
containers-workflows-workshop

Release 0.2

KrisDavie, kobelavaerts, tmuylder

Mar 29, 2023

CONTENTS

1	General context	3
2	Practical information	5
3	Objectives	7
4	Prerequisites	9
5	Requirements	11
6	Exercises	13
6.1	Installations	13
6.2	Containers	14
6.3	Nextflow	14

Welcome to our Nextflow workshop! We are very happy to have you here.

GENERAL CONTEXT

This is the third edition of this workshop, jointly organised by the VIB Bioinformatics Core and ELIXIR Belgium.

- The first session (13 & 14 March 2023) is dedicated to Containers (Docker & Singularity) which are great tools for code portability and reproducibility of your analysis. You will learn how to use containers and how to build a container from scratch, share it with others and how to re-use and modify existing containers.
- The second session (30 & 31 March 2023) is focused on Nextflow for building scalable and reproducible bioinformatics pipelines and running them on a personal computer, cluster and cloud. Starting from the basic concepts we will build our own simple pipeline and add new features with every step, all in the new DSL2 language. On the second day, we will utilise all the gathered knowledge to build a small-scale microbiomics pipeline.

This website contains the course materials and outline for the second session.

The presentation which goes alongside this material can be found [here](#).

PRACTICAL INFORMATION

Schedule day 1:

- 9:30 - 11:00 - session
- 11:00 - 11:15 - break
- 11:15 - 12:45 - session
- 12:45 - 13:45 - lunch
- 13:45 - 15:15 - session
- 15:15 - 15:30 - break
- 15:30 - 17:00 - session

We aim to complete up to and including exercise 2.5 during this day

Schedule day 2:

- 9:30 - 11:00 - session
- 11:00 - 11:15 - break
- 11:15 - 12:45 - session
- 12:45 - 13:45 - lunch
- 13:45 - 17:00 - project

OBJECTIVES

The objectives of the Nextflow workshop are the following:

- Understand Nextflow's basic concepts & syntax: channels, processes, modules, workflows, etc.
- Execute local and publicly available pipelines with different executors and environments
- Write and run Nextflow pipelines
- Write and modify config files for storing parameters related to computing hardware as well as pipeline dependent parameters

PREREQUISITES

Being comfortable working with the CLI (command-line interface) in a Linux-based environment.

REQUIREMENTS

The (technical) installation requirements are described in the [installations](#) section.

EXERCISES

The exercises and solutions are available in [this GitHub repository](#).

6.1 Installations

Please read this page carefully **before** the start of the workshop.

There are two options for following this workshop: (1) do the installations yourself & be in control of everything, (2) use the VMs that we have created with the installations already done. In the former case, you will have to download [Nextflow](#), [Docker](#) and [Singularity](#). In the latter case, you can follow the instructions below.

6.1.1 Provided infrastructure

We have created a VM with Azure Labs for each participant. This means that each participant has access to a dedicated Azure Labs VM. This is an Ubuntu 18.04 virtual machine containing all the software needed for the workshop. Each VM can be accessed during the workshop hours and for an extra 10 hours. This extra time allows you to experiment with the materials in your spare time.

There are several ways for connecting to the VMs, however we advise to connect with the VM through a SSH connection in VSCode following these instructions:

- Download Visual Studio Code ([link](#))
- Add the following extensions for a seamless integration of Nextflow and the VM in VScode:
 - In VSCode, navigate to the ‘Extensions’ tab, search for the following packages and install them:
 - ‘Remote - SSH’ (ms-vscode-remote.remote-ssh).
 - ‘Nextflow’ (nextflow.nextflow)
 - ‘Docker’ (ms-azuretools.vscode-docker).
- Create the connection to the VM:
 - In VScode, navigate to the new ‘Remote Explorer’ icon on the left side bar, find the dropdown menu ‘WSL Targets’ and switch to ‘SSH Targets’.
 - Click on ‘+’, in the pop-up navigation bar and paste the SSH key (see below) & follow the instructions.
- To obtain the SSH keys:
 - Navigate to the registration URL of Microsoft Azure Lab that you received in your mailbox and select or make a Microsoft account

- Find the VM *containers-workflow-2022* in your Azure Labs, start the Lab and choose a password. After some waiting, you can copy the SSH key.
- An elaborate description is available [here](#).

In a VScode Terminal (select Terminal and then New Terminal), test the infrastructure with the following command `nextflow -h`. The server will be empty, however we will populate it with the course materials during the workshop.

6.2 Containers

The course materials (presentations) are available [here](#).

The exercises are available in [this GitHub repository](#). In order to get the data, run the `download-data.sh` script in the `data/` folder.

6.3 Nextflow

This tutorial aims to get you familiarized with Nextflow. After this course you should be able to understand workflow pipelines that are written in Nextflow and write simple pipelines yourself! Here's an overview of the materials that we will cover:

- General introduction to Nextflow
- Building blocks of Nextflow: processes, channels and operators, workflows and modules
- Executing pipelines
- Creating our first Nextflow script(s)
- Managing configurations: parameters, portability, executors

6.3.1 Building blocks

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

In the `data/` folder we have already installed some data for you to use in the following exercises.

Introduction

Writing pipelines to automate processes is not something new, Bash scripts are probably one of the oldest forms of pipelines where we concatenate processes. Let's have a look at an example:

```
#!/bin/bash

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

Starting with a shebang line, the `blastp` command is piped through multiple times to eventually result in an output file `sequences.txt`.

Question

What is the downside of similar relatively simple pipelines?

Solution

There are a couple of suboptimal things happening here:

- Will it use the available resources optimally?
- Which versions of the tools are being used?
- Will it work on my machine (cfr. installation of tools)?
- Can we scale it to HPC clusters or Cloud environments?
- What if the pipeline fails somewhere in the middle, we need to restart the pipeline from the beginning?

In response to that, workflow managers such as Nextflow were built, aimed to deal with more complex situations. Nextflow is designed around the idea that Linux has many simple but powerful command-line and scripting tools that, when chained together, facilitate complex data manipulations.

By definition, Nextflow is a reactive workflow framework and a programming Domain Specific Language that eases the writing of data-intensive computational pipelines[1]. Nextflow scripting is an extension of the Groovy programming language, which in turn is a super-set of the Java programming language. Groovy can be considered as Python for Java in a way that simplifies the writing of code and is more approachable.



Why (not)?

Nextflow is not the only player in the field[2], however there are good reasons to opt for it.

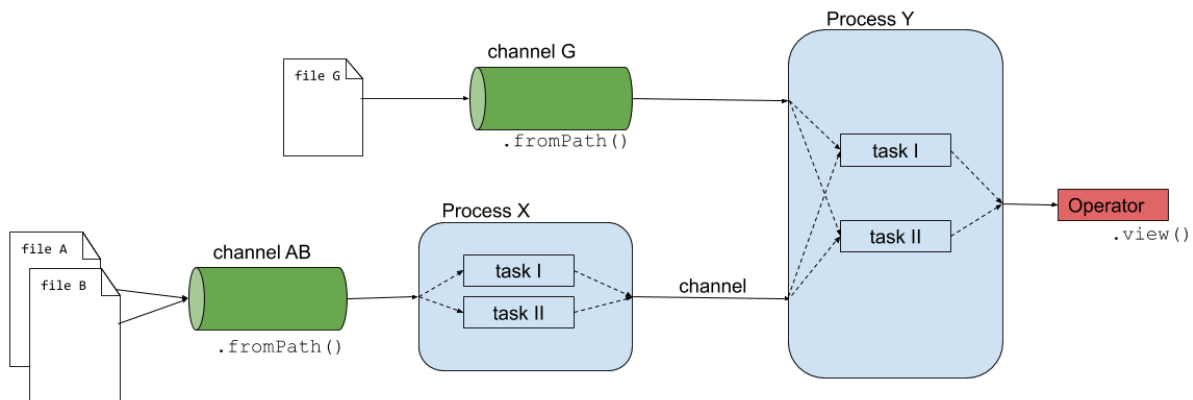
- Parallelization: processes are automatically scheduled based on available resources
- Scalability: simple scaling from local to HPC-cluster usage
- Portability: run across different platforms
- Reproducible: native support for containers, conda environments, and interaction with Git.
- Re-usability: with the introduction of modules it becomes (theoretically) simple to re-use processes written in other pipelines
- Community[3]: even though the community is never a reason why to choose for a tool (functionality is more important), it is still very relevant to know that when you are facing problems, there are people out there ready to help you out.

Some thoughts or disadvantages from my personal point of view. It takes some time to get used to the syntax of the Groovy language. As flexible as it is, as complex it gets. Often it's difficult to trace down the exact problem of a failure of a pipeline script, especially in the beginning. It's probably not the first thing you should be concerned of if you're doing a one-time analysis.

Main abstractions

Nextflow consists of four main components: channels, operators, processes and workflows.

- *Channels*: contain the input of the workflows used by the processes. Channels connect processes/operators with each other.
- *Operators*: transform the content of channels by applying functions or transformations. Usually operators are applied on channels to get the input of a process in the right format.
- *Processes*: define the piece of script that is actually being run (e.g. an alignment process with STAR).
- *Workflows*: call the processes as functions with channels as input arguments, only processes defined in the workflow are run. Workflows were introduced in DSL2.



The script `exercises/01_building_blocks/firstscript.nf` is using these three components and gives an idea of how Nextflow scripts are being build.

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

// Creating channels
numbers_ch = Channel.of(1,2,3)
strings_ch = Channel.of('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 {
```

(continues on next page)

(continued from previous page)

```
valuesToFile(numbers_ch, strings_ch)
}
```

Note: Besides these main building blocks, we also already highlight the existence of the `params` parameters. In the previous code block we explicitly defined some input values in the channels. However, we can define the input values into a parameter instead, that is passed on to the channel.

```
// create a parameter 'input_read'
params.input_read = '/path/to/read_1.fq'

// use the input_read parameter as an input for the channel
input_read_ch = Channel.fromPath(params.input_read)
```

Here `params.input_read = '/path/to/read_1.fq'` will create a parameter `input_read` and give it the value `'/path/to/read_1.fq'` which is used as an input for the channel. We will later see that these parameters can then be overwritten on runtime.

1. Channels

The input of the analysis is stored in a channel, these are generally files like sequencing, reference fasta, annotation files, etc. however the input can be of any kind like numbers, strings, lists, etc. To have a complete overview, we refer to the official documentation[4]. Here are some examples of how a channel is being created:

```
# Channel consisting of strings
strings_ch = Channel.of('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')
```

These channels can then be used by operators or serve as an input for the processes.

Exercise 1.1

Inspect and edit the `exercises/01_building_blocks/template.nf` script. Create a channel consisting of multiple paired-end files. For more information, read [fromFilePairs](#).

Once the Nextflow script is saved, run it with: `nextflow run exercises/01_building_blocks/template.nf`.

Solution 1.1

The solution is available in the file `exercises/01_building_blocks/solutions/1.1_template-paired-end.nf`.

Note that the content of the channel is constructed in a following manner:

```
[common-name, [/path/to/read1.fq, /path/to/read2.fq]]
```

This is a **tuple** qualifier which we will use a lot during this workshop and discuss later again.

2. Operators

Operators are necessary to transform the content of channels in a format that is necessary for usage in the processes. There is a plethora of different operators[5], however only a handful are used extensively. Here are some examples that you might come across:

- **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 (output in a list data-type).

Example: `exercises/01_building_blocks/operator_collect.nf`

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

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

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

Example: `exercises/01_building_blocks/operator_mix.nf`

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

c1 .mix(c2,c3)
  .view()

# possible output
a
1
2
b
3
z
```

- `map`: e.g. when you would like to run your own function on each item in a channel.
 - The map operator is expressed as a `closure` (`{ ... }`)
 - By default, the items in the channel are referenced by the variable `it`. This can be changed by using the `map { item -> ... }` syntax.
 - All functions available on the item, are available on the `it` variable within the closure.
 - When an element is a list or tuple, you can use the `it[0]`, `it[1]`, etc. syntax to access the individual elements of your item.

Example: `exercises/01_building_blocks/operator_map.nf`

```
Channel
.of( 1, 2, 3, 4, 5 )
.map { it * it }
.subscribe onNext: { println it }, onComplete: { println 'Done' }

# outputs
1
4
9
16
25
Done
```

Exercise 1.2

Create a channel from a csv-file (`input.csv`) and use an operator to view the contents. Generate the channel for the `input.csv`-file which you can find in the `exercises/01_building_blocks/` folder and contains the following content:

sampleId	Read 1	Read 2
01	data/ggal_gut_1.fq.gz	data/ggal_gut_2.fq.gz
02	data/ggal_liver_1.fq.gz	data/ggal_liver_2.fq.gz

Test your Nextflow script with: `nextflow run <name>.nf`.

Solution 1.2

The solution is available in the file `exercises/01_building_blocks/solutions/1.2_template-csv.nf`

The file is imported with `.fromPath()`, followed by the `splitCsv()` operator where we set the header to `True`. The last step will output how the channels are constructed. Each row is transformed into a tuple with the first element as a variable `sampleId`, the second as `forward_read` and the third as `reverse_read`.

```
samples_ch = Channel
    .fromPath('exercises/01_building_blocks/input.csv') // make sure that
    ↳ the path towards the file is correct
    .splitCsv(header:true)
```

Exercise 1.3

Building on exercise 1.2 and using the map operator, create 2 channels, one containing the sampleId and the forward read as a tuple and the second containing the sampleId and reverse read as a tuple. Use the view operator to inspect the contents of these channels.

Solution 1.3

The solution is available in the file `exercises/01_building_blocks/solutions/1.3_template-csv-map.nf`

3. Processes

Processes are the backbone of the pipeline. They represent each individual subpart of the analysis. In the code-snippet below, you can see that it consists of a couple of blocks: directives, input, output, when-clause and the script itself.

```
process < name > {  
  
    [ directives ]  
  
    input:  
    < process inputs >  
  
    output:  
    < process outputs >  
  
    when:  
    < condition >  
  
    [script|shell|exec]:  
    < user script to be executed >  
}
```

Here are a couple of examples of processes:

Writing a file

Creating an output file `results.txt` with inputs from channels `nums` and `strs`

```
process valuesToFile {  
    input:  
    val nums  
    val strs  
  
    output:  
    path 'result.txt'  
  
    script:  
    """  
    echo $nums and $strs > result.txt  
    """  
}
```

(continues on next page)

(continued from previous page)

```

    """
}

```

FastQC

Quality control process with fastqc

```

process fastqc {
  input:
  tuple val(sample), path(reads)

  output:
  path("*_fastqc.{zip,html}")

  script:
  """
  fastqc ${reads}
  """
}

```

Salmon

Quantifying in mapping-based mode with salmon

```

process salmon_quant {
  input:
  path index
  tuple val(pair_id), path(reads)

  output:
  path pair_id

  script:
  """
  salmon quant --threads $task.cpus --libType=U -i $index -1 ${reads[0]} -2 ${reads[1]}
  ↪ -o $pair_id
  """
}

```

Trimming & quality filtering reads

Trimming adapters & quality filtering with trimmomatic

```

process trimmomatic {
  // directives
  publishDir "$params.outdir/trimmed-reads", mode: 'copy', overwrite: true
  label 'low'
  container 'quay.io/biocontainers/trimmomatic:0.35--6'
}

```

(continues on next page)

(continued from previous page)

```

input:
tuple val(sample), path(reads)

output:
tuple val("${sample}"), path("${sample}*_P.fq"), emit: trim_fq
tuple val("${sample}"), path("${sample}*_U.fq"), emit: untrim_fq

script:
"""
    trimmomatic PE -threads $params.threads ${reads[0]} ${reads[1]} ${sample}1_P.fq $
↪ ${sample}1_U.fq ${sample}2_P.fq ${sample}2_U.fq $params.slidingwindow $params.avgqual
"""
}

```

The **input** declaration block defines the channels where the process expects to receive its data. The input definition starts with an input qualifier followed by the input name ([more information](#)). The most frequently used qualifiers are `val`, `path` and `tuple`, respectively representing a value (e.g. numbers or strings), a path towards a file and a combination of input values having one of the available qualifiers (e.g. tuple containing a value and two files).

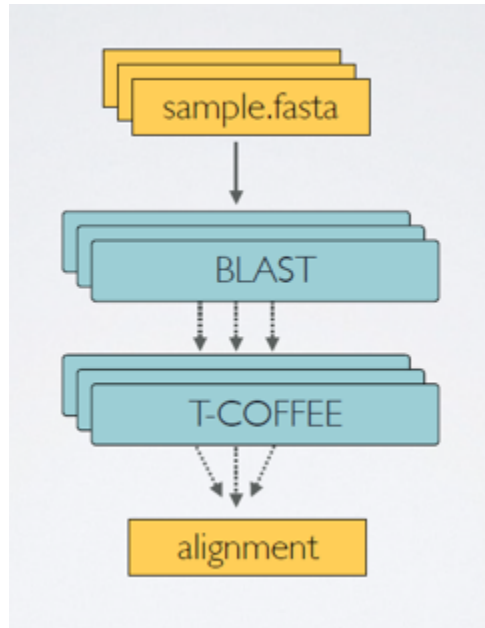
Warning: The keyword `from` is a remainder of DSL1 and is not used in DSL2. Therefore we can neglect this keyword in this course even though we will see it appears a lot in older tutorials (and in the official Nextflow documentation).

The **output** declaration block defines the channels created by the process to send out the results produced. They are build similar as the input declarations, using a qualifier (e.g. `val`, `path` and `tuple`) followed by the generated output. The output of a process usually serves as the input of another process, hence with the `emit` option we can make a name identifier that can be used to reference the output (as a channel) in the external scope. In the `trimmomatic` example we can access the generated filtered and trimmed paired reads in the external scope as such: `trimmomatic.out.trim_fq`.

Directives are defined at the top of the process (see `trimmomatic` example) and can be any of the [following long list of possibilities](#). We can define the directory where the outputs should be published, add labels or tags, define containers used for the virtual environment of the process, and much more. We will discover some of the possibilities along the way.

Conditionals are not considered in this course.

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.



Let's exemplify this by running the script `exercises/01_building_blocks/fifo.nf` and inspect the order that the channels are being processed.

```

NEXTFLOW ~ version 20.10.0
Launching `fifo.nf` [nauseous_mahavira] - revision: a71d904cf6
[-          ] process > whosfirst -
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
  
```

A script, as part of the process, can be written in any language (bash, Python, Perl, Ruby, etc.). This allows to add self-written scripts in the pipeline. The script can be written in the process itself, or can be present as a script in another folder and is run from the process here. An example can be found in `exercises/01_building_blocks/hellofrompython.nf`.

```

#!/usr/bin/env nextflow

process python {

    script:
    """
    #!/usr/bin/env python3

    firstWord = 'hello'
  
```

(continues on next page)

(continued from previous page)

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

Check the output of the script in the `.command.out` file of the work-directory.

Note: The work-directory of the last process can be seen in the output of nextflow.

[f6/4916cd] process > python [100%] 1 of 1 ✓

In this case, the output would be in the directory starting `work/f6/4916cd...`

Exercise 1.4

A tag directive can be added at the top of the process definition and allows you to associate each process execution with a custom label. Hence, it is really useful for logging or debugging. Add a tag for `num` and `str` in the process of the script `exercises/01_building_blocks/firstscript.nf` and inspect the output.

Solution 1.4

The process should be adapted, containing the following tag line in the directives.

```
// Defining the process that is executed
process valuesToFile {
    tag "$nums,$strs"

    input:
    val nums
    val str

    output:
    path 'result.txt'

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

When you execute the pipeline, the processes overwrite into one line and it is not very clear in which hashed work directory the outputs are. Therefore, you can use the following to follow the execution of your pipeline:

```
nextflow run exercises/01_building_blocks/firstscript.nf -bg > nf.log
tail -f nf.log
```

4. Workflows

Defining processes will not produce anything, because you need another part that actually calls the process and connects it to the input channel. Thus, in the `workflow`, the processes are called as functions with input arguments being the channels.

The output that is generated in a process, needs to be emitted (`emit`) in order to serve as an input for a next process. The `trimmomatic` process defined above emits the paired trimmed and unpaired trimmed (not passing the filtering thresholds) reads as two separate outputs, `trim_fq` and `untrim_fq` respectively. The following workflow calls the `trimmomatic` process with `reads` as its input channel. Now we can access the output of this process using `trimmomatic.out.trim_fq`.

```
workflow {
    trimmomatic(reads)
}
```

Extra exercises

Extra exercise 1

Use the view operator on the output of the `valuesToFile` process in the script `exercises/01_building_blocks/firstscript.nf`. For this, you will first need to add an `emit` argument to the output of the process. More information is available in the documentation [here](#).

Solution 1

```
...
process ...
    output:
    path 'result.txt', emit: result_ch
...

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

Extra exercise 2

You need to execute a hypothetical task for each record in a CSV file. Write a Nextflow script containing the following:

1. Create a channel for the input (`input.csv`):
 - Read the CSV file line-by-line using the `splitCsv` operator, then use the `map` operator to return a tuple with the required field for each line. Finally use the resulting channel as input for the process.
2. Create a process that:
 - Accepts a tuple as input channel with the information from the csv-file.

- Has the following script: `echo your_command --sample $sampleId --reads $read1 $read2`

3. Create a workflow that calls the process with the input channel.

Given the file `input.csv` (in the exercises folder) with the following content:

sampleId	Read 1	Read 2
01	data/ggal_gut_1.fq.gz	data/ggal_gut_2.fq.gz
02	data/ggal_liver_1.fq.gz	data/ggal_liver_2.fq.gz

Solution 2

Find the solution also in `split-csv.nf`. Inspect the command that has ran in the intermediate `work/` directory following the hashed folders and look in the file `.command.sh`.

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

params.input_csv = 'input.csv'

samples_ch = Channel
    .fromPath(params.input_csv)
    .splitCsv(header:true)
    .map{ row -> tuple(row.sampleId, file(row.forward_read), file(row.
↪reverse_read)) }

process split_csv {
    input:
        tuple val(sampleId), file(read1), file(read2)

    script:
        """
        echo your_command --sample $sampleId --reads $read1 $read2
        """
}

workflow {
    samples_ch.view()
    split_csv(samples_ch)
}
```

Futher reading on DSL2

Nextflow recently went through a big make-over. The premise of the next version, using DSL2, is to make the pipelines more modular and simplify the writing of complex data analysis pipelines.

Here is a list of the major changes:

- Since version 22.03.0-edge, Nextflow defaults to using DSL2 and you do not need to manually enable it.
- When using DSL1 each channel could only be consumed once, this is omitted 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 input arguments being the channels.
- Regarding the processes, the new DSL separates the definition of a process from its invocation. This means that in DSL1 the process was defined and also run when the script was invoked, however in DSL2, the definition of a process does not necessarily mean that it will be run.
- Moreover, within processes there are no more references to channels (i.e. **from** and **into**). The channels are passed as inputs to the processes which are defined and invoked in the **workflow**.

6.3.2 Executing pipelines

Executing our first pipeline

If we want to run a Nextflow script in its most basic form, we will use the following command:

```
nextflow run <pipeline-name.nf>
```

with `<pipeline-name.nf>` the name of our pipeline, e.g. `exercises/02_run_first_script/firstscript.nf`. Inspect the script `firstscript.nf` again and notice how the channels and process are being created, how the workflow calls the process as a function with the channels as input arguments, how they are passed on as the processes' inputs, to the script section and then given to the output.

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

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

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

    output:
        path 'result.txt', emit: result_ch

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

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

Nextflow will generate an output that has a standard lay-out:

```
N E X T F L O W ~ version 22.04.5
Launching `exercises/02_run_first_script/firstscript.nf` [distracted_almeida] DSL2 -
↪revision: 1a87b5fe26
```

(continues on next page)

(continued from previous page)

```
executor > local (2)
[eb/9af3b0] process > valuesToFile (2) [100%] 2 of 2 ✓
/home/training/git/nextflow-workshop/work/c8/b5f6c2d2a5932f77d5bc53320b8a5d/result.txt
/home/training/git/nextflow-workshop/work/eb/9af3b0384ef96c011b4da69e86fca7/result.txt
```

The output consists of:

- Version of nextflow
- Information regarding the script that has ran with an identifier name
- Hash with process ID, progress and caching information
- Optional output printed to the screen as defined in the script (if present)

Question

When we run this script, the result file will not be present in our folder structure. Where will the output of this script be stored?

The results are stored in the results file as described in the two last lines. By default the results of a process are stored in the `work/` directory in subfolders with names defined by the hashes. Besides the output that we generated, also a bunch of hidden `.command.*` files are present in the hashed `work` folders:

```
| - work/
|   |
|   | - c8
|   |   |
|   |   | - b5f6c2d2a5932f77d5bc53320b8a5d
|   |   |   |
|   |   |   | - .command.begin
|   |   |   | - .command.err
|   |   |   | - .command.log
|   |   |   | - ...
|   |   |
|   |   | - eb
|   |   |   |
|   |   |   | - 9af3b0384ef96c011b4da69e86fca7
|   |   |   |   |
|   |   |   |   | - ...
|   |   |   |
|   |   |   | - ...
|   |   |
|   |   | - ...
|   |
|   | - ...
|
| - ...
```

`.command.log`

`.command.log`, contains the log of the command execution. Often is identical to `.command.out`

.command.out

`.command.out`, contains the standard output of the command execution

.command.err

`.command.err`, contains the standard error of the command execution

.command.begin

`.command.begin`, contains what has to be executed before `.command.sh`

.command.sh

`.command.sh`, contains the block of code indicated in the process script block

.command.run

`.command.run`, contains the code made by nextflow for the execution of `.command.sh` and contains environmental variables, eventual invocations of linux containers etc

.exitcode

`.exitcode`, contains the exitcode of the process, this is typically 0 if everything is ok, another value if there was a problem.

Pipeline parameters vs Nextflow options

There are two types of parameters!

Pipeline parameters are the parameters used in the pipeline script (e.g. `params.reads`). They are related to the pipeline and can be modified/overwritten on the command-line with a **double dash**: e.g parameter `params.reads` in the `fastqc.nf` script can be set as `--reads` in the command-line.

There are more ways to set your pipeline parameters, for example in a `nextflow.config` file. This can be useful when there are many parameters to a pipeline, or if you want to save the parameters for reuse later. More information about this can be found [here](#).

Nextflow options are set in the command-line with a **single dash** and are predefined in Nextflow's language. Here are some examples:

- `-bg` runs the workflow in the background.
- `-resume` resumes the parameter from where it failed last time and uses cached information from the `work/` directory.
- `-with-report` creates a report of how the pipeline ran (performance, memory usages etc.).
- `-work-dir` overwrite the name of the directory where intermediate result files are written.
- ...

We will discover these options while going through the course materials.

Knowing where to find a pipeline and which one to use.

Before thinking of writing our own (plausibly) complex pipeline, we can also think about importing one. Several repositories exist that store Nextflow pipelines (non-exhaustive list):

- Some curated nextflow pipelines are available on [awesome-nextflow](#).
- Pipelines from the [nf-core community](#).
- Pipelines from [WorkflowHub](#) (this is a currently ongoing effort).
- VSN-Pipelines for single cell analysis [VSN-Pipelines](#) (Currently not updated)

Import a pipeline

Imagine that we set our eyes on the [nextflow-io/rnaseq-nf](#) pipeline. A toy workflow for the analysis of (once again) RNAseq data.

There are different possibilities to pull a publicly available pipeline at a git-based hosting code system (GitHub, GitLab or BitBucket). One of them is to pull the pipeline using `nextflow pull`, like so:

```
nextflow pull nextflow-io/rnaseq-nf
```

The latest version of the pipeline is written in DSL2. Imagine that you would like to run the last DSL1 version of the pipeline (v1.2), we can pull this specific version using:

```
nextflow pull nextflow-io/rnaseq-nf -r v1.2
```

Nextflow enables to pull any specific tag, release or commit. To pull the pipeline from (1) a given branch and at a (2) specific git commit, we use the following:

```
nextflow pull nextflow-io/rnaseq-nf -r master
nextflow pull nextflow-io/rnaseq-nf -r 98ffd10a76
```

The workflows will not be cloned in the folder from where we launched these commands. Instead, it is available in the folder `~/.nextflow/assets/`, e.g. for the `nextflow-io/rnaseq-nf` pipeline in `~/.nextflow/assets/nextflow-io/rnaseq-nf/`. If we would want to have the workflows available (for further editing), we can use `nextflow clone`, similar to how `git` works.

The `-r` option can also be used directly with `nextflow run` rather than running `nextflow pull` first.

After importing our pipeline of interest, we can run it on the command-line using the `nextflow run <pipeline-name>` command, with `<pipeline-name>` being the name of the pipeline we just imported.

Note: When you use `nextflow run` without pulling the pipeline first (`nextflow pull`), Nextflow will check GitHub for a corresponding repository, if one exists it will pull it and run it locally.

`nextflow run nextflow-io/rnaseq-nf` will result in an error due to uninstalled tools on our VMs. To fix this, simply add the parameter `-with-docker`. We will discover what is happening when we enable this setting later`.

Extra exercises

Warning: The `nextflow-io/rnaseq-nf` pipeline was recently upgraded to require Nextflow-22.08.2-edge which changes how the `conda` directive works. As we are on an earlier version of Nextflow, you will need to use the version in this git commit `1ca363c8`.

Extra exercise 1

Run the publicly available pipeline `nextflow-io/rnaseq-nf`. Try to modify the name of the folder where results are stored by using a different parameter on the command-line.

Solution 1

The directory with the final results:

```
nextflow run nextflow-io/rnaseq-nf --outdir 'myAwesomeResults' -r 1ca363c8 -with-docker
```

or, the directory with temporary files (used for caching):

```
nextflow run nextflow-io/rnaseq-nf -w 'myAwesomeResults' -r 1ca363c8 -with-docker
```

Extra exercise 2

Which pipeline parameters are defined, can you modify these in the `rnaseq-nf` pipeline?

Solution 2

The reads, transcriptome, `outdir` and `multiqc` parameters.

Extra exercises 3

3.1 How many pipelines are currently available in `nf-core`? How many are under development, released, and archived?

3.2 Find the pipeline for performing ATAC-seq data analysis in `nf-core`.

- What is the current/latest version of the pipeline?
- How many versions are available to download?
- How many and which parameter(s) is(are) **required** to run the pipeline?
- What is the default output directory's name?
- What happens if you do not specify a profile (`-profile`)?

3.3 In the `nextflow-io awesome pipelines`, look for the featured BABS-aDNaseq workflow:

- What tool is used for calling variants?
- What version of Nextflow is it advised to use?
- How do you download the BABS-aDNaseq pipeline locally?

Solution 3

3.1. As of 20/10/2022: 71 pipelines are available, of which 40 are released, 25 are under development, and 6 are archived.

3.2 [link](#)

- 1.2.2 (20/10/2022)
- 5 versions: current (1.2.2), 1.2.1, 1.2.0, 1.1.0, and 1.0.0.
- Only one required parameter: `--input` (Path to comma-separated file containing information about the samples in the experiment)
- `./results` (parameter `--outdir`)
- If `-profile` is not specified, the pipeline will run locally and expect all software to be installed and available on the PATH. More information is available [here](#).

