Templates Documentation

Release 1

Diogo N. Silva

Dec 10, 2018

Getting Started

FLOWCRAFT

A NextFlow pipeline assembler for genomics.

CHAPTER 1

Overview

FlowCraft is an assembler of pipelines written in nextflow for analyses of genomic data. The premisse is simple:

Software are container blocks → Build your lego-like pipeline → Execute it (almost) anywhere.

1.1 What is Nextflow

If you do not know nextflow, be sure to check it out. It's an awesome framework based on the dataflow programming model used for building parallelized, scalable and reproducible workflows using software containers. It provides an abstraction layer between the execution and the logic of the pipeline, which means that the same pipeline code can be executed on multiple platforms, from a local laptop to clusters managed with SLURM, SGE, etc. These are quite attractive features since genomic pipelines are increasingly executed on large computer clusters to handle large volumes of data and/or tasks. Moreover, portability and reproducibility are becoming central pillars in modern data science.

1.2 What FlowCraft does

FlowCraft is a python engine that automatically builds nextflow pipelines by assembling pre-made ready-to-use *components*. These components are modular pieces of software or scripts, such as `fastqc`, `trimmomatic`, `spades`, etc, that are written for nextflow and have a set of attributes, such as input and output types, parameters, directives, etc. This modular nature allows them to be freely connected as long as they respect some basic rules, such as the input type of a component must match with the output type of the preceding component. In this way, nextflow processes can be written only once, and FlowCraft is the magic glue that connects them, handling the linking and forking of channels automatically. Moreover, each component is associated with a docker image, which means that there is no need to install any dependencies at all and all software runs on a transparent and reliable box. To illustrate:

- A linear genome assembly pipeline can be easily built using FlowCraft with the following pipeline string:

  ```
  trimmomatic fastqc spades
  ```

Which will generate all the necessary files to run the nextflow pipeline on any linux system that has nextflow and a container engine.

- You can easily add more components to perform assembly polishing, in this case, `pilon`:

```
trimmomatic fastqc spades pilon
```

- If a new assembler comes along and you want to switch that component in the pipeline, its as easy as replacing `spades` (or any other component):

```
trimmomatic fastqc skesa pilon
```

- And you can also fork the output of a component into multiple ones. For instance, we could annotate the resulting assemblies with multiple software:

```
trimmomatic fastqc spades pilon (abricate | prokka)
```

- Or fork the execution of a pipeline early on to compare different software:

```
trimmomatic fastqc (spades pilon | skesa pilon)
```

This will fork the output of `fastqc` into `spades` and `skesa`, and the pipeline will proceed independently in these two new 'lanes'.

- Directives for each process can be dynamically set when building the pipeline, such as the cpu/RAM usage or the software version:

```
trimmomatic={'cpus':'4'} fastqc={'version':'0.11.5'} skesa={'memory':'10GB'} pilon (abricate | prokka)
```

- And extra input can be directly inserted in any part of the pipeline. For example, it is possible to assemble genomes from both fastq files and SRR accessions (downloaded from public databases) in a single workflow:

```
download_reads trimmomatic={'extra_input':'reads'} fastqc skesa pilon
```

This pipeline can be executed by providing a file with accession numbers (`--accessions` parameter by default) **and** fastq reads, using the `--reads` parameter defined with the `extra_input` directive.

1.3 Who is FlowCraft for

FlowCraft can be useful for bioinformaticians with varied levels of expertise that need to executed genomic pipelines often and potentially in different platforms. Building and executing pipelines requires no programming knowledge, but familiarization with nextflow is highly recommended to take full advantage of the generated pipelines.

At the moment, the available pre-made processes are mainly focused on bacterial genome assembly simply because that was how we started. However, our goal is to expand the library of existing components to other commonly used tools in the field of genomics and to widen the applicability and usefulness of FlowCraft pipelines.

1.4 Why not just write a Nextflow pipeline?

In many cases, building a static nextflow pipeline is sufficient for our goals. However, when building our own pipelines, we often felt the need to add dynamism to this process, particularly if we take into account how fast new tools arise and existing ones change. Our biological goals also change over time and we might need different pipelines to answer different questions. FlowCraft makes this very easy by having a set of pre-made and ready-to-use components that can be freely assembled. By using components (`fastqc`, `trimmomatic`) as its atomic elements, very complex pielines that take full advantage of nextflow can be built with little effort. Moreover, these components have explicit and standardized input and output types, which means that the addition of new modules does not require any changes in the existing code base. They just need to take into account how data will be received by the process and how data may be emitted from the process, to ensure that it can link with other components.

However, why not both?

FlowCraft generates a complete Nextflow pipeline file, which ca be used as a starting point for your customized processes!

CHAPTER 2

Installation

2.1 User installation

FlowCraft is available as a bioconda package, which already comes with nextflow:

```
conda install flowcraft
```

Alternatively, you can install only FlowCraft, via pip:

```
pip install flowcraft
```

You will also need a container engine (see *Container engine* below)

2.2 Container engine

All components of FlowCraft are executed in docker containers, which means that you'll need to have a container engine installed. The container engines available are the ones supported by Nextflow:

- Docker,
- Singularity
- Shifter (undocumented)

If you already have any one of these installed, you are good to go. If not, you'll need to install one. We recommend singularity because it does not require the processes to run on a separate root daemon.

2.2.1 Singularity

Singularity is available to download and install here. Make sure that you have singularity v2.5.x or higher. Note that singularity should be installed as root and available on the machine(s) that will be running the nextflow processes.

Important: Singularity is available as a bioconda package. However, conda installs singularity in user space without root privileges, which may prevent singularity images from being correctly downloaded. **Therefore it is not recommended that you install singularity via bioconda**.

2.2.2 Docker

Docker can be installed following the instructions on the website: https://www.docker.com/community-edition#/download. To run docker as anon-root user, you'll need to following the instructions on the website: https://docs.docker.com/install/linux/linux-postinstall/#manage-docker-as-a-non-root-user

2.3 Developer installation

If you are looking to contribute to FlowCraft or simply interested in tweaking it, clone the github repository and its submodule and then run setup.py:

```
git clone https://github.com/assemblerflow/flowcraft.git
cd flowcraft
git submodule update --init --recursive
python3 setup.py install
```

CHAPTER 3

About

FlowCraft is developed by the Molecular Microbiology and Infection Unit (UMMI) at the Instituto de Medicina Molecular Joao Antunes.

This project is licensed under the GPLv3 license. The source code of FlowCraft is available at https://github.com/assemblerflow/flowcraft and the webservice is available at https://github.com/assemblerflow/flowcraft-webapp.

CHAPTER 4

Basic Usage

FlowCraft has currently two execution modes, `build` and `inspect`, that are used to build and inspect the nextflow pipeline, respectively. However, a `report` mode is also being developed.

4.1 Build

4.1.1 Assembling a pipeline

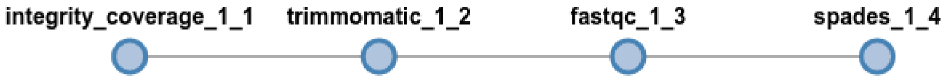
Pipelines are generated using the `build` mode of FlowCraft and the `-t` parameter to specify the *components* inside quotes:

```
flowcraft build -t "trimmomatic fastqc spades" -o my_pipe.nf
```

All components should be written inside quotes and be space separated. This command will generate a linear pipeline with three components on the current working directory (for more features and tips on how pipelines can be built, see the *pipeline building* section). **A linear pipeline means that there are no bifurcations between components, and the input data will flow linearly.**

The rationale of how the data flows across the pipeline is simple and intuitive. Data enters a component and is processed in some way, which may result on the creation of result files (stored in the `results` directory) and reports files (stored in the `reports` directory) (see *Results and reports* below). If that component has an `output_type`, it will feed the processed data into the next component (or components) and this will repeated until the end of the pipeline.

If you are interesting in checking the pipeline DAG tree, open the `my_pipe.html` file (same name as the pipeline with the html extension) in any browser.



The integrity_coverage component is a dependency of trimmomatic, so it was automatically added to the pipeline.

Important: Not all pipeline configurations will work. **You always need to ensure that the output type of a component matches the input type of the next component**, otherwise FlowCraft will exit with an error.

4.1.2 Pipeline directory

In addition to the main nextflow pipeline file (my_pipe.nf), FlowCraft will write several auxiliary files that are necessary for the pipeline to run. The contents of the directory should look something like this:

```
$ ls
bin                lib           my_pipe.nf       params.config    templates
containers.config  my_pipe.html  nextflow.config  profiles.config  resources.config ␣
↪user.config
```

You do not have to worry about most of these files. However, the *.config files can be modified to change several aspects of the pipeline run (see *Pipeline configuration* for more details). Briefly:

- params.config: Contains all the available parameters of the pipeline (see *Parameters* below). These can be changed here, or provided directly on run-time (e.g.: nextflow run --fastq value).
- resources.config: Contains the resource directives of the pipeline processes, such as cpus, allocated RAM and other nextflow process directives.
- containers.config: Specifies the container and version tag of each process in the pipeline.
- profiles.config: Contains a number of predefined profiles of executor and container engine.
- user.config: Empty configuration file that is not over-written if you build another pipeline in the same directory. Used to set persistent configurations across different pipelines.

4.1.3 Parameters

The parameters of the pipeline can be viewed by running the pipeline file with nextflow and using the --help option:

```
$ nextflow run my_pipe.nf --help
N E X T F L O W  ~  version 0.30.1
Launching `my_pipe.nf` [kickass_mcclintock] - revision: 480b3455ba

============================================================
                F L O W C R A F T
============================================================
Built using flowcraft v1.2.1.dev1


Usage:
    nextflow run my_pipe.nf

       --fastq                     Path expression to paired-end fastq files.␣
↪(default: fastq/*_{1,2}.*) (default: 'fastq/*_{1,2}.*')

       Component 'INTEGRITY_COVERAGE_1_1'
       ----------------------------------
```

(continues on next page)

(continued from previous page)

```
       --genomeSize_1_1            Genome size estimate for the samples in Mb. It is␣
↪used to estimate the coverage and other assembly parameters andchecks (default: 1)
       --minCoverage_1_1           Minimum coverage for a sample to proceed. By␣
↪default it's setto 0 to allow any coverage (default: 0)

       Component 'TRIMMOMATIC_1_2'
       ---------------------------
       --adapters_1_2              Path to adapters files, if any. (default: 'None')
       --trimSlidingWindow_1_2     Perform sliding window trimming, cutting once the␣
↪average quality within the window falls below a threshold (default: '5:20')
       --trimLeading_1_2           Cut bases off the start of a read, if below a␣
↪threshold quality (default: 3)
       --trimTrailing_1_2          Cut bases of the end of a read, if below a␣
↪threshold quality (default: 3)
       --trimMinLength_1_2         Drop the read if it is below a specified length ␣
↪(default: 55)

       Component 'FASTQC_1_3'
       ----------------------
       --adapters_1_3              Path to adapters files, if any. (default: 'None')

       Component 'SPADES_1_4'
       ----------------------
       --spadesMinCoverage_1_4     The minimum number of reads to consider an edge in␣
↪the de Bruijn graph during the assembly (default: 2)
       --spadesMinKmerCoverage_1_4 Minimum contigs K-mer coverage. After assembly␣
↪only keep contigs with reported k-mer coverage equal or above this value (default:␣
↪2)
       --spadesKmers_1_4           If 'auto' the SPAdes k-mer lengths will be␣
↪determined from the maximum read length of each assembly. If 'default', SPAdes will␣
↪use the default k-mer lengths.  (default: 'auto')
```

All these parameters are specific to the components of the pipeline. However, the main input parameter (or parameters) of the pipeline is always available. **In this case, since the pipeline started with fastq paired-end files as the main input, the `--fastq` parameter is available.** If the pipeline started with any other input type or with more than one input type, the appropriate parameters will appear (more information in the *raw input types* section).

The parameters are composed by their name (`adapters`) followed by the ID of the process it refers to (`_1_2`). The IDs can be consulted in the DAG tree (See *Assembling a pipeline*). This is done to prevent issues when duplicating components and, as such, **all parameters will be independent between different components**. This behaviour can be changed when building the pipeline by using the `--merge-params` option (See *Merge parameters*).

Note: The `--merge-params` option of the `build` mode will merge all parameters with identical names (*e.g.:* `--genomeSize_1_1` and `--genomeSize_1_5` become simply `--genomeSize`) . This is usually more appropriate and useful in linear pipelines without component duplication.

Providing/modifying parameters

These parameters can be provided on run-time:

```
nextflow run my_pipe.nf --genomeSize_1_1 5 --adapters_1_2 "/path/to/adapters"
```

or edited in the `params.config` file:

```
params {
    genomeSize_1_1 = 5
    adapters_1_2 = "path/to/adapters"
}
```

Most parameters in FlowCraft's components already come with sensible defaults, which means that usually you'll only need to provide a small number of arguments. In the example above, the `--fastq` is the only parameter required. I have placed fastq files on the `data` directory:

```
$ ls data
sample_1.fastq.gz  sample_2.fastq.gz
```

We'll need to provide the pattern to the fastq files. This pattern is perhaps a bit confusing at first, but it's necessary for the correct inference of the paired:

```
--fastq "data/*_{1,2}.*"
```

In this case, the pairs are separated by the "_1." or "_2." substring, which leads to the pattern `*_{1,2}.*`. Another common nomenclature for paired fastq files is something like `sample_R1_L001.fastq.gz`. In this case, an acceptable pattern would be `*_R{1,2}_*`.

Important: Note the quotes around the fastq path pattern. These quotes are necessary to allow nextflow to resolve the pattern, otherwise your shell might try to resolve it and provide the wrong input to nextflow.

4.2 Execution

Once you build your pipeline with Flowcraft you have a standard nextflow pipeline ready to run. Therefore, all you need to do is:

```
nextflow run my_pipe.nf --fastq "data/*_{1,2}.*
```

4.2.1 Changing executor and container engine

The default run mode of an FlowCraft pipeline is to be executed locally and using the singularity container engine. In nextflow terms, this is equivalent to have `executor = "local"` and `singularity.enabled = true`. If you want to change these settings, you can modify the `nextflow.config` file, or use one of the available profiles in the `profiles.config` file. These profiles provide a combination of common `<executor>_<container_engine>` that are supported by nextflow. Therefore, if you want to run the pipeline on a cluster with SLURM and shifter, you'll just need to specify the ‘‘ slurm_shifter‘‘ profile:

```
nextflow run my_pipe.nf --fastq "data/*_{1,2}.*" -profile slurm_shifter
```

Common executors include:

- `slurm`
- `sge`
- `lsf`
- `pbs`

Other container engines are:

- docker
- singularity
- shifter

4.2.2 Docker images

All components of FlowCraft are executed in containers, which means that the first time they are executed in a machine, **the corresponding image will have to be downloaded**. In the case of docker, images are pulled and stored in var/lib/docker by default. In the case of singularity, the nextflow.config generated by FlowCraft sets the cache dir for the images at $HOME/.singularity_cache. Note that when an image is downloading, nextflow does not display any informative message, except for singularity where you'll get something like:

```
Pulling Singularity image docker://ummidock/trimmomatic:0.36-2 [cache /home/
↪diogosilva/.singularity_cache/ummidock-trimmomatic-0.36-2.img]
```

So, if a process seems to take too long to run the first time, it's probably because the image is being downloaded.

4.2.3 Results and reports

As the pipeline runs, processes may write result and report files to the results and reports directories, respectively. For example, the reports of the pipeline above, would look something like this:

```
reports
├── coverage_1_1
│   └── estimated_coverage_initial.csv
├── fastqc_1_3
│   ├── FastQC_2run_report.csv
│   ├── run_2
│   │   ├── sample_1_0_summary.txt
│   │   └── sample_1_1_summary.txt
│   ├── sample_1_1_trim_fastqc.html
│   └── sample_1_2_trim_fastqc.html
└── status
    ├── master_fail.csv
    ├── master_status.csv
    └── master_warning.csv
```

The estimated_coverage_initial.csv file contains a very rough coverage estimation for each sample, the fastqc* directory contains the html reports and summary files of FastQC for each sample, and the status directory contains a log of the status, warnings and fails of each process for each sample.

The actual results for each process that produces them, are stored in the results directory:

```
results
├── assembly
│   └── spades_1_4
│       └── sample_1_trim_spades3111.fasta
└── trimmomatic_1_2
    ├── sample_1_1_trim.fastq.gz
    └── sample_1_2_trim.fastq.gz
```

If you are interested in checking the actual environment where the execution of a particular process occurred for any given sample, you can inspected the pipeline_stats.txt file in the root of the pipeline directory. This file contains rich information about the execution of each process, including the working directory:

```
task_id hash        process         tag          status       exit     start                   ␣
↪       container                            cpus     duration      realtime      queue
↪%cpu    %mem     rss      vmem
5       7c/cae270    trimmomatic_1_2 sample_1     COMPLETED    0        2018-04-12␣
↪11:42:29.599 docker:ummidock/trimmomatic:0.36-2  2        1m 25s        1m 17s        - ␣
↪      329.3%  1.1%     1.5 GB  33.3 GB
```

The hash column contains the start of the current working directory of that process. In the example below, the directory would be:

```
work/7c/cae270*
```

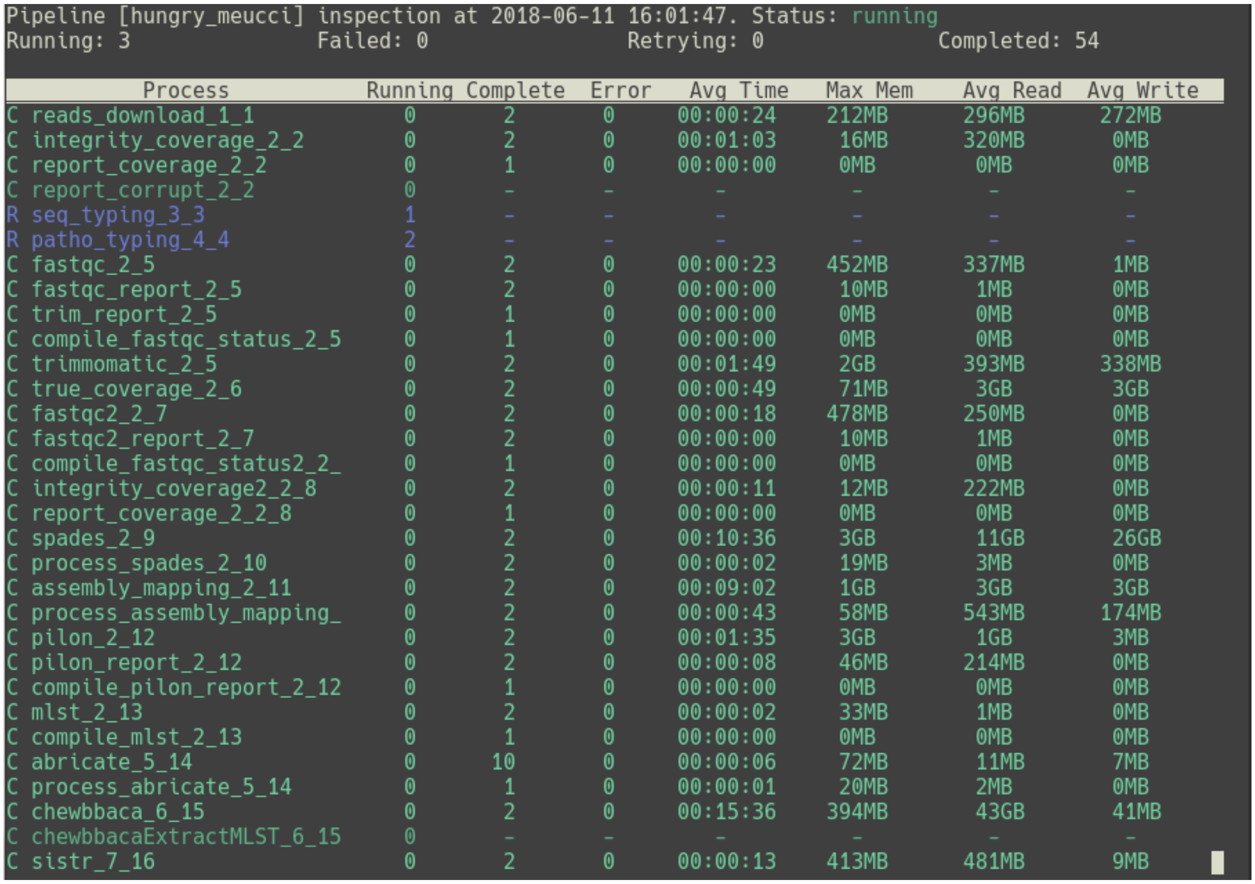
4.3 Inspect

FlowCraft has two options (overview and broadcast) for inspecting the progress of a pipeline that is running locally, either in a personal computer or a server machine. In both cases, the progress of the pipeline will be continuously updated in real-time.

4.3.1 In a terminal

To open inspect in the terminal just write the following command **on the folder that the pipeline is running**:

```
flowcraft inspect
```



```
Pipeline [hungry_meucci] inspection at 2018-06-11 16:01:47. Status: running
Running: 3               Failed: 0               Retrying: 0               Completed: 54

          Process              Running Complete  Error   Avg Time   Max Mem    Avg Read  Avg Write
C reads_download_1_1              0       2        0     00:00:24   212MB      296MB      272MB
C integrity_coverage_2_2          0       2        0     00:01:03    16MB      320MB      0MB
C report_coverage_2_2             0       1        0     00:00:00   0MB        0MB        0MB
C report_corrupt_2_2              0       -        -        -        -          -          -
R seq_typing_3_3                  1       -        -        -        -          -          -
R patho_typing_4_4                2       -        -        -        -          -          -
C fastqc_2_5                      0       2        0     00:00:23   452MB      337MB      1MB
C fastqc_report_2_5               0       2        0     00:00:00    10MB      1MB        0MB
C trim_report_2_5                 0       1        0     00:00:00   0MB        0MB        0MB
C compile_fastqc_status_2_5       0       1        0     00:00:00   0MB        0MB        0MB
C trimmomatic_2_5                 0       2        0     00:01:49   2GB        393MB      338MB
C true_coverage_2_6               0       2        0     00:00:49    71MB      3GB        3GB
C fastqc2_2_7                     0       2        0     00:00:18   478MB      250MB      0MB
C fastqc2_report_2_7              0       2        0     00:00:00    10MB      1MB        0MB
C compile_fastqc_status2_2_       0       1        0     00:00:00   0MB        0MB        0MB
C integrity_coverage2_2_8         0       2        0     00:00:11    12MB      222MB      0MB
C report_coverage_2_2_8           0       1        0     00:00:00   0MB        0MB        0MB
C spades_2_9                      0       2        0     00:10:36   3GB         11GB      26GB
C process_spades_2_10             0       2        0     00:00:02    19MB      3MB        0MB
C assembly_mapping_2_11           0       2        0     00:09:02   1GB        3GB        3GB
C process_assembly_mapping_       0       2        0     00:00:43    58MB      543MB      174MB
C pilon_2_12                      0       2        0     00:01:35   3GB        1GB        3MB
C pilon_report_2_12               0       2        0     00:00:08    46MB      214MB      0MB
C compile_pilon_report_2_12       0       1        0     00:00:00   0MB        0MB        0MB
C mlst_2_13                       0       2        0     00:00:02    33MB      1MB        0MB
C compile_mlst_2_13               0       1        0     00:00:00   0MB        0MB        0MB
C abricate_5_14                   0      10        0     00:00:06    72MB       11MB      7MB
C process_abricate_5_14           0       1        0     00:00:01    20MB      2MB        0MB
C chewbbaca_6_15                  0       2        0     00:15:36   394MB       43GB      41MB
C chewbbacaExtractMLST_6_15       0       -        -        -        -          -          -
C sistr_7_16                      0       2        0     00:00:13   413MB      481MB      9MB
```

`overview` is the default behavior of this module, but it can also be called like this:

```
flowcraft inspect -m overview
```

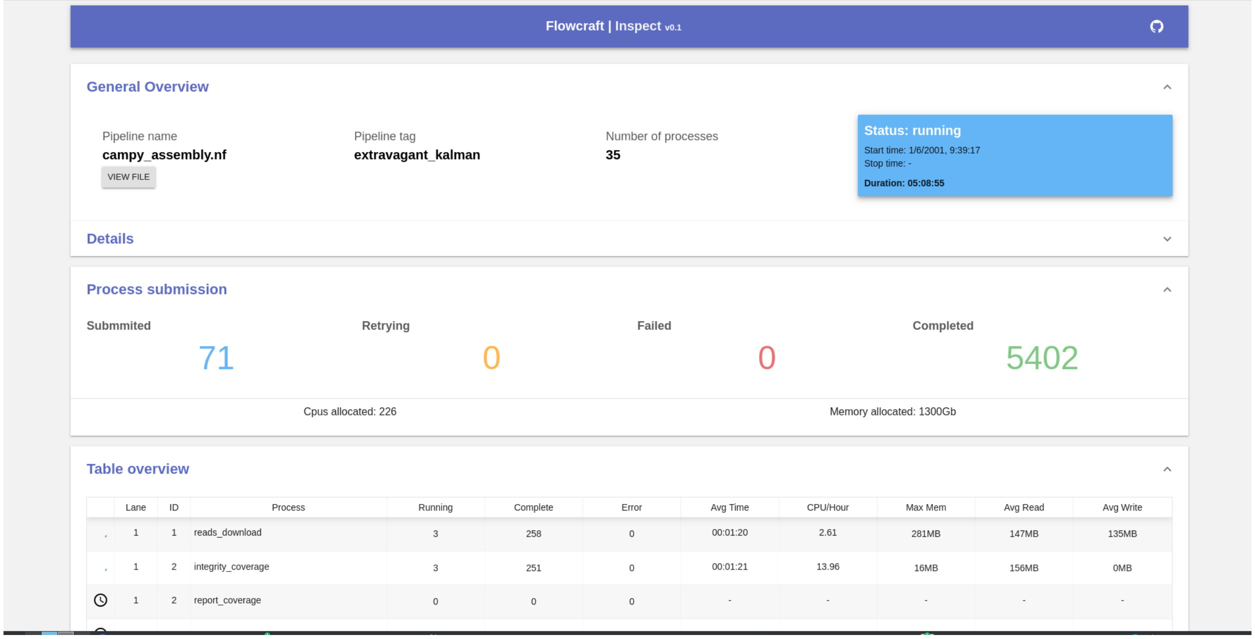
Note: To exit the inspection just type `q` or `ctrl+c`.

