in shell commands be automatically quoted (see the docs). This is usefull when dealing with filenames that contain whitespaces.

Changed

- Snakemake now deletes output files before job exection. Further, it touches output files after job execution. This solves various problems with slow NFS filesystems.
- A bug was fixed that caused dynamic output rules to be executed multiple times when forcing their execution with -R.
- A bug causing double uploads with remote files was fixed. Various additional bug fixes related to remote files.
- Various minor bug fixes.

[3.7.1] - 2016-05-16

Changed

- Fixed a missing import of the multiprocessing module.

4.28.102 [3.7.0] - 2016-05-05

Added

- The entries in `resources` and the `threads` job attribute can now be callables that must return `int` values.
- Multiple `--cluster-config` arguments can be given to the Snakemake command line. Later one override earlier ones.
- In the API, multiple `cluster_config` paths can be given as a list, alternatively to the previous behaviour of expecting one string for this parameter.
- When submitting cluster jobs (either through `--cluster` or `--drmaa`), you can now use `--max-jobs-per-second` to limit the number of jobs being submitted (also available through Snakemake API). Some cluster installations have problems with too many jobs per second.
- Wildcard values are now printed upon job execution in addition to input and output files. # ## Changed
- Fixed a bug with HTTP remote providers.

[3.6.1] - 2016-04-08

Changed

- Work around missing RecursionError in Python < 3.5
- Improved conversion of numpy and pandas data structures to R scripts.
- Fixed locking of working directory.

4.28.103 [3.6.0] - 2016-03-10

Added

- onstart handler, that allows to add code that shall be only executed before the actual workflow execution (not on dryrun).
- Parameters defined in the cluster config file are now accessible in the job properties under the key "cluster".
- The wrapper directive can be considered stable. # ## Changed
- Allow to use rule/job parameters with braces notation in cluster config.
- Show a proper error message in case of recursion errors.
- Remove non-empty temp dirs.
- Don't set the process group of Snakemake in order to allow kill signals from parent processes to be propagated.
- Fixed various corner case bugs.
- The params directive no longer converts a list `l` implicitly to `" ".join(l)`.

[3.5.5] - 2016-01-23

Added

- New experimental wrapper directive, which allows to refer to re-usable wrapper scripts. Wrappers are provided in the Snakemake Wrapper Repository.
- David Koppstein implemented two new command line options to constrain the execution of the DAG of job to sub-DAGs (–until and –omit-from). # ## Changed
- Fixed various bugs, e.g. with shadow jobs and –latency-wait.

[3.5.4] - 2015-12-04

Changed

- The params directive now fully supports non-string parameters. Several bugs in the remote support were fixed.

[3.5.3] - 2015-11-24

Changed

- The missing remote module was added to the package.

[3.5.2] - 2015-11-24

Added

- Support for easy integration of external R and Python scripts via the new script directive.
- Chris Tomkins-Tinch has implemented support for remote files: Snakemake can now handle input and output files from Amazon S3, Google Storage, FTP, SFTP, HTTP and Dropbox.
- Simon Ye has implemented support for sandboxing jobs with shadow rules.

Changed

- Manuel Holtgrewe has fixed dynamic output files in combination with multiple wildcards.
- It is now possible to add suffixes to all shell commands with shell.suffix(“mysuffix”).
- Job execution has been refactored to spawn processes only when necessary, resolving several problems in combination with huge workflows consisting of thousands of jobs and reducing the memory footprint.
- In order to reflect the new collaborative development model, Snakemake has moved from my personal bitbucket account to http://snakemake.bitbucket.org.

[3.4.2] - 2015-09-12

Changed

- Willem Ligtenberg has reduced the memory usage of Snakemake.
- Per Unneberg has improved config file handling to provide a more intuitive overwrite behavior.
- Simon Ye has improved the test suite of Snakemake and helped with setting up continuous integration via Codeship.
- The cluster implementation has been rewritten to use only a single thread to wait for jobs. This avoids failures with large numbers of jobs.
- Benchmarks are now writing tab-delimited text files instead of JSON.
- Snakemake now always requires to set the number of jobs with -j when in cluster mode. Set this to a high value if your cluster does not have restrictions.
- The Snakemake Conda package has been moved to the bioconda channel.
- The handling of Symlinks was improved, which made a switch to Python 3.3 as the minimum required Python version necessary.

[3.4.1] - 2015-08-05

Changed

- This release fixes a bug that caused named input or output files to always be returned as lists instead of single files.

4.28.104 [3.4] - 2015-07-18

Added

- This release adds support for executing jobs on clusters in synchronous mode (e.g. qsub -sync). Thanks to David Alexander for implementing this.
- There is now vim syntax highlighting support (thanks to Jay Hesselberth).
- Snakemake is now available as Conda package.

Changed

- Lots of bugs have been fixed. Thanks go to e.g. David Koppstein, Marcel Martin, John Huddleston and Tao Wen for helping with useful reports and debugging.

See here for older changes.

4.29 License

Snakemake is licensed under the MIT License: