BioLizard

Welcome to the

NEXTFLOW WORKSHOP

Tuur Muyldermans: tuur.muyldermans@lizard.bio
Steff Taelman: steff.taelman@lizard.bio

section.lead h1 {
text-align: center;
}

Welcome to the nextflow workshop

Tuur Muyldermans - tuur.muyldermans@lizard.bio
Steff Taelman - steff.taelman@lizard.bio

Objective

- Understand Nextflow syntax
- Understand workflow pipelines
- Write simple pipelines yourself!

Overview:

- Introduction
- Basic concepts: processes, channels and operators
- Creating our first Nextflow script(s)
- Managing configurations: parameters, portability, execution
- Creating reports

1. Building blocks

In the first chapter we will elaborate on how Nextflow is designed, its advantages and disadvantages, the basic components, etc.

Bash scripts

```
#!/bin/bash

...

# Download each fasta read sequence file into the directory
for file in $LIST; do
    echo "Downloading $file"
        wget -P ../data -np ${rawdatalink}/$file
done

...
```

Workflow managers

“ **Nextflow** is a reactive workflow framework and a programming Domain Specific Language that eases the writing of data-intensive computational pipelines. ”

$$\begin{tikzcd} \text{Java} \arrow[r] & \text{Groovy} \arrow[r] & \text{nextflow} \end{tikzcd}$$

Alternatives: link

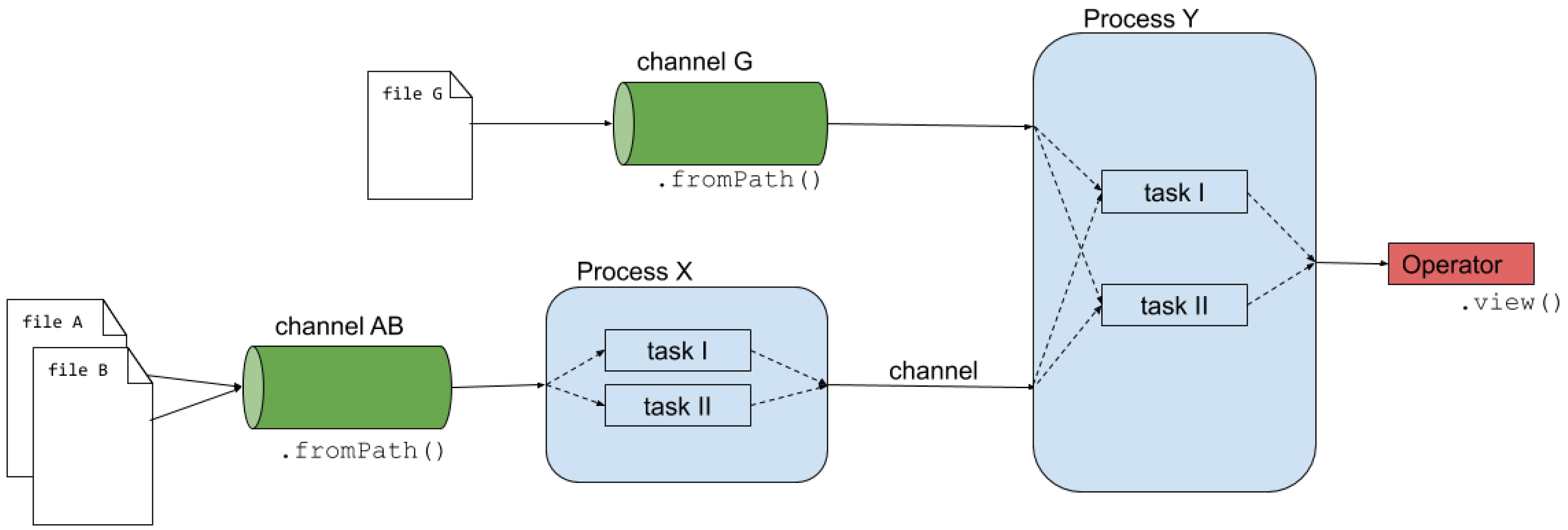
Why Nextflow?

- Parallelization
- Scalability
- Portability
- Reproducible
- Continuous checkpoints
- Modularity
- Community

Why not?

- Syntax of the Groovy language, yet another language
- Flexibility also comes with cost of complexity
- Nitpicking details in failure of scripts

2. Basic concepts



```
#!/usr/bin/env nextflow
nextflow.enable.dsl=2

// Creating channels
numbers_ch = Channel.from(1,2,3)
strings_ch = Channel.from('a','b')

// Defining the process that is executed
process valuesToFile {
    input:
    val nums
    val strs

    output:
    path 'result.txt'

    """
    echo $nums and $strs > result.txt
    """
}

// Running a workflow with the defined processes
workflow{
    valuesToFile(numbers_ch, strings_ch)
}
```

to communicate with each other

- Channels can be used by operators or serve as an input for the processes

```
# Channel consisting of strings
strings_ch = Channel.from('This', 'is', 'a', 'channel')

# Channel consisting of a single file
file_ch = Channel.fromPath('data/sequencefile.fastq')

# Channel consisting of multiple files by using a wildcard *
multfiles_ch = Channel.fromPath('data/*.fastq')

# Channel consisting of multiple paired-end files by using wildcard * and opti
paired_ch = Channel.fromFilePairs('data/*{1,2}.fastq')
```

Further reading: Nextflow's documentation.

The input of the analysis is stored in a channel, these

2.2 Operators

- Transform content of channels
- A plethora of operators exists, only a handful used extensively
- Examples: `.view()`, `.ifEmpty()`, `.splitFasta()`, `.print()`, etc. etc. etc.

- `collect` : e.g. when using a channel consisting of multiple independent files (e.g. fastq-files) and need to be assembled for a next process.

```
Channel
    .from( 1, 2, 3, 4 )
    .collect()
    .view()

# outputs
[1,2,3,4]
```

Further reading: Nextflow's documentation

- `mix` : e.g. when assembling items from multiple channels into one channel for a next process (e.g. multiqc)

```
c1 = Channel.from( 1,2,3 )
c2 = Channel.from( 'a','b' )
c3 = Channel.from( 'z' )

c1 .mix(c2,c3)

# outputs
1
2
3
'a'
'b'
'z'
```

Further reading: Nextflow's documentation represent each individual subpart of the analysis and contain hence one of the (many) processes in a pipeline: think about fastqc, trimmomatic, star or hisat alignment, counting. In the code-snippet below, you can see that it consists of a couple of blocks: directives, input, output, when clause and the script.

```
process < name > {

   [ directives ]

   input:
    < process inputs >

   output:
    < process outputs >

   when:
```

Processes

- Executed independently
- Isolated from any other process
- FIFO queues

Each process is executed independently and isolated from any other process. They communicate via asynchronous FIFO queues, i.e. one process will wait for the output of another and then runs reactively when the channel has contents.

then given to the output.

Output:

When we run this script, the result file will not be present in our folder structure. Question: look at the output... Can you guess where to find the result?

```
N E X T F L O W  ~  version 20.07.1
Launching `02-basic-concepts/firstscript.nf` [elegant_curie] - revision: 9f886
executor >  local (2)
executor >  local (2)
[5e/195314] process > valuesToFile (2) [100%] 2 of 2 ✓
results file: /path/to/work/51/7023ee62af2cb4fdd9ef654265506a/result.txt
results file: /path/to/work/5e/195314955591a705e5af3c3ed0bd5a/result.txt
```

The output consists of:

- Version of nextflow

```
-... user group    0 Nov 26 15:20 .command.out*
-... user group 3187 Nov 26 15:20 .command.run*
-... user group   53 Nov 26 15:20 .command.sh*
-... user group    3 Nov 26 15:20 .exitcode*
```