4.3.2 In a browser

It is also possible to track the pipeline progress in a browser in any device using the flowcraft web application. **To do so, the following command should be run in the folder where the pipeline is running**:

```
flowcraft inspect -m broadcast
```

This will output an URL to the terminal that can be opened in a browser. This is an example of the screen that is displayed once the url is opened:



General Overview

Pipeline name
campy_assembly.nf

Pipeline tag
extravagant_kalman

Number of processes
35

Status: running
Start time: 1/6/2001, 9:39:17
Stop time: -
Duration: 05:08:55

Details

Process submission

Submited	Retrying	Failed	Completed
71	0	0	5402

Cpus allocated: 226

Memory allocated: 1300Gb

Table overview

	Lane	ID	Process	Running	Complete	Error	Avg Time	CPU/Hour	Max Mem	Avg Read	Avg Write
	1	1	reads_download	3	258	0	00:01:20	2.61	281MB	147MB	135MB
	1	2	integrity_coverage	3	251	0	00:01:21	13.96	16MB	156MB	0MB
🕓	1	2	report_coverage	0	0	0	-	-	-	-	-

Important: This pipeline inspection will be available for **anyone** via the provided URL, which means that the URL can be shared with anyone and/or any device with a browser. **However, the inspection section will only be available while the `flowcraft inspect -m broadcast` command is running. Once this command is cancelled, the data will be erased from the service and the URL will no longer be available**.

4.3.3 Want to know more?

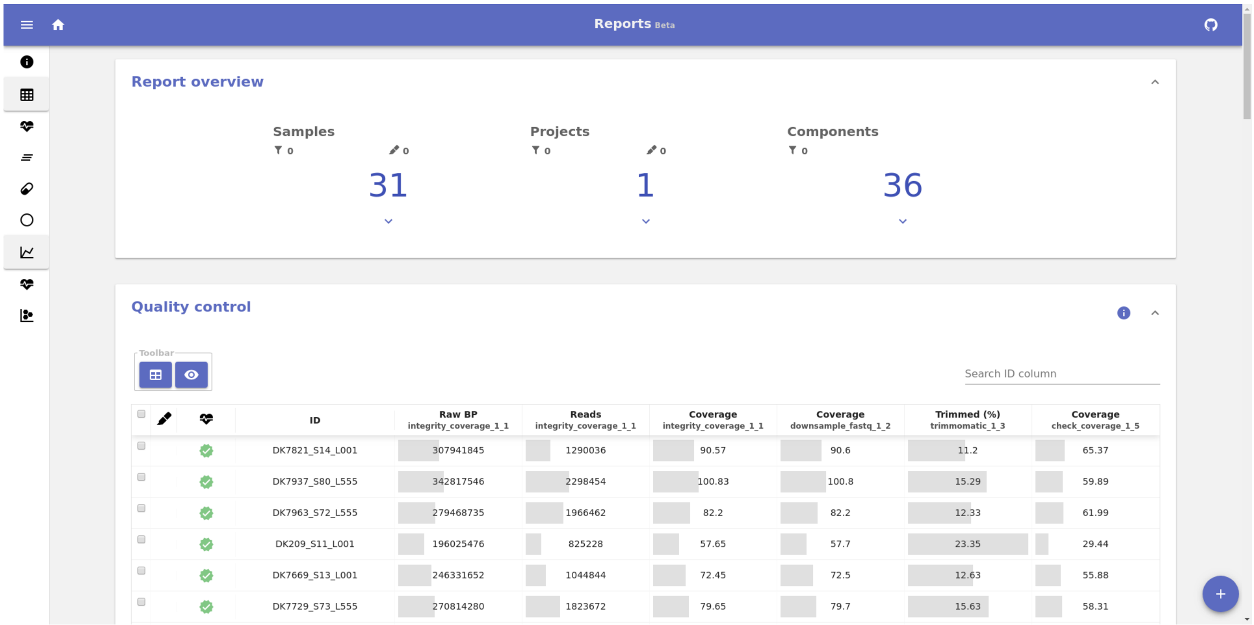
Pipeline inspection is the full documentation of the `inspect` mode.

4.4 Reports

The reporting of a FlowCraft pipeline is saved on a JSON file that is stored in `pipeline_reports/pipeline_report.json`. To visualize the reports you'll just need to execute the following command in the folder where the pipeline was executed:

```
flowcraft report
```

This will output an URL to the terminal that can be opened in a browser. This is an example of the screen that is displayed once the url is opened:



□			ID	Raw BP integrity_coverage_1_1	Reads integrity_coverage_1_1	Coverage integrity_coverage_1_1	Coverage downsample_fastq_1_2	Trimmed (%) trimmomatic_1_3	Coverage check_coverage_1_5
□		✓	DK7821_S14_L001	307941845	1290036	90.57	90.6	11.2	65.37
□		✓	DK7937_S80_L555	342817546	2298454	100.83	100.8	15.29	59.89
□		✓	DK7963_S72_L555	279468735	1966462	82.2	82.2	12.33	61.99
□		✓	DK209_S11_L001	196025476	825228	57.65	57.7	23.35	29.44
□		✓	DK7669_S13_L001	246331652	1044844	72.45	72.5	12.63	55.88
□		✓	DK7729_S73_L555	270814280	1823672	79.65	79.7	15.63	58.31

The actual layout and content of the reports will depend on the pipeline you build and it will only provide the information that is directly related to your pipeline components.

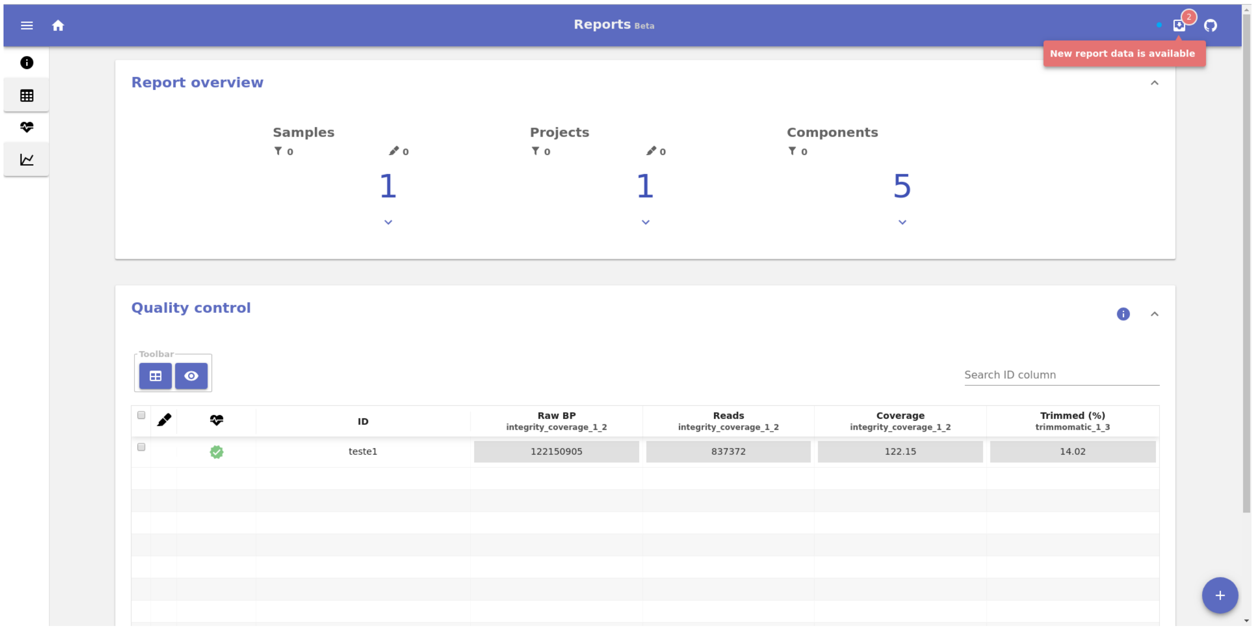
Important: This pipeline report will be available for **anyone** via the provided URL, which means that the URL can be shared with anyone and/or any device with a browser. **However, the report section will only be available while the `flowcraft report` command is running. Once this command is cancelled, the data will be erased from the service and the URL will no longer be available**.

4.4.1 Real time reports

The reports of any FlowCraft pipeline can be monitored in real-time using the `--watch` option:

```
flowcraft report --watch
```

This will output an URL exactly as in the previous section and will render the same reports page with a small addition. In the top right of the screen in the navigation bar, there will be a new icon that informs the user when new reports are available:



New report data is available

			ID	Raw BP integrity_coverage_1_2	Reads integrity_coverage_1_2	Coverage integrity_coverage_1_2	Trimmed (%) trimmomatic_1_3
□		✓	teste1	122150905	837372	122.15	14.02

4.4.2 Local visualization

The FlowCraft report JSON file can also be visualized locally by drag and dropping it into the FlowCraft web application page, currently hosted at http://www.flowcraft.live/reports

4.4.3 Offline visualization

The complete FlowCraft report is also available as a standalone HTML file that can be visualized offline. This HTML file, stored in `pipeline_reports/pipeline_report.html`, can be opened in any modern browser.

CHAPTER 5

Pipeline building

FlowCraft offers a few extra features when building pipelines using the `build` execution mode.

5.1 Raw input types

The first component (or components) you place at the start of the pipeline determine the raw input type, and the parameter for providing input data. The input type information is provided in the documentation page of each component. For instance, if the first component is FastQC, which has an input type of `FastQ`, the parameter for providing the raw input data will be `--fastq`. Here are the currently supported input types and their respective parameters:

- `FastQ`: `--fastq`
- `Fasta`: `--fasta`
- `Accessions`: `--accessions`

5.2 Merge parameters

By default, parameters in a FlowCraft pipeline are unique and independent between different components, even if the parameters have the same name and/or the components are the same. This allows for the execution of the same software using different parameters in a single workflow. The `params.config` of these pipelines will look something like:

```
params {
    /*
    Component 'trimmomatic_1_2'
    --------------------------
    */
    adapters_1_2 = 'None'
    trimSlidingWindow_1_2 = '5:20'
    trimLeading_1_2 = 3
    trimTrailing_1_2 = 3
    trimMinLength_1_2 = 55
```

(continues on next page)

(continued from previous page)

```
    /*
    Component 'fastqc_1_3'
    ---------------------
    */
    adapters_1_3 = 'None'
}
```

Notice that the `adapters` parameter occurs twice and can be independently set in each component.

If you want to override this behaviour, FlowCraft has a `--merge-params` option that merges all parameters with the same name in a single parameter, which is then equally applied to all components. So, if we generate the pipeline above with this option:

```
flowcraft build -t "trimmomatic fastqc" -o pipe.nf --merge-params
```

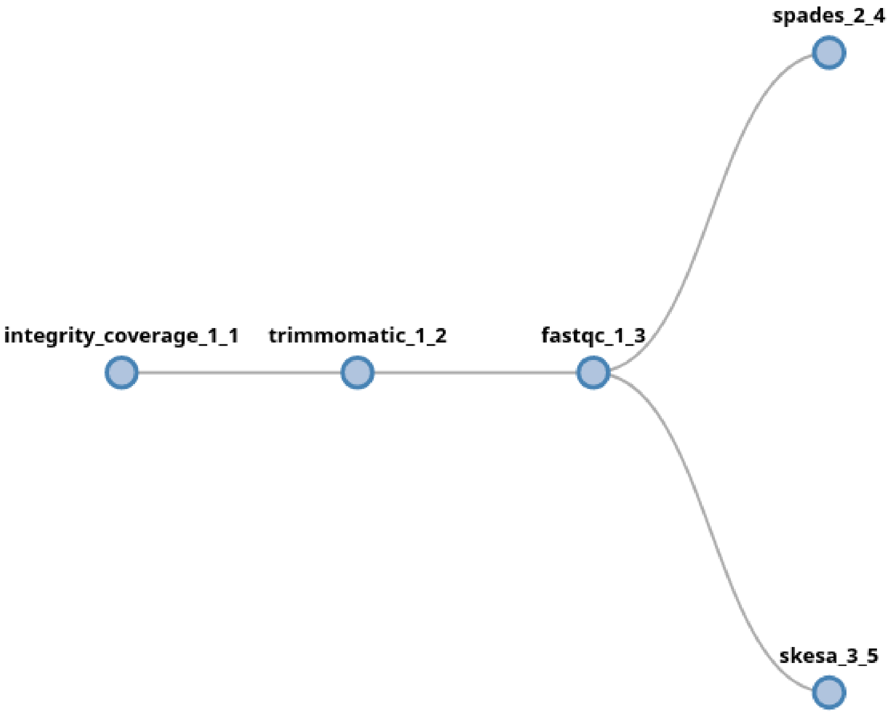
Then, the `params.config` will become:

```
params {
    adapters = 'None'
    trimSlidingWindow = '5:20'
    trimLeading = 3
    trimTrailing = 3
    trimMinLength = 5
}
```

5.3 Forks

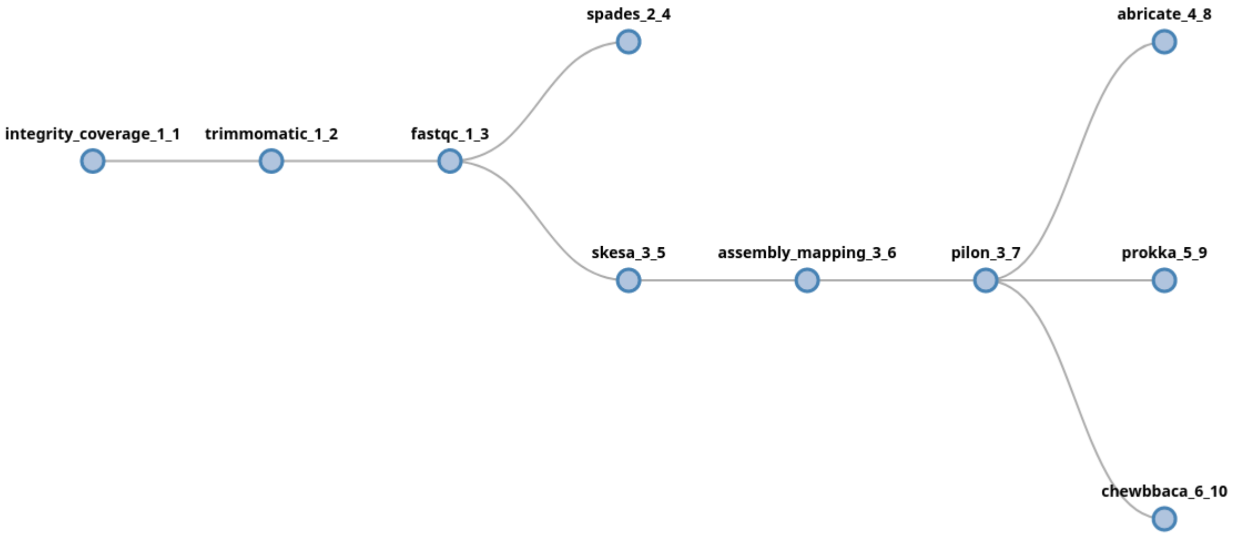
The output of any component in an FlowCraft pipeline can be forked into two or more components, using the following fork syntax:

```
trimmomatic fastqc (spades | skesa)
```



In this example, the output of `fastqc` will be fork into two new *lanes*, which will proceed independently from each other. In this syntax, a fork is triggered by the `(` symbol (and the corresponding closing `)`) and each lane will be separated by a `|` symbol. There is no limitation to the number of forks or lanes that a pipeline has. For instance, we could add more components after the `skesa` module, including another fork:

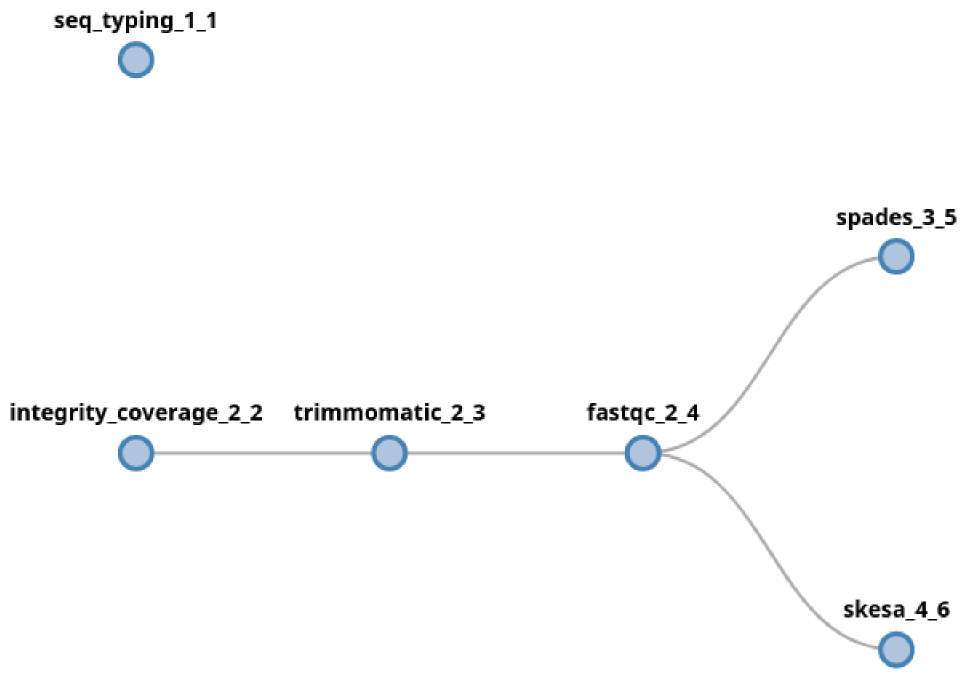
```
trimmomatic fastqc (spades | skesa pilon (abricate | prokka | chewbbaca))
```



In this example, data will be forked after `fastqc` into two new lanes, processed by `spades` and `skesa`. In the skesa lane, data will continue to flow into the `pilon` component and its output will fork into three new lanes.

It is also possible to start a fork at the beggining of the pipeline, which basically means that the pipeline will have multiple starting points. If we want to provide the raw input two multiple process, the fork syntax can start at the beginning of the pipeline:

```
(seq_typing | trimmomatic fastqc (spades | skesa))
```



In this case, since both initial components (`seq_typing` and `integrity_coverage`) received fastq files as input, the data provided via the `--fastq` parameter will be forked and provided to both processes.

Note: Some components have dependencies which need to be included previously in the pipeline. For instance, `trimmomatic` requires `integrity_coverage` and `pilon` requires `assembly_mapping`. By default, FlowCraft will insert any missing dependencies right before the process, which is why these components appear in the figures above.

Warning: Pay special attention to the syntax of the pipeline string when using forks. However, when unable to parse it, FlowCraft will do its best to inform you where the parsing error occurred.

5.4 Directives

Several directives with information on cpu usage, RAM, version, etc. can be specified for each individual component when building the pipeline using the `={}` notation. These directives are written to the `resources.config` and `containers.config` files that are generated in the pipeline directory. You can pass any of the directives already supported by nextflow (https://www.nextflow.io/docs/latest/process.html#directives), but the most commonly used include:

- `cpus`
- `memory`
- `queue`

In addition, you can also pass the `container` and `version` directives which are parsed by FlowCraft to dynamically change the container and/or version tag of any process.

Here is an example where we specify cpu usage, allocated memory and container version in the pipeline string:

```
flowcraft build -t "fastqc={'version':'0.11.5'} \
                        trimmomatic={'cpus':'2'} \
                        spades={'memory':'\'10GB\''}" -o my_pipeline.nf
```

When a directive is not specified, it will assume the default value of the nextflow directive.

Warning: Take special care not to include any white space characters inside the directives field. Common mistakes occur when specifying directives like `fastqc={'version':  '0.11.5'}`.

Note: The values specified in these directives are placed in the respective config files exactly as they are. For instance, `spades={'memory':'10GB'}"` will appear in the config as `spades.memory = 10Gb`, which will raise an error in nextflow because `10Gb` should be a string. Therefore, if you want a string you'll need to add the `'` as in this example: `spades={'memory':'\'10GB\''}"`. The reason why these directives are not automatically converted is to allow the specification of dynamic computing resources, such as `spades={'memory':'{10.Gb*task.attempt}'}"`

5.5 Extra inputs

By default, only the first process (or processes) in a pipeline will receive the raw input data provided by the user. However, the `extra_input` special directive allows one or more processes to receive input from an additional parameter that is provided by the user:

```
reads_download integrity_coverage={'extra_input':'local'} trimmomatic spades
```

The default main input of this pipeline is a text file with accession numbers for the `reads_download` component. The `extra_input` creates a new parameter, named `local` in this example, that allows us to provide additional input data to the `integrity_coverage` component directly:

```
nextflow run pipe.nf --accessions accession_list.txt --local "fastq/*_{1,2}.*"
```

What will happen in this pipeline, is that the fastq files provided to the `integrity_coverage` component will be mixed with the ones provided by the `reads_download` component. Therefore, if we provide 10 accessions and 10 fastq samples, we'll end up with 20 samples being processed by the end of the pieline.

It is important to note that the extra input parameter expected data compliant with the input type of the process. If files other than fastq files would be provided in the pipeline above, this would result in a pipeline error.

If the `extra_input` directive is used on a component that has a different input type from the first component in the pipeline, it is possible to use the `default` value:

```
trimmomatic spades abricate={'extra_input':'default'}
```

In this case, the input type of the first component if fastq and the input type of `abricate` is fasta. The `default` value will make available the default parameter for fasta raw input, which is `fasta`:

```
nextflow run pipe.nf --fastq "fastq/*_{1,2}.*" --fasta "fasta/*.fasta"
```

5.6 Pipeline file

Instead of providing the pipeline components via the command line, you can specify them in a text file:

```
# my_pipe.txt
trimmomatic fastqc spades
```

And then provide the pipeline file to the `-t` parameter:

```
flowcraft build -t my_pipe.txt -o my_pipe.nf
```

Pipeline files are usually more readable, particularly when they become more complex. Consider the following example:

```
integrity_coverage (
    spades={'memory':'\'50GB\''} |
    skesa={'memory':'\'40GB\'','cpus':'4'} |
    trimmomatic fastqc (
        spades pilon (abricate={'extra_input':'default'} | prokka) |
        skesa pilon (abricate | prokka)
    )
)
```

In addition to be more readable, it is also easier to edit, re-use and share.

CHAPTER 6

Pipeline configuration

When a nextflow pipeline is built with FlowCraft, a number of configuration files are automatically generated in the same directory. They are all imported at the end of the `nextflow.config` file and are sorted by their configuration role. All configuration files are overwritten if you build another pipeline in the same directory, with the exception of the `user.config` file, which is meant to be a persistent configuration file.

6.1 Parameters

The `params.config` file includes all available paramenters for the pipeline and their respective default values. Most of these parameters already contain sensible defaults.

6.2 Resources

The `resources.config` file includes the majority of the directives provided for each process, including `cpus` and `memory`. You'll note that each process name has a suffix like `_1_1`, which is a unique process identifier composed of `<lane>_<process_number>`. This ensures that even when the same component is specified multiple times in a pipeline, you'll still be able to set directives for each one individually.

6.3 Containers

The `containers.config` file includes the container directive for each process in the pipeline. These containers are retrieved from dockerhub, if they do not exist locally yet. You can change the container string to any other value, but it should point to an image that exist on dockerhub or locally.

6.4 Profiles

The `profiles.config` file includes a set of pre-made profiles with all possible combinations of executors and container engines. You can add new ones or modify existing one.

6.5 User configutations

The `user.config` file is configuration file that is not overwritten when a new pipeline is build in the same directory. It can contain any configuration that is supported by nextflow and will overwrite all other configuration files.

CHAPTER 7

Pipeline inspection

FlowCraft offers an `inspect` mode for tracking the progress of a nextflow pipeline either directly in a terminal (`overview`) or by broadcasting information to the flowcraft web application (`broadcast`).