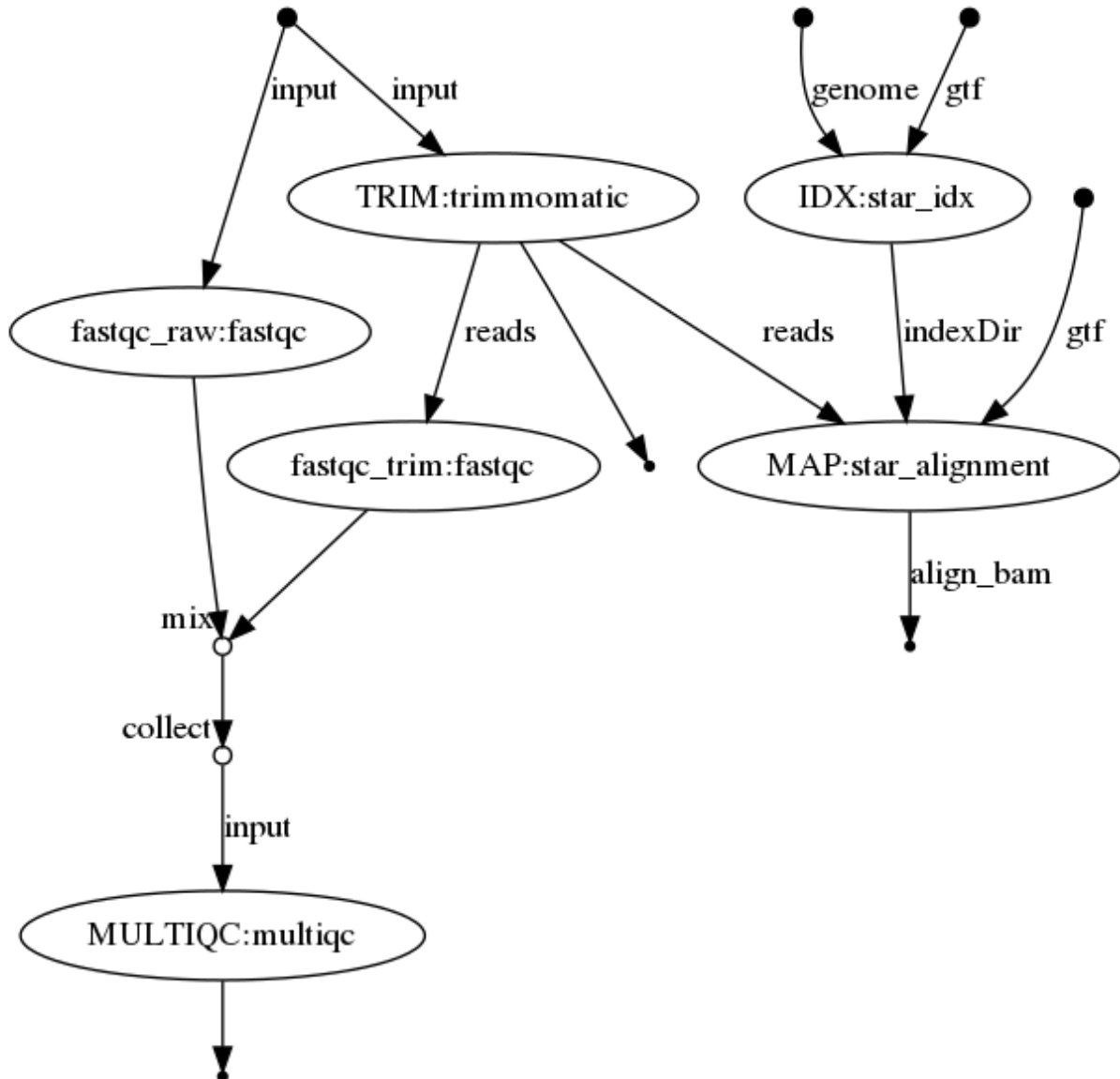
3.3 [link](#).

- `samtools mpileup`
- version 0.30.2 (Note that the current version is 22.10.0 (20/10/2022))
- `git clone https://github.com/crickbabs/BABS-aDNASeq` (or `nextflow clone`)

6.3.3 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. The figure below was generated with Nextflow and represents the processes that we will build and the overview of the dataflow from the beginning to the end of the workflow.

Channel.fromFilePairs



Quality control with FastQC

The following script can be found and run in `exercises/03_first_pipeline/fastqc.nf`.

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

params.reads = "$launchDir/data/*.fq.gz"

/**
 * Quality control fastq
 */

reads_ch = Channel
    .fromPath( params.reads )
```

(continues on next page)

(continued from previous page)

```
process fastqc {  
  
    input:  
    path read  
  
    script:  
    """  
    fastqc ${read}  
    """  
}  
  
workflow {  
    fastqc(reads_ch)  
}
```

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.

Note:

- `$launchDir`: The directory from where the script is launched (replaces `$baseDir` in version >20).
 - There is a great flexibility in the Nextflow (Groovy) language: writing of whitespaces, newlines where channels are created,...
-

Let's first run this script with the following command. If you have `htop` installed, keep an eye on the distribution of the workload and notice how Nextflow parallelises the jobs.

```
nextflow run exercises/03_first_pipeline/fastqc.nf
```

Note: The process in `exercises/03_first_pipeline/fastqc.nf` specifies a container, and the `nextflow.config` file in the same folder activates the use of docker. If this directive was not there or docker was not enabled, you would need to make sure that the tool `fastQC` is installed. Conda is already installed and activated, it allows us to easily install `fastqc` with the following command `conda install -c bioconda fastqc`.

In the following steps we will add new features to this script:

Exercise 2.1

- Overwrite the parameter `reads` on runtime (when running Nextflow on the command-line) so that it only takes `ggal_gut_1.fq.gz` as an input read.
- Additionally, FastQC generates a html- and zip-file for each read. Where are these output files located?

Solution 2.1

- `nextflow run exercises/03_first_pipeline/fastqc.nf --reads data/ggal_gut_1.fq.gz`
 - The output files are stored in the `work/` directory following the generated hashes. The hash at the beginning of each process reveals where you can find the result of each process.
-

Exercise 2.2

Change the the script in order to accept & work with paired-end reads. For this we will need to:

- Adapt something in the reads parameter (`params.reads`)
- Change how the channel is generated
- Change the input declaration in the process (from `path` to a `tuple`).

Solution 2.2

The solution is given in `exercises/03_first_pipeline/solutions/2.2_fastqc.nf`. Note that if you run this script, only two processes will be launched, one for each paired-end reads dataset.

Exercise 2.3

Run the script with: `nextflow run exercises/03_first_pipeline/fastqc.nf -bg > log`. What does the `-bg > log` mean? What would the advantage be?

Solution 2.3

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

Exercise 2.4

Check if the files exist (`checkIfExists`) upon creating the channels and invoke an error by running the nextflow script with wrong reads, e.g. `nextflow run exercises/03_first_pipeline/fastqc.nf --reads wrongfilename`.

Solution 2.4

The solution is given in `exercises/03_first_pipeline/solutions/2.4_fastqc.nf`

Exercise 2.5

Control where and how the output is stored. Have a look at the directive `publishDir`. Nextflow will only store the files that are defined in the output declaration block of the process, therefore we now also need to define the output. Put a copy of the output files in a new folder that contains only these results.

Solution 2.5

The solution is given in `exercises/03_first_pipeline/solutions/2.5_fastqc.nf`

- 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 be made available in the defined directory.
- 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 your pipeline will fail complaining about files not being present.

Warning: Files are copied into the specified directory in an asynchronous manner, thus they may not be immediately available in the published directory at the end of the process execution. For this reason files published by a process must not be accessed by other downstream processes.

The final FastQC script, with some additional comments is provided in `exercises/03_first_pipeline/solutions/fastqc_final.nf`.

Quality filtering with `trimmomatic`

Now we will add the next step in our pipeline, which is **trimming and filtering the low quality reads**. For this process, we will use the tool `trimmomatic`.

The `fastqc.nf` script was extended with the `trimmomatic` process and is available in `exercises/03_first_pipeline/trimmomatic.nf`.

- A number of parameters have been added related to the `trimmomatic` process
- The process `trimmomatic` with its inputs and outputs and the script has been created
- The `workflow` now also contains the process `trimmomatic`, called as a function

In the output declaration block, we are introducing a new option: `emit`. Defining a process output with `emit` allows us to use it as a named channel in the external scope.

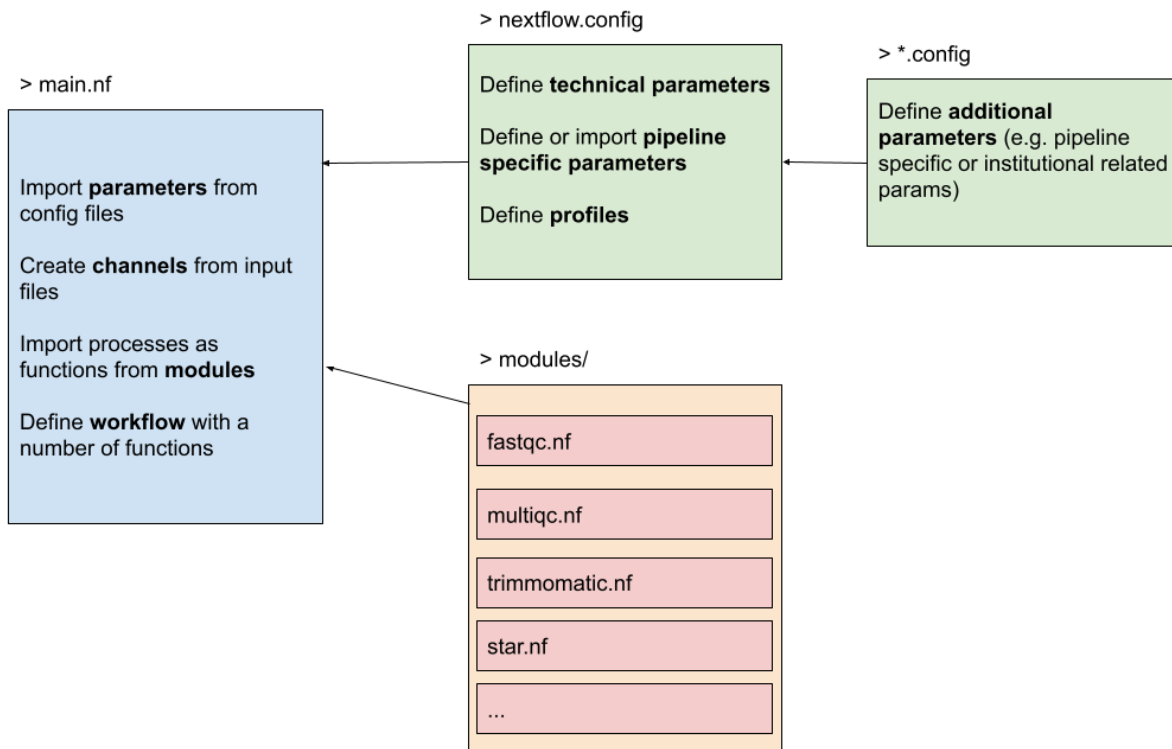
At this point we're interested in the result of the `trimmomatic` process. Hence, we want to verify the quality of the reads with another `fastqc` process. Re-run `fastqc` on the filtered read sequences by adding it in the workflow of `trimmomatic.nf`. Use the parameter `-resume` to restart the pipeline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

Hmm, 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. It means that processes can only be used once in a workflow. This means that we need to come up with a smarter solution (see below).

Modules

Until now, we have written the processes and the workflow in the same file. However, if we want to be truly modular, we can write a library of modules and import a specific component from that library. A module can contain the definition of a function, process and workflow definitions.

The figure below gives an overview of how the structure could look like. On the left we have the main Nextflow script (`main.nf`) that defines the parameters, channels and the workflow. It imports the processes from the modules, in this case available in a folder `modules/`. The configuration file `nextflow.config` will be further discussed in the next chapter.



A module is generally imported with

```
include {<process-name>} from './path/to/modules/script.nf'
```

with `<process-name>` the name of the process defined in the `script.nf`. The origin of the module defined by a relative path must start with `./`, alternatively use `projectDir` to use the absolute path. Navigate to the `modules` folder and find a script called `fastqc.nf`. This script consists of a process and a workflow. This module can be imported into our pipeline script (main workflow) like this:

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

This doesn't overcome the problem that we can only use a process once. However, when including a module component it's possible to specify a name alias. This allows the inclusion and the invocation of the same component multiple times in your script using different names. For example:

```
include { fastqc as fastqc_raw; fastqc as fastqc_trim } from "${projectDir}/modules/
↳fastqc"
```

Now we're ready to use a process, defined in a module, multiple times in a workflow.

Investigate & run the script `exercises/03_first_pipeline/modules.nf` which contains the following code snippet

```
...
include { fastqc as fastqc_raw; fastqc as fastqc_trim } from "${projectDir}/../../
modules/fastqc"
include { trimmomatic } from "${projectDir}/../../modules/trimmomatic"

// Running a workflow with the defined processes here.
workflow {
  read_pairs_ch.view()
  fastqc_raw(read_pairs_ch)
  trimmomatic(read_pairs_ch)
  fastqc_trim(trimmomatic.out.trim_fq)
}
```

Similarly as described above, we can extend this pipeline and map our trimmed reads on a reference genome. First, we'll have to create an index for our genome and afterwards we can map our reads onto it. These modules are called from the main script `RNAseq.nf`.

Exercise 2.6

In the folder `modules/` find the script `star.nf` which contains two processes: `star_index` and `star_alignment`. Complete the script `RNAseq.nf` so it includes these processes and hence the pipeline is extended with an indexing and alignment step. The parameters used in the modules are already defined for you.

Solution 2.6

Solution in `exercises/03_first_pipeline/solutions/2.6_RNAseq.nf`. The following lines were added.

```
genome = Channel.fromPath(params.genome)
gtf = Channel.fromPath(params.gtf)

include { star_idx; star_alignment } from "${projectDir}/../../modules/star"

workflow {
  ...
  star_idx(genome, gtf)
  star_alignment(trimmomatic.out.trim_fq, star_idx.out.index, gtf)
}
```

Exercise 2.7

In the folder `modules/` find the script `multiqc.nf`. Import the process in the main script so we can call it as a function in the workflow. This process expects all of the zipped and html files from the fastqc processes (raw & trimmed) as one input. Thus it is necessary to use the operators `.mix()` and `.collect()` on the outputs of `fastqc_raw` and `fastqc_trim` to generate one channel with all the files.

Solution 2.7

Solution in `exercises/03_first_pipeline/solutions/2.7_RNAseq.nf`. The following lines were added.

```
include { multiqc } from "${projectDir}/../modules/multiqc"

workflow {
  ...
  multiqc_input = fastqc_raw.out.fastqc_out
    .mix(fastqc_trim.out.fastqc_out)
    .collect()

  multiqc(multiqc_input)
}
```

This pipeline is still subject to optimizations which will be further elaborated in the next chapter.

Subworkflows

The workflow keyword allows the definition of **sub-workflow** components that enclose the invocation of one or more processes and operators. Here we have created a sub-workflow for a hypothetical `hisat` aligner.

```
workflow hisat {
  hisat_index(arg1)
  hisat_alignment(arg1, arg2)
}
```

These sub-workflows allow us to use this workflow from within another workflow. The workflow that does not carry any name is considered to be the main workflow and will be executed implicitly. This is thus the entry point of the pipeline, however alternatively we can overwrite it by using the `-entry` parameter. The following code snippet defines two sub-workflows and one main workflow. If we would only be interested in the star alignment workflow, then we would use `nextflow run pipeline.nf -entry star`.

```
workflow star {
  take:
  arg1
  arg2
  arg3

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

(continues on next page)