- .exitcode, contains 0 if everything is ok, another value if there was a problem.
- .command.log, contains the log of the command execution. Often is identical to .command.out
- .command.out, contains the standard output of the command execution
- .command.err, contains the standard error of the command execution
- .command.begin, contains what has to be executed before .command.sh

```
N E X T F L O W  ~  version 20.07.1
Launching `02-basic-concepts/fifo.nf` [nauseous_mahavira] - revision: a71d904c
[-        ] process > whosfirst [  0%] 0 of 2
This is job number 6
This is job number 3
This is job number 7
This is job number 8
This is job number 5
This is job number 4
This is job number 1
This is job number 2
This is job number 9
executor >  local (10)
[4b/aff57f] process > whosfirst (10) [100%] 10 of 10
```

Earlier, we described that Nextflow uses an asynchronous FIFO principle. Let's exemplify this by running the script `02-basic-consepts/fifo.nf` and inspect the order that the channels are being

language (bash, Python, Perl, Ruby, etc.). This allows to add self-written scripts in the pipeline.
They can be defined in the process script itself as given in the example here, or they could also exist as a script in another folder and be run here like python script.py

- Any language (bash, Python, Perl, Ruby, etc.)
- Defined in the process or command to run the script

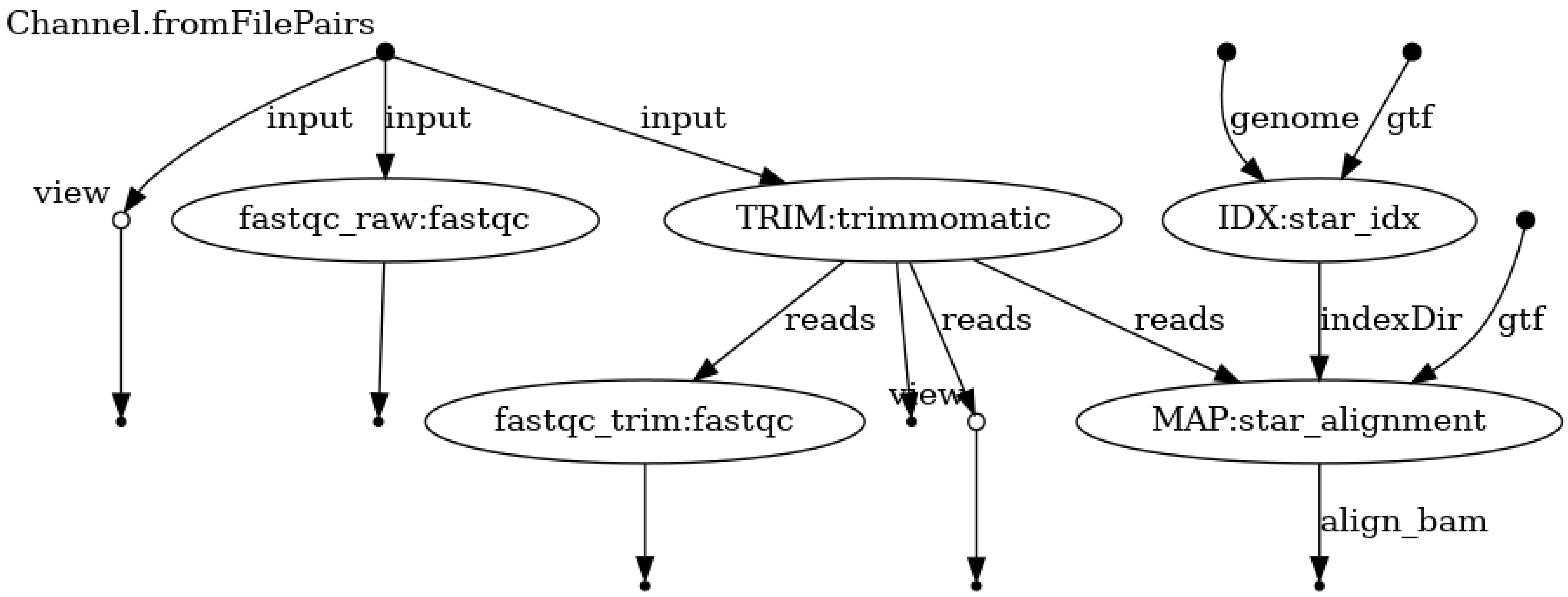
```
#!/usr/bin/env nextflow

process python {

    """
    #!/usr/bin/python3
```

