
Pipeliner Documentation

Release 1

Anthony Federico, Stefano Monti

Apr 10, 2019

Contents

1	Requirements	1
1.1	Testing Nextflow	1
1.2	Installing Anaconda	2
1.3	Pre-Packaged Conda Environment	2
1.3.1	Yaml File	2
1.4	Setting up Pipeliner	2
2	Basic Usage	5
2.1	Framework Stucture	5
2.2	Pipeline Configuration	6
2.2.1	Config Inheritance	6
2.2.2	Data Input and Output	6
2.2.3	Basic Options	7
2.2.4	Providing an Index	7
2.2.5	Starting from Bams	7
2.2.6	Temporary Files	7
2.2.7	Skipping Steps	7
2.3	Process Configuration	7
2.3.1	Descriptive Arguments	8
2.3.2	Xargs	8
2.3.3	Ainj	9
2.4	Pipeline Execution	9
2.5	Output and Results	9
2.5.1	Sample Folders	10
2.5.2	Expression Matrix	10
2.5.3	Bam Files	10
2.5.4	Alignment Index	10
2.5.5	Reports	10
3	Pipeline Structure	11
3.1	Configuration File	11
3.1.1	File Paths	11
3.1.2	Executor and Compute Resources	12
3.1.3	Pipeline Options and Parameters	12
3.2	Pipeline Script	12
3.3	Template Processes	12
3.4	Output	14

4	Existing Pipelines	15
4.1	RNA-seq	15
4.1.1	Check Reads <code>check_reads</code>	15
4.1.2	Genome Indexing <code>hisat_indexing/star_indexing</code>	15
4.1.3	Pre-Quality Check <code>pre_fastqc</code>	15
4.1.4	Pre-MultiQC <code>pre_multiqc</code>	15
4.1.5	Read Trimming <code>trim_galore</code>	16
4.1.6	Read Mapping <code>hisat_mapping/star_mapping</code>	16
4.1.7	Reformat Reference <code>gtftobed</code>	16
4.1.8	Mapping Quality <code>rseqc</code>	16
4.1.9	Quantification <code>counting</code>	16
4.1.10	Expression Matrix <code>expression_matrix</code>	16
4.1.11	Expression Features <code>expression_features</code>	17
4.1.12	Expression Set <code>expression_set</code>	17
4.1.13	Summary Report <code>multiqc</code>	17
4.2	scRNA-seq	17
4.2.1	Check Reads <code>check_reads</code>	17
4.2.2	Genome Indexing <code>hisat_indexing/star_indexing</code>	17
4.2.3	Quality Check <code>fastqc</code>	17
4.2.4	Whitelist <code>whitelist</code>	18
4.2.5	Extract <code>extract</code>	18
4.2.6	Read Mapping <code>hisat_mapping/star_mapping</code>	18
4.2.7	Reformat Reference <code>gtftobed</code>	18
4.2.8	Mapping Quality <code>rseqc</code>	18
4.2.9	Quantification <code>counting</code>	18
4.2.10	Summary Report <code>multiqc</code>	18
4.3	DGE	19
4.3.1	Quantification <code>counting</code>	19
4.3.2	Expression Matrix <code>expression_matrix</code>	19
4.3.3	Sample Renaming <code>rename_samples</code>	19
4.3.4	Summary Report <code>multiqc</code>	19
5	Extending Pipelines	21
5.1	General Workflow	21
5.2	Configuration Inheritance	21
5.3	Template Process Injections	22
5.4	Testing Module	23

CHAPTER 1

Requirements

The *Pipelinier* framework requires *Nextflow* and *Anaconda*. *Nextflow* can be used on any POSIX compatible system (Linux, OS X, etc). It requires BASH and [Java 8 \(or higher\)](#) to be installed. Third-party software tools used by individual pipelines will be installed and managed through a Conda virtual environment.

1.1 Testing Nextflow

Before continuing, test to make sure your environment is compatible with a Nextflow executable.

Note: You will download another one later when you clone the repository

Make sure your Java installation is version 8 or higher:

```
java -version
```

Create a new directory and install/test *Nextflow*:

```
mkdir nf-test
cd nf-test
curl -s https://get.nextflow.io | bash
./nextflow run hello
```

Output:

```
N E X T F L O W ~ version 0.31.0
Launching `nextflow-io/hello` [sad_curran] - revision: d4c9ea84de [master]
[warm up] executor > local
[4d/479eec] Submitted process > sayHello (4)
[a8/4bc038] Submitted process > sayHello (2)
[17/5be64e] Submitted process > sayHello (3)
[ee/0d879f] Submitted process > sayHello (1)
```

(continues on next page)

(continued from previous page)

```
Hola world!
Ciao world!
Hello world!
Bonjour world!
```

1.2 Installing Anaconda

Pipeliner uses virtual environments managed by *Conda*, which is available through [Anaconda](#). Download the distribution pre-packaged with Python 2.7.

Make sure conda is installed and updated:

```
conda --version
conda update conda
```

Tip: If this is your first time working with *Conda*, you may need to edit your configuration paths to ensure *Anaconda* is invoked when calling `conda`

1.3 Pre-Packaged Conda Environment

1.3.1 Yaml File

Clone Pipeliner:

```
git clone https://github.com/montilab/pipeliner
```

Environment for Linux:

```
conda env create -f pipeliner/envs/linux_env.yml
```

Environment for OS X:

```
conda env create -f pipeliner/envs/osx_env.yml
```

Note: Copies of pre-compiled binaries are hosted/maintained at <https://anaconda.org/Pipeliner/repo>

Warning: For those installing on the Shared Computing Cluster (SCC) at Boston University, instructions on how to setup a private conda environment can be [here](#).

1.4 Setting up Pipeliner

Tip: It is recommended to clone Pipeliner to a directory path that does not contain spaces

With all prerequisites, one can quickly setup Pipeliner by cloning the repository, configuring local paths to toy datasets, activating the conda environment, and downloading the Nextflow executable:

```
# Clone Pipeliner
git clone https://github.com/montilab/pipeliner

# Activate conda environment
source activate pipeliner

# Configure local paths to toy datasets
python pipeliner/scripts/paths.py

# Move to pipelines directory
cd pipeliner/pipelines

# Download nextflow executable
curl -s https://get.nextflow.io | bash

# Run RNA-seq pipeline with toy data
./nextflow rnaseq.nf -c rnaseq.config
```

The output should look like this:

```
N E X T F L O W ~ version 0.31.1
Launching `rnaseq.nf` [nasty_pauling] - revision: cd3f572ab2
[warm up] executor > local
[31/1b2066] Submitted process > pre_fastqc (ggal_alpha)
[23/de6d60] Submitted process > pre_fastqc (ggal_theta)
[7c/28ee53] Submitted process > pre_fastqc (ggal_gamma)
[97/9ad6c1] Submitted process > check_reads (ggal_alpha)
[ab/c3eedf] Submitted process > check_reads (ggal_theta)
[2d/050633] Submitted process > check_reads (ggal_gamma)
[1d/f3af6d] Submitted process > pre_multiqc
[32/b1db1d] Submitted process > hisat_indexing (genome_reference.fa)
[3b/d93c6d] Submitted process > trim_galore (ggal_alpha)
[9c/3fa50b] Submitted process > trim_galore (ggal_theta)
[62/25fce0] Submitted process > trim_galore (ggal_gamma)
[66/ccc9db] Submitted process > hisat_mapping (ggal_alpha)
[28/69fff5] Submitted process > hisat_mapping (ggal_theta)
[5c/5ed2b6] Submitted process > hisat_mapping (ggal_gamma)
[b4/e559ab] Submitted process > gtftobed (genome_annotation.gtf)
[bc/6f490c] Submitted process > rseqc (ggal_alpha)
[71/80aa9e] Submitted process > rseqc (ggal_theta)
[17/ca0d9f] Submitted process > rseqc (ggal_gamma)
[d7/7d391b] Submitted process > counting (ggal_alpha)
[df/936854] Submitted process > counting (ggal_theta)
[11/143c2c] Submitted process > counting (ggal_gamma)
[31/4c11f9] Submitted process > expression_matrix
[1f/3af548] Submitted process > multiqc
Success: Pipeline Completed!
```


2.1 Framework Stucture

Pipeline is a framework with various moving parts to support the development of multiple sequencing pipelines. The following is a simplified example of its directory structure:

```
/pipeline
├── /docs
├── /envs
├── /scripts
├── /tests
└── /pipelines
    ├── /configs
    ├── /scripts
    ├── /templates
    ├── /toy_data
    ├── /rnaseq.nf
    └── /rnaseq.config
```

docs Markdown and Restructured Text documentaion files associated with Pipeliner and existing pipelines

envs Yaml files and scripts required to reproduce Conda environments

scripts Various helper scripts for framework setup and maintenance

tests Python testing module for multi-pipeline automatic test execution and reporting

pipelines/configs Base config files inherited by pipeline configurations

pipelines/scripts Various helper scripts for pipeline processes

pipelines/templates Template processes inherited by pipeline workflows

pipelines/toy_data Small datasets for rapid development and testing of pipelines. These datasets are modifications from original [RNA-seq](#) and [scRNA-seq](#) datasets.

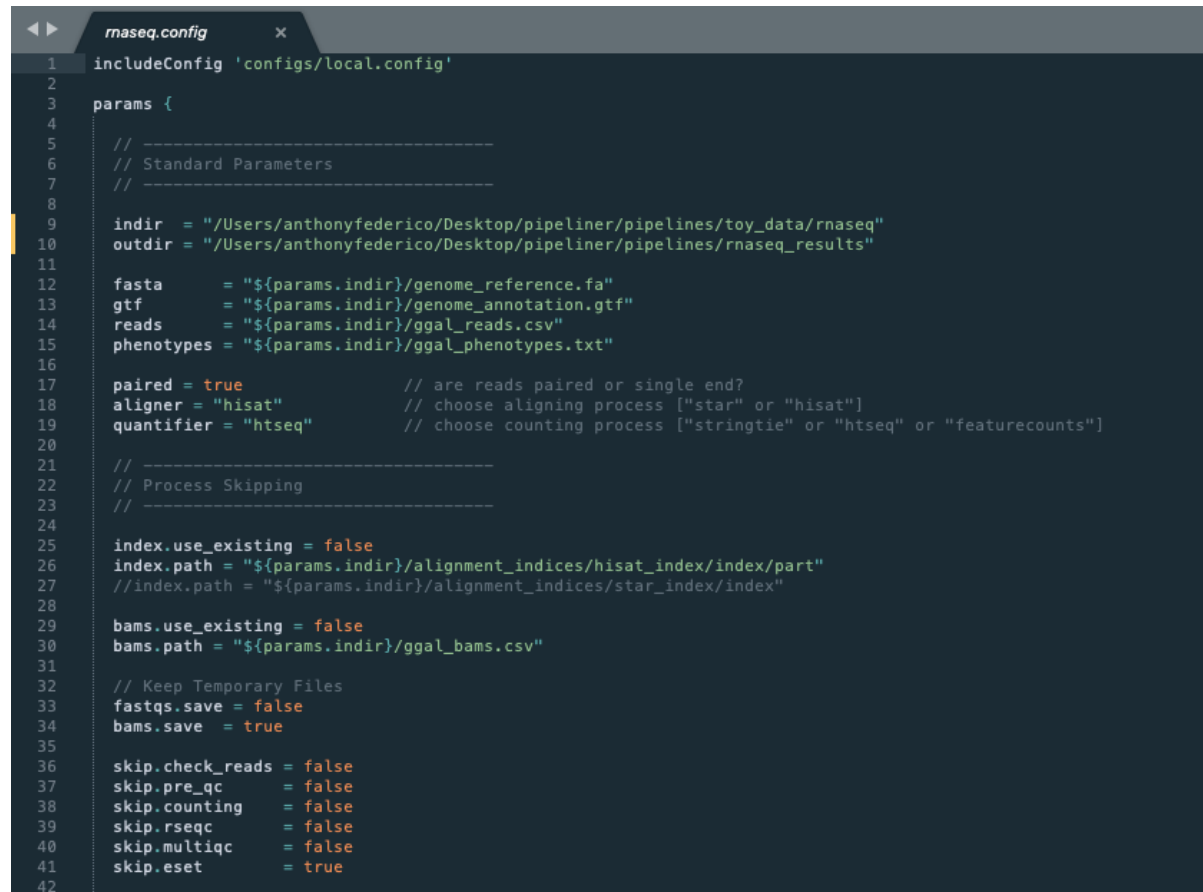
pipelines/rnaseq.nf Nextflow script for the RNA-seq pipeline

pipelines/rnaseq.config Configuration file for the RNA-seq pipeline

2.2 Pipeline Configuration

Note: These examples are applicable to all pipelines

In the previous section, we gave instructions for processing the RNA-seq toy dataset. In that example, the configuration options were all preset, however with real data, these settings must be reconfigured. Therefore the configuration file is typically the first thing a user will have to modify to suit their needs. The following is a screenshot of the first half of the RNA-seq configuration file.



```

1 includeConfig 'configs/local.config'
2
3 params {
4
5     // -----
6     // Standard Parameters
7     // -----
8
9     indir = "/Users/anthonyfederico/Desktop/pipeliner/pipelines/toy_data/rnaseq"
10    outdir = "/Users/anthonyfederico/Desktop/pipeliner/pipelines/rnaseq_results"
11
12    fasta = "${params.indir}/genome_reference.fa"
13    gtf = "${params.indir}/genome_annotation.gtf"
14    reads = "${params.indir}/ggal_reads.csv"
15    phenotypes = "${params.indir}/ggal_phenotypes.txt"
16
17    paired = true // are reads paired or single end?
18    aligner = "hisat" // choose aligning process ["star" or "hisat"]
19    quantifier = "htseq" // choose counting process ["stringtie" or "htseq" or "featurecounts"]
20
21    // -----
22    // Process Skipping
23    // -----
24
25    index.use_existing = false
26    index.path = "${params.indir}/alignment_indices/hisat_index/index/part"
27    //index.path = "${params.indir}/alignment_indices/star_index/index"
28
29    bams.use_existing = false
30    bams.path = "${params.indir}/ggal_bams.csv"
31
32    // Keep Temporary Files
33    fastqs.save = false
34    bams.save = true
35
36    skip.check_reads = false
37    skip.pre_qc = false
38    skip.counting = false
39    skip.rseqc = false
40    skip.multiqc = false
41    skip.eset = true
42

```

2.2.1 Config Inheritance

Line 1: Configuration files can inherit basic properties that are reused across many pipelines. We have defined several inheritable configuration files that are reused repeatedly. These include configs for running pipelines on local machines, Sun Grid Engine clusters, in Docker environments, and on AWS cloud computing.

2.2.2 Data Input and Output

Lines 9-15 All data paths are defined in the configuration file. This includes specifying where incoming data resides as well as defining where to output all data produced by the pipeline.

2.2.3 Basic Options

Lines 17-19 These are pipeline specific parameters that make large changes to how the data is processed.

2.2.4 Providing an Index

Lines 25-27 A useful feature of a pipeline is the ability to use an existing alignment index.

2.2.5 Starting from Bams

Lines 29-30 Another useful feature of a pipeline is the ability to skip pre-processing steps and start directly from the bam files. This allows users to start their pipeline from the counting step.

2.2.6 Temporary Files

Lines 33-34 By default, bam files are saved after alignment for future use. This can be useful, however these files are quite large and serve only as an intermediate step. Therefore, users can opt-out of storing them.

2.2.7 Skipping Steps

Lines 36-41 Users can skip entire pipeline steps and mix and match options that suit their need. Note that not all combination of steps are compatible.

2.3 Process Configuration

While the first half of the configuration is dedicated to controlling the pipeline, the second half is dedicated to modifying specific steps. We call these process-specific settings or parameters.

```

47 // Trim Galore
48 // -----
49 trim_galore.quality          = 20
50 trim_galore.custom_adaptors  = false
51 trim_galore.adapter1         = ""
52 trim_galore.adapter2         = ""
53 trim_galore.xargs             = ""
54
55 // Hisat Aligner
56 // -----
57 hisat_indexing.cpus          = 4
58 hisat_mapping.cpus           = 4
59 hisat_mapping.mp              = "6,2"
60 hisat_mapping.sp              = "2,1"
61 hisat_mapping.rdg             = "5,3"
62 hisat_mapping.rfg             = "5,3"
63
64 // Star Aligner
65 // -----
66 star_indexing.sjdb_overhang   = 149
67 star_indexing.cpus            = 4
68 star_mapping.cpus             = 4
69 star_mapping.twopassMode       = "Basic"
70 star_mapping.outfilter_multimap_nmax = 20
71 star_mapping.outfilter_mismatch_nmax = 999
72 star_mapping.outfilter_mismatch_relmax = 0.04
73 star_mapping.align_intron_min = 20
74 star_mapping.align_intron_max = 100000
75 star_mapping.align_mates_gapmax = 100000
76 star_mapping.align_sjoverhang_min = 1
77
78 // Htseq
79 // -----
80 htseq.type                     = "exon"
81 htseq.mode                     = "union"
82 htseq.idattr                   = "gene_id"
83 htseq.order                    = "pos"
84 htseq.xargs                    = ""
85 htseq.ainj                     = ""
86
87 // Feature Counts
88 // -----
89 feature_counts.type            = "exon"
90 feature_counts.id               = "gene_id"
91 feature_counts.xargs            = ""
92 feature_counts.ainj             = ""
93
94 // Stringtie
95 // -----
96 stringtie.xargs                 = ""
97 stringtie.ainj                  = ""
98 }
99

```

2.3.1 Descriptive Arguments

Variables for common parameters used in each process are explicitly typed out. For example, `trim_galore.quality` refers to the quality threshold used by Trim Galore and `feature_counts.id` refers to the gene id that Feature Counts refers to in the gtf file header. These variable names match the same variable names given in the original documentation of each tool. Therefore, one can refer to their individual documentation for more information.

2.3.2 Xargs

Because some software tools have hundreds of arguments, they cannot all be listed in the configuration file. Therefore, another variable called `xargs` can be used to extend the flexibility of each tool. Users can add additional arguments

as a string that will be injected into the shell command.

2.3.3 Ainj

Sometimes, users may want to add additional processing steps to a process without modifying the pipeline script or template directly. This can be done with the variable called `ainj` that injects a secondary shell command after the original template process.

2.4 Pipeline Execution

When the configuration file is set, run the pipeline with:

```
./nextflow rnaseq.nf -c rnaseq.config
```

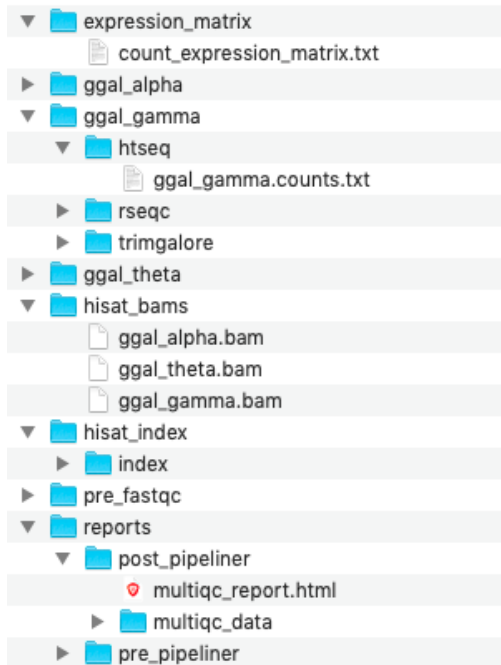
If the pipeline encounters an error, start from where it left off with:

```
./nextflow rnaseq.nf -resume -c rnaseq.config
```