(continued from previous page)

```
workflow hisat2 {
  take:
  arg1
  arg2

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

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

Note: The `take:` declaration block defines the input channels of the sub-workflow, `main:` is the declaration block that contains the processes (functions) and is required in order to separate the inputs from the workflow body. These options are useful when the pipeline is growing with multiple entry-levels to keep a tidy overview.

Extra exercises

Extra exercise 1

Extend the workflow pipeline with a final note printed on completion of the workflow. Read more about workflow introspection [here](#).

Solution 1

The solution is given in `exercises/03_first_pipeline/solutions/ex.1_RNAseq.nf`

Extra exercise 2

Adapt the `exercises/03_first_pipeline/solutions/ex.1_RNAseq.nf` script so it uses Salmon as an aligner and quantifier. In our temporary solution the alignment with Star has been replaced with Salmon, it would be better to create a subworkflow so you can choose upon `-entry` to work with Star or Salmon.

Solution 2

The solution is given in `exercises/03_first_pipeline/solutions/ex.2_RNAseq.nf`.

Extra exercise 3

Write a Nextflow script for a tool that you use in your research. Use the same approach with parameters, channels, process in a module, and a workflow.

Solution 3

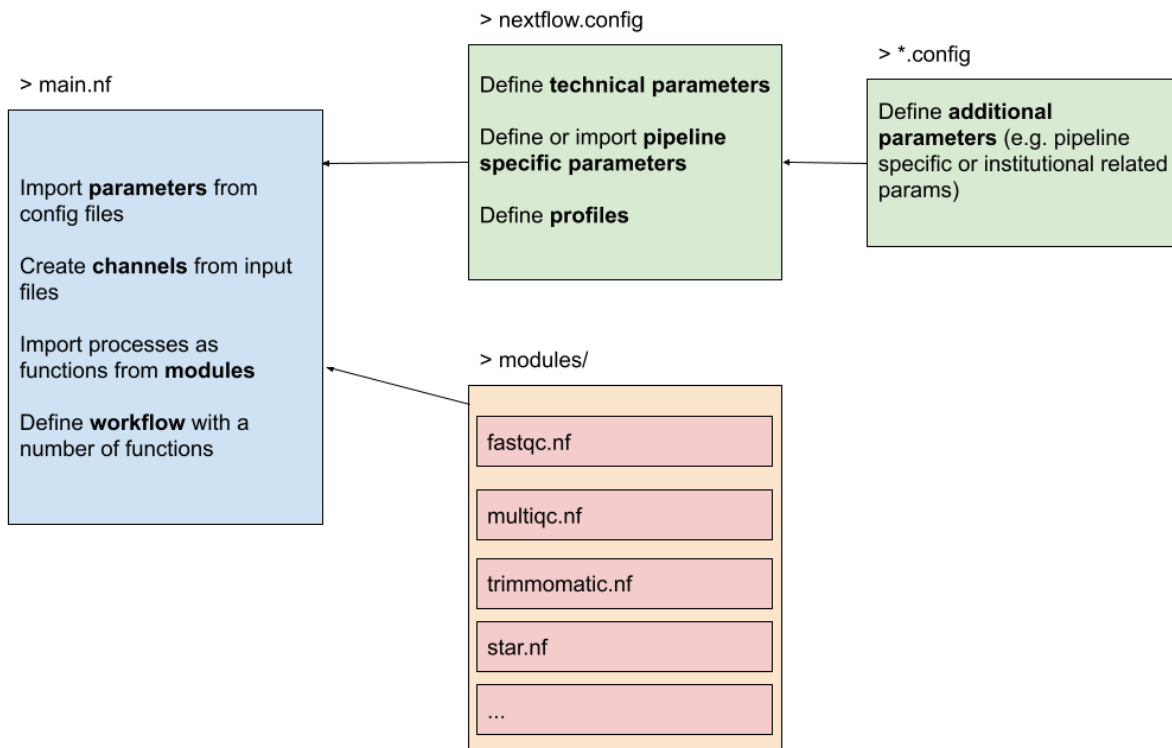
If you are stuck, don't hesitate to ask for help!

6.3.4 Configuration files

Managing configurations

Pipeline configuration properties are defined in a file named `nextflow.config` situated in the pipeline execution directory. This file can be used to define technical and project parameters, e.g. which executor to use, the processes' environment variables, pipeline parameters etc. Hence, the configuration file allows to separate these variables from the nextflow workflow script and makes the scripts more flexible and modular.

Let's have a look again at the structure of the workflow. The `nextflow.config` defines the technical and pipeline parameters and are imported in the `main.nf` script. Actually, we can write any number of `*.config` file and import them in the general `nextflow.config` which is then imported by default in the `main.nf`.



Technical parameters

Executors

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 using the `local` executor. In the example below we start with defining the processes' allowed memory- and cpu-usage. This list can be further extended with parameters such as time, queue, etc.

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

It's also possible to create labels that can be chosen and used for each process separately. In the example below we can use the label `high` as a directive in a process and hence allow more resources for that particular process (see `star.nf`). These labels are added in the directives of the processes as we did in our modules.

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

The local executor is very useful for pipeline development and testing purposes, but for real world computational pipelines an HPC or cloud platform is often required, these may not allow direct access to the machines where your code will run and instead provide systems to submit tasks with such as `pbs` or `SLURM`. The executor can be defined as `process.executor = 'local'` in the snippet above. If we want to use a different executor we could use e.g. `azurebatch` or `awsbatch`, however this goes also hand in hand with the parameters that are applicable for that specific executor. [This config](#) file contains the minimal set of required parameters for the Azure case (we will discuss the profiles soon).

Hence, you can write your pipeline script once and have it running on your computer, a cluster resource manager or the cloud by simply changing the executor definition in the Nextflow configuration file. As these configurations are often a one-time effort, managed by a local IT/admin person, we refer to the [official documentation](#).

Here is an overview of supported executors:



Portability

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 configuration file.

In the former case, simply define the container image name in the process directives. In the snippet below, we defined a container that already exists in [DockerHub](#). Dockerhub is also the default location where Nextflow will search for the existence of this container if it doesn't exist locally.

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

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

In the latter case, write the following line in the `nextflow.config` file:

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

We're referring to a Docker container image that exists on [Dockerhub](#). Notice however that all the tools and dependencies necessary during your pipeline, need to be present in this image. To run a pipeline script with this Docker container image, you would use the following command: `nextflow run example.nf -with-docker`.

Ultimately, the parameter `-with-docker` does not need to be defined on runtime and it should use the Docker container in the background at all times, for this purpose we can set `docker.enabled = true` option in the config file.

Note: Another interesting parameter to consider adding to the configuration file is the `docker.runOptions = '-u \$(id -u):\$(id -g)'`. This allows us to create files with permissions on user-level instead of the default root-level

files.

Singularity/Apptainer:

Similar to docker, using a singularity or apptainer image does not require you to have to adapt the pipeline script. You can run with Singularity container using the following command-line parameter: `-with-singularity [singularity-image-file]` (Apptainer support is also present), 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:

```
singularity.cacheDir = "/path/to/singularity"
```

If you want to avoid entering the Singularity image as a command line parameter, you can define it in the Nextflow configuration file. For example you can add the following lines in the `nextflow.config` file:

```
process.container = '/path/to/singularity.img'
singularity.enabled = true
```

Profiles

To create some structure in the config files and quickly select the parameters necessary for the infrastructure we are running the workflow on, the concept of **profiles** was introduced. Each profile contains a set of parameters and is selected on runtime using the `-profile` option.

Combining all of the above results in one nice looking config file:

```
profiles {
  standard {
    process {
      executor = 'local'
      withLabel: 'low' {
        memory='1G'
        cpus='1'
        time='6h'
      }
      withLabel: 'med' {
        memory='2G'
        cpus='2'
      }
      withLabel: 'high' {
        memory = '8G'
        cpus='8'
      }
    }
  }

  azure {
    process {
      executor = 'azurebatch'
    }
  }

  conda { params.enable_conda = true }
```

(continues on next page)

(continued from previous page)

```

docker {
  // Enabling docker
  docker.enabled = true
  docker.runOptions = '-u ${id -u}:${id -g}'
}

singularity {
  // Enabling singularity
  singularity.enabled = true
  singularity.autoMounts = true
  singularity.cacheDir = "$launchDir/singularity"
}
}

```

Here are some examples of how we can run the workflow:

- Locally with conda:

```
nextflow run main.nf -profile standard,conda
```

- Locally with docker:

```
nextflow run main.nf -profile standard,docker
```

- On Microsoft Azure with Docker:

```
nextflow run main.nf -profile azure,docker
```

Pipeline parameters

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

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

The parameters can be defined with `params.<name> = <value>` or join them all in one long list as such:

```

// Define project parameters needed for running the pipeline
params {
  // General parameters
  projdir = "/path/to/data"
  refdir = "/path/to/references"
  outdir = "/path/to/data-analysis"

  // Reference genome and annotation files
  genome = "${refdir}/Drosophila_melanogaster.BDGP6.dna.fa"
  gtf = "${refdir}/Drosophila_melanogaster.BDGP6.85.sample.gtf"

  // Input parameters
  reads = "${projdir}/*{1,2}.fq.gz"

  ...
}

```

Extra exercises

Extra exercise 1

Complete the `nextflow.config` and `params.config` files in the `exercises/04_configs/` folder. These config files should accompany the script `exercises/04_configs/RNaseq.nf`. The command to run this pipeline should be: `nextflow run exercises/04_configs/RNaseq.nf -profile docker`.

Solution 1

The solution is available in the `exercises/04_configs/solutions/` folder.

Note: Try changing a parameter in the config file and see the updated value printed by the workflow. Also try leaving an option out of the config file to see how the defaults in the script are then used, when they'd normally be overwritten.

Extra exercise 2

Run the `nextflow-io/rnaseq-nf` locally with Docker.

Solution 2

```
nextflow run nextflow-io/rnaseq-nf -r 1ca363c8 -profile standard,docker
```

The local executor will be chosen and it is hence not necessary to select the standard profile.

Extra exercise 3

In the previous extra exercise we ran a Nextflow pipeline residing on GitHub. Imagine that we want to run this pipeline, however we need to do some minor configurations to it. Let's say that we want to change the docker profile. Find a way to edit the `nextflow.config` file and change the contents of docker profile so it includes the following:

```
...
docker.enabled = true
docker.runOptions = '-u ${id -u}:${id -g}'
```

Solution 3

To change anything in the configuration file, the `nextflow.config` file needs to be edited. There are two options for this: in the assets where the pipeline is stored or by cloning the pipeline in our local folder structure. For this, you can use the following command: `nextflow clone <pipeline-name>` to clone (download) the pipeline locally. Then, open an editor and change the `nextflow.config` file so it contains the following:

Warning: Watch out for nested git folders!

```

profiles {
  docker {
    docker.enabled = true
    docker.runOptions = '-u \$(id -u):\$(id -g)'
  }
}

```

6.3.5 Creating reports

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

1. Workflow report

After running the nextflow pipeline script with the option `-with-report`, find the html report in the folder from where you launched the pipeline.