3. Creating our first pipeline

In this chapter we will build a basic RNA-seq pipeline consisting of quality controls, trimming of reads and mapping to a reference genome (excl. counting). We will build the pipeline step by step, starting from quality control with FastQC. We will start with DSL1 for this process. After exploring Nextflow's flexibility on the quality control process, we will switch to the newer DSL2 language and extend our pipeline script with the other processes


Channel.fromFilePairs
input
input
input
view
fastqc_raw:fastqc
TRIM:trimmomatic
genome
gtf
IDX:star_idx
gtf
reads
reads
reads
indexDir
fastqc_trim:fastqc
view
MAP:star_alignment
align_bam

- Assign input into `params`
- Comments are always useful
- Create a channel for input files - serve as input for process
- Operator on a channel `.view()`

The first line of our script is always a shebang line, declaring the environment where the OS can find the software (i.e. Nextflow). Generally, the input files and parameters of the processes are first assigned into *parameters* which allows flexibility in the pipeline. Input files are then assigned to channels and they serve as input for the process.

Implicit parallellisation:
Run the following and keep an eye on the workload distribution (`htop`).

```
nextflow run 03-first-pipeline/fastqc_1.nf
```

Let's add some new features:

1. `nextflow run 03-first-pipeline/fastqc_1.nf -bg > log`

ANSWER: run in the background and push output of nextflow to the log file. No need of explicitly using nohup, screen or tmux.

ANSWER: the hash at the beginning of each process reveals where you can find the result of each process.

Let's add some new features:

1. `nextflow run 03-first-pipeline/fastqc_1.nf -bg > log`
2. Adapt file for handling read pairs. Which parameter would you use on runtime to overwrite the inputfiles?

ANSWER: nextflow run --reads data/WT_lib1_R1.fq.gz 03-first-pipeline/fastqc_1.nf .

Let's add some new features:

1. `nextflow run 03-first-pipeline/fastqc_1.nf -bg > log`
2. Adapt file for handling read pairs. Which parameter would you use on runtime to overwrite the inputfiles?
3. Print parameters using `println` & check if the files exist when creating the channels.

ANSWER: println section to describe your workflow

ANSWER: checkIfExists

ANSWER: invoke the checkIfExists-error by running the nextflow script with wrong reads: `nextflow run 03-first-pipeline/fastqc_3.nf --reads wrongfilename` publishDir .

ANSWER: If the output is to be used by another process, and the files are being moved, they won't be accessible for the next process and hence you're pipeline will fail complaining about files not being present. Files published by a process must not be accessed by other downstream processes

DIFFERENT MODES:

- Without any additional arguments, a hyperlink will be created to the files stored in the `work/` directory, with mode set to copy (`mode: 'copy'`) the files will

3.2 Moving towards DSL2

- Make the pipelines more modular
- Simplify the writing of complex data analysis pipelines

- Introduction of `workflow` & modules

Here is a list of the major changes:

- Following the shebang line, the nf-script wil start with the following line: `nextflow.enable.dsl=2` (not to be mistaken with *preview*).
- When using DSL1 each channel could only be consumed once, this is ommited in DSL2. Once created, a channel can be consumed indefinitely.
- A process on the other hand can still only be used once in DSL2
- A new term is introduced: `workflow`. In the workflow, the processes are called as functions with

Quality control with `FastQC` (DSL2)

Inspect: `03-first-pipeline/dsl2-fastqc.nf`

```
// Running a workflow with the defined processes here.
workflow {
        read_pairs_ch.view()
        fastqc(read_pairs_ch)
}
```

Trimming with `trimmomatic` (DSL2)