Note: This mode was design for nextflow pipelines generated by FlowCraft. It should be possible to inspect any nextflow pipeline, provided that the requirements below are met, but compatibility it's not guaranteed.

How it works: Simply run `flowcraft inspect -m <mode>` in the directory where the pipeline is running. In either run mode, FlowCraft will keep running (until you cancel it) and continuously update the progress of a pipeline. If the pipeline is interrupted or fails for some reason, FlowCraft should be able to correctly reset the inspection automatically when resuming its execution.

7.1 Requirements for inspect

While the `inspect` mode is running, it will parse the information written into two files that are generated by nextflow:

- `.nextflow.log`: The log file that is automatically generated by nextflow.
- `trace file`: The trace file that is generated by nextflow when using the `-with-trace` option. By default, it searches for the `pipeline_stats.txt` file, but this can be changed using the `-i` option.

7.2 Trace fields

FlowCraft parses several fields of the trace file, but only a few are mandatory for its execution. If the trace file does not contain any of the optional fields, that information will simply not appear on the terminal or web app. Nevertheless, to take full advantage of the inspect mode, the following trace fields should be present:

- **Mandatory:**

  - `tag`: The tag of the nextflow process. Flowcraft assumes that this is a string with only the sample name (e.g.: *SampleA*). While this is not strictly required, providing strings with other information (e.g.: *Running bowtie for sampleA*) may result in some inconsistencies in the inspection.
  - `task_id`: The task ID is used to skip entries that have already been parsed.

- **Optional:**
  - `hash`: Used to get the work directory the process execution.
  - `cpus`, `%cpu`, `memory`, `rss`, `rchar` and `wchar`: Used for statistics of computational resources.

Note: Any additional fields present in the trace file are ignored.

7.3 Usage

```
flowcraft inspect --help
usage: flowcraft inspect [-h] [-i TRACE_FILE] [-r REFRESH_RATE]
                         [-m {overview,broadcast}] [-u URL] [--pretty]

optional arguments:
  -h, --help            show this help message and exit
  -i TRACE_FILE         Specify the nextflow trace file.
  -r REFRESH_RATE       Set the refresh frequency for the continuous inspect
                        functions
  -m {overview,broadcast}, --mode {overview,broadcast}
                        Specify the inspection run mode.
  -u URL, --url URL     Specify the URL to where the data should be broadcast
  --pretty              Pretty inspection mode that removes usual reporting
                        processes.
```

- `-i`: Used to specify the path to the trace file that should be parsed. By default, FlowCraft will try to parse the `pipeline_stats.txt` file in current working directory.
- `-r`: Sets the time interval in seconds between each parsing of the relevant nextflow files. By default it is set to `0.01`.
- `-m`: The inspection mode. `overview` is the terminal display while `broadcast` sends the data to FlowCraft's web service.
- `-u`: The URL of FlowCraft's web service. By default it is already set to the main service and you do not need to specify it. It is only useful when the service is running on local host or in other custom instance.
- `--pretty`: By default the inspection shows the progress of all processes in the pipeline. Using this option filters the processes to the most relevant ones of FlowCraft's pipelines.

CHAPTER 8

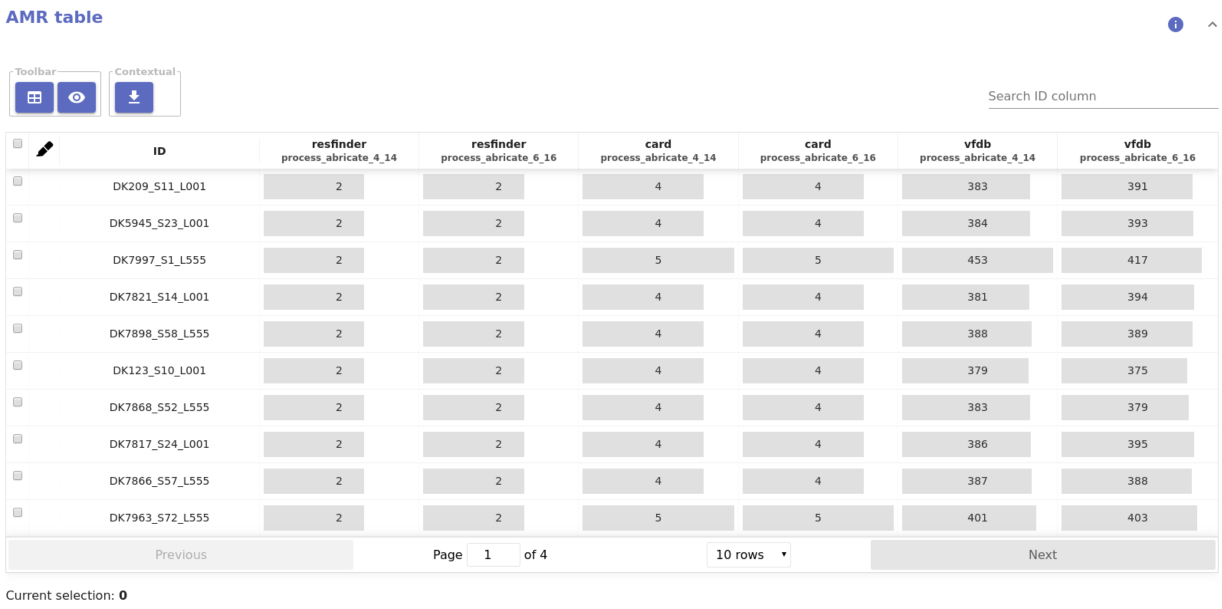
Pipeline reports

8.1 abricate

8.1.1 Table data

AMR table:

- **<abricate database>**: Number of hits for a particular given database



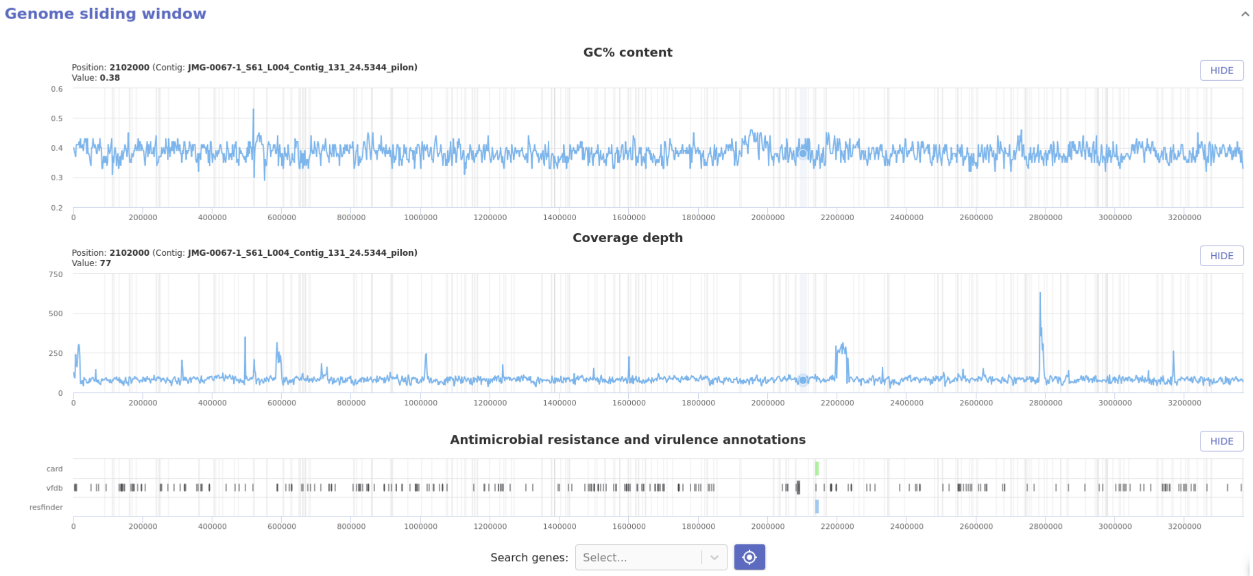
AMR table

□	✎	ID	resfinder process_abricate_4_14	resfinder process_abricate_6_16	card process_abricate_4_14	card process_abricate_6_16	vfdb process_abricate_4_14	vfdb process_abricate_6_16
□		DK209_S11_L001	2	2	4	4	383	391
□		DK5945_S23_L001	2	2	4	4	384	393
□		DK7997_S1_L555	2	2	5	5	453	417
□		DK7821_S14_L001	2	2	4	4	381	394
□		DK7898_S58_L555	2	2	4	4	388	389
□		DK123_S10_L001	2	2	4	4	379	375
□		DK7868_S52_L555	2	2	4	4	383	379
□		DK7817_S24_L001	2	2	4	4	386	395
□		DK7866_S57_L555	2	2	4	4	387	388
□		DK7963_S72_L555	2	2	5	5	401	403

Current selection: **0**

8.1.2 Plot data

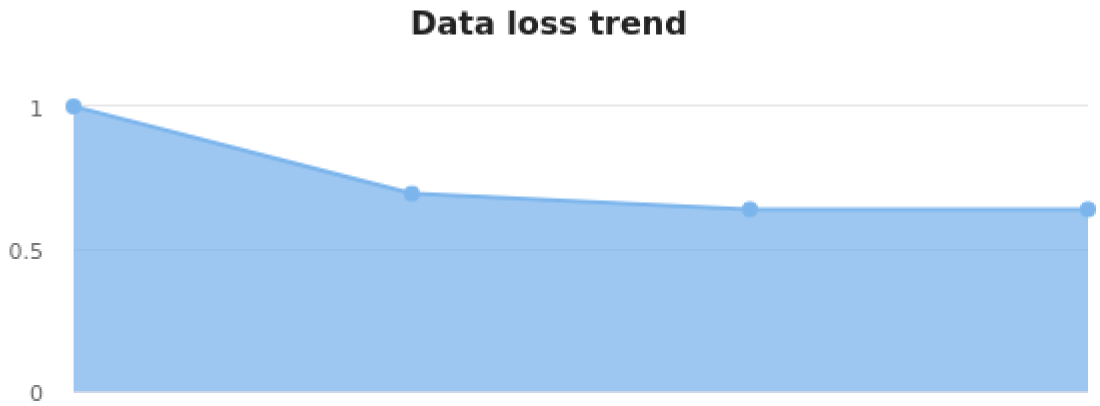
- **Sliding window AMR annotation**: Provides annotation of Abricate hits for each database along the genome. This report component is only available when the `pilon` component was used downstream of `abricate`.



8.2 assembly_mapping

8.2.1 Plot data

- **Data loss chart**: Gives a trend of the data loss (in total number of base pairs) across components that may filter this data.



8.2.2 Warnings

Assembly table:

- When the number of contigs exceeds the threshold of 100 contigs per 1.5Mb.

8.2.3 Fails

Assembly table:

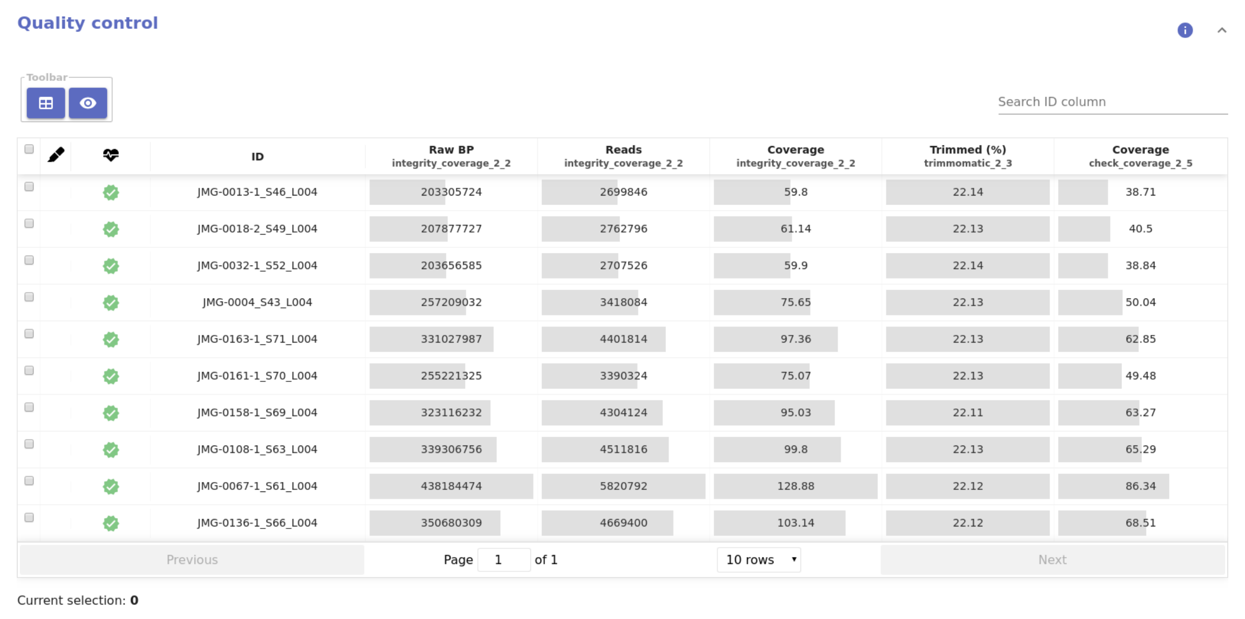
- When the assembly size if smaller than 80% or larger than 150% of the expected genome size.

8.3 check_coverage

8.3.1 Table data

Quality control table:

- **Coverage**: Estimated coverage based on the number of base pairs and the expected genome size.



□			ID	Raw BP integrity_coverage_2_2	Reads integrity_coverage_2_2	Coverage integrity_coverage_2_2	Trimmed (%) trimmomatic_2_3	Coverage check_coverage_2_5
□		✓	JMG-0013-1_S46_L004	203305724	2699846	59.8	22.14	38.71
□		✓	JMG-0018-2_S49_L004	207877727	2762796	61.14	22.13	40.5
□		✓	JMG-0032-1_S52_L004	203656585	2707526	59.9	22.14	38.84
□		✓	JMG-0004_S43_L004	257209032	3418084	75.65	22.13	50.04
□		✓	JMG-0163-1_S71_L004	331027987	4401814	97.36	22.13	62.85
□		✓	JMG-0161-1_S70_L004	255221325	3390324	75.07	22.13	49.48
□		✓	JMG-0158-1_S69_L004	323116232	4304124	95.03	22.11	63.27
□		✓	JMG-0108-1_S63_L004	339306756	4511816	99.8	22.13	65.29
□		✓	JMG-0067-1_S61_L004	438184474	5820792	128.88	22.12	86.34
□		✓	JMG-0136-1_S66_L004	350680309	4669400	103.14	22.12	68.51

Current selection: **0**

8.3.2 Warnings

Quality control table:

- When the enconding and phred score cannot be guessed from the FastQ file(s).

8.3.3 Fails

Quality control table:

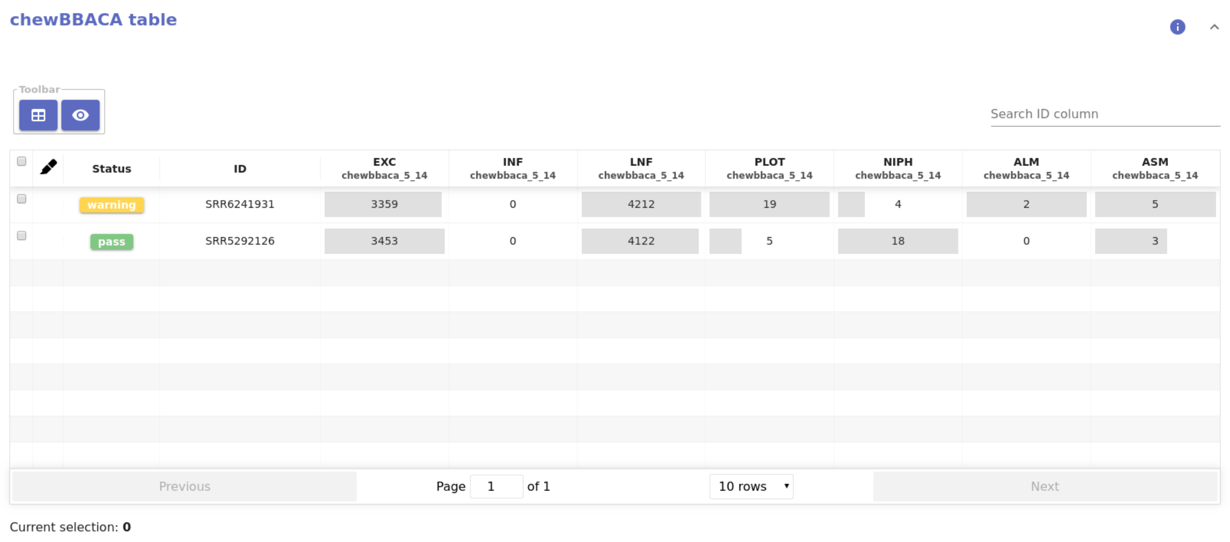
- When the sample has lower estimated coverage than the provided coverage threshold.

8.4 chewbbaca

8.4.1 Table data

Chewbbaca table:

- Table with the summary statistics of ChewBBACA allele calling, including the number of exact matches, inferred loci, loci not found, etc.



chewBBACA table

□	Status	ID	EXC chewbbaca_5_14	INF chewbbaca_5_14	LNF chewbbaca_5_14	PLOT chewbbaca_5_14	NIPH chewbbaca_5_14	ALM chewbbaca_5_14	ASM chewbbaca_5_14
□	warning	SRR6241931	3359	0	4212	19	4	2	5
□	pass	SRR5292126	3453	0	4122	5	18	0	3

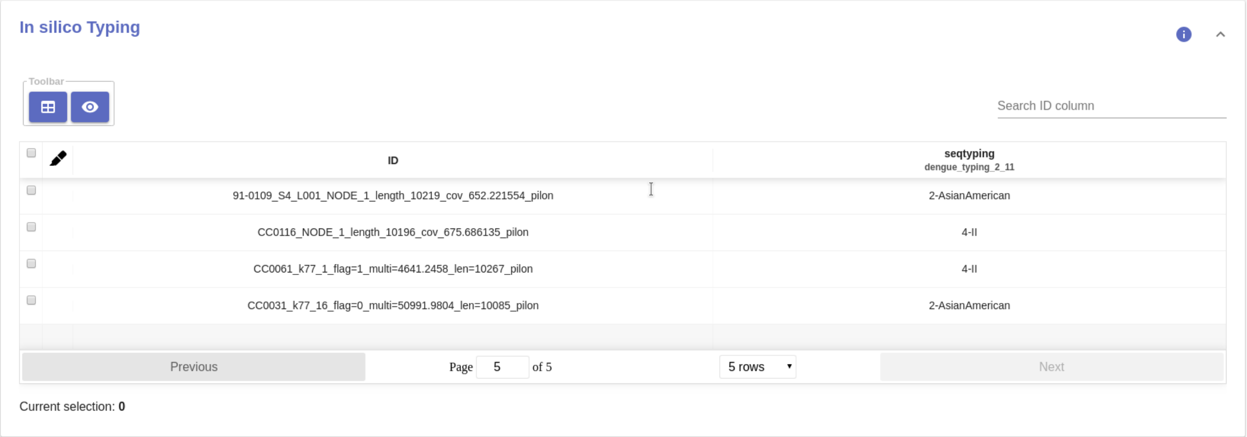
Current selection: **0**

8.5 dengue_typing

8.5.1 Table data

Typing table:

- **seqtyping**: The sequence typing result (serotypy-genotype).



In silico Typing

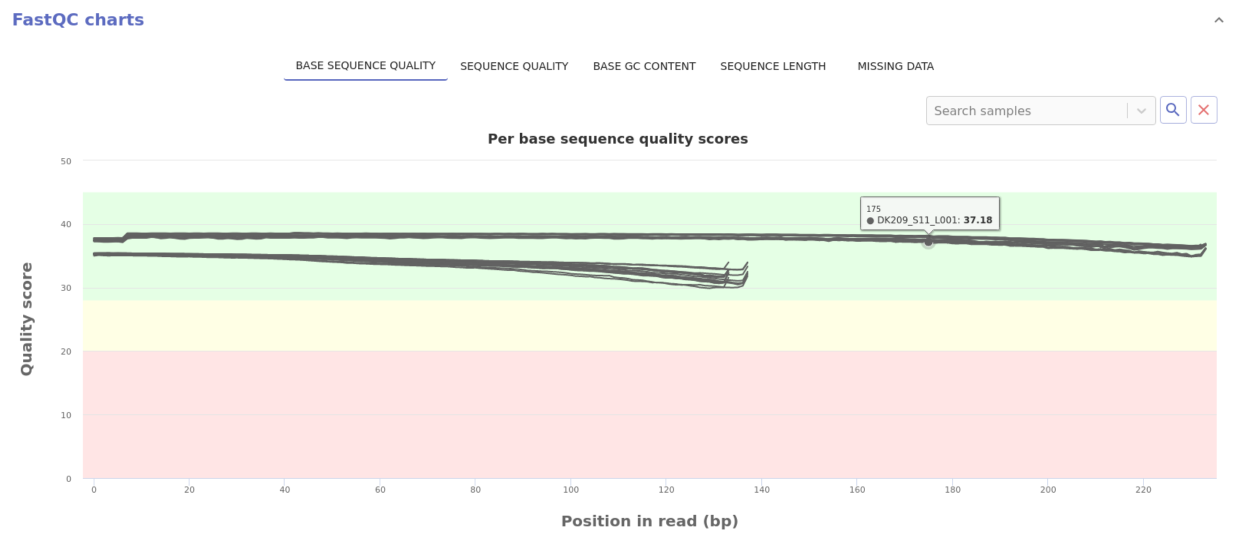
□	ID	seqtyping dengue_typing_2_11
□	91-0109_S4_L001_NODE_1_length_10219_cov_652.221554_pilon	2-AsianAmerican
□	CC0116_NODE_1_length_10196_cov_675.686135_pilon	4-II
□	CC0061_k77_1_flag=1_multi=4641.2458_len=10267_pilon	4-II
□	CC0031_k77_16_flag=0_multi=50991.9804_len=10085_pilon	2-AsianAmerican

Current selection: **0**

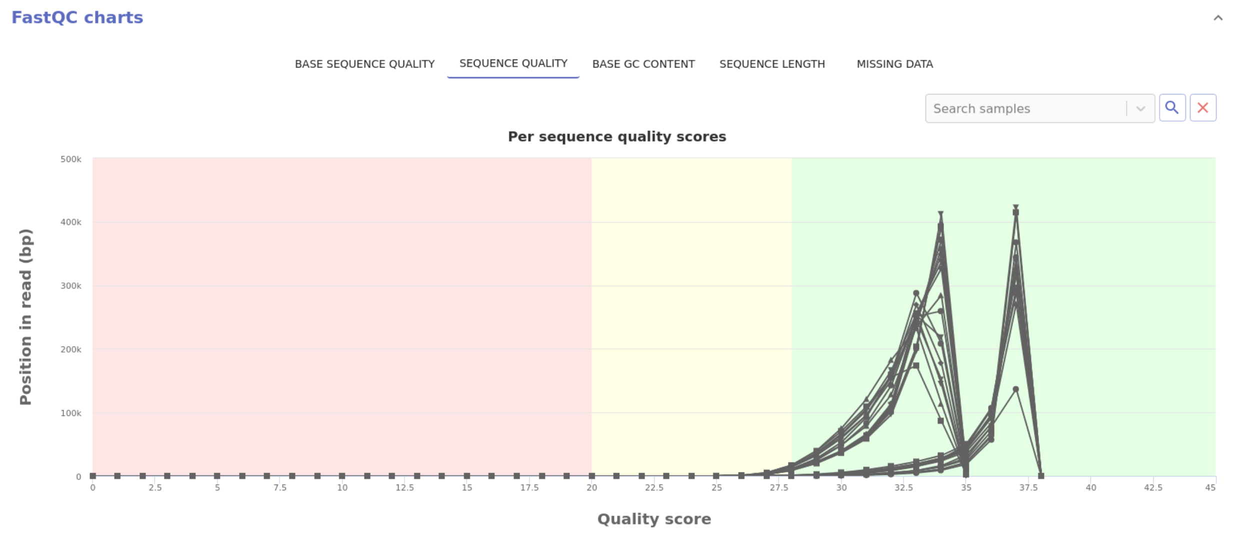
8.6 fastqc

8.6.1 Plot data

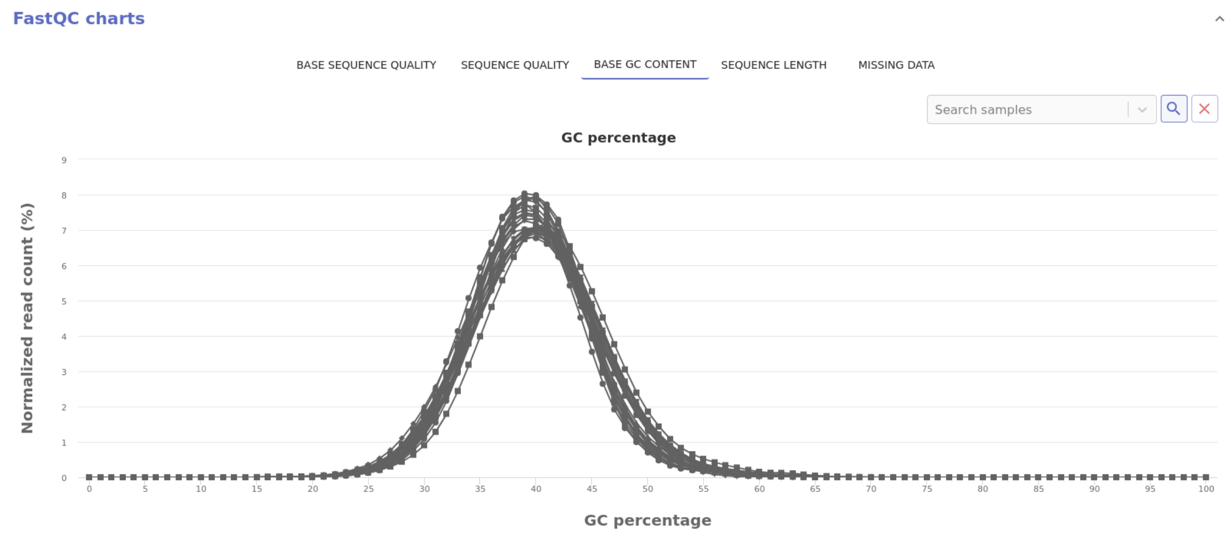
- **Base sequence quality**: The average quality score across the read length.



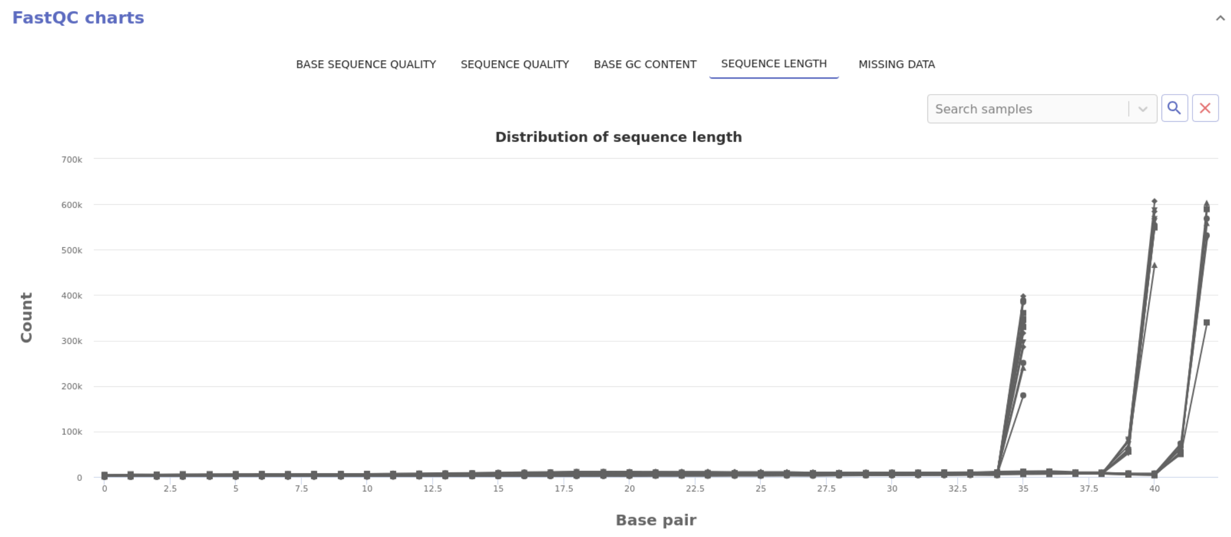
- **Sequence quality**: Distribution of the mean sequence quality score.



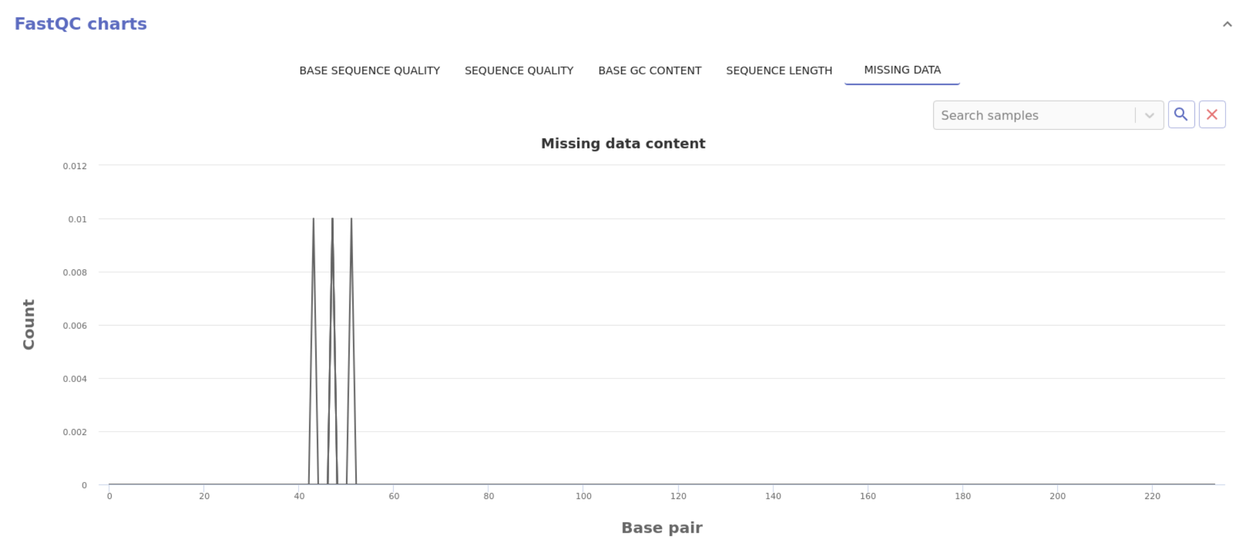
- **Base GC content**: Distribution of the GC content of each sequence.



- **Sequence length**: Distribution of the read sequence length.



- **Missing data**: Normalized count of missing data across the read length.



8.6.2 Warnings

The following FastQC categories will issue a warning when they have a `WARN` flag:

- Per base sequence quality.
- Overrepresented sequences.

The following FastQC categories will issue a warning when do not have a `PASS` flag:

- Per base sequence content.

8.6.3 Fails

The following FastQC categories will issue a fail when they have a `FAIL` flag:

- Per base sequence quality.
- Overrepresented sequences.
- Sequence length distribution.
- Per sequence GC content.

The following FastQC categories will issue a fail when the do not have a `PASS` flag:

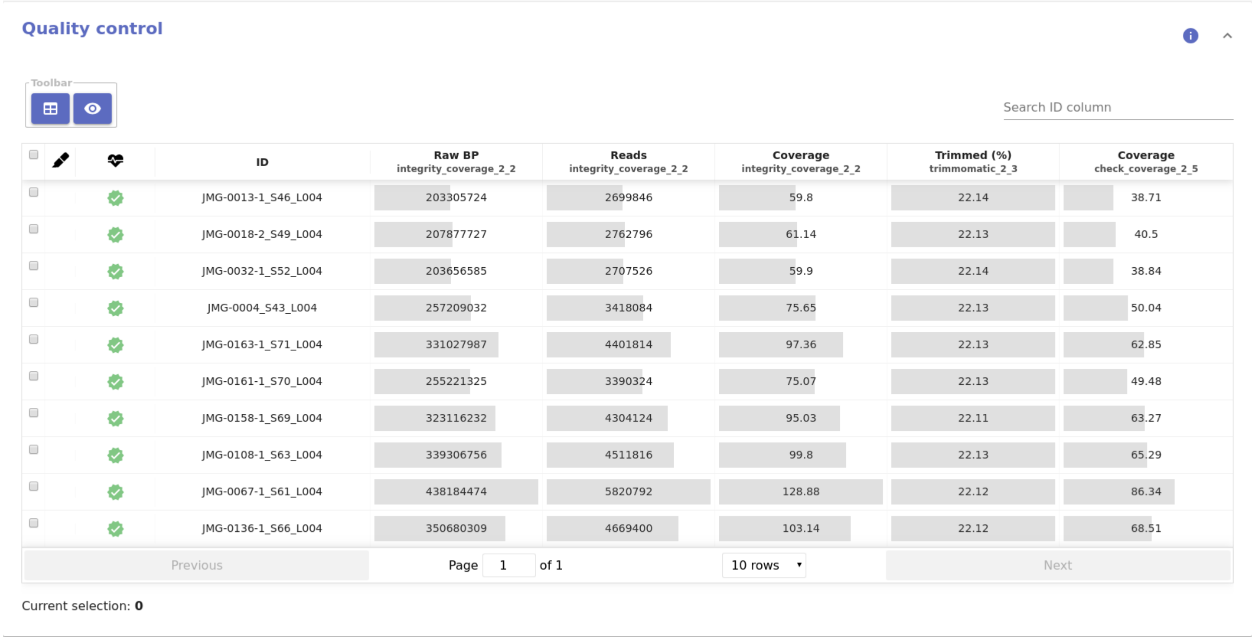
- Per base N content.
- Adapter content.

8.7 fastqc_trimmomatic

8.7.1 Table data

Quality control table:

- **Trimmed (%)**: Percentage of trimmed base pairs.



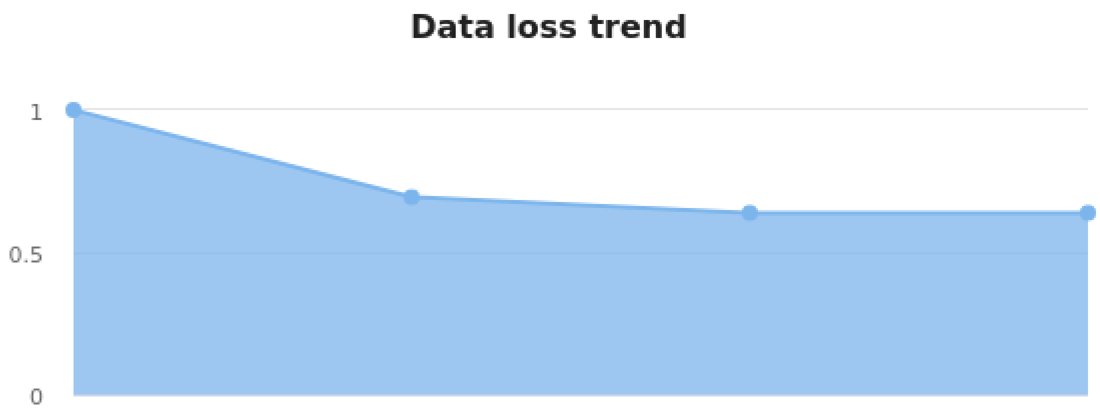
Quality control

□	✎	♥	ID	Raw BP integrity_coverage_2_2	Reads integrity_coverage_2_2	Coverage integrity_coverage_2_2	Trimmed (%) trimmomatic_2_3	Coverage check_coverage_2_5
□		✓	JMG-0013-1_S46_L004	203305724	2699846	59.8	22.14	38.71
□		✓	JMG-0018-2_S49_L004	207877727	2762796	61.14	22.13	40.5
□		✓	JMG-0032-1_S52_L004	203656585	2707526	59.9	22.14	38.84
□		✓	JMG-0004_S43_L004	257209032	3418084	75.65	22.13	50.04
□		✓	JMG-0163-1_S71_L004	331027987	4401814	97.36	22.13	62.85
□		✓	JMG-0161-1_S70_L004	255221325	3390324	75.07	22.13	49.48
□		✓	JMG-0158-1_S69_L004	323116232	4304124	95.03	22.11	63.27
□		✓	JMG-0108-1_S63_L004	339306756	4511816	99.8	22.13	65.29
□		✓	JMG-0067-1_S61_L004	438184474	5820792	128.88	22.12	86.34
□		✓	JMG-0136-1_S66_L004	350680309	4669400	103.14	22.12	68.51

Current selection: **0**

8.7.2 Plot data

- **Data loss chart**: Gives a trend of the data loss (in total number of base pairs) across components that may filter this data.



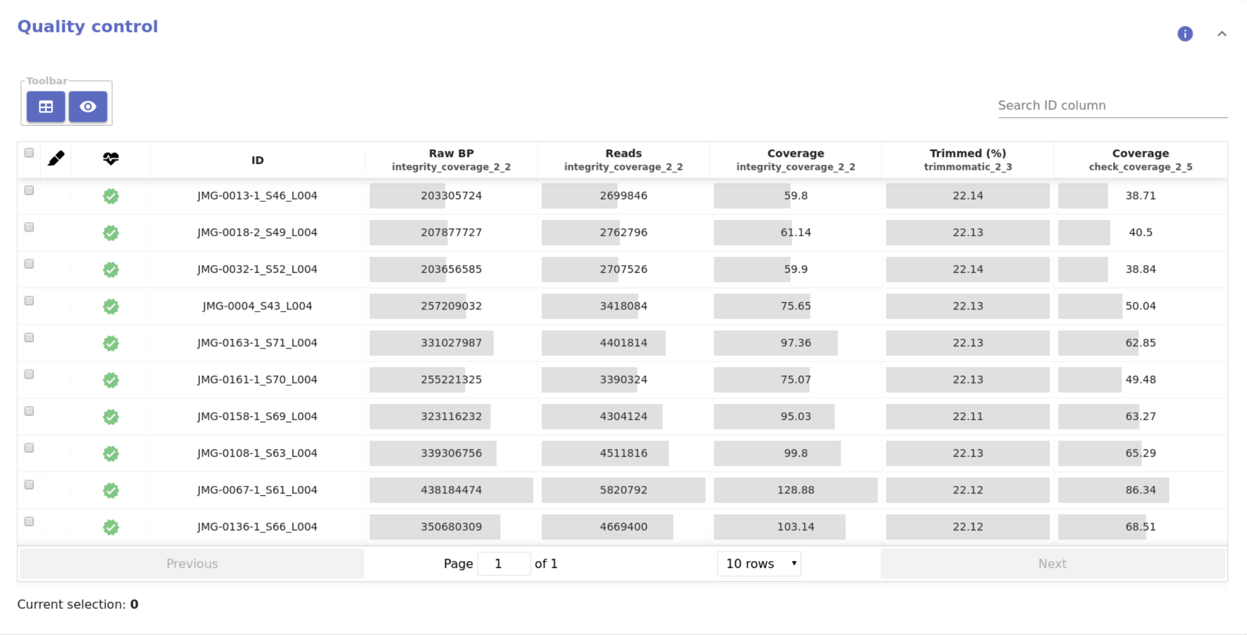
8.8 integrity_coverage

8.8.1 Table data

Quality control table:

- **Raw BP**: Number of raw base pairs from the FastQ file(s).

- **Reads**: Number of reads in the FastQ file(s)
- **Coverage**: Estimated coverage based on the number of base pairs and the expected genome size.



Quality control

Toolbar

Search ID column

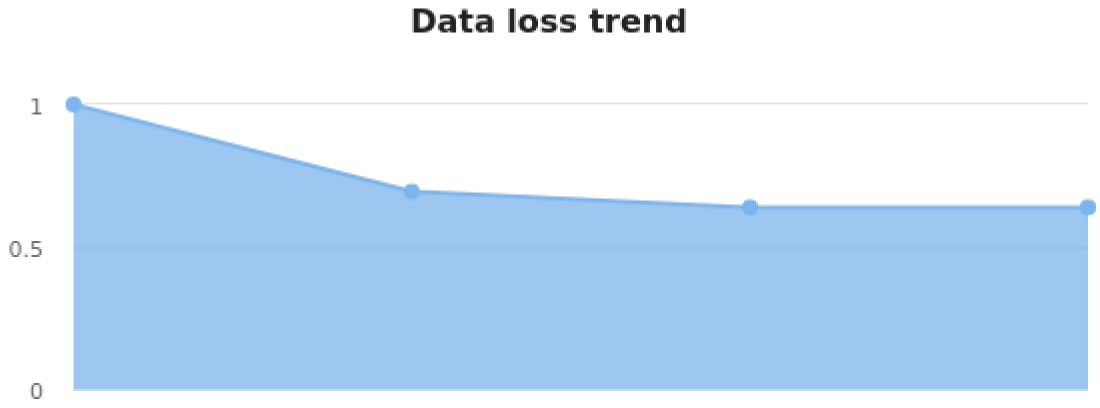
□			ID	Raw BP integrity_coverage_2_2	Reads integrity_coverage_2_2	Coverage integrity_coverage_2_2	Trimmed (%) trimmomatic_2_3	Coverage check_coverage_2_5
□		✓	JMG-0013-1_S46_L004	203305724	2699846	59.8	22.14	38.71
□		✓	JMG-0018-2_S49_L004	207877727	2762796	61.14	22.13	40.5
□		✓	JMG-0032-1_S52_L004	203656585	2707526	59.9	22.14	38.84
□		✓	JMG-0004_S43_L004	257209032	3418084	75.65	22.13	50.04
□		✓	JMG-0163-1_S71_L004	331027987	4401814	97.36	22.13	62.85
□		✓	JMG-0161-1_S70_L004	255221325	3390324	75.07	22.13	49.48
□		✓	JMG-0158-1_S69_L004	323116232	4304124	95.03	22.11	63.27
□		✓	JMG-0108-1_S63_L004	339306756	4511816	99.8	22.13	65.29
□		✓	JMG-0067-1_S61_L004	438184474	5820792	128.88	22.12	86.34
□		✓	JMG-0136-1_S66_L004	350680309	4669400	103.14	22.12	68.51

Previous Page 1 of 1 10 rows Next

Current selection: **0**

8.8.2 Plot data

- **Data loss chart**: Gives a trend of the data loss (in total number of base pairs) across components that may filter this data.



8.8.3 Warnings

Quality control table:

- When the enconding and phred score cannot be guessed from the FastQ file(s).

8.8.4 Fails

Quality control table:

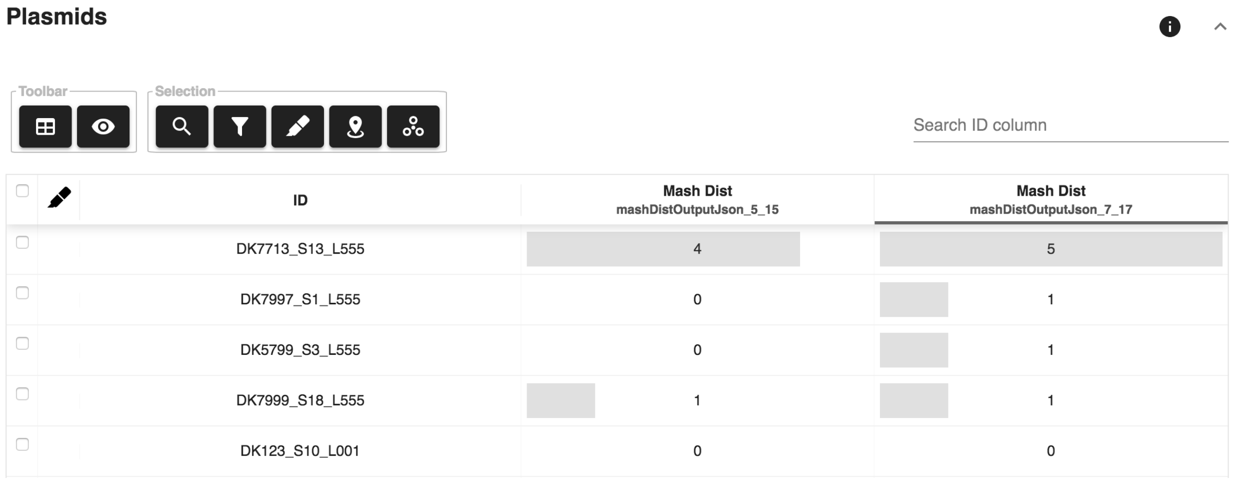
- When the sample has lower estimated coverage than the provided coverage threshold.

8.9 mash_dist

8.9.1 Table data

Plasmids table:

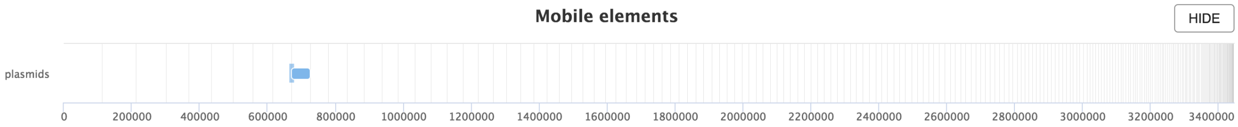
- **Mash Dist**: Number of plasmid hits



□		ID	Mash Dist mashDistOutputJson_5_15	Mash Dist mashDistOutputJson_7_17
□		DK7713_S13_L555	4	5
□		DK7997_S1_L555	0	1
□		DK5799_S3_L555	0	1
□		DK7999_S18_L555	1	1
□		DK123_S10_L001	0	0

8.9.2 Plot data

- **Sliding window Plasmid annotation**: Provides annotation of plasmid hits along the genome assembly. This report component is only available when the `mash_dist` component is used.

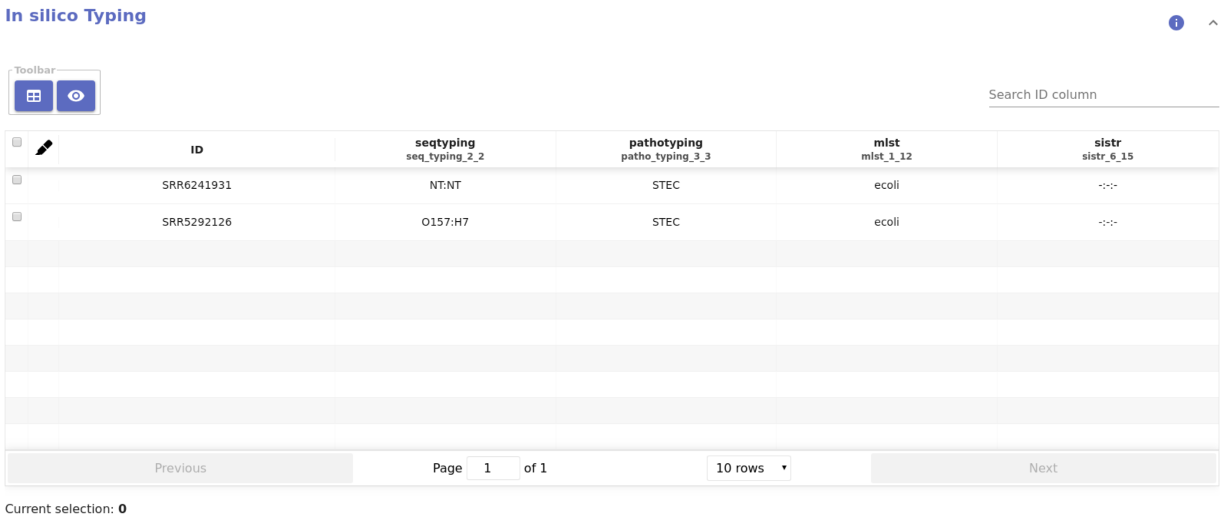


8.10 mlst

8.10.1 Table data

Typing table:

- **MLST species**: The inferred species name.
- **MLST ST**: The inferred sequence type.



In silico Typing

	ID	seqtyping seq_typing_2_2	pathotyping patho_typing_3_3	mlst mlst_1_12	sistr sistr_6_15
□	SRR6241931	NT:NT	STEC	ecoli	-:-:-
□	SRR5292126	O157:H7	STEC	ecoli	-:-:-

Previous Page 1 of 1 10 rows Next

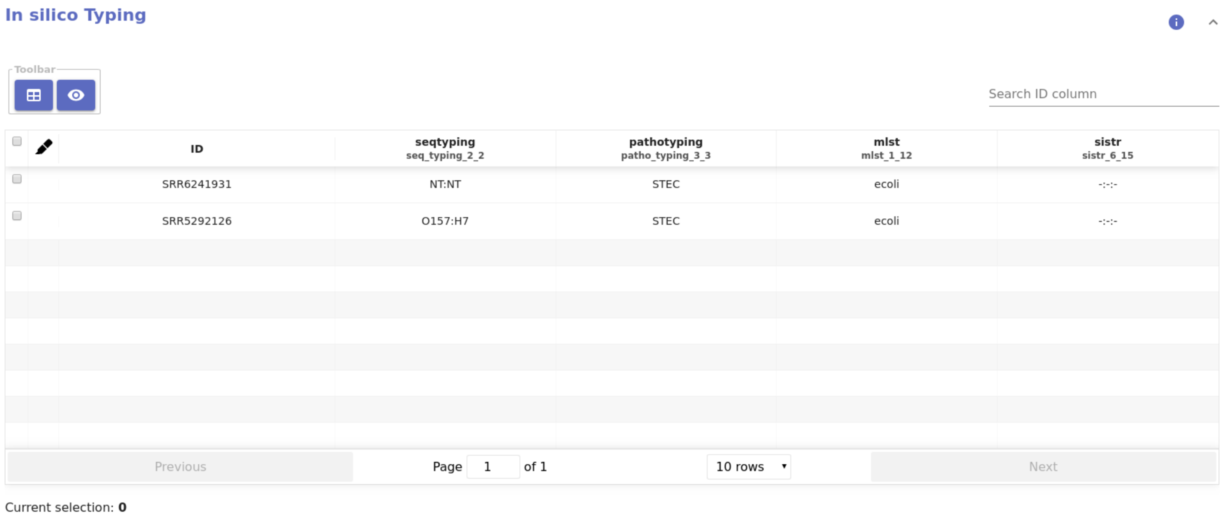
Current selection: **0**

8.11 patho_typing

8.11.1 Table data

Typing table:

- **Patho_typing**: The pathotyping result.



In silico Typing

	ID	seqtyping seq_typing_2_2	pathotyping patho_typing_3_3	mlst mlst_1_12	sistr sistr_6_15
□	SRR6241931	NT:NT	STEC	ecoli	-:-:-
□	SRR5292126	O157:H7	STEC	ecoli	-:-:-

Previous Page 1 of 1 10 rows Next

Current selection: **0**

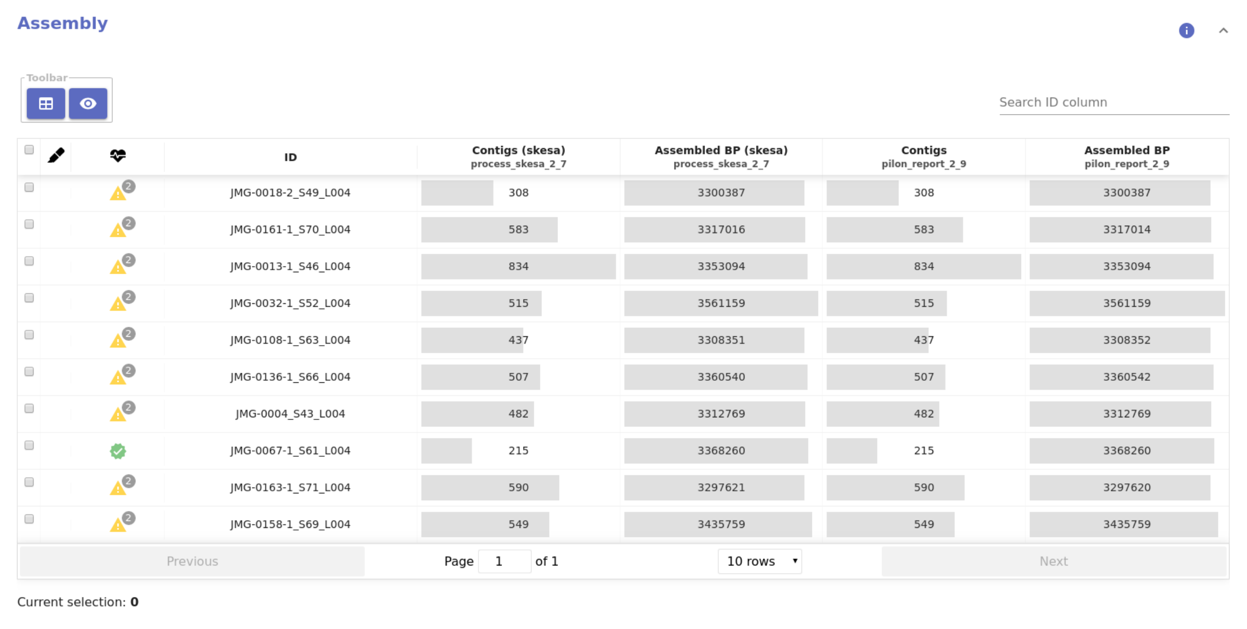
8.12 pilon

8.12.1 Table data

Quality control table:

- **Contigs**: Number of assembled contigs.

- **Assembled BP**: Total number of assembled base pairs.



Assembly

Toolbar

Search ID column

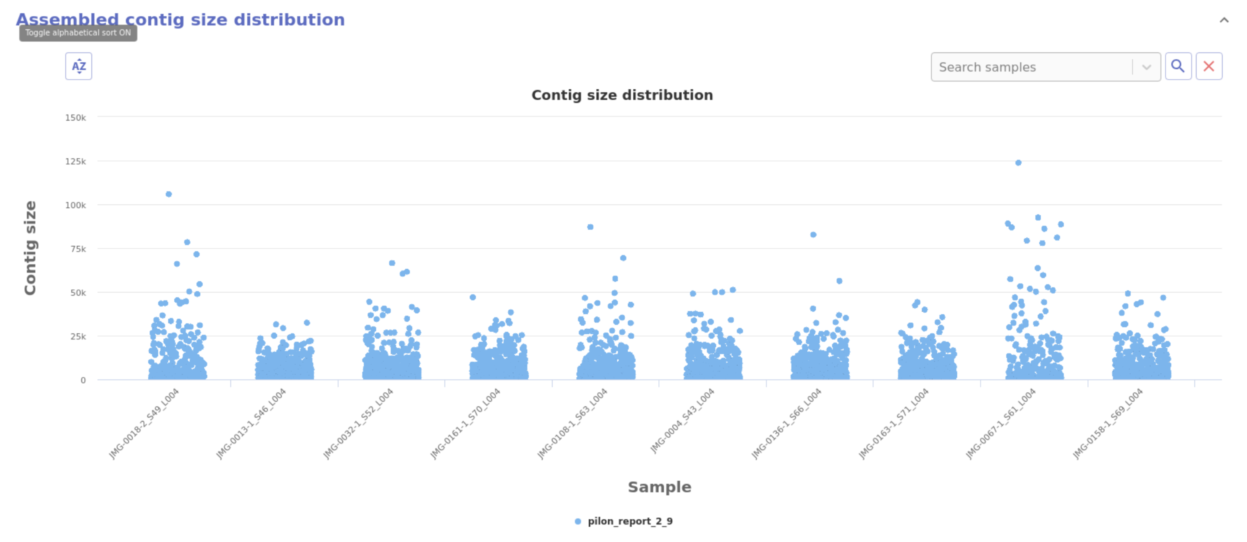
□	✎	♥	ID	Contigs (skesa) process_skesa_2_7	Assembled BP (skesa) process_skesa_2_7	Contigs pilon_report_2_9	Assembled BP pilon_report_2_9
□		⚠ 2	JMG-0018-2_S49_L004	308	3300387	308	3300387
□		⚠ 2	JMG-0161-1_S70_L004	583	3317016	583	3317014
□		⚠ 2	JMG-0013-1_S46_L004	834	3353094	834	3353094
□		⚠ 2	JMG-0032-1_S52_L004	515	3561159	515	3561159
□		⚠ 2	JMG-0108-1_S63_L004	437	3308351	437	3308352
□		⚠ 2	JMG-0136-1_S66_L004	507	3360540	507	3360542
□		⚠ 2	JMG-0004_S43_L004	482	3312769	482	3312769
□		✓	JMG-0067-1_S61_L004	215	3368260	215	3368260
□		⚠ 2	JMG-0163-1_S71_L004	590	3297621	590	3297620
□		⚠ 2	JMG-0158-1_S69_L004	549	3435759	549	3435759

Previous Page 1 of 1 10 rows Next

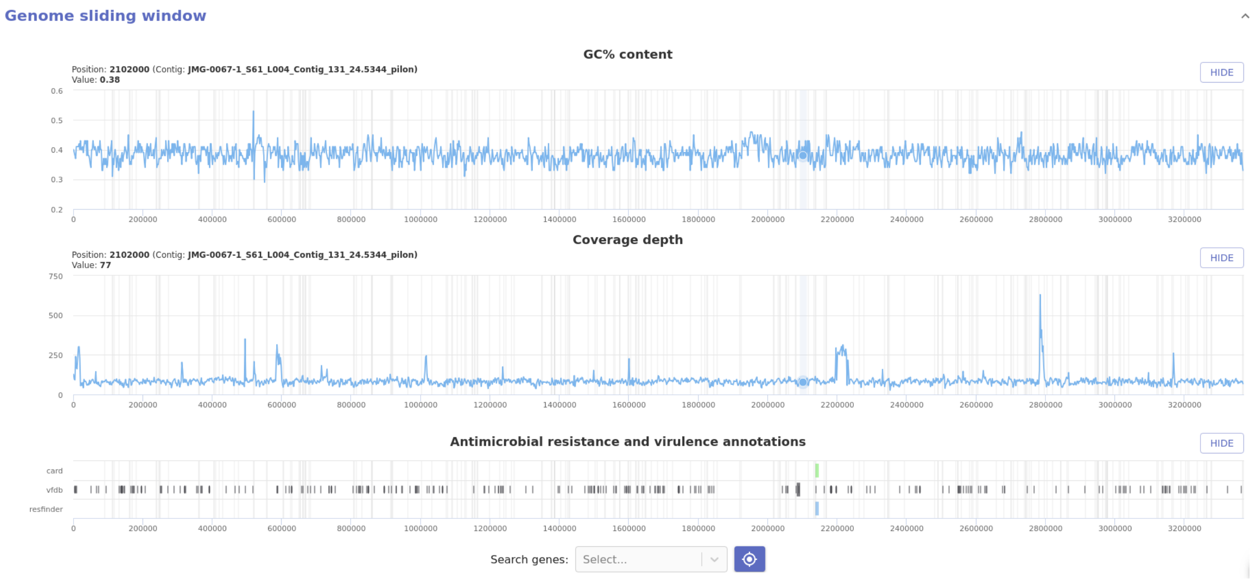
Current selection: **0**

8.12.2 Plot data

- **Contig size distribution**: Distribution of the size of each assembled contig.



- **Sliding window coverage and GC content**: Provides coverage and GC content metrics along the genome using a sliding window approach and two synchronised charts.



8.12.3 Warnings

Quality control table:

- When the enconding and phred score cannot be guessed from the FastQ file(s).

8.12.4 Fails

Quality control table:

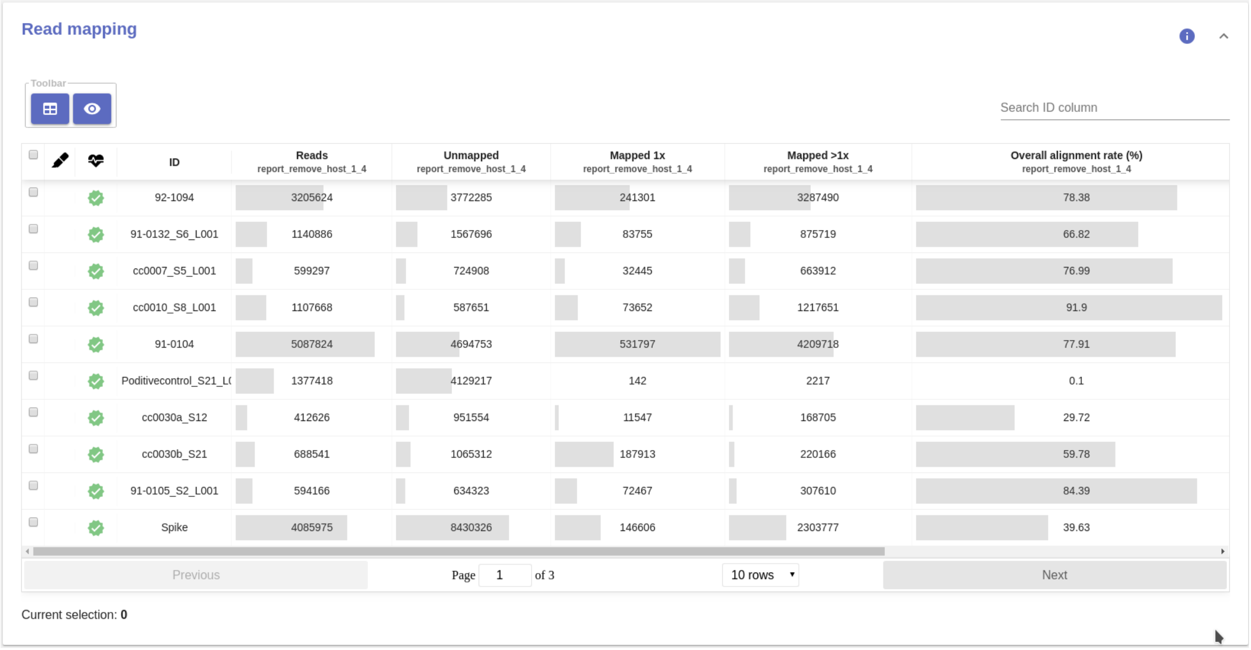
- When the sample has lower estimated coverage than the provided coverage threshold.

8.13 process_mapping

8.13.1 Table data

Read mapping table:

- **Reads**: Number reads in the the FastQ file(s).
- **Unmapped**: Number of unmapped reads
- **Mapped 1x**: Number of reads that aligned, concordantly and discordantly, exactly 1 time
- **Mapped >1x**: Number of reads that aligned, concordantly or disconrdantly, more than 1 times
- **Overall alignment rate** (%): Overall alignment rate



Read mapping

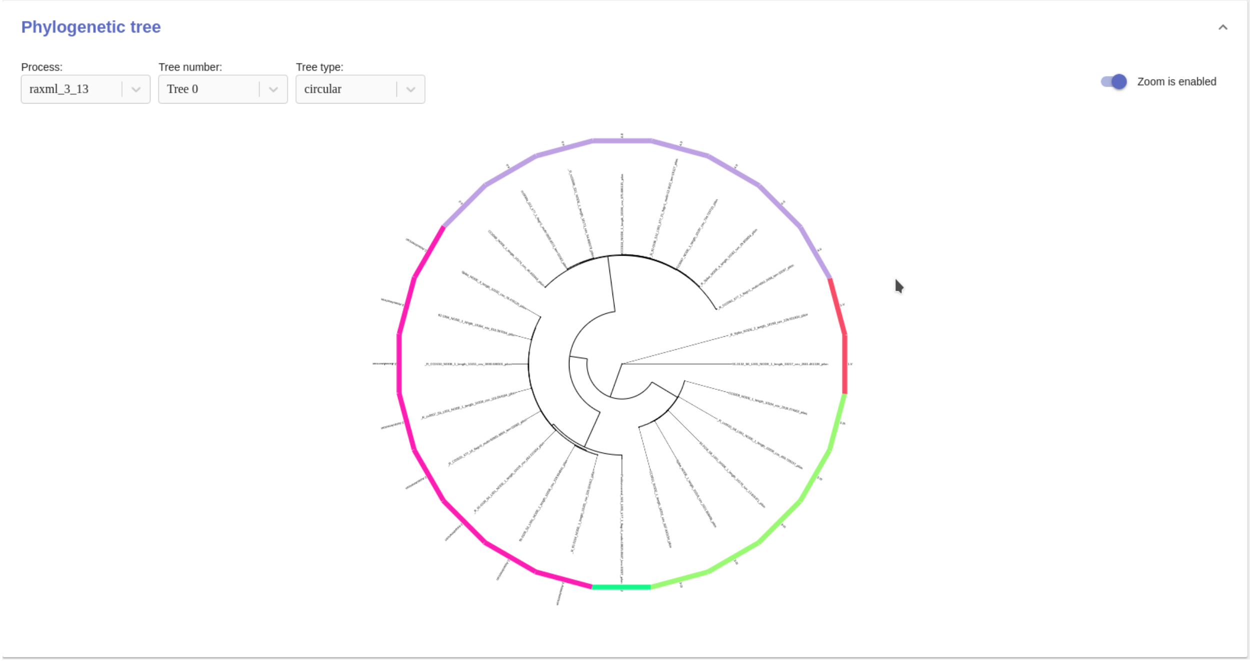
□			ID	Reads report_remove_host_1_4	Unmapped report_remove_host_1_4	Mapped 1x report_remove_host_1_4	Mapped >1x report_remove_host_1_4	Overall alignment rate (%) report_remove_host_1_4
□		✓	92-1094	3205624	3772285	241301	3287490	78.38
□		✓	91-0132_S6_L001	1140886	1567696	83755	875719	66.82
□		✓	cc0007_S5_L001	599297	724908	32445	663912	76.99
□		✓	cc0010_S8_L001	1107668	587651	73652	1217651	91.9
□		✓	91-0104	5087824	4694753	531797	4209718	77.91
□		✓	Poditivecontrol_S21_L(	1377418	4129217	142	2217	0.1
□		✓	cc0030a_S12	412626	951554	11547	168705	29.72
□		✓	cc0030b_S21	688541	1065312	187913	220166	59.78
□		✓	91-0105_S2_L001	594166	634323	72467	307610	84.39
□		✓	Spike	4085975	8430326	146606	2303777	39.63

Current selection: **0**

8.14 process_newick

8.14.1 Tree data

Phylogenetic reconstruction with bootstrap values for the provided tree.



Phylogenetic tree

Process: raxml_3_13

Tree number: Tree 0

Tree type: circular

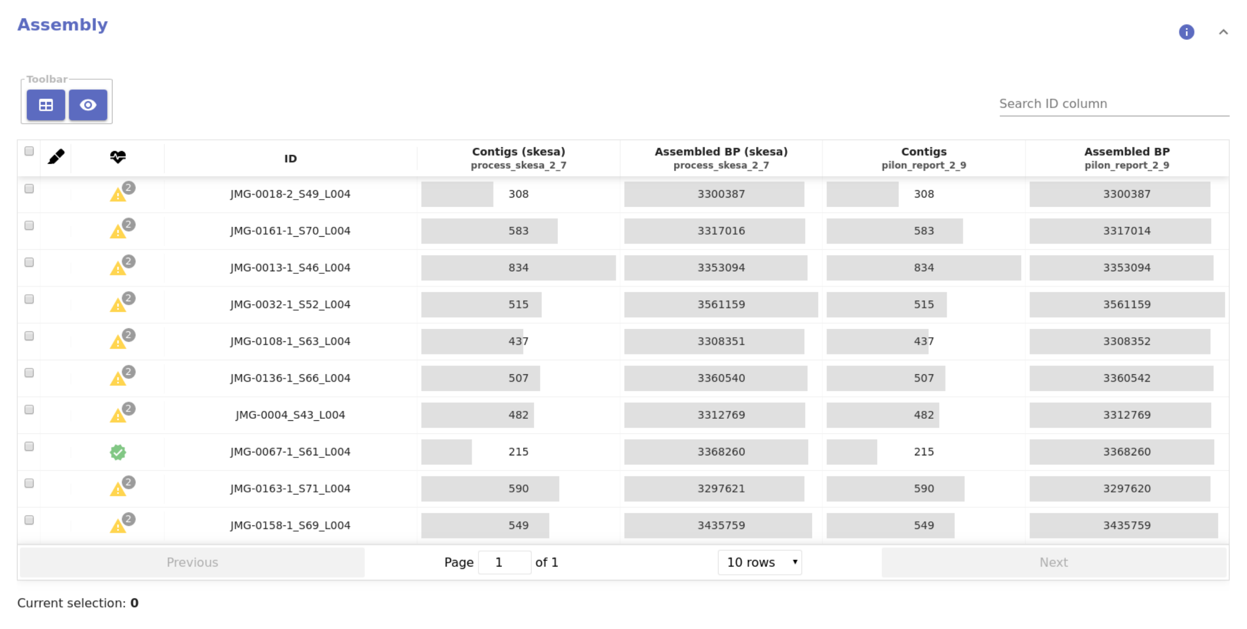
Zoom is enabled

8.15 process_skesa

8.15.1 Table data

Quality control table:

- **Contigs (skesa)**: Number of assembled contigs.
- **Assembled BP**: Total number of assembled base pairs.



□			ID	Contigs (skesa) process_skesa_2_7	Assembled BP (skesa) process_skesa_2_7	Contigs pilon_report_2_9	Assembled BP pilon_report_2_9
□		⚠2	JMG-0018-2_S49_L004	308	3300387	308	3300387
□		⚠2	JMG-0161-1_S70_L004	583	3317016	583	3317014
□		⚠2	JMG-0013-1_S46_L004	834	3353094	834	3353094
□		⚠2	JMG-0032-1_S52_L004	515	3561159	515	3561159
□		⚠2	JMG-0108-1_S63_L004	437	3308351	437	3308352
□		⚠2	JMG-0136-1_S66_L004	507	3360540	507	3360542
□		⚠2	JMG-0004_S43_L004	482	3312769	482	3312769
□		✓	JMG-0067-1_S61_L004	215	3368260	215	3368260
□		⚠2	JMG-0163-1_S71_L004	590	3297621	590	3297620
□		⚠2	JMG-0158-1_S69_L004	549	3435759	549	3435759

Current selection: **0**

8.15.2 Warnings

Assembly table:

- When the number of contigs exceeds the threshold of 100 contigs per 1.5Mb.

8.15.3 Fails

Assembly table:

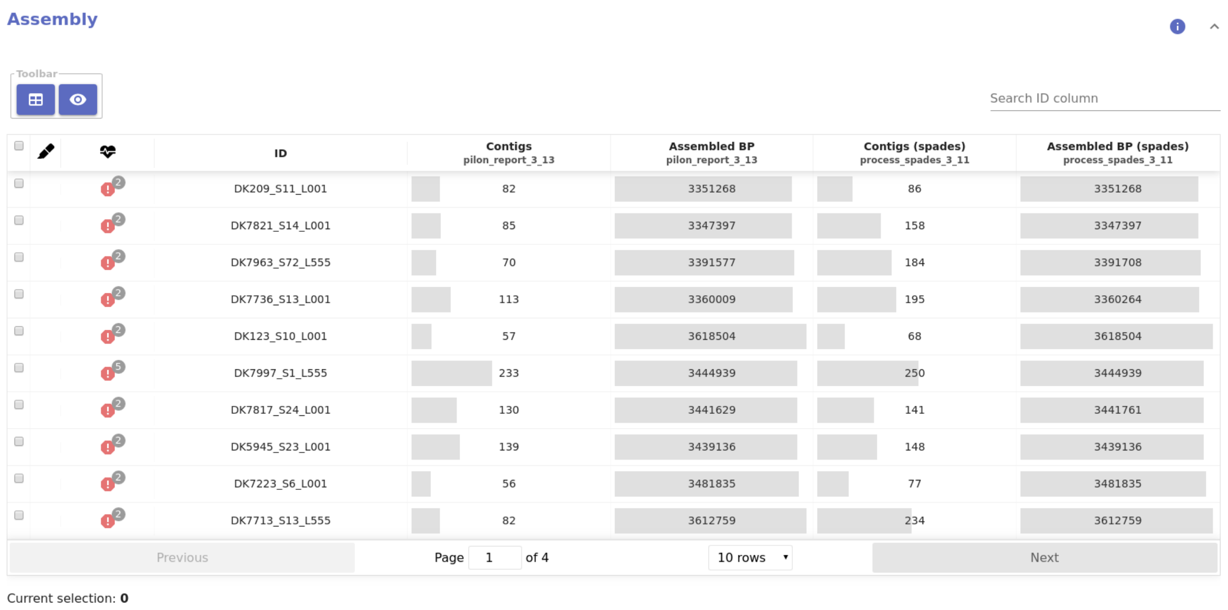
- When the assembly size if smaller than 80% or larger than 150% of the expected genome size.

8.16 process_spades

8.16.1 Table data

Quality control table:

- **Contigs (spades)**: Number of assembled contigs.
- **Assembled BP**: Total number of assembled base pairs.



Assembly

Toolbar

Search ID column

□	✎	♥	ID	Contigs pilon_report_3_13	Assembled BP pilon_report_3_13	Contigs (spades) process_spades_3_11	Assembled BP (spades) process_spades_3_11
□		! 2	DK209_S11_L001	82	3351268	86	3351268
□		! 2	DK7821_S14_L001	85	3347397	158	3347397
□		! 2	DK7963_S72_L555	70	3391577	184	3391708
□		! 2	DK7736_S13_L001	113	3360009	195	3360264
□		! 2	DK123_S10_L001	57	3618504	68	3618504
□		! 5	DK7997_S1_L555	233	3444939	250	3444939
□		! 2	DK7817_S24_L001	130	3441629	141	3441761
□		! 2	DK5945_S23_L001	139	3439136	148	3439136
□		! 2	DK7223_S6_L001	56	3481835	77	3481835
□		! 2	DK7713_S13_L555	82	3612759	234	3612759

Previous | Page 1 of 4 | 10 rows ▾ | Next

Current selection: **0**

8.16.2 Warnings

Assembly table:

- When the number of contigs exceeds the threshold of 100 contigs per 1.5Mb.

8.16.3 Fails

Assembly table:

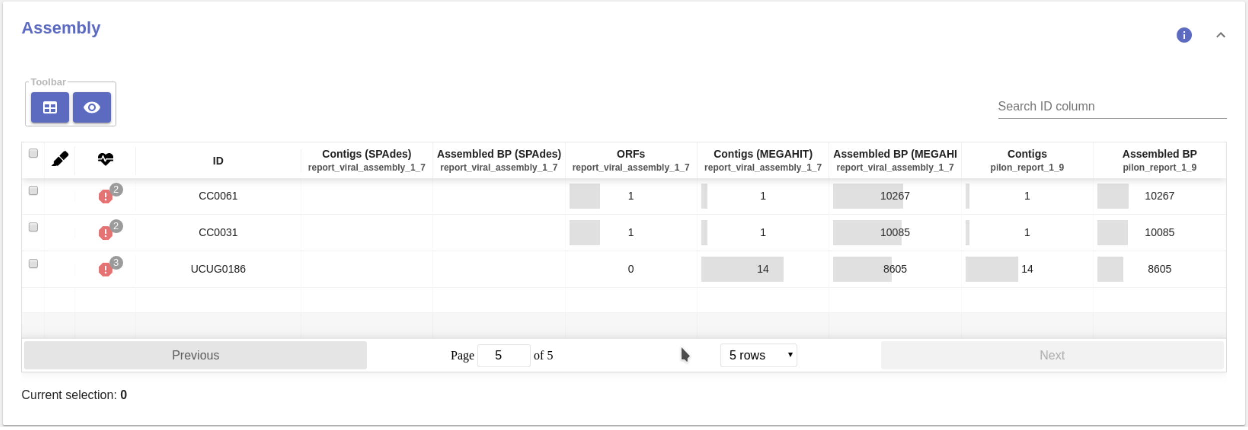
- When the assembly size if smaller than 80% or larger than 150% of the expected genome size.