Warning: If running Pipeliner on a high performance cluster environment such as Sun Grid Engine, ensure that Nextflow is initially executed on a node that allows for long-running processes.

2.5 Output and Results

One the pipeline has finished, all results will be directed to a single output folder specified in the configuration file.



2.5.1 Sample Folders

Each sample contains its own individual folder that holds temporary and processed data that was created by each process. In the screenshot, one can see the gene counts file specific to sample `gga1_gamma` that was generated by HTSeq.

2.5.2 Expression Matrix

The expression matrix folder contains the final count matrix as well as other normalized gene by sample matrices.

2.5.3 Bam Files

If the configuration file is set to store bam files, they will show up in the results directory.

2.5.4 Alignment Index

If an alignment index is built from scratch, it will be saved to the results directory so that it can be reused during future pipeline runs.

2.5.5 Reports

After a successful run, two reports are generated. A report conducted using the original data before any pre-processing steps as well as a final report run after the entire pipeline has finished. This allows one to see any potential issues that existed in the data before the pipeline as well as if those issues were resolved after the pipeline.

Pipeline Structure

The file paths for all data fed to a pipeline are specified in the configuration file. To ease the development process, Pipeline includes toy datasets for each of the pipelines. This example will cover the RNA-seq pipeline.

Note: Data for this pipeline is located in **pipelines/toy_data/rna-seq**

Users must provide the following files:

- Sequencing files or alignment files
- Comma-delimited file containing file paths to reads/bams
- Genome reference file
- Genome annotation file

3.1 Configuration File

The configuration file is where all file paths are specified and pipeline processes are parameterized. The configuration can be broken into three sections, including file paths, executor and compute resources, and pipeline options and parameters.

3.1.1 File Paths

The configuration file specifies where to find all of the input data. Additionally, it provides a path to an output directory where the pipeline will output results. The following is a typical example for the RNA-seq configuration file:

```
indir   = "/Users/anthonyfederico/pipeliner/pipelines/toy_data/rna-seq"
outdir  = "/Users/anthonyfederico/pipeliner/pipelines/rna-seq-results"
fasta   = "${params.indir}/genome_reference.fa"
gtf      = "${params.indir}/genome_annotation.gtf"
reads   = "${params.indir}/ggal_reads.csv"
```

3.1.2 Executor and Compute Resources

An abstraction layer between *Nextflow* and *Pipeliner* logic enables platform independence and seamless compatibility with high performance computing executors. This allows users to execute pipelines on their local machine or through a computing cluster by simply specifying in the configuration file.

Pipeliner provides two base configuration files that can be inherited depending if a pipeline is being executing using local resources or a Sun Grid Engine (SGE) queuing system.

If the latter is chosen, pipeline processes will be automatically parallelized. Additionally, each individual process can be allocated specific computing resource instructions when nodes are requested.

Local config example:

```
process {
  executor = 'local'
}
```

Cloud computing config example:

```
process {
  executor = 'sge'
  scratch = true

  $trim_galore.clusterOptions      = "-P montilab -l h_rt=24:00:00 -pe omp 8"
  $star_mapping.clusterOptions     = "-P montilab -l h_rt=24:00:00 -l mem_total=94G -
  ↪pe omp 16"
  $counting.clusterOptions         = "-P montilab -l h_rt=24:00:00 -pe omp 8"
  $expression_matrix.clusterOptions = "-P montilab -l h_rt=24:00:00 -pe omp 8"
  $multiqc.clusterOptions          = "-P montilab -l h_rt=24:00:00 -pe omp 8"
}
```

3.1.3 Pipeline Options and Parameters

The rest of the configuration file is dedicated to the different pipeline options and process parameters that can be specified. Some important examples include the following:

```
# General pipeline parameters
aligner      = "hisat"
quantifier   = "htseq"

# Process-specific parameters
htseq.type   = "exon"
htseq.mode   = "union"
htseq.idattr = "gene_id"
htseq.order  = "pos"
```

3.2 Pipeline Script

3.3 Template Processes

Pipelines written in *Nextflow* consist of a series of processes. Processes specify data I/O and typically wrap around third-party software tools to process this data. Processes are connected through channels – asynchronous FIFO queues – which manage the flow of data throughout the pipeline.

Processes have the following basic structure:

```
process <name> {

    input:
    <process inputs>

    output:
    <process outputs>

    script:
    <user script to be executed>
}
```

Often, the script portion of the processes are reused by various sequencing pipelines. To help standardize pipeline development and ensure good practices are propagated to all pipelines, template processes are defined and inherited by pipeline processes.

Note: Templates are located in **pipelines/templates**

For example, these two processes execute the same code:

```
# Without inheritance
process htseq {

    input:
    <process inputs>

    output:
    <process outputs>

    script:
    '''
    samtools view ${bamfiles} | htseq-count - ${gtf} \\
    --type ${params.htseq.type} \\
    --mode ${params.htseq.mode} \\
    --idattr ${params.htseq.idattr} \\
    --order ${params.htseq.order} \\
    > counts.txt
    '''
}

# With inheritance
process htseq {

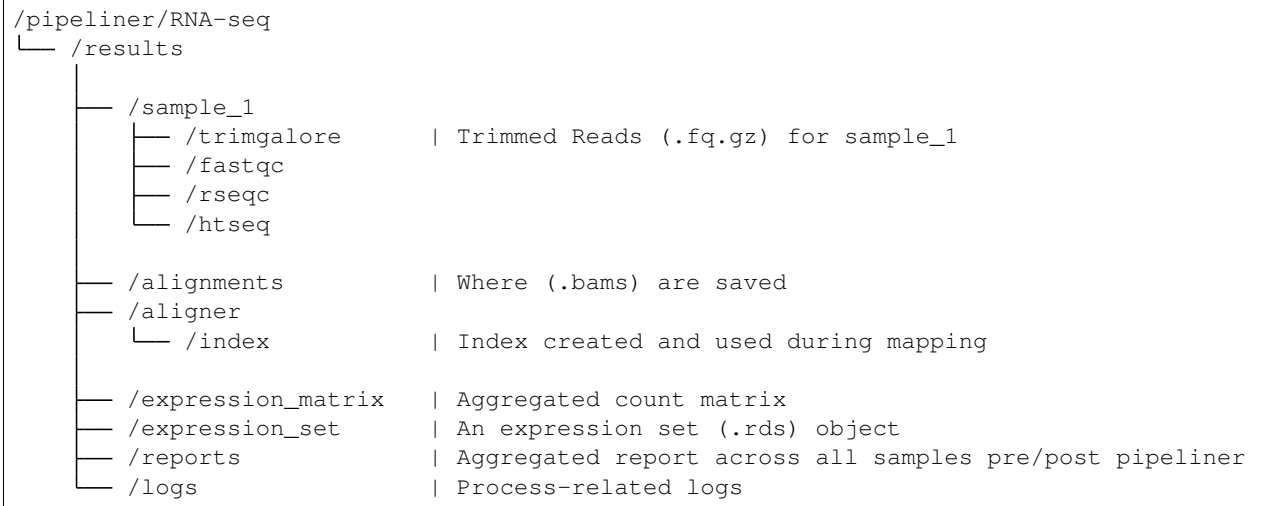
    input:
    <process inputs>

    output:
    <process outputs>

    script:
    template 'htseq.sh'
}
```

3.4 Output

The RNA-seq pipeline output has the following basic structure:



Each sample will have its own directory with sample-specific data and results for each process. Additionally, sequencing alignment files and the indexed reference genome will be saved for future use if specified. Summary reports pre/post-workflow can be found inside the reports directory.

4.1 RNA-seq

4.1.1 Check Reads `check_reads`

input List of read files (*.fastq*)
output None
script Ensures correct format of sequencing read files

4.1.2 Genome Indexing `hisat_indexing/star_indexing`

input Genome reference file (*.fa*) | Genome annotation file (*.gtf*)
output Directory containing indexed genome files
script Uses either *STAR* or *HISAT2* to build an indexed genome

4.1.3 Pre-Quality Check `pre_fastqc`

input List of read files (*.fastq*)
output Report files (*.html*)
script Uses *FastQC* to check quality of read files

4.1.4 Pre-MultiQC `pre_multiqc`

input Log files (*.log*)
output Summary report file (*.html*)

script Uses *MultiQC* to generate a summary report

4.1.5 Read Trimming `trim_galore`

input List of read files (*.fastq*)

output Trimmed read files (*.fastq*) | Report files (*.html*)

script Trims low quality reads with *TrimGalore* and checks quality with *FastQC*

4.1.6 Read Mapping `hisat_mapping/star_mapping`

input List of read files (*.fastq*) | Genome annotation file (*.gtf*) | Directory containing indexed reference genome files

output A list of alignment files (*.bam*) | Log files (*.log*)

script Uses either *STAR* or *HISAT2* to align reads to a reference genome

4.1.7 Reformat Reference `gt_ftobed`

input Genome annotation file (*.gtf*)

output Genome annotation file (*.bed*)

script Converts genome annotation file from GTF to BED format

4.1.8 Mapping Quality `rseqc`

input A list of alignment files (*.bam*)

output Report files (*.txt*)

script Uses *RSeQC* to check quality of alignment files

4.1.9 Quantification `counting`

input A list of alignment files (*.bam*) | Genome annotation file (*.gtf*)

output Read counts (*.txt*) | Log files (*.txt*)

script Uses either *StringTie*, *HTSeq*, or *featureCounts* to quantify reads

4.1.10 Expression Matrix `expression_matrix`

input A list of count files (*.txt*)

output An expression matrix (*.txt*)

script Reformats a list of count files into a *genes x samples* matrix

4.1.11 Expression Features `expression_features`

input Genome annotation file (*.gtf*) | An expression matrix (*.txt*)
output Gene feature data (*.txt*)
script Parses the genome annotation file for gene feature data

4.1.12 Expression Set `expression_set`

input An expression matrix (*.txt*) | Gene feature data (*.txt*) | Sample phenotypic data (*.txt*)
output An expression set object (*.rds*)
script Creates an expression set object with eData, fData, and pData attributes

4.1.13 Summary Report `multiqc`

input Log files and summary reports from all processes
output A summary report (*.html*)
script Uses *MultiQC* to generate a summary report

4.2 scRNA-seq

4.2.1 Check Reads `check_reads`

input List of read files (*.fastq*)
output None
script Ensures correct format of sequencing read files

4.2.2 Genome Indexing `hisat_indexing/star_indexing`

input Genome reference file (*.fa*) | Genome annotation file (*.gtf*)
output Directory containing indexed genome files
script Uses either *STAR* or *HISAT2* to build an indexed genome

4.2.3 Quality Check `fastqc`

input List of read files (*.fastq*)
output Report files (*.html*)
script Uses *FastQC* to check quality of read files

4.2.4 Whitelist `whitelist`

input List of read files (*.fastq*)
output A table of white listed barcodes (*.txt*)
script Uses *UMI-tools* to extract and identify true cell barcodes

4.2.5 Extract `extract`

input List of read files (*.fastq*) | A table of white listed barcodes (*.txt*)
output Extracted read files (*.fastq*)
script Uses *UMI-tools* to extract barcode from reads and append to read name

4.2.6 Read Mapping `hisat_mapping/star_mapping`

input List of read files (*.fastq*) | Genome annotation file (*.gtf*) | Directory containing indexed reference genome files
output A list of alignment files (*.bam*) | Log files (*.log*)
script Uses either *STAR* or *HISAT2* to align reads to a reference genome

4.2.7 Reformat Reference `gtftobed`

input Genome annotation file (*.gtf*)
output Genome annotation file (*.bed*)
script Converts genome annotation file from GTF to BED format

4.2.8 Mapping Quality `rseqc`

input A list of alignment files (*.bam*)
output Report files (*.txt*)
script Uses *RSeQC* to check quality of alignment files

4.2.9 Quantification `counting`

input A list of alignment files (*.bam*) | Genome annotation file (*.gtf*)
output Read counts (*.txt*) | Log files (*.txt*)
script Uses *featureCounts* to quantify reads

4.2.10 Summary Report `multiqc`

input Log files and summary reports from all processes
output A summary report (*.html*)
script Uses *MultiQC* to generate a summary report

4.3 DGE

4.3.1 Quantification counting

input A list of alignment files (*.bam*) | Genome annotation file (*.gtf*)

output Read counts (*.txt*) | Log files (*.txt*)

script Uses *featureCounts* to quantify reads

4.3.2 Expression Matrix `expression_matrix`

input A list of count files (*.txt*)

output An expression matrix (*.txt*)

script Reformats a list of count files into a *genes x samples* matrix

4.3.3 Sample Renaming `rename_samples`

input An expression matrix (*.txt*)

output An expression matrix (*.txt*)

script Renames samples in expression matrix based on a user-supplied table

4.3.4 Summary Report `multiqc`

input Log files and summary reports from all processes

output A summary report (*.html*)

script Uses *MultiQC* to generate a summary report

Extending Pipelines

5.1 General Workflow

The framework provides multiple resources for the user to extend and create sequencing pipelines. The first is toy datasets for all available pipelines including sequencing files, alignment files, genome reference and annotation files, as well as phenotypic data. Additionally, there are pre-defined scripts, processes, and configuration files that can be inherited and easily modified for various pipelines. Together, users can rapidly develop flexible and scalable pipelines. Lastly, there is a testing module enabling users to frequently test a series of different configurations with each change to the codebase.

5.2 Configuration Inheritance

An important property of configuration files is that they are inheritable. This allows developers to focus solely on the configuration components that are changing with each pipeline execution. Typically there are four components of a configuration file including the following.