```
nextflow run exercises/05_reports/RNaseq.nf -with-report -profile docker
```

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

2. DAG

Use the option `-with-dag` to create a visualization of the workflow. By default and without any arguments, it will create a `.dot`-file that contains a description of the workflow, however to get a visualization we need to use an extra argument (e.g. `rnaseq.html`). This visualization is a nice overview of the workflow processes and how they are chained together and can be especially useful as a starting point to unravel more complex pipelines.

```
nextflow run exercises/05_reports/RNaseq.nf -with-dag rnaseq.html -profile docker
```

Note: As of Nextflow 22.04, the DAG can also be output in mermaid format, more information can be found [here](#).

3. Timeline Report

After running the nextflow pipeline script with the option `-with-timeline`, find the html report in the folder from where you launched the pipeline.

```
nextflow run exercises/05_reports/RNaseq.nf -with-timeline -profile docker
```

This report summarizes the execution time of each process in your pipeline. It can be used to identify bottlenecks and to optimize the pipeline. More information about the format of the timeline report can be found [here](#).

4. Tower

Adding the parameter `-with-tower` enables the Seqera Tower service and will output the reports to a browser-based platform. More about Tower below.

Tower

The Tower service, supported and developed by Seqera Labs, allows to monitor the workloads from a browser. Pipelines can be deployed on any local, cluster or cloud environment using the intuitive *launchpad* interface. Furthermore, it is also possible to manage teams and organizations, control project costs, and more. With ongoing improvements to the Tower platform, it is a very powerful platform worth checking out.

To start using Tower, first create an account on tower.nf. Then, we need to set the access token in our environment (on our VMs):

```
export TOWER_ACCESS_TOKEN=<YOUR ACCESS TOKEN>
export NXF_VER=22.10.7
```

Verify the Nextflow version (NXF_VER) with `nextflow -v`. The access token can be obtained from clicking on the top-right profile icon, select *Your tokens* and create *New token*.

Tower is undergoing a lot of changes, hence we refer to this useful video. More information is also available at seqera.io.

Exercise 1

Run the `RNAseq.nf` pipeline again, this time also make the reports (both html-report and a visualization of the pipeline)

Solution 1

The command that we need for this is the following.

```
nextflow run exercises/05_reports/RNAseq.nf -profile docker -with-report -with-dag_
↪rnaseq.html
```

To view the report and the dag, you will need to download the files to your local machine.

6.3.6 Project

For the second half of day two, each of you will build a small-scale metagenomics pipeline from scratch to test out what you've learned so far.

Concept

In metagenomics, environmental samples are taken and examined for the micro-organisms that can be found in them. In its most basic form, this is done by extracting DNA from each sample and selectively amplifying and sequencing the 16S rRNA-encoding gene, which is unique to bacteria and archaea. By examining the variety in sequences we get from sequencing only this gene, we can start to tell which specific micro-organisms are present within a sample.

In this project, we'd like to combine publicly available tools and some basic R scripts to:

1. Check the quality of the reads
2. Trim primers from them and filter low-quality reads
3. Find the unique 16S sequence variants & get a grasp of the diversity of the samples.

Warning: Before you start the project, make sure to `cd` to the project directory and work there to maintain a clean working environment.

Data

For this project, we will use data from *Vandeputte et al. (2017)*: a well-known study from the VIB centre for microbiology that was published in Nature. This study took faecal samples from 135 participants to examine their gut microbiota.

To keep computation times to a minimum, we will work with two subsets of this data. You can download this data by running the `get_project_data.sh` script that has been provided for you.

```
bash get_project_data.sh
```

Our metagenomics pipeline

Step 0: Preparation

We'd like to construct a pipeline that executes quality control, trimming, filtering and the finding of unique sequence in an automated fashion, parallelising processes wherever possible. We'd also like to run the whole thing in docker containers so we don't have to worry about dependencies.

Objective 1

Set up a `main.nf` script in which you will build your pipeline that uses nextflow's DSL2 and which reads in the forward and reverse reads for each of the five samples in the `data1`-directory into a channel.

All the docker containers we will need are already publicly available, so don't worry about having to write Dockerfiles yourself

Note: The following docker containers will work well with Nextflow for the pipeline you're going to create:

- fastqc: `biocontainers/fastqc:v0.11.9_cv8`
 - DADA2: `blekhmanlab/dada2:1.26.0`
 - Python: `python:slim-bullseye`
 - Cutadapt: `kfdrc/cutadapt:latest`
-

Step 1: Quality Control

After pulling in and setting up the data, we're first interested in examining the quality of the sequencing data we got.

Objective 2

Write a process which executes FastQC over the raw samples.

As we're not really looking forward to inspecting each FastQC report individually, we should pool these in a single report using MultiQC.

Objective 3

Write a second process that executes MultiQC on the FastQC output files.

If this all works, you should be able to take a look at the outputted `.html` report, in which you should see stats for 10 sets of reads (forward and reverse for each of the 5 samples).

Step 2: Trimming and filtering

Looking at the MultiQC report, our reads don't look that fantastic at this point, so we should probably do something about that.

We can use a publicly available tool called Cutadapt:

- to trim off the primers
- to trim and filter low-quality reads
- to remove very short reads and reads containing unknown bases (*i.e.*, 'N')

Cutadapt however requires us to specify the forward and reverse primers, as well as their reverse **complements**. The forward and reverse primers we can find in the paper: GTGCCAGCMGCCGCGGTAA and GGACTACHVHHHTWTCTAAT, the reverse complements of these are TTACCGCGGCKGCTGGCAC and ATTAGAWADDDBDGTAGTCC respectively.

Objective 4

Write a process that executes Cutadapt to filter and trim the reads.

Hint

In bash, the code for this would look something like this:

```
cutadapt -a ^FW_PRIMER...REVERSECOMP_RV_PRIMER \\  
        -A ^RV_PRIMER...REVERSECOMP_FW_PRIMER \\  
        --quality-cutoff 28 \\  
        --max-n 0 \\  
        --minimum-length 30 \\  
        --output SAMPLE_R1_TRIMMED.FASTQ --paired-output SAMPLE_  
↪R2_TRIMMED.FASTQ \\  
        SAMPLE_R1.FASTQ SAMPLE_R2.FASTQ
```

Step 3: Re-evaluate

As hopefully Cutadapt has done its job, we'd now like to take another look at the quality report of the preprocessed reads to see if this has improved the stats.

Objective 5

Write a workflow in your `main.nf` file which runs FastQC and MultiQC on the raw reads, filters and trims these reads using Cutadapt, and then reruns FastQC and MultiQC on the preprocessed reads.

Hint

Combine the FastQC and MultiQC processes into a named workflow.

Step 4: Find unique sequences and plot

To closely examine amplicon sequencing data and to extract the unique 16S sequence variants from these, there is an incredibly useful package in R called DADA2. You have been provided with a small R script (`reads2counts.r`) which uses this package to count the abundance of each unique sequence in each sample. Based on these abundances, the script can compare samples to each other and can construct a distance tree (also known as a dendrogram):



The script takes the preprocessed forward & reverse reads (in no specific order) as input arguments on the command line.

Objective 6

Write and incorporate a process that executes this Rscript and outputs the `counts_matrix.csv` and `dendrogram.png` files.

Hint

The container that you use should have the R-package ‘DADA2’ installed.

You now have successfully written your own microbiomics pipeline!

Step 5: Rinse and repeat

To see if all this effort in automatisisation was really worth it, you should run your pipeline on another dataset to see if it works as well.

Objective 7

Run your pipeline on the sequencing data in the `data2` directory.

If you have time left

There are a few things left that you can implement in your pipeline so others can more easily work with it as well.

- Use directives to output data from different processes to separate directories.
- Make a cool header that displays every time you run your pipeline using the `log.info` command.
- Add an `onComplete` printout to your pipeline that tells the user where they can find the output files.
- Speed up the slow processes in your pipeline by allocating more cpus and memory to them.
- Have nextflow create a report when you run the pipeline to see some cool stats.

The course materials are focused on the newer version of Nextflow DSL2. This is the newest version of the Nextflow language and the *de-facto* standard for writing new pipelines in Nextflow. In fact, DSL1 is supposed to be fading out and chances are that the support for DSL1 will be gone within a near future. Must you have any questions regarding pipelines written in DSL1, feel free to ask any questions during the workshop.

6.3.7 References and further reading

Here are some great tips for learning and to get inspired for writing your own pipelines:

- 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](#))
- Tutorial by Andrew Severin ([link](#))