8.17 process_viral_assembly

8.17.1 Table data

Quality control table:

- **Contigs (SPAdes)**: Number of assembled contigs.
- **Assembled BP (SPAdes)**: Total number of assembled base pairs.
- **ORFs**: Number of complete ORFs in the assembly.
- **Contigs (MEGAHIT)**: Number of assembled contigs.
- **Assembled BP (MEGAHIT)**: Total number of assembled base pairs.



Assembly

Toolbar

Search ID column

□			ID	Contigs (SPAdes) report_viral_assembly_1_7	Assembled BP (SPAdes) report_viral_assembly_1_7	ORFs report_viral_assembly_1_7	Contigs (MEGAHIT) report_viral_assembly_1_7	Assembled BP (MEGAHI report_viral_assembly_1_7	Contigs pilon_report_1_9	Assembled BP pilon_report_1_9
□		! 2	CC0061			1	1	10267	1	10267
□		! 2	CC0031			1	1	10085	1	10085
□		! 3	UCUG0186			0	14	8605	14	8605

Previous Page 5 of 5 5 rows ▾ Next

Current selection: **0**

8.17.2 Fails

Assembly table:

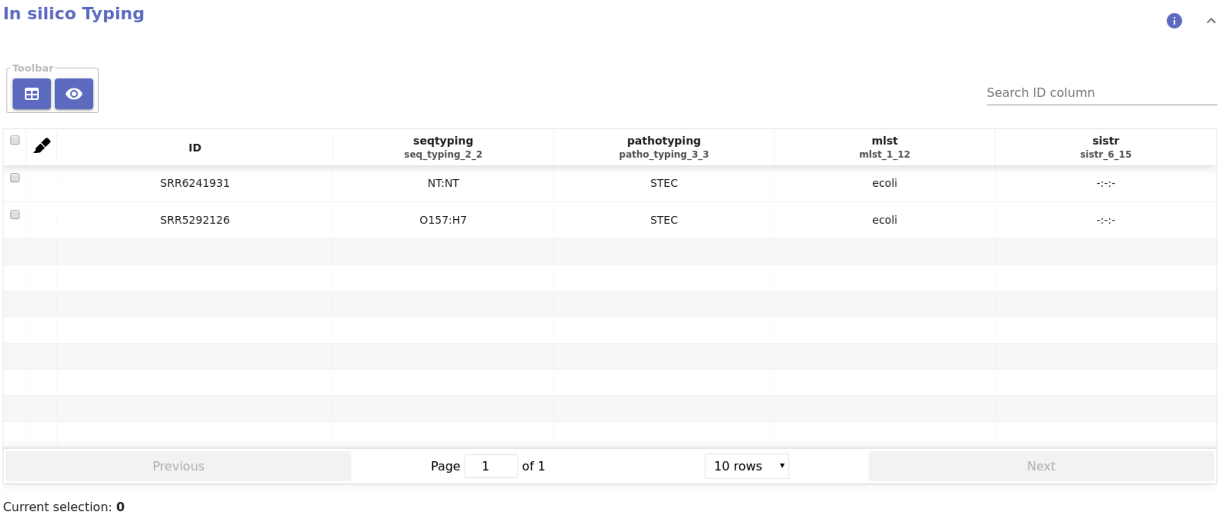
- When the assembly size if smaller than 80% or larger than 150% of the expected genome size.

8.18 seq_typing

8.18.1 Table data

Typing table:

- **seqtyping**: The sequence typing result.



In silico Typing

Toolbar

Search ID column

□		ID	seqtyping seq_typing_2_2	pathotyping patho_typing_3_3	mlst mlst_1_12	sistr sistr_6_15
□		SRR6241931	NT:NT	STEC	ecoli	-:-:-
□		SRR5292126	O157:H7	STEC	ecoli	-:-:-

Previous Page 1 of 1 10 rows ▾ Next

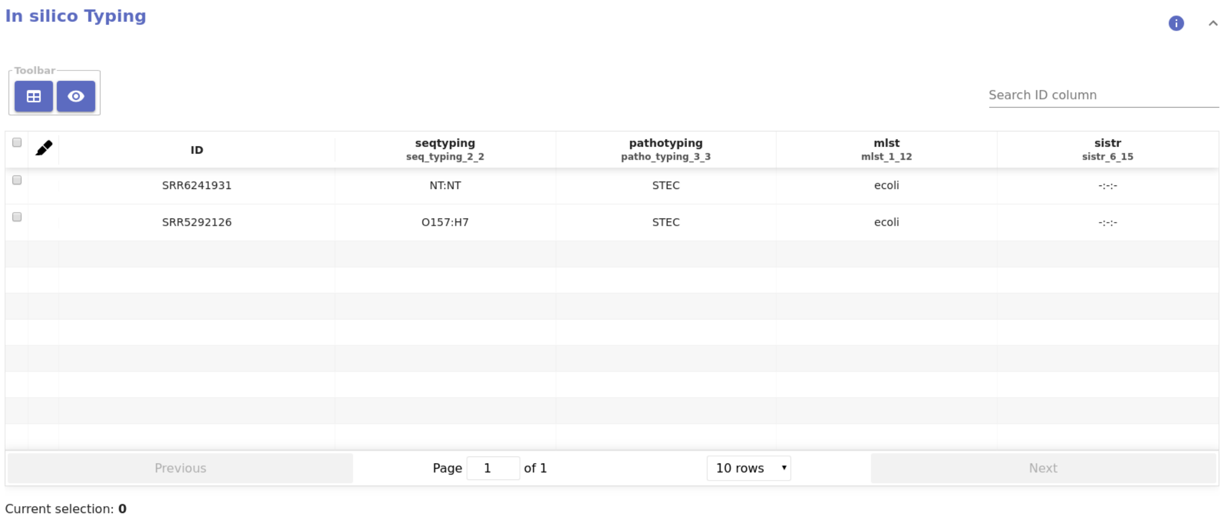
Current selection: **0**

8.19 sistr

8.19.1 Table data

Typing table:

- **sistr**: The sequence typing result.



In silico Typing

□		ID	seqtyping seq_typing_2_2	pathotyping patho_typing_3_3	mlst mlst_1_12	sistr sistr_6_15
□		SRR6241931	NT:NT	STEC	ecoli	-:-:-
□		SRR5292126	O157:H7	STEC	ecoli	-:-:-

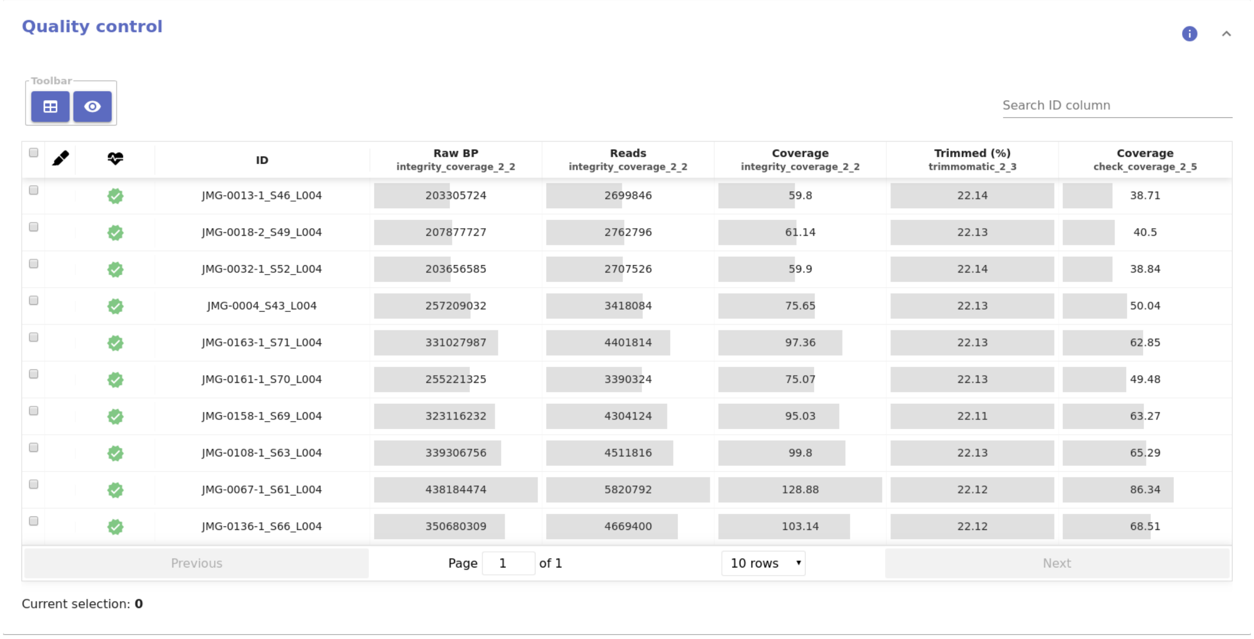
Current selection: **0**

8.20 trimmomatic

8.20.1 Table data

Quality control table:

- **Trimmed (%)**: Percentage of trimmed base pairs.



Quality control

Toolbar

Search ID column

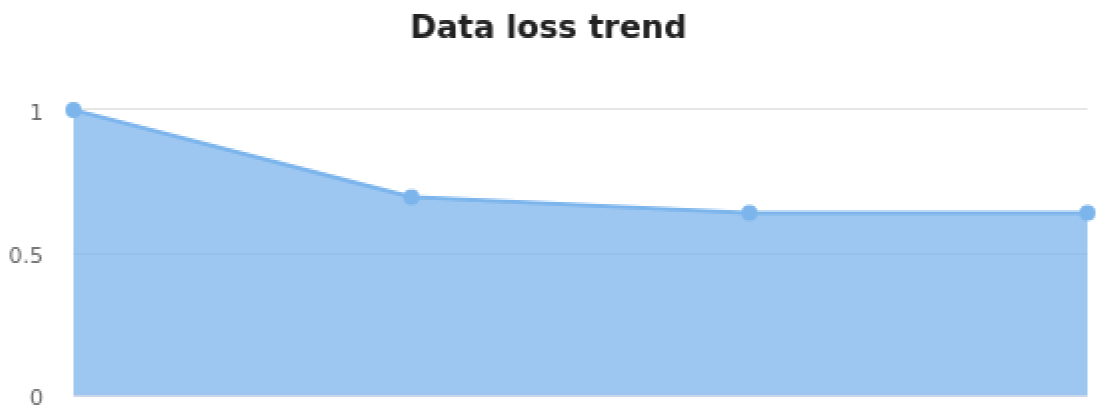
□			ID	Raw BP integrity_coverage_2_2	Reads integrity_coverage_2_2	Coverage integrity_coverage_2_2	Trimmed (%) trimmomatic_2_3	Coverage check_coverage_2_5
□		✓	JMG-0013-1_S46_L004	203305724	2699846	59.8	22.14	38.71
□		✓	JMG-0018-2_S49_L004	207877727	2762796	61.14	22.13	40.5
□		✓	JMG-0032-1_S52_L004	203656585	2707526	59.9	22.14	38.84
□		✓	JMG-0004_S43_L004	257209032	3418084	75.65	22.13	50.04
□		✓	JMG-0163-1_S71_L004	331027987	4401814	97.36	22.13	62.85
□		✓	JMG-0161-1_S70_L004	255221325	3390324	75.07	22.13	49.48
□		✓	JMG-0158-1_S69_L004	323116232	4304124	95.03	22.11	63.27
□		✓	JMG-0108-1_S63_L004	339306756	4511816	99.8	22.13	65.29
□		✓	JMG-0067-1_S61_L004	438184474	5820792	128.88	22.12	86.34
□		✓	JMG-0136-1_S66_L004	350680309	4669400	103.14	22.12	68.51

Previous Page 1 of 1 10 rows ▾ Next

Current selection: **0**

8.20.2 Plot data

- **Data loss chart**: Gives a trend of the data loss (in total number of base pairs) across components that may filter this data.

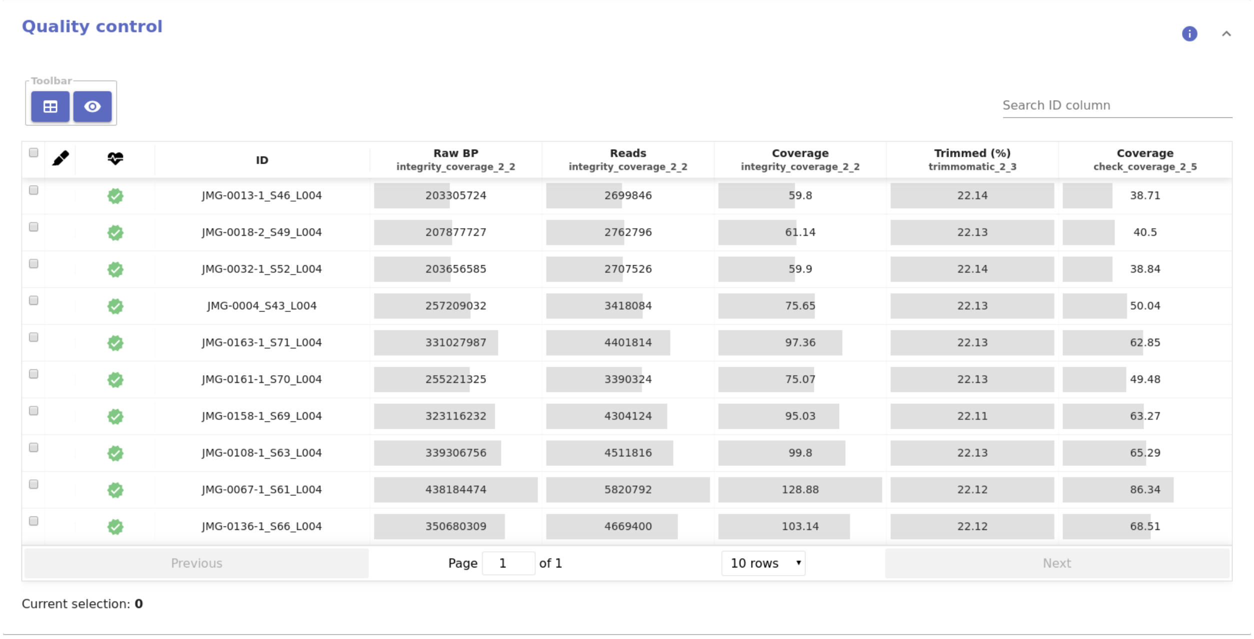


8.21 true_coverage

8.21.1 Table data

Quality control table:

- **True Coverage**: Estimated coverage based on read mapping on MLST genes.



□	✎	♥	ID	Raw BP integrity_coverage_2_2	Reads integrity_coverage_2_2	Coverage integrity_coverage_2_2	Trimmed (%) trimmomatic_2_3	Coverage check_coverage_2_5
□		✓	JMG-0013-1_S46_L004	203305724	2699846	59.8	22.14	38.71
□		✓	JMG-0018-2_S49_L004	207877727	2762796	61.14	22.13	40.5
□		✓	JMG-0032-1_S52_L004	203656585	2707526	59.9	22.14	38.84
□		✓	JMG-0004_S43_L004	257209032	3418084	75.65	22.13	50.04
□		✓	JMG-0163-1_S71_L004	331027987	4401814	97.36	22.13	62.85
□		✓	JMG-0161-1_S70_L004	255221325	3390324	75.07	22.13	49.48
□		✓	JMG-0158-1_S69_L004	323116232	4304124	95.03	22.11	63.27
□		✓	JMG-0108-1_S63_L004	339306756	4511816	99.8	22.13	65.29
□		✓	JMG-0067-1_S61_L004	438184474	5820792	128.88	22.12	86.34
□		✓	JMG-0136-1_S66_L004	350680309	4669400	103.14	22.12	68.51

Current selection: **0**

8.21.2 Fails

Quality control table:

- When the sample has lower estimated coverage than the provided coverage threshold.

CHAPTER 9

Components

These are the currently available FlowCraft components with a short description of their tasks. For a more detailed information, follow the links of each component.

9.1 Download

- components/reads_download: Downloads reads from the SRA/ENA public databases from a list of accessions.
- components/fasterq_dump: Downloads reads from the SRA public databases from a list of accessions, using `fasterq-dump`.

9.2 Reads Quality Control

- components/check_coverage: Estimates the coverage for each sample and filters FastQ files according to a specified minimum coverage threshold.
- components/fastqc: Runs FastQC on paired-end FastQ files.
- components/fastqc_trimmomatic: Runs Trimmomatic on paired-end FastQ files informed by the FastQC report.
- components/filter_poly: Runs PrinSeq on paired-end FastQ files to remove low complexity sequences.
- components/integrity_coverage: Tests the integrity of the provided FastQ files, provides the option to filter FastQ files based on the expected assembly coverage and provides information about the maximum read length and sequence encoding.
- components/trimmomatic: Runs Trimmomatic on paired-end FastQ files.
- components/downsample_fastq: Subsamples fastq files up to a target coverage depth.

9.3 Assembly

- components/megahit: Assembles metagenomic paired-end FastQ files using megahit.
- components/metaspades: Assembles metagenomic paired-end FastQ files using metaSPAdes.
- components/skesa: Assembles paired-end FastQ files using skesa.
- components/spades: Assembles paired-end FastQ files using SPAdes.

9.4 Post-assembly

- components/pilon: Corrects and filters assemblies using Pilon.
- components/process_skesa: Processes the assembly output from Skesa and performs filtering base on quality criteria of GC content k-mer coverage and read length.
- components/process_spades: Processes the assembly output from Spades and performs filtering base on quality criteria of GC content k-mer coverage and read length.

9.5 Binning

- components/maxbin2: An automatic tool for binning metagenomic sequences

9.6 Annotation

- components/abricate: Performs anti-microbial gene screening using abricate.
- components/card_rgi: Performs anti-microbial resistance gene screening using CARD rgi (with contigs as input).
- components/prokka: Performs assembly annotation using prokka.

9.7 Distance Estimation

- components/mash_dist: Executes mash distance against a reference index plasmid database and generates a *JSON* for pATLAS. This component calculates pairwise distances between sequences (one from the database and the query sequence). However if a different database is provided it can use mash dist for other purposes.
- components/mash_screen: Performs mash screen against a reference index plasmid database and generates a JSON input file for pATLAS. This component searches for containment of a given sequence in read sequencing data. However if a different database is provided it can use mash screen for other purposes.
- components/fast_ani: Performs pairwise comparisons between fastas,

given a multifasta as input for fastANI. It will split the multifasta into single fastas that will then be provided as a matrix. The output will be the all pairwise comparisons that pass the minimum of 50 aligned sequences with a default length of 200 bp.

- components/mash_sketch_fasta: Performs mash sketch for fasta files.
- components/mash_sketch_fastq: Performes mash sketch for fastq files.

9.8 Mapping

- components/assembly_mapping: Performs a mapping procedure of FastQ files into a their assembly and performs filtering based on quality criteria of read coverage and genome size.
- components/bowtie: Align short paired-end sequencing reads to long reference sequences
- components/mapping_patlas: Performs read mapping and generates a JSON input file for pATLAS.
- components/remove_host: Performs read mapping with bowtie2 against the target host genome (default hg19) and removes the mapping reads
- components/retrieve_mapped: Retrieves the mapped reads of a previous bowtie2 mapping process.

9.9 Taxonomic Profiling

- components/kraken: Performs taxonomic identification with kraken on FastQ files (minikrakenDB2017 as default database)
- components/midas_species: Performs taxonomic identification on FastQ files at the species level with midas (requires database)

9.10 Typing

- components/chewbbaca: Performs a core-genome/whole-genome Multilocus Sequence Typing analysis on an assembly using ChewBBACA.
- components/metamlst: Checks the Sequence Type of metagenomic reads using Multilocus Sequence Typing.
- components/mlst: Checks the Sequence Type of an assembly using Multilocus Sequence Typing.
- components/patho_typing: *In silico* pathogenic typing from raw illumina reads.
- components/seq_typing: Determines the type of a given sample from a set of reference sequences.
- components/sistr: Serovar predictions from whole-genome sequence assemblies by determination of antigen gene and cgMLST gene alleles.
- components/momps: Multi-locus sequence typing for Legionella pneumophila from assemblies and reads.

CHAPTER 10

General orientation

10.1 Codebase structure

The most important elements of FlowCraft's directory structure are:

- **generator:**
  - components: Contains the Process classes for each component
  - templates: Contains the nextflow jinja template files for each component
  - engine.py: The engine of FlowCraft that builds the pipeline
  - process.py: Contains the abstract Process class that is inherited
  - by all component classes
  - pipeline_parser.py: Functions that parse and check the pipeline string
  - recipe.py: Class responsible for creating recipes
- templates: A git submodule of the templates repository that contain the template scripts for the components.

10.2 Code style

- **Style**: the code base of flowcraft should adhere (the best it can) to the PEP8 style guidelines.
- **Docstrings**: code should be generally well documented following the numpy docstring style.
- **Quality**: there is also an integration with the codacy service to evaluate code quality, which is useful for detecting several coding issues that may appear.

10.3 Testing

Tests are performed using pytest and the source files are stored in the `flowcraft/tests` directory. Tests must be executed on the root directory of the repository

10.4 Documentation

Documentation source files are stored in the `docs` directory. The general configuration file is found in `docs/conf.py` and the entry point to the documentation is `docs/index.html`.

CHAPTER 11

Process creation guidelines

11.1 Basic process creation

The addition of a new process to FlowCraft requires three main steps:

1. *Create process template*: Create a jinja2 template in `flowcraft.generator.templates` with the nextflow code.
2. *Create Process class*: Create a `Process` subclass in `flowcraft.generator.process` with information about the process (e.g., expected input/output, secondary inputs, etc.).

11.1.1 Create process template

First, create the nextflow template that will be integrated into the pipeline as a process. This file must be placed in `flowcraft.generator.templates` and have the `.nf` extension. In order to allow the template to be dynamically added to a pipeline file, we use the jinja2 template language to substitute key variables in the process, such as input/output channels.

An example created as a `my_process.nf` file is as follows:

```
some_channel_{{ pid }} = Channel.value(params.param1{{ param_id}})
other_channel_{{ pid }} = Chanel.fromPath(params.param2{{ param_id}})

process myProcess_{{ pid }} {

    {% include "post.txt" ignore missing %}

    publishDir "results/myProcess_{{ pid }}", pattern: "*.tsv"

    input:
    set sample_id, <data> from {{ input_channel }}
    val x from some_channel_{{ pid }}
    file y from other_channel_{{ pid }}
```

(continues on next page)

(continued from previous page)

```
    val direct_from_parms from Channel.value(params.param3{{param_id}}

    // The output is optional
    output:
    set sample_id, <data> into {{ output_channel }}
    {% with task_name="abricate" %}
    {%- include "compiler_channels.txt" ignore missing -%}
    {% endwith %}

    """
    <process code/commands>
    """
}

{{ forks }}
```

The fields surrounded by curly brackets are jinja placeholders that will be dynamically substituted when building the pipeline. They will ensure that the processes and potential forks correctly link with each other and that channels are unique and correctly linked. This example contains all placeholder variables that are currently supported by FlowCraft.

{{pid}}

Used as a unique process identifier that prevent issues from process and channel duplication in the pipeline. Therefore, is should be appended to each process and channel name as `_{{ pid }}` (note the underscore):

```
some_channel_{{ pid }}
process myProcess_{{ pid }}
```

{{param_id}}

Same as the **{{ pid }}**, but sets the identified for nextflow `params`. It should be appended to each `param` as `{{ param_id }}`. This will allow parameters to be specific to each component in the pipeline:

```
Channel.value(params.param1{{ param_id}})
```

Note that the parameters used in the template, should also be defined in the Process class params attribute (see *Parameters*).

{% include "post.txt" %}

Inserts `beforeScript` and `afterScript` statements to the process that sets environmental variables and a series of *dotfiles* for the process to log their status, warnings, fails and reports (see *Dotfiles* for more information). It also includes scripts for sending requests to REST APIs (only when certain pipeline parameters are used).

{{input_channel}}

All processes must include **one and only one** input channel. In most cases, this channel should be defined with a two element tuple that contains the sample ID and then the actual data file/stream. We suggest the sample ID variable to be named `sample_id` as a standard. If other name variable name is specified and you include the `compiler_channels.txt` in the process, you'll need to change the sample ID variable (see *Sample ID variable*).

{{output_channel}}

Terminal processes may skip the output channel entirely. However, if you want to link the main output of this process with subsequent ones, this placeholder must be used **only once**. Like in the input channel, this channel should be defined with a two element tuple with the sample ID and the data. The sample ID must match the one specified in the `input_channel`.