Executor parameters:

```
process {  
    executor = "local"  
}
```

Input data file paths:

```
indir  = "/Users/anthonyfederico/pipeliner/pipelines/toy_data/rna-seq"  
outdir = "/Users/anthonyfederico/pipeliner/pipelines/rna-seq-results"
```

Pipeline parameters:

```
aligner      = "hisat"  
quantifier   = "htseq"
```

Process-specific parameters:

```
htseq.type    = "exon"
htseq.mode    = "union"
htseq.idattr  = "gene_id"
htseq.order   = "pos"
```

When developing, typically the only parameters that will be changing are pipeline parameters when testing the full scope of flexibility. Therefore, the development configuration file will look something like the following:

```
// paired / hisat / featurecounts

includeConfig "local.config"
includeConfig "dataio.config"

paired        = true
aligner       = "hisat"
quantifier    = "featurecounts"
skip.counting = false
skip.rseqc    = false
skip.multiqc  = false
skip.eset     = false

includeConfig "parameters.config"
```

5.3 Template Process Injections

Note: Sometimes it's better to create a new template rather than heavily modify an existing one

Each pipeline is essentially a series of modules - connected through minimal Nextflow scripting - that execute pre-defined template processes. While templates are generally defined to be applicable to multiple pipelines and are parameterized in a configuration file, they have two additional components contributing to their flexibility.

The following is an example of a template process for the third-party software tool *featureCounts*:

```
1 featureCounts \\  
2  
3 # Common flags directly defined by the user  
4 -T ${params.feature_counts.cpus} \\  
5 -t ${params.feature_counts.type} \\  
6 -g ${params.feature_counts.id} \\  
7  
8 # Flags handled by the pipeline  
9 -a ${gtf} \\  
10 -o "counts.raw.txt" \\  
11  
12 # Arguments indirectly defined by the user  
13 ${feature_counts_sargs} \\  
14  
15 # Extra arguments  
16 ${params.feature_counts.xargs} \\  
17  
18 # Input data  
19 ${bamfiles};
```

(continues on next page)

(continued from previous page)

```

20
21 # After injection
22 ${params.feature_counts.ainj}
    
```

Lines 4-6 These are *common* keyword arguments that can be set to string/int/float types by the user and passed *directly* from the configuration file to the template. The *params* prefix in the variable means it is initialized in the configuration file.

Lines 9-10 These are flags that are typically non-dynamic and handled internally by the pipeline.

Line 13 These are *common* flags that must be *indirectly* defined by the user. For example, `featureCounts` requires a `-p` flag for paired reads. Because `params.paired` is a boolean, it makes more sense for the pipeline to create a string of supplemental arguments indirectly defined by the configuration file.

```

feature_counts_sargs = ""
if (params.paired) {
    feature_counts_sargs = feature_counts_sargs.concat("-p ")
}
    
```

Line 16 These are *uncommon* keyword arguments or flags that can be pass *directly* from the configuration file to the template. Because some software tools can include hundreds of arguments, we explicitly state common arguments, but allow the user to additionally insert any unlimited number of additional arguments to maximize flexibility.

For example, the user might want to perform a one-off test of the pipeline where they remove duplicate reads and only count fragments that have a length between 50-600 base pairs. These options can be injected into the template by simply defining `params.feature_counts.xargs = "--ignoreDup -d 50 -D 600"` in the configuration file.

Line 19 These are required arguments such as input data handled internally by the pipeline.

Line 22 These are code injections - typically one-liner cleanup commands - that can be injected after the main script of a template. For example, the output of `featureCounts` is a *genes x samples* matrix and the user may want to try sorting rows by gene names. Setting `params.feature_counts.ainj` to `"sort -n -k1,1 counts.raw.txt > counts.raw.txt;"` would accomplish such a task.

After parameterization, the final result would look something like this:

```

1 featureCounts -T 1 -t "exon" -g "gene_id" \
2 -a "path/to/reference_annotation.gtf" \
3 -o "counts.raw.txt" \
4 -p --ignoreDup -d 50 -D 600 \
5 s1.bam s2.bam s3 bam;
6 sort -n -k1,1 counts.raw.txt > counts.raw.txt;
    
```

5.4 Testing Module

Each major change to a pipeline should be followed with a series of tests. Because pipelines are so flexible, it's infeasible to manually test even a limited set of typical configurations. To solve this problem we include an automated testing module.

Users can automatically test a series of configuration files by specifying a directory of user-defined tests:

```

/pipeliner
└─ /tests
    
```

(continues on next page)

(continued from previous page)

```
└─ /configs
   └─ /rnaseq
      └─ /t1.config
      └─ /t2.config
      └─ /t3.config
```

To run these series of tests, users can execute `python pipeliner/tests/test.py rnaseq` which will search for the directory `pipeliner/tests/configs/rnaseq` and automatically pair and run each configuration file with a pipeline script named `rnaseq.nf`.

Note: The directory name of tests must be the same as the pipeline script they are paired with

Warning: You must execute `test.py` from the `/pipelines` directory because Nextflow requires its executable to be in the working directory. Therefore the testing command will look like `python ../tests/test.py rnaseq`