Inspect: `03-first-pipeline/dsl2-trimming.nf`

- Introducing: output and `emit`.
- Accessing an output of a process with `<processname>.out.<emitname>` Defining a process output with `emit` allows us to use it as a channel in the external scope.

```
nextflow run 03-first-pipeline/dsl2-trimming.nf
nextflow run 03-first-pipeline/dsl2-trimming.nf
```

Output:

```
Error: Process fastqc has been already used --
If you need to reuse the same component include it with a
different name or include in a different workflow context
```

At this point we're interested in the result of the `trimmomatic` process. Hence, we want to verify the quality of the reads with another `fastqc` process. Re-run `fastqc` on the filtered read sequences by adding it in the workflow of `03-first-pipeline/dsl2-trimming.nf`. Use the parameter `-resume` to restart the pipeline from where it stopped the last time.

- Sub-workflows with workflow keyword definition
- Allows use of sub-workflows within workflow
- Main workflow does not carry a name and is executed implicitly

The workflow keyword (which is basically a name for a workflow) allows the definition of **sub-workflow** components that enclose the invocation of one or more processes and operators. It also allows you to use this workflow from within another workflow. The workflow that does not cary any name is considered to be the main workflow and will be executed implicitly. An alternative workflow entry can be specified using

```
  arg3

  main:
  star_index(arg1, arg2)
  star_alignment(arg1, arg2, arg3)
}

workflow hisat2{
  take:
  arg1
  arg2

  main:
  hisat_index(arg1)
  hisat_alignment(arg1, arg2)
}

workflow {
  star(arg1, arg2, arg3)
  hisat2(arg1, arg2)
}
```

`fastqc.nf` . This script consists of a process and a workflow. This module can be imported into our pipeline script (main workflow) like this:

- Write components in a module (component = process, workflow, functions)
- Import a specific component from that module:

```
include {QC} from './modules/fastqc.nf'
```

This line is quite specific. The workflow is defined within the curly brackets, the origin of the module defined by a relative path must start with `./` .

Example: `modules/fastqc.nf` .

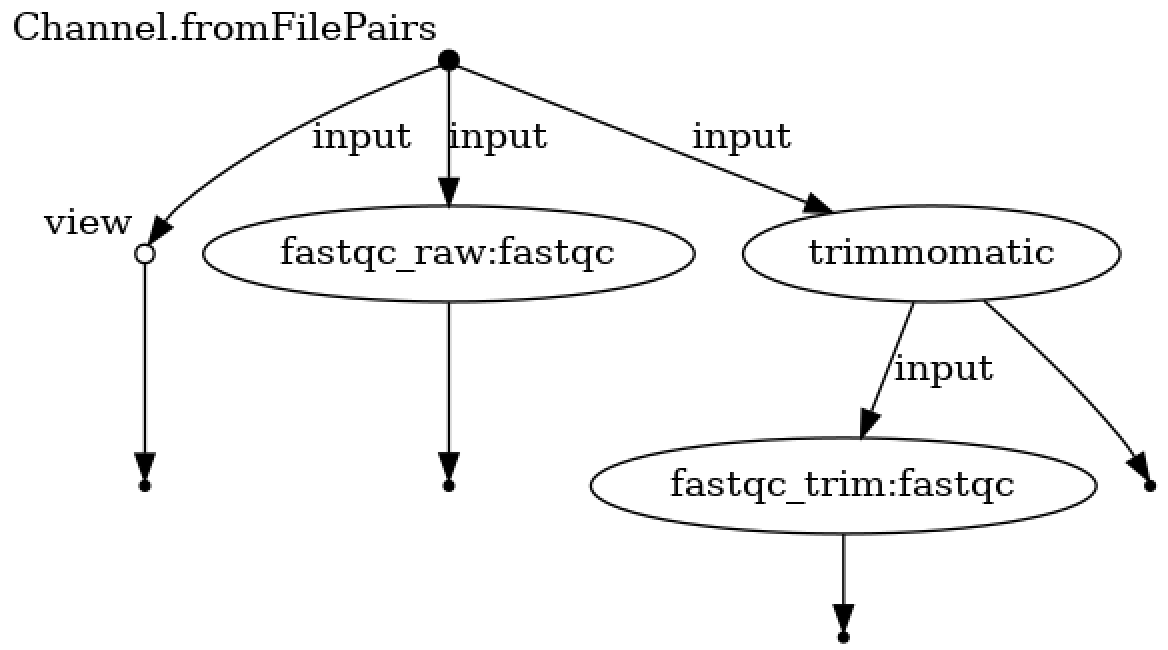
Inspect `03-first-pipeline/dsl2-subworkflow.nf`:

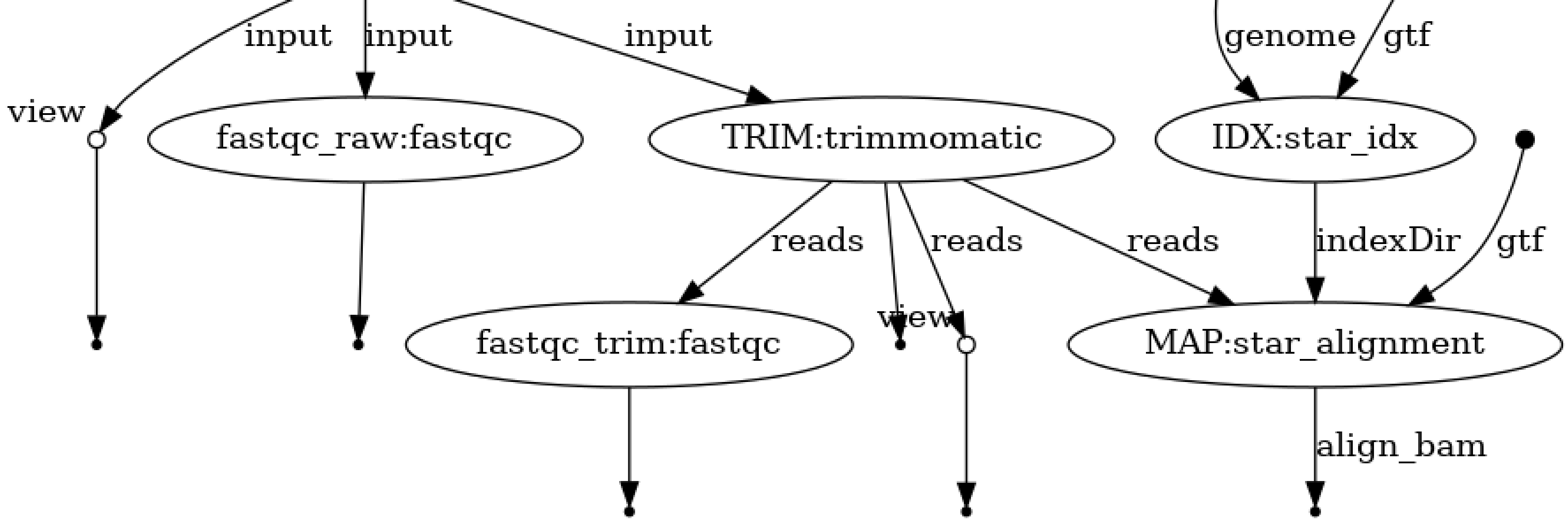
- Fastqc process has been removed from this script
- Fastqc is imported from `modules/fastqc.nf`
- Trimmomatic process is still internally described

```
nextflow run 03-first-pipeline/dsl2-subworkflow.nf
```

Run `03-first-pipeline/dsl2-subworkflow.nf` which contains the `trimmomatic` process internally and imports the `fastqc` process from the modules library `module/fastqc.nf`.

`03-first-pipeline/dsl2-subworkflow.nf` :





Similarly as described above, we can extend this pipeline and map our trimmed reads on a reference genome. First, we'll have to index our reads and afterwards we can map our reads. In the folder `modules/` find the script `star.nf` which contains two processes: `star_index` and `star_alignment`.

These modules are called from the main script `03-`

4. Managing configurations

trimmomatic or STAR)

- Separate these variables from pipeline - more modular

```
params.reads = "$launchDir/data/*{1,2}.fq.gz"

process {
    memory='1G'
    cpus='1'
}
```

Pipeline configuration properties are defined in a file named `nextflow.config` in the pipeline execution directory. This file can be used to define technical and project depeding parameters, e.g. which executor to use, the processes' environment variables, pipeline

```
// Define technical resources below:
process {
    withLabel: 'low' {
        memory='1G'
        cpus='1'
        time='6h'
    }
    withLabel: 'med' {
        memory='2G'
        cpus='2'
    }
    withLabel: 'high' {
        memory = '8G'
        cpus='8'
    }
}
```

It's also possible to create labels that can be chosen and used for each process separately. In the example

- Separate analysis parameteres in a separate file

```
includeConfig "/path/to/params.config"
```

Imagine that you want to separate analysis parameters in a separate file, this is possible by creating a `params.config` file and include it in the `nextflow.config` file as such:

Portability & reproducibility

- Support for Conda, Docker & Singularity

As discussed before, Nextflow is especially useful thanks to its portability and reproducibility, i.e. the native support for containers and environment managers. There are two options for attaching containers to your pipeline. Either you define a dedicated container image for each process individually, or you define one container for all processes together in the configurations file.

where Nextflow will search for the existence of this container if it doesn't exist locally.

1. In process directives:

```
process quality-control {
    container 'biocontainers/fastqc:v0.11.9_cv7'

    """
    fastqc ...
    """
}
```

2. In `nextflow.config` file:

```
process.container = 'vibbioinfocore/analysispipeline:latest'
```

We're referring to a Docker container image that exists

- Or add the following to `nextflow.config` -file:
  `docker.enabled = true`

Note: to set the correct user- and group-settings: `docker.runOptions = '-u \$(id -u):\$(id -g)'`

Ultimately, the parameter `-with-docker` does not need to be defined and it should use the Docker container in the background at all times, for this purpose also use the `docker.enabled = true` option in the config file. Another interesting parameter to consider addin to the configuration file is the `docker.runOptions = '-u \$(id -u):\$(id -g)'`. This

Similarly with a singularity image for which you do not have to adapt the pipeline script. You can run with Singularity container using the following command-line parameter: `-with-singularity [singularity-image-file]`, where the image is downloaded from Dockerhub as well, built on runtime and then stored in a folder `singularity/`. Re-using a singularity image is possible with:

Or extend `nextflow.config` -file with:

```
singularity.cacheDir = "/path/to/singularity" // centralised caching directory
process.container = 'singularity.img' // define the image
singularity.enabled = true  // enable running with singularity
```

If you want to avoid entering the Singularity image as kubernetes amazon web services Google Cloud Platform

While a *process* defines *what* command or script has to be executed, the *executor* determines *how* that script is actually run on the target system. In the Nextflow framework architecture, the executor is the component that determines the system where a pipeline process is run and it supervises its execution.

If not otherwise specified, processes are executed on the local computer. The local executor is very useful for pipeline development and testing purposes, but for real world computational pipelines on HPC or cloud

5. Creating reports

Nextflow has an embedded function for reporting a number of informations about the resources needed by each job and the timing. Just by adding a parameter on run-time, different kind of reports can be created.

```
nextflow run example.nf -with-docker -with-report
```

This report describes the usage of resources and job durations and give an indication of bottlenecks in the pipeline.

2. DAG: visualization of the pipeline (dependency: graphviz)
   Use the option `-with-dag` to create a visualization of the workflow. By default it will create a `.dot`-file that contains a description of the workflow, however use e.g. `rnaseq.PNG` as an argument to create a figure. This visualization is a nice overview of the

Questions

Further reading & references:

- Nextflow's official documentation (link)
- Reach out to the community on Gitter (link)
- Curated collection of patterns (link)
- Workshop focused on DSL2 developed by CRG Bioinformatics Core (link)
- Tutorial exercises (DSL1) developed by Seqera (link)
- Curated ready-to-use analysis pipelines by NF-core (link)
- Model example pipeline on Variant Calling Analysis