{% include “compiler_channels.txt %}

This will include the special channels that will compile the status/logging of the processes throughout the pipeline. **You must include the whole block** (see *Status channels*):

```
{% with task_name="abricate" %}
{%- include "compiler_channels.txt" ignore missing -%}
{% endwith %}
```

{{forks}}

Inserts potential forks of the main output channel. It is **mandatory** if the `output_channel` is set.

Complete example

As an example of a complete process, this is the template of `spades.nf`:

```
IN_spades_opts_{{ pid }} = Channel.value([params.spadesMinCoverage{{ param_id }},
↪params.spadesMinKmerCoverage{{ param_id }}])
IN_spades_kmers_{{pid}} = Channel.value(params.spadesKmers{{ param_id }})

process spades_{{ pid }} {

    // Send POST request to platform
    {% include "post.txt" ignore missing %}

    tag { fastq_id + " getStats" }
    publishDir 'results/assembly/spades/', pattern: '*_spades.assembly.fasta', mode:
↪'copy'

    input:
    set fastq_id, file(fastq_pair), max_len from {{ input_channel }}.join(SIDE_max_
↪len_{{ pid }})
    val opts from IN_spades_opts_{{ pid }}
    val kmers from IN_spades_kmers_{{ pid }}

    output:
    set fastq_id, file('*_spades.assembly.fasta') optional true into {{ output_
↪channel }}
    set fastq_id, val("spades"), file(".status"), file(".warning"), file(".fail")␣
↪into STATUS_{{ pid }}
    file ".report.json"

    script:
    template "spades.py"
}
```

(continues on next page)

(continued from previous page)

```
{{ forks }}
```

11.1.2 Create Process class

The process class will contain the information that FlowCraft will use to build the pipeline and assess potential conflicts/dependencies between process. This class should be created in one the category files in the *flowcraft.generator.components* module (e.g.: `assembly.py`). If the new component does not fit in any of the existing categories, create a new one that imports `flowcraft.generator.process.Process` and add your new class. This class should inherit from the `Process` base class:

```
class MyProcess(Process):

    def __init__(self, **kwargs):

        super().__init__(**kwargs)

        self.input_type = "fastq"
        self.output_type = "fasta"
```

This is the simplest working example of a process class, which basically needs to inherit the parent class attributes (the `super` part). Then we only need to define the expected input and output types of the process. There are no limitations to the input/output types. However, a pipeline will only build successfully when all processes correctly link the output with the input type.

Depending on the process, other attributes may be required:

- *Parameters*: Parameters provided by the user to be used in the process.
- *Secondary inputs*: Channels created from parameters provided by the user.
- Secondary *Link start* and *Link end*: Secondary links that connect secondary information between two processes.
- *Dependencies*: List of other processes that may be required for the current process.
- *Directives*: Default information for RAM/CPU/Container directives and more.

11.1.3 Add to available components

Contrary to previous implementation (version <= 1.3.1), the available components are now retrieved automatically by FlowCraft and there is no need to add the process to any dictionary (previous `process_map`). In order for the component to be accessible to `flowcraft build` the process template name in `snake_case` must match the process class in `CamelCase`. For instance, if the process template is named `my_process.nf`, the process class must be `MyProcess`, then the FlowCraft will be able to automatically add it to the list of available components.

Note: Note that the template string does not include the `.nf` extension.

11.2 Process attributes

This section describes the main attributes of the `Process` class: what they do and how do they impact the pipeline generation.

11.2.1 Input/Output types

The input_type and output_type attributes set the expected type of input and output of the process. There are no limitations to the type of input/output that are provided. However, processes will only link when the output of one process matches the input of the subsequent process (unless the ignore_type attribute is set to True). Otherwise, FlowCraft will raise an exception stating that two processes could not be linked.

Note: The input/ouput types that are currently used are fastq, fasta.

11.2.2 Parameters

The params attribute sets the parameters that can be used by the process. For each parameter, a default value and a description should be provided. The default value will be set in the params.config file in the pipeline directory and the description will be used to generated the custom help message of the pipeline:

```
self.params = {
    "genomeSize": {
        "default": 2.1,
        "description": "Expected genome size (default: params.genomeSiz)
    },
    "minCoverage": {
        "default": 15,
        "description": "Minimum coverage to proceed (default: params.minCoverage)"
    }
}
```

These parameters can be simple values that are not feed into any channel, or can be automatically set to a secondary input channel via *Secondary inputs* (see below).

They can be specified when running the pipeline like any nextflow parameter (e.g.: --genomeSize 5) and used in the nextflow process as usual (e.g.: params.genomeSize).

Note: These pairs are then used to populate the params.config file that is generated in the pipeline directory. Note that the values are replaced literally in the config file. For instance, "genomeSize": 2.1, will appear as genomeSize = 2.1, whereas "adapters": "'None'" will appear as adapters = 'None'. If you want a value to appear as a string, the double and single quotes are necessary.

11.2.3 Secondary inputs

Warning: The secondary_inputs attribute has been deprecated since **v1.2.1.** Instead, specify the secondary channels directly in the nextflow template files.

Any process can receive one or more input channels in addition to the main channel. These are particularly useful when the process needs to receive additional options from the parameters scope of nextflow. These additional inputs can be specified via the secondary_inputs attribute, which should store a list of dictionaries (a dictionary for each input). Each dictionary should contains a key:value pair with the name of the parameter (params) and the definition of the nextflow channel (channel). Consider the example below:

```
self.secondary_inputs = [
        {
            "params": "genomeSize",
            "channel": "IN_genome_size = Channel.value(params.genomeSize)"
        },
        {
            "params": "minCoverage",
            "channel": "IN_min_coverage = Channel.value(params.minCoverage)"
        }
    ]
```

This process will receive two secondary inputs that are given by the `genomeSize` and `minCoverage` parameters. These should be also specified in the `params` attribute (See *Parameters* above).

For each of these parameters, the dictionary also stores how the channel should be defined at the beginning of the pipeline file. Note that this channel definition mentions the parameters (e.g. `params.genomeSize`). An additional best practice for channel definition is to include one or more sanity checks to ensure that the provided arguments are correct. These checks can be added in the nextflow template file, or literally in the `channel` string:

```
self.secondary_inputs = [
    {
        "params": "genomeSize",
        "channel":
                "IN_genome_size = Channel.value(params.genomeSize)"
                "map{it -> it.toString().isNumber() ? it : exit(1, \"The genomeSize␣
↪parameter must be a number or a float. Provided value: '${params.genomeSize}'\")}"
        }
```

11.2.4 Extra input

The `extra_input` attribute is mostly a user specified directive that allows the injection of additional input data from a parameter into the main input channel of the process. When a pipeline is defined as:

```
process1 process2={'extra_input':'var'}
```

FlowCraft will expose a new `var` parameter, setup an extra input channel and mix it with `process2` main input channel. A more detailed explanation follows below.

First, FlowCraft will create a nextflow channel from the parameter name provided via the `extra_input` directive. The channel string will depend on the input type of the process (this string is fetched from the `RAW_MAPPING` attribute). For instance, if the input type of `process2` is `fastq`, the new extra channel will be:

```
IN_var_extraInput = Channel.fromFilePairs(params.var)
```

Since the same extra input parameter may be used by more than one process, the `IN_var_extraInput` channel will be automatically forked into the final destination channels:

```
// When there is a single destination channel
IN_var_extraInput.set{ EXTRA_process2_1_2 }
// When there are multiple destination channels for the same parameter
IN_var_extraInput.into{ EXTRA_process2_1_2; EXTRA_process3_1_3 }
```

The destination channels are the ones that will be actually mixed with the main input channels:

```
process process2 {
    input:
    (...) main_channel.mix(EXTRA_process2_1_2)
}
```

In these cases, the processes that receive the extra input will process the data provided by the preceding channel **AND** by the parameter. The data provided via the extra input parameter does not have to wait for the `main_channel`, which means that they can run in parallel, if there are enough resources.

11.2.5 Compiler

The `compiler` attribute allows one or more channels of the process to be fed into a compiler process (See *Compiler processes*). These are special processes that collect information from one or more processes to execute a given task. Therefore, this parameter can only be used when there is an appropriate compiler process available (the available compiler processes are set in the `compilers` dictionary). In order to provide one or more channels to a compiler process, simply add a key:value to the attribute, where the key is the id of the compiler process present in the `compilers` dictionary and the value is the list of channels:

```
self.compiler["patlas_consensus"] = ["mappingOutputChannel"]
```

11.2.6 Link start

The `link_start` attribute stores a list of strings of channel names that can be used as secondary channels in the pipeline (See the *Secondary links between process* section). By default, this attribute contains the main output channel, which means that every process can fork the main channel to one or more receiving processes.

11.2.7 Link end

The `link_end` attribute stores a list of dictionaries with channel names that are meant to be received by the process as secondary channel **if** the corresponding *Link start* exists in the pipeline. Each dictionary in this list will define one secondary channel and requires two key:value pairs:

```
self.link_end({
    "link": "SomeChannel",
    "alias": "OtherChannel")
})
```

If another process exists in the pipeline with `self.link_start.extend(["SomeChannel"])`, FlowCraft will automatically establish a secondary channel between the two processes. If there are multiple processes receiving from a single one, the channel from the later will for into any number of receiving processes.

11.2.8 Dependencies

If a process depends on the presence of one or more processes upstream in the pipeline, these can be specific via the `dependencies` attribute. When building the pipeline if at least one of the dependencies is absent, FlowCraft will raise an exception informing of a missing dependency.

11.2.9 Directives

The directives attribute allows for information about cpu/RAM usage and container to be specified for each nextflow process in the template file. For instance, considering the case where a Process has a template with two nextflow processes:

```
process proc_A_{{ pid }} {
    // stuff
}

process proc_B_{{ pid }} {
    // stuff
}
```

Then, information about each process can be specified individually in the directives attribute:

```
class myProcess(Process):
    (...)
    self.directives = {
        "proc_A": {
            "cpus": 1
            "memory": "4GB"
        },
        "proc_B": {
            "cpus": 4
            "container": "my/container"
            "version": "1.0.0"
        }
    }
```

The information in this attribute will then be used to build the resources.config (containing the information about cpu/RAM) and containers.config (containing the container images) files. Whenever a directive is missing, such as the container and version from proc_A and memory from proc_B, nothing about them will be written into the config files and they will use the **default pipeline values**:

- cpus: 1
- memory: 1GB
- container: flowcraft_base image

11.2.10 Ignore type

The ignore_type attribute, controls whether a match between the input of the current process and the output of the previous one is enforced or not. When there are multiple terminal processes that fork from the main channel, there is no need to enforce the type match and in that case this attribute can be set to False.

11.2.11 Process ID

The process ID, set via the pid attribute, is an arbitrarily and incremental number that is awarded to each process depending on its position in the pipeline. It is mainly used to ensure that there are no duplicated channels even when the same process is used multiple times in the same pipeline.

11.2.12 Template

The `template` attribute is used to fetch the jinja2 template file that corresponds to the current process. The path to the template file is determined as follows:

```
join(<template directory>, template + ".nf")
```

11.2.13 Status channels

The status channels are special channels dedicated to passing information regarding the status, warnings, fails and logging from each process (see *Dotfiles* for more information). They are used only when the nextflow template file contains the appropriate jinja2 placeholder:

```
output:
{% with task_name="<nextflow_template_name>" %}
{%- include "compiler_channels.txt" ignore missing -%}
{% endwith %}
```

By default, every `Process` class contains a `status_channels` list attribute that contains the `template` string:

```
self.status_channels = ["STATUS_{}".format(template)]
```

If there is only one nextflow process in the template and the `task_name` variable in the template matches the `template` attribute, then it's all automatically set up.

If the template file contains **more than one nextflow process** definition, multiple placeholders can be provided in the template:

```
process A {
    (...)
    output:
    {% with task_name="A" %}
    {%- include "compiler_channels.txt" ignore missing -%}
    {% endwith %}
}

process B {
    (...)
    output:
    {% with task_name="B" %}
    {%- include "compiler_channels.txt" ignore missing -%}
    {% endwith %}
}
```

In this case, the `status_channels` attribute would need to be changed to:

```
self.status_channels = ["A", "B"]
```

Sample ID variable

In case you change the standard nextflow variable that stores the sample ID in the input of the process (`sample_id`), you also need to change it for the `compiler_channels` placeholder:

```
process A {

input:
set other_id, data from {{ input_channel }}

output:
{% with task_name="B", sample_id="other_id" %}
{%- include "compiler_channels.txt" ignore missing -%}
{% endwith %}

}
```

11.3 Advanced use cases

11.3.1 Compiler processes

Compilers are special processes that collect data from one or more processes and perform a given task with that compiled data. They are automatically included in the pipeline when at least one of the source channels is present. In the case there are multiple source channels, they are merged according to a specified operator.

Creating a compiler process

The creation of the compiler process is simpler than that of a regular process but follows the same three steps.

1. Create a nextflow template file in `flowcraft.generator.templates`:

```
process fullConsensus {

    input:
    set id, file(infile_list) from {{ compile_channels }}

    output:
    <output channels>

    script:
    """
    <commands/code/template>
    """

}
```

The only requirement is the inclusion of a `compiler_channels` jinja placeholder in the main input channel.

2. Create a Compiler class in the `flowcraft.generator.process` module:

```
class PatlasConsensus(Compiler):

    def __init__(self, **kwargs):

        super().__init__(**kwargs)
```

This class must inherit from `Compiler` and does not require any more changes.

3. Map the compiler template file to the class in `compilers` attribute:

```
self.compilers = {
"patlas_consensus": {
    "cls": pc.PatlasConsensus,
    "template": "patlas_consensus",
    "operator": "join"
    }
}
```

Each compiler should contain a key:value entry. The key is the compiler id that is then specified in the `compiler` attribute of the component classes. The value is a json/dict object that species the compiler class in the `cls` key, the template string in the `template` string and the operator used to join the channels into the compiler via the `operator` key.

How a compiler process works

Consider the case where you have a compiler process named `compiler_1` and two processes, `process_1` and `process_2`, both of which feed a single channel to `compiler_1`. This means that the class definition of these processes include:

```
class Process_1(Process):
    (...)
    self.compiler["compiler_1"] = ["channel1"]

class Process_2(Process):
    (...)
    self.compiler["compiler_1"] = ["channel2"]
```

If a pipeline is built with at least one of these process, the `compiler_1` process will be automatically included in the pipeline. If more than one channel is provided to the compiler, they will be merged with the specified operator:

```
process compiler_1 {

    input:
    set sample_id, file(infile_list) from channel2.join(channel1)

}
```

This will allow the output of multiple separate process to be processed by a single process in the pipeline, and it automatically adjusts according to the channels provided to the compiler.

11.3.2 Secondary links between process

In some cases, it might be necessary to perform additional links between two or more processes. For example, the maximum read length might be gathered in one process, and that information may be required by a subsequent process. These secondary channels allow this information to be passed between theses channels.

These additional links are called secondary channels and they may be explicitly or implicitly declared.

Explicit secondary channels

To create an explicit secondary channel, the origin or source of this channel must be declared in the nextflow process that sends it:

```
// secondary channels can be created inside the process
output:
<main output> into {{ output_channel }}
<secondary output> into SIDE_max_read_len_{{ pid }}

// or outside
SIDE_phred_{{ pid }} = Channel.create()
```

Then, we add the information that this process has a secondary channel start via the `link_start` list attribute in the corresponding `flowcraft.generator.process.Process` class:

```
class MyProcess(Process):

    (...)

    self.link_start.extend(["SIDE_max_read_len", "SIDE_phred"])
```

Notice that we extend the `link_start` list, instead of simply assigning. This is because all processes already have the main channel as an implicit link start (See *Implicit secondary channels*).

Now, any process that is executed after this one can receive this secondary channel.

For another process to receive this channel, it will be necessary to add this information to the process class(es) via the `link_end` list attribute:

```
class OtherProcess(Process):

    (...)

    self.link_end.append({
        "link": "SIDE_phred",
        "alias": "OtherName"
    })
```

Notice that now we append a dictionary with two key:values. The first, *link* must match a string from the *link_start* list (in this case, *SIDE_phred*). The second, *alias*, will be the channel name in the receiving process nextflow template (which can be the same as the *link* value).

Now, we only need to add the secondary channel to the nextflow template, as in the example below:

```
input:
<main_input> from {{ input_channel }}.mix(OtherName_{{ pid}})
```

Implicit secondary channels

By default, the main output of the channels is declared as a secondary channel start. This means that any process can receive the main output channel as a a secondary channel of a subsequent process. This can be useful in situations were a post-assembly process (has `assembly` as expected input and output) needs to receive the last channel with fastq files:

```
class AssemblyMapping(Process):

    (...)

    self.link_end.append({
        "link": "MAIN_fq",
```

(continues on next page)

(continued from previous page)

```
        "alias": "_MAIN_assembly"
    })
```

In this example, the AssemblyMapping process will receive a secondary channel with from the last process that output fastq files into a channel called _MAIN_assembly. Then, this channel is received in the nextflow template like this:

```
input:
<main input> from {{ input_channel }}.join(_{{ input_channel }})
```

Implicit secondary channels can also be used to fork the last output channel into multiple terminal processes:

```
class Abricate(Process):

    (...)

    self.link_end.append({
        "link": "MAIN_assembly",
        "alias": "MAIN_assembly"
    })
```

In this case, since MAIN_assembly is already the prefix of the main output channel of this process, there is no need for changes in the process template:

```
input:
<main input> from {{ input_channel }}
```

CHAPTER 12

Template creation guidelines

Though none of these guidelines are mandatory nor required, their usage is highly recommended for several reasons:

- Consistency in the outputs of the templates throughout the pipeline, particularly the status and report dotfiles (see *Dotfiles* section);
- Debugging purposes;
- Versioning;
- Proper documentation of the template scripts.

12.1 Preface header

After the script shebang, a header with a brief description of the purpose and expected inputs and outputs should be provided. A complete example of such description can be viewed in `flowcraft.templates.integrity_coverage`.

12.1.1 Purpose

Purpose section contains a brief description of the script's objective. E.g.:

```
Purpose
-------

This module is intended parse the results of FastQC for paired end FastQ \
samples.
```

12.1.2 Expected input

Expected input section contains a description of the variables that are provided to the main function of the template script. These variables are defined in the input channels of the process in which the template is supposed to be executed.

E.g.:

```
Expected input
--------------

The following variables are expected whether using NextFlow or the
:py:func:`main` executor.

- ``mash_output`` : String with the name of the mash screen output file.
    - e.g.: ``'sortedMashScreenResults_SampleA.txt'``
```

This means that the process that will execute this channel will have the input defined as:

```
input:
file(mash_output) from <channel>
```

12.1.3 Generated output

Generated output section contains a description of the output files that the template script is intended to generated. E.g.:

```
Generated output
----------------

The generated output are output files that contain an object, usually a string.

- ``fastqc_health`` : Stores the health check for the current sample. If it
    passes all checks, it contains only the string 'pass'. Otherwise, contains
    the summary categories and their respective results
```

These can then be passed to the output channel(s) in the nextflow process:

```
output:
file(fastqc_health) into <channel>
```

Note: Since templates can be re-used by multiple processes, not all generated outputs need to be passed to output channels. Depending on the job of the nextflow process, it may catch none or all of the output files generated by the template.

12.2 Versioning and logging

FlowCraft has a specific `logger` (*`get_logger()`*) and versioning system that can be imported from *`flowcraft.templates.flowcraft_utils`*:

```
# the module that imports the logger and the decorator class for versioning
# of the script itself and other software used in the script
from flowcraft_utils.flowcraft_base import get_logger, MainWrapper
```

12.2.1 Logger

A *logger* function is also required to add logs to the script. The logs are written to the `.command.log` file in the work directory of each process.

First, the logger must be called, for example, after the **imports** as follows:

```
logger = get_logger(__file__)
```

Then, it may be used at will, using the default logging levels . E.g.:

```
logger.debug("Information tha may be important for debugging")
logger.info("Information related to the normal execution steps")
logger.warning("Events that may require the attention of the developer")
logger.error("Module exited unexpectedly with error:\\n{}".format(
            traceback.format_exc()))
```

12.2.2 MainWrapper decorator

This `MainWrapper` class decorator allows the program to fetch information on the script version, build and template name. For example:

```
# This can also be declared after the imports
__version__ = "1.0.0"
__build__ = "15012018"
__template__ = "process_abricate-nf"
```

The `MainWrapper` should decorate the main function of the script. E.g.:

```
@MainWrapper
def main():
    #some awesome code
    ...
```

Besides searching for the script's version, build and template name this decorator will also search for a specific set of functions that start with the substring `__get_version`. For example:

```
def __get_version_fastqc():

    try:

    cli = ["fastqc", "--version"]
    p = subprocess.Popen(cli, stdout=PIPE, stderr=PIPE)
    stdout, _ = p.communicate()

    version = stdout.strip().split()[1][1:].decode("utf8")

    except Exception as e:
        logger.debug(e)
        version = "undefined"

    # Note that it returns a dictionary that will then be written to the .versions
    # dotfile
    return {
        "program": "FastQC",
        "version": version,
```

(continues on next page)

(continued from previous page)

```
        # some programs may also contain build.
    }
```

These functions are used to fetch the version, name and other relevant information from third-party software and the only requirement is that they return a dictionary with **at least** two key:value pairs:

- `program`: String with the name of the program.
- `version`: String with the version of the program.

For more information, refer to the *build_versions()* method.

12.3 Nextflow *.command.sh*

When these templates are used as a Nextflow template they are executed as a `.command.sh` file in the work directory of each process. In this case, we recommended the inclusion of an **if statement** to parse the arguments sent from nextflow to the python template. For example, imagine we have a path to a file name to pass as argument between nextflow and the required template:

```
# code check for nextflow execution
if __file__.endswith(".command.sh"):
    FILE_NAME = '$Nextflow_file_name'
    # logger output can also be included here, for example:
    logger.debug("Running {} with parameters:".format(
        os.path.basename(__file__)))
    logger.debug("FILE_NAME: {}".format(FILE_NAME))
```

Then, we could use this variable as the argument of a function, such as:

```
def main(FILE_NAME):
    #some awesome code
    ...
```

This way, we can use this function with nextflow arguments or without them, as is the case when the templates are used as standalone modules.

12.4 Use numpy docstrings

`FlowCraft` uses numpy docstrings to document code. Use this link for reference.

CHAPTER 13

Recipe creation guidelines

Recipes are pre-made pipeline strings that may be associated with specific parameters and directives and are used to rapidly build a certain type of pipeline.

Instead of building a pipeline like:

```
-t "integrity_coverage fastqc_trimmomatic fastqc spades pilon"
```

The user simply can specific a recipe with that pipeline:

```
-r assembly
```

13.1 Recipe creation

The creation of new recipes is a very simple and straightforward process. You need to create a new file in the `flowcraft/generator/recipes` folder with any name and create a basic class with three attributes:

```
try:
    from generator.recipe import Recipe
except ImportError:
    from flowcraft.generator.recipe import Recipe


class Innuca(Recipe):

    def __init__(self):
        super().__init__()

        # Recipe name
        self.name = "innuca"

        # Recipe pipeline
        self.pipeline_str = <pipeline string>
```

(continues on next page)

(continued from previous page)

```
        # Recipe parameters and directives
        self.directives = { <directives> }
```

And that's it! Now there is a new recipe available with the `innuca` name and we can build this pipeline using the option `-r innuca`.

13.1.1 Name

This is the name of the recipe, which is used to make a match with the recipe name provided by the user via the `-r` option.

13.1.2 Pipeline_str

The pipeline string as if provided via the `-t` option.

13.1.3 Directives

A dictionary containing the parameters and directives for each process in the pipeline string. **Setting this attribute is optional and components that are not specified here will assume their default values**. In general, each element in this dictionary should have the following format:

```
self.directives = {
    "component_name": {
        "params": {
            "paramA": "value"
        },
        "directives": {
            "directiveA": "value"
        }
    }
}
```

This will set the provided parameters and directives to the component, but it is possible to provide only one.

A more concrete example of a real component and directives follows:

```
self.pipeline_str = "integrity_coverage fastqc"

# Set parameters and directives only for integrity_coverage
# and leave fastqc with the defaults
self.directives = {
    "integrity_coverage": {
        "params": {
            "minCoverage": 20
        },
        "directives": {
            "memory": "1GB"
        }
    }
}
```

Duplicate components

In some cases, the same component may be present multiple times in the pipeline string of a recipe. In these cases, directives can be assigned to each individual component by adding a `#<id>` suffix to the component:

```
self.pipeline_str = "integrity_coverage ( trimmomatic spades#1 | spades#2)"

self.directives = {
    "spades#1": {
        "directives": {
            "memory": "10GB"
        }
    },
    "spades#2": {
        "directives": {
            "version": "3.7.0"
        }
    }
}
```

CHAPTER 14

Docker containers guidelines

All FlowCraft components require a docker container in order to be executed, thus if a new component is added, a docker image should be added as well and uploaded to .. _docker hub: https://hub.docker.com/ in order to be available to pull in other machines. Although this can be done in any personal repository, we recommend that this docker images are added to an already existing .. _FlowCraft github repository: https://github.com/assemblerflow/docker-imgs (called here `Official`) so that docker builds can be automated with github integration. Also, the centralization of all images will allow other contributors to easily access and edit these containers instead of forking from one side to another every time a container needs to be changed/updated.

14.1 Official FlowCraft Docker images

14.1.1 Writing docker images

Official FlowCraft Docker images are available in .. _this github repository: https://github.com/assemblerflow/docker-imgs . If you want to add your image to this repository please fork it and make a Pull Request (PR) with the requested new image or create an issue asking to be added to the organization as a contributor.

14.1.2 Building docker images

Then, after the image has been added to the FlowCraft .. _docker-imgs https://github.com/assemblerflow/docker-imgs github repository, they can be built through .. _FlowCraft docker hub https://hub.docker.com/u/flowcraft/dashboard/ .

Tag naming

Each time a docker image is built using the automated build of docker hub it should follow this nomenclature: `version-patch`. This is used to avoid the override of previous builds for the same images, allowing for instance users to use different version of the same software using the same docker image but with different tags.

- `Version`: Is a string with tree letters like this: `1.1.1`. Versions should

change every time a new software is added the container.

- `Patch`: Is a number that follows a - after the version. Patches should

change every time a change does not affect the software inside it. For example, updates to database related files required by some of the software inside the container.

14.2 Unofficial FlowCraft Docker images

Although we **strongly** recommend that all images are stored in FlowCraft .. _docker-imgs https://github.com/assemblerflow/docker-imgs github repo, it is not mandatory to do it. Images can be built in another github repo and also use another docker hub repository to build the images. However, do make sure that you define it correctly in the directives of the process as explained in *Directives*.

CHAPTER 15

Dotfiles

Several dotfiles (files prefixed by a single `.`, as in `.status`) are created at the beginning of every nextflow process that has the following placeholder (see *Create process template*):

```
process myProcess {
    {% include "post.txt" ignore missing %}
    (...)
}
```

The actual script that creates the dotfiles is found in `flowcraft/bin`, is called `set_dotfiles.sh` and executes the following command:

```
touch .status .warning .fail .report.json .versions
```

15.1 Status

The `.status` file simply stores a string with the run status of the process. The supported status are:

- `pass`: The process finished successfully
- `fail`: The process ran without unexpected issues but failed due to some quality control check
- `error`: The process exited with an unexpected error.

15.2 Warning

The `.warning` file stores any warnings that may occur during the execution of the process. There is no particular format for the warning messages other than that each individual warning should be in a separate line.

15.3 Fail

The `.fail` file stores any fail messages that may occur during the execution of the process. When this occurs, the `.status` channel must have the `fail` string as well. As in the warning dotfile, there is no particular format for the fail message.

15.4 Report JSON

Important: The general specification of the report JSON changed in version 1.2.2. See the issue tracker for details.

The `.report.json` file stores any information from a given process that is deemed worthy of being reported and displayed at the end of the pipeline. Any information can be stored in this file, as long as it is in JSON format, but there are a couple of recommendations that are necessary to follow for them to be processed by a reporting web app (Currently hosted at flowcraft-webapp). However, if data processing will be performed with custom scripts, feel free to specify your own format.

15.4.1 Information for tables

Information meant to be displayed in tables should be in the following format:

```
json_dic = {
    "tableRow": [{
        "sample": "A",
        "data": [{
            "header": "Raw BP",
            "value": 123,
            "table": "qc"
        }, {
            "header": "Coverage",
            "value": 32,
            "table": "qc"
        }]
    }, {
        "sample": "B",
        "data": [{
            "header": "Coverage",
            "value": 35,
            "table": "qc"
        }]
    }]
}
```

This provides table information for multiple samples in the same process. In this case, data for two samples is provided. For each sample, values for one or more headers can be provided. For instance, this report provides information about the **Raw BP** and **Coverage** for sample **A** and this information should go to the **qc** table. If any other information is relevant to build the table, feel free to add more elements to the JSON.

15.4.2 Information for plots

Information meant to be displayed in plots should be in the following format:

```
json_dic = {
    "plotData": [{
        "sample": "strainA",
        "data": {
            "sparkline": 23123,
            "otherplot": [1,2,3]
         }
    }],
}
```

As in the table JSON, *plotData* should be an array with an entry for each sample. The data for each sample should be another JSON where the keys are the *plot signatures*, so that we know to which plot the data belongs. The corresponding values are whatever data object you need.

15.4.3 Other information

Other than tables and plots, which have a somewhat predefined format, there is not particular format for other information. They will simply store the data of interest to report and it will be the job of a downstream report app to process that data into an actual visual report.

15.5 Versions

The `.version` dotfile should contain a list of JSON objects with the version information of the programs used in any given process. There are only two required key:value pairs:

- `program`: String with the name of the software/script/template
- `version`: String with the version of said software.

As an example:

```
version = {
    "program": "abricate"
    "version": "0.3.7"
}
```

Key:value pairs with other metadata can be included at will for downstream processing.

CHAPTER 16

Pipeline reporting

This section describes how the reports of a FlowCraft pipeline are generated and collected at the end of a run. These reports can then be sent to the FlowCraft web application where the results are visualized.

Important: Note that if the nextflow process reports add new types of data, one or more React components need to be added to the web application for them to be rendered.

16.1 Data collection

The data for the pipeline reports is collected from three dotfiles in each nextflow process (they should be present in each work sub directory):

- **.report.json**: Contains report data (See *Report JSON* for more information).
- **.versions**: Contains information about the versions of the software used (See *Versions* for more information).
- **.command.trace**: Contains resource usage information.

The **.command.trace** file is generated by nextflow when the **trace** scope is active. The **.report.json** and **.version** files are specific to FlowCraft pipelines.

16.1.1 Generation of dotfiles

Both **report.json** and **.versions** empty dotfiles are automatically generated by the `{% include "post.txt" ignore missing %}` placeholder, specified in the *Create process template* section. Using this placeholder in your processes is all that is needed.

16.1.2 Collection of dotfiles

The **.report.json**, **.versions** and **.command.trace** files are automatically collected and sent to dedicated report channels in the pipeline by the `{%- include "compiler_channels.txt" ignore missing -%}` placeholder, specified in the *process creation* section. Placing this placeholder in your processes will generate the following line in the output channel specification:

```
set {{ sample_id|default("sample_id") }}, val("{{ task_name }}_{{ pid }}"), val("{{␣
↪pid }}"), file(".report.json"), file(".versions"), file(".command.trace") into␣
↪REPORT_{{task_name}}_{{ pid }}
```

This line collects several metadata associated with the process along with the three dotfiles.

16.1.3 Compilation of dotfiles

As mentioned in the previous section, the dotfiles and other relevant metadata for are sent through special report channels to a FlowCraft component that is responsible for compiling all the information and generate a single report file at the end of each pipeline run.

This component is specified in `flowcraft.generator.templates.report_compiler.nf` and it consists of two nextflow processes:

- First, the **report** process receives the data from each executed process that sends report data and runs the `flowcraft/bin/prepare_reports.py` script on that data. This script will simply merge metadata and dotfiles information in a single JSON file. This file contains the following keys:
  - `reportJson`: The data in **.report.json** file.
  - `versions`: The data in **.versions** file.
  - `trace`: The data in **.command.trace** file.
  - `processId`: The process ID
  - `pipelineId`: The pipeline ID that defaults to one, unless specified in the parameters.
  - `projectid`: The project ID that defaults to one, unless specified in the parameters.
  - `userId`: The user ID that defaults to one, unless specified in the parameters.
  - `username`: The user name that defaults to *user*, unless specified in the parameters
  - `processName`: The name of the flowcraft component.
  - `workdir`: The work directory where the process was executed.
- Second, all JSON files created in the process above are merged and a single reports JSON file is created. This file will contains the following structure:

```
reportJSON = {
    "data": {
        "results": [<array of report JSONs>]
    }
}
```

CHAPTER 17

Reports

17.1 Report JSON specification

The report JSON is quite flexibly on the information it can contain. Here are some guidelines to promote consistency on the reports generated by each component. In general, the reports file is an array of JSON objects that contain relevant information for each executed process in the pipeline:

```
reportFile = [{<processA/tagA reports>}, {<processB/tagB reports>}, ... ]
```

17.1.1 Nextflow metadata

The nextflow metada is automatically added to the reportFile as a single JSON entry with the `nfMetadata` key that contains the following information:

```
"nfMetadata": {
    "scriptId": "${workflow.scriptId}",
    "scriptName": "${workflow.scriptId}",
    "profile": "${workflow.profile}",
    "container": "${workflow.container}",
    "containerEngine": "${workflow.containerEngine}",
    "commandLine": "${workflow.commandLine}",
    "runName": "${workflow.runName}",
    "sessionId": "${workflow.sessionId}",
    "projectDir": "${workflow.projectDir}",
    "launchDir": "${workflow.launchDir}",
    "start_time": "${workflow.start}"
}
```

Note: Unlike the remaining JSON entries in the report file, which are generated for each process execution, the `nfMetadata` entry is generated only once per project execution.

17.1.2 Root

The reports contained in the `reports.json` file for each process execution are added to the root object:

```
{
    "pipelineId": 1,
    "processId": pid,
    "processName": task_name,
    "projectid": RUN_NAME,
    "reportJson": reports,
    "runName": RUN_NAME,
    "scriptId": SCRIPT_ID,
    "versions": versions,
    "trace": trace,
    "userId": 1,
    "username": "user",
    "workdir": dirname(abspath(report_json))
}
```

The other key:values are added automatically when the reports are compiled for each process execution.

17.1.3 Versions

Inside the root, the signature key for software version information is `versions`:

```
"versions": [{
    "program": "progA",
    "version": "1.0.0",
    "build": "1"
}, {
    "program": "progB",
    "version": "2.1"
}]
```

Only the `program` and `version` keys are mandatory.

17.1.4 ReportJson

Table data

Inside `reportJson`, the signature key for table data is `tableRow`:

```
 "reportJson": {
     "tableRow": [{
         "sample": "strainA",
         "data": [{
             "header": "Raw BP",
             "value": 123,
             "table": "qc",
         }, {
             "header": "Coverage",
             "value": 32,
             "table": "qc"
         }],
         "sample": "strainB",
```

(continues on next page)

(continued from previous page)

```
         "data": [{
             "header": "Raw BP",
             "value": 321,
             "table": "qc",
         }, {
             "header": "Coverage",
             "value": 22,
             "table": "qc"
         }]
     }]
}
```

`tableRow` should contain an array of JSON for each sample with two key:value pairs:

- `sample`: Sample name
- `data`: Table data (see below).

`data` should be an array of JSON with at least three key:value pairs:

- `header`: Column header
- `value`: The data value
- `table`: Informs to which table this data should go.

Note: Available `table` keys: `typing`, `qc`, `assembly`, `abricate`, `chewbbaca`.

Plot data

Inside `reportJson`, the signature key for plot data is `plotData`:

```
"reportJson": {
    "plotData": [{
        "sample": "strainA",
        "data": {
            "sparkline": 23123,
            "otherplot": [1,2,3]
         }
    }],
}
```

`plotData` should contain an array of JSON for each sample with two key:value pairs:

- `sample`: Sample name
- `data`: Plot data (see below).

`data` should contain a JSON object with the plot signatures as keys, and the relevant plot data as value. This data can be any object (integer, float, array, JSON, etc). **It will be up to the components in the flowcraft web application to parse this data and generate the appropriate chart.**

Warnings and fails

Inside `reportJson`, the signature key for warnings is `warnings` and for failures is `fail`:

```
"reportJson": {
    "warnings": [{
        "sample": "strainA",
        "table": "qc",
        "value": ["message 1", "message 2"]
    }],
    "fail": [{
        "sample": "strainA",
        "table": "assembly",
        "value": ["message 1"]
    }]
}
```

`warnings`/`fail` should contain an array of JSON for each sample with two key:value pairs:

- `sample`: Sample name
- `value`: An array with one or more string messages.
- `table` **[optional]**: If a table signature is provided, the warning/fail messages information will appear on that table. Otherwise, it will appear as a general warning/error that is associated to the sample but not to any particular table.

CHAPTER 18

flowcraft package

18.1 Subpackages

18.1.1 flowcraft.generator package

Subpackages

flowcraft.generator.components package

Submodules

flowcraft.generator.components.annotation module

flowcraft.generator.components.assembly module

flowcraft.generator.components.assembly_processing module

flowcraft.generator.components.distance_estimation module

flowcraft.generator.components.downloads module

flowcraft.generator.components.metagenomics module

flowcraft.generator.components.mlst module

flowcraft.generator.components.patlas_mapping module

flowcraft.generator.components.reads_quality_control module

flowcraft.generator.components.typing module

Module contents

Submodules

exception flowcraft.generator.error_handling.**SanityError**(*value*)
Bases: exceptions.Exception

Class to raise a custom error for sanity checks

exception flowcraft.generator.error_handling.**InspectionError**(*value*)
Bases: exceptions.Exception

exception flowcraft.generator.error_handling.**ReportError**(*value*)
Bases: exceptions.Exception

exception flowcraft.generator.error_handling.**RecipeError**(*value*)
Bases: exceptions.Exception

flowcraft.generator.footer_skeleton module

flowcraft.generator.header_skeleton module

flowcraft.generator.inspect module

flowcraft.generator.pipeline_parser module

flowcraft.generator.pipeline_parser.**guess_process**(*query_str*, *process_map*)
Function to guess processes based on strings that are not available in process_map. If the string has typos and is somewhat similar (50%) to any process available in flowcraft it will print info to the terminal, suggesting the most similar processes available in flowcraft.

Parameters

query_str: str The string of the process with potential typos

process_map: The dictionary that contains all the available processes

flowcraft.generator.pipeline_parser.**remove_inner_forks**(*text*)
Recursively removes nested brackets

This function is used to remove nested brackets from fork strings using regular expressions

Parameters

text: str The string that contains brackets with inner forks to be removed

Returns

text: str the string with only the processes that are not in inner forks, thus the processes that belong to a given fork.

flowcraft.generator.pipeline_parser.**empty_tasks**(*p_string*)
Function to check if pipeline string is empty or has an empty string

Parameters

p_string: str

String with the definition of the pipeline, e.g.:: 'processA processB processC(ProcessD | ProcessE)'

flowcraft.generator.pipeline_parser.**brackets_but_no_lanes**(*p_string*)
Function to check if a LANE_TOKEN is provided but no fork is initiated. Parameters ——— p_string: str

String with the definition of the pipeline, e.g.:: 'processA processB processC(ProcessD | ProcessE)'

`flowcraft.generator.pipeline_parser.`**`brackets_insanity_check`**(*p_string*)

This function performs a check for different number of '(' and ')' characters, which indicates that some forks are poorly constructed.

Parameters

p_string: str

String with the definition of the pipeline, e.g.:: 'processA processB processC(ProcessD | ProcessE)'

`flowcraft.generator.pipeline_parser.`**`lane_char_insanity_check`**(*p_string*)

This function performs a sanity check for multiple '|' character between two processes.

Parameters

p_string: str

String with the definition of the pipeline, e.g.:: 'processA processB processC(ProcessD | ProcessE)'

`flowcraft.generator.pipeline_parser.`**`final_char_insanity_check`**(*p_string*)

This function checks if lane token is the last element of the pipeline string.

Parameters

p_string: str

String with the definition of the pipeline, e.g.:: 'processA processB processC(ProcessD | ProcessE)'

`flowcraft.generator.pipeline_parser.`**`fork_procs_insanity_check`**(*p_string*)

This function checks if the pipeline string contains a process between the fork start token or end token and the separator (lane) token. Checks for the absence of processes in one of the branches of the fork ['|)' **and '(|**'] and for the existence of a process before starting a fork (in an inner fork) ['|('].

Parameters

p_string: str

String with the definition of the pipeline, e.g.:: 'processA processB processC(ProcessD | ProcessE)'

`flowcraft.generator.pipeline_parser.`**`start_proc_insanity_check`**(*p_string*)

This function checks if there is a starting process after the beginning of each fork. It checks for duplicated start tokens ['(('].

Parameters

p_string: str

String with the definition of the pipeline, e.g.:: 'processA processB processC(ProcessD | ProcessE)'

`flowcraft.generator.pipeline_parser.`**`late_proc_insanity_check`**(*p_string*)

This function checks if there are processes after the close token. It searches for everything that isn't "|" or ")" after a ")" token.

Parameters

p_string: str

String with the definition of the pipeline, e.g.:: 'processA processB processC(ProcessD | ProcessE)'

`flowcraft.generator.pipeline_parser.`**`inner_fork_insanity_checks`**(*pipeline_string*)

This function performs two sanity checks in the pipeline string. The first check, assures that each fork contains a lane token '|', while the second check looks for duplicated processes within the same fork.

Parameters

pipeline_string: str

String with the definition of the pipeline, e.g.:: 'processA processB processC(ProcessD | ProcessE)'

`flowcraft.generator.pipeline_parser.`**`insanity_checks`**(*pipeline_str*)

Wrapper that performs all sanity checks on the pipeline string

Parameters

pipeline_str [str] String with the pipeline definition

`flowcraft.generator.pipeline_parser.`**`parse_pipeline`**(*pipeline_str*)

Parses a pipeline string into a list of dictionaries with the connections between processes

Parameters

pipeline_str [str]

String with the definition of the pipeline, e.g.:: 'processA processB processC(ProcessD | ProcessE)'

Returns

pipeline_links [list]

`flowcraft.generator.pipeline_parser.`**`get_source_lane`**(*fork_process*, *pipeline_list*)

Returns the lane of the last process that matches fork_process

Parameters

fork_process [list] List of processes before the fork.

pipeline_list [list] List with the pipeline connection dictionaries.

Returns

int Lane of the last process that matches fork_process

`flowcraft.generator.pipeline_parser.`**`get_lanes`**(*lanes_str*)

From a raw pipeline string, get a list of lanes from the start of the current fork.

When the pipeline is being parsed, it will be split at every fork position. The string at the right of the fork position will be provided to this function. It's job is to retrieve the lanes that result from that fork, ignoring any nested forks.

Parameters

lanes_str [str] Pipeline string after a fork split

Returns

lanes [list] List of lists, with the list of processes for each lane

`flowcraft.generator.pipeline_parser.`**`linear_connection`**(*plist*, *lane*)

Connects a linear list of processes into a list of dictionaries

Parameters

plist [list] List with process names. This list should contain at least two entries.

lane [int] Corresponding lane of the processes

Returns

res [list] List of dictionaries with the links between processes

`flowcraft.generator.pipeline_parser.`**`fork_connection`**`(`*source*, *sink*, *source_lane*, *lane*`)`

Makes the connection between a process and the first processes in the lanes to which it forks.

The `lane` argument should correspond to the lane of the source process. For each lane in `sink`, the lane counter will increase.

Parameters

source [str] Name of the process that is forking

sink [list] List of the processes where the source will fork to. Each element corresponds to the start of a lane.

source_lane [int] Lane of the forking process

lane [int] Lane of the source process

Returns

res [list] List of dictionaries with the links between processes

`flowcraft.generator.pipeline_parser.`**`linear_lane_connection`**`(`*lane_list*, *lane*`)`

Parameters

lane_list [list] Each element should correspond to a list of processes for a given lane

lane [int] Lane counter before the fork start

Returns

res [list] List of dictionaries with the links between processes

`flowcraft.generator.pipeline_parser.`**`add_unique_identifiers`**`(`*pipeline_str*`)`

Returns the pipeline string with unique identifiers and a dictionary with references between the unique keys and the original values

Parameters

pipeline_str [str] Pipeline string

Returns

str Pipeline string with unique identifiers

dict Match between process unique values and original names

`flowcraft.generator.pipeline_parser.`**`remove_unique_identifiers`**`(`*identifiers_to_tags*, *pipeline_links*`)`

Removes unique identifiers and add the original process names to the already parsed pipelines

Parameters

identifiers_to_tags [dict] Match between unique process identifiers and process names

pipeline_links: list Parsed pipeline list with unique identifiers

Returns

list Pipeline list with original identifiers

flowcraft.generator.process module

flowcraft.generator.process_details module

`flowcraft.generator.process_details.`**`colored_print`**(*msg*, *color_label='white_bold'*)

This function enables users to add a color to the print. It also enables to pass end_char to print allowing to print several strings in the same line in different prints.

Parameters

color_string: str

The color code to pass to the function, which enables color change as well as background color change.

msg: str The actual text to be printed

end_char: str The character in which each print should finish. By default it will be "

".

`flowcraft.generator.process_details.`**`procs_dict_parser`**(*procs_dict*)

This function handles the dictionary of attributes of each Process class to print to stdout lists of all the components or the components which the user specifies in the -t flag.

Parameters

procs_dict: dict A dictionary with the class attributes for all the components (or components that are used by the -t flag), that allow to create both the short_list and detailed_list. Dictionary example: {"abyss": {'input_type': 'fastq', 'output_type': 'fasta', 'dependencies': [], 'directives': {'abyss': {'cpus': 4, 'memory': '{ 5.GB * task.attempt }', 'container': 'flowcraft/abyss', 'version': '2.1.1', 'scratch': 'true'}}}

`flowcraft.generator.process_details.`**`proc_collector`**(*process_map*, *args*, *pipeline_string*)

Function that collects all processes available and stores a dictionary of the required arguments of each process class to be passed to procs_dict_parser

Parameters

process_map: dict The dictionary with the Processes currently available in flowcraft and their corresponding classes as values

args: argparse.Namespace The arguments passed through argparser that will be access to check the type of list to be printed

pipeline_string: str the pipeline string

flowcraft.generator.recipe module

Module contents

Placeholder for Process creation docs

18.1.2 flowcraft.templates package

Subpackages

flowcraft.templates.flowcraft_utils package

Submodules

flowcraft.templates.flowcraft_utils.flowcraft_base module

`flowcraft.templates.flowcraft_utils.flowcraft_base.`**`get_logger`**(*filepath*, *level=10*)

`flowcraft.templates.flowcraft_utils.flowcraft_base.`**`log_error`**()

Nextflow specific function that logs an error upon unexpected failing

`class` `flowcraft.templates.flowcraft_utils.flowcraft_base.`**`MainWrapper`**(*f*)

Methods

`build_versions`()	Writes versions JSON for a template file

__call__	

`build_versions`()

Writes versions JSON for a template file

This method creates the JSON file `.versions` based on the metadata and specific functions that are present in a given template script.

It starts by fetching the template metadata, which can be specified via the `__version__`, `__template__` and `__build__` attributes. If all of these attributes exist, it starts to populate a JSON/dict array (Note that the absence of any one of them will prevent the version from being written).

Then, it will search the template scope for functions that start with the substring `__set_version` (For example ‘‘def __set_version_fastqc()‘). These functions should gather the version of an arbitrary program and return a JSON/dict object with the following information:

```
{
    "program": <program_name>,
    "version": <version>
    "build": <build>
}
```

This JSON/dict object is then written in the `.versions` file.

Module contents

Submodules

flowcraft.templates.assembly_report module

Purpose

This module is intended to provide a summary report for a given assembly in Fasta format.

Expected input

The following variables are expected whether using NextFlow or the `main()` executor.

- **`sample_id`** [Sample Identification string.]
  - e.g.: `'SampleA'`
- **`assembly`** [Path to assembly file in Fasta format.]
  - e.g.: `'assembly.fasta'`

Generated output

- **`${sample_id}_assembly_report.csv`** [CSV with summary information of the assembly.]
  - e.g.: `'SampleA_assembly_report.csv'`

Code documentation

`class` `flowcraft.templates.assembly_report.`**`Assembly`**(*assembly_file*, *sample_id*)

Class that parses and filters an assembly file in Fasta format.

This class parses an assembly file, collects a number of summary statistics and metadata from the contigs and reports.

Parameters

assembly_file [str] Path to assembly file.

sample_id [str] Name of the sample for the current assembly.

Methods

`get_coverage_sliding`(coverage_file[, window])	**Parameters**
`get_gc_sliding`([window])	Calculates a sliding window of the GC content for the assembly
`get_summary_stats`([output_csv])	Generates a CSV report with summary statistics about the assembly

summary_info = None
OrderedDict: Initialize summary information dictionary. Contains keys:

- `ncontigs`: Number of contigs
- `avg_contig_size`: Average size of contigs
- `n50`: N50 metric
- `total_len`: Total assembly length
- `avg_gc`: Average GC proportion
- `missing_data`: Count of missing data characters

contigs = None
OrderedDict: Object that maps the contig headers to the corresponding sequence

contig_coverage = None
OrderedDict: Object that maps the contig headers to the corresponding list of per-base coverage

sample = None
str: Sample id

contig_boundaries = None
dict: Maps the boundaries of each contig in the genome

get_summary_stats(*output_csv=None*)
Generates a CSV report with summary statistics about the assembly

The calculated statistics are:

- Number of contigs
- Average contig size
- N50
- Total assembly length
- Average GC content
- Amount of missing data

Parameters

output_csv: str Name of the output CSV file.

get_gc_sliding(*window=2000*)
Calculates a sliding window of the GC content for the assembly

Returns

gc_res [list] List of GC proportion floats for each data point in the sliding window

get_coverage_sliding(*coverage_file*, *window=2000*)

Parameters

