Session Johnson & Johnson - 04 December 2020

Tuur Muyldermans - tuur.muyldermans@vib.be
Alexander Botzki - alexander.botzki@vib.be

Overview:

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

1. Introduction

Bash scripts

```
#!/bin/bash

blastp -query sample.fasta -outfmt 6 \
        | head -n 10 \
        | cut -f 2 \
        | blastdbcmd -entry - > sequences.txt
```

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

Why (not)? (1/2)

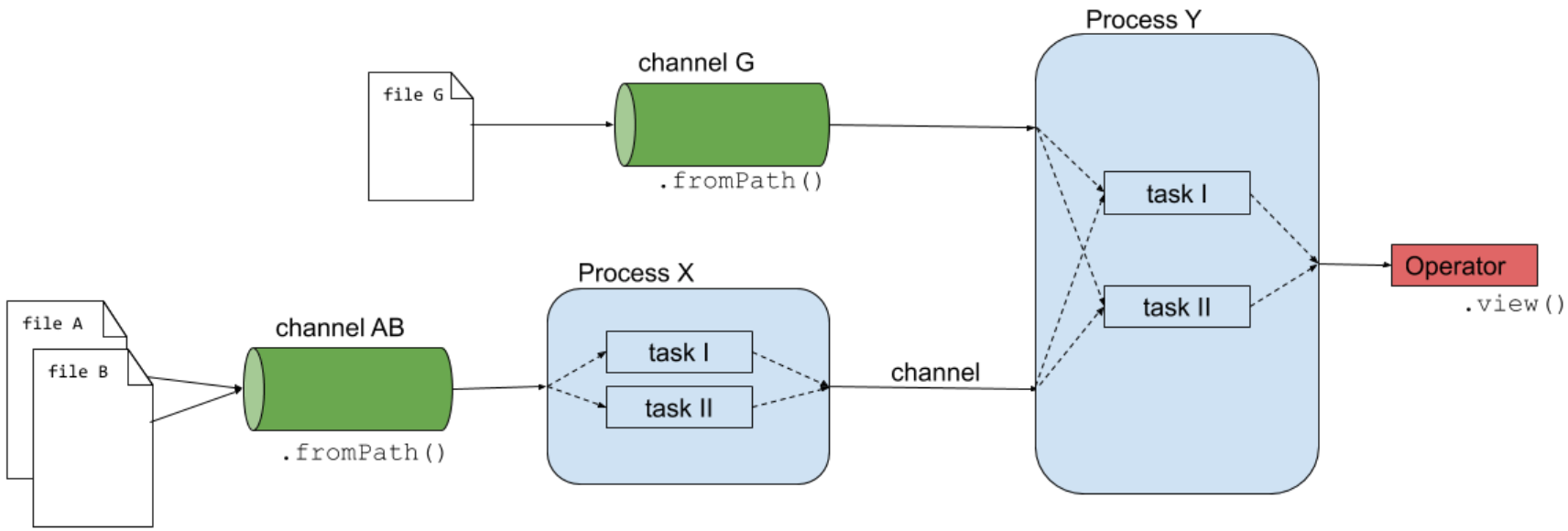
- Parallelization: processes are automatically scheduled based on available resources
- Scalability: simple scaling from local to HPC-cluster usage
- Portability: run across different platforms
- Reproducible: native support for containers, conda environments, and interaction with Git.
- Continuous checkpoints for resuming / expanding pipelines (which is usually the case for workflow pipelines)
- Re-usability: DSL2 and its modules will allow re-using of other scripts
- Community: nf-core, Gitter, etc.

Why (not)? (2/2)

Alternatives: link

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

2. Basic concepts


file G
channel G
.fromPath()
Process Y
task I
task II
Operator
.view()
file A
file B
channel AB
.fromPath()
Process X
task I
task II
channel

Example: inspect `02-basic-concepts/firstscript.nf`

2.1 Channels

- Input of the analysis is stored in a channel (files, strings, numbers, etc.)
- unidirectional async queues that allows the processes 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 options {x,y}
paired_ch = Channel.fromFilePairs('data/*{1,2}.fastq')
```

Further reading: Nextflow's documentation.

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

2.3 Processes

```
process < name > {

   [ directives ]

   input:
    < process inputs >

   output:
    < process outputs >

   when:
    < condition >

   [script|shell|exec]:
   < user script to be executed >

}
```

Processes

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

2.4 Running our first pipeline:

```
nextflow run firstscript.nf
```

Output:

```
N E X T F L O W  ~  version 20.07.1
Launching `02-basic-concepts/firstscript.nf` [elegant_curie] - revision: 9f886cc00a
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
```

Output-files stored in the work-directory.

Besides the output, also a bunch of hidden `.command.*` files are present:

```
-... user group    0 Nov 26 15:20 .command.begin*
-... user group 1797 Nov 26 15:20 .command.err*
-... user group 1826 Nov 26 15:20 .command.log*
-... 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*
```

FIFO-principle

```
nextflow run 02-basic-consepts/fifo.nf
```

Output:

```
N E X T F L O W  ~  version 20.07.1
Launching `02-basic-concepts/fifo.nf` [nauseous_mahavira] - revision: a71d904cf6
[-        ] 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
```

Self-written scripts

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

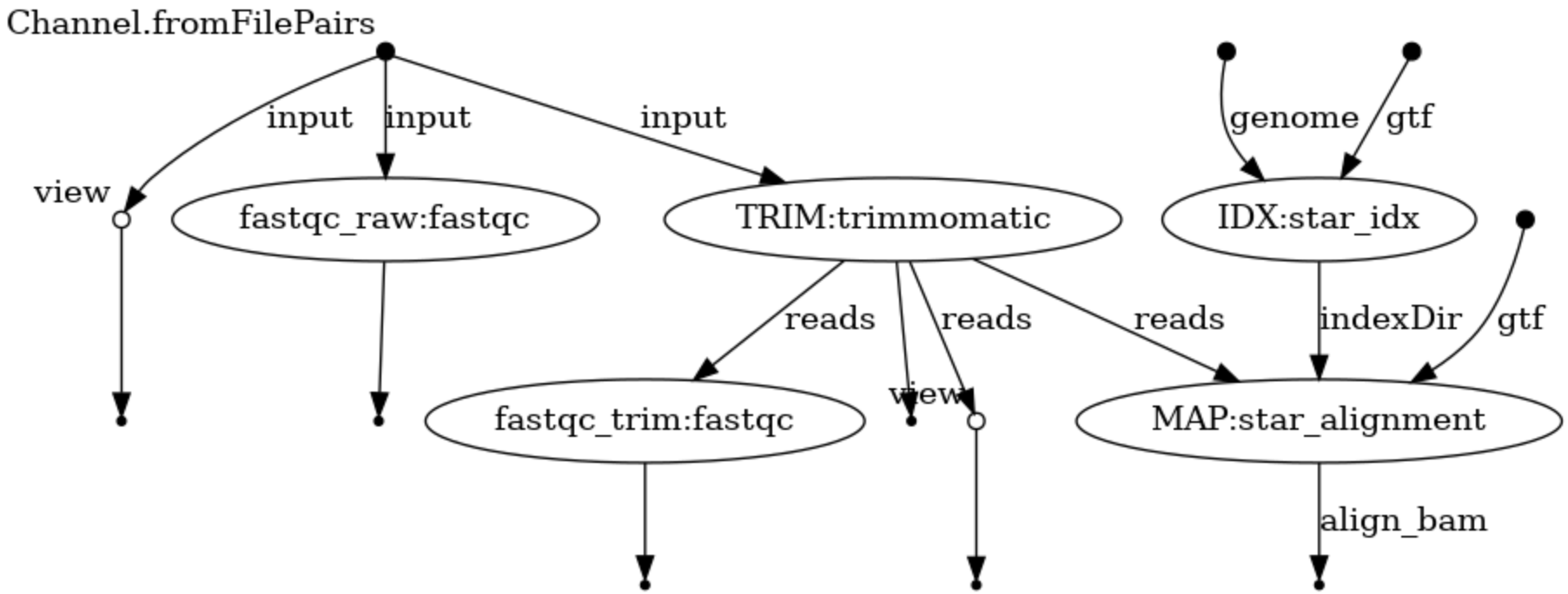
```
#!/usr/bin/env nextflow

process python {

    """
    #!/usr/bin/python3

    firstWord = 'hello'
    secondWord = 'folks'
    print(f'{firstWord} {secondWord}')
    """
}
```

3. Creating our first pipeline


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

3.1 Using DSL1 (example: FastQC)

Inspect: `03-first-pipeline/fastqc_1.nf`

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

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`

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?

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.

Let's add some new features:

1. `nextflow run 03-first-pipeline/fastqc_1.nf -bg > log`. Where are the output files?
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.
4. Create a directory where the files can be stored with `publishDir`.

3.2 Moving towards DSL2

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

Overview of the changes (DSL2 vs DSL1):

- `nextflow.enable.dsl=2`
- In DSL2:
  - Channel can be used indefinitely (vs only once in DSL1)
  - Process can (still) only be used once
  - DSL2 separates the definition of a process from its invocation
  - Within processes no more references to channels (i.e. `from` and `into`)
- Introduction of `workflow` & modules

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>`

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

Quality control on trimmed reads with `FastQC` (DSL2)

Rerun and uncomment the last fastqc-process in the workflow:

```
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
```

3.3 Sub-workflows

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

```
workflow star{
  take:
  arg1
  arg2
  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)
}
```

3.4 Modules

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

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

Example: `modules/fastqc.nf`.

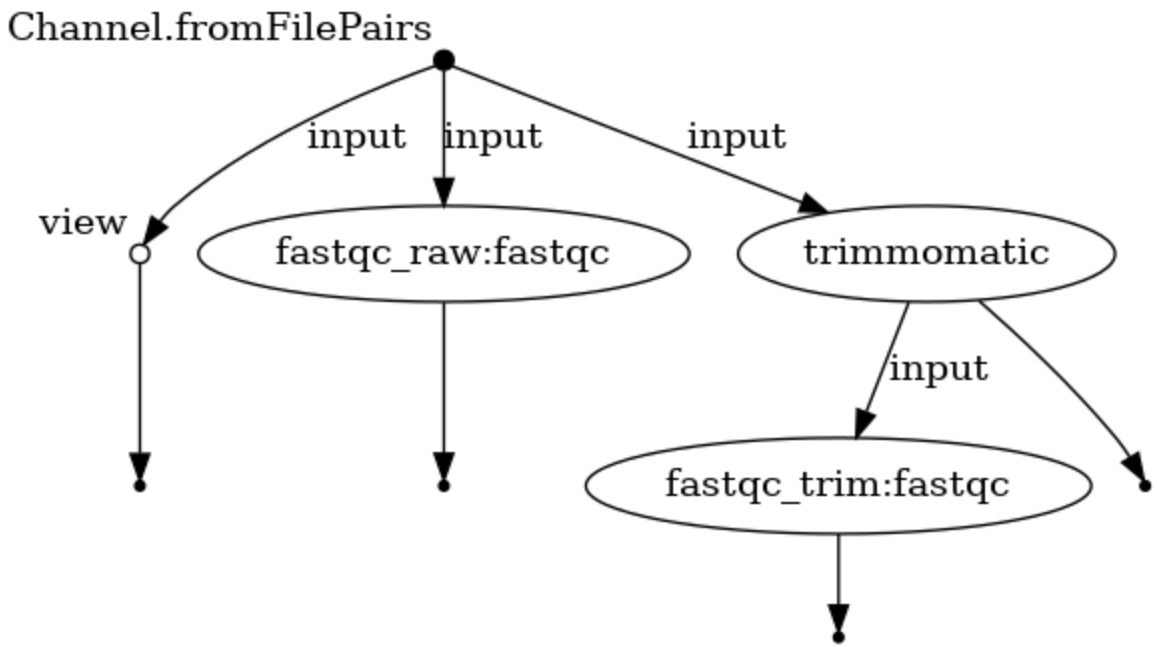
```
include { QC as fastqc_raw; QC as fastqc_trim } from "${launchDir}/modules/fastqc")
```

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

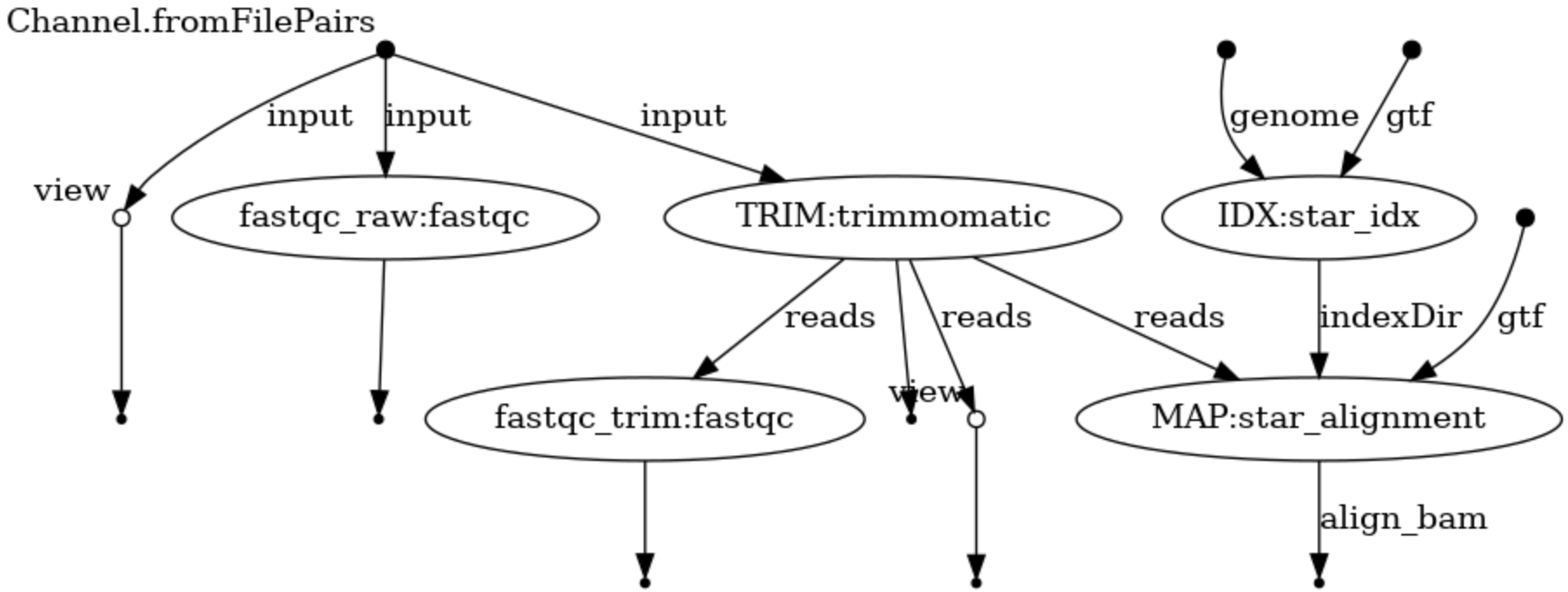
- Fastqc process has been removed from this script
- Fastqc is imported from `modules/fastqc.nf`
- Trimmomatic process is still internally described

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

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



RNAseq workflow example



4. Managing configurations

Configuration files

- Pipeline configuration properties are defined in `nextflow.config`
- Technical parameters (profiles):
  - Executor, CPUs, memory, etc.
- Pipeline specific parameters:
  - Input files, process related parameters (e.g. trimmomatic or STAR)
- Separate these variables from pipeline - more modular

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

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

- Add labels that allow different resources for a particular process

```
// 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'
    }
}
```

- Separate analysis parameteres in a separate file

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

Portability & reproducibility

- Support for Conda, Docker & Singularity

Using Docker:

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'
```

- Run pipeline with Docker container: `nextflow run example.nf -with-docker`
- 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)'`

Using Singularity:

```
nextflow run example.nf -with-singularity [singularity-image-file]
```

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
```

Executors


Schedulers
GRID ENGINE
Sun Grid Engine
(SGE)
slurm
workload manager
LSF
Platform Load Sharing
Facility
PBS Works™
Portable Batch System
HTCondor
High Throughput Computing
Cloud platforms
kubernetes
amazon
web services™
Google Cloud Platform

5. Creating reports

1. Workflow report (html)

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

2. DAG: visualization of the pipeline (dependency: graphviz)

```
nextflow run example.nf -with-dag <filename.PNG>
```

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 with NGS RNA-Seq data developed by CRG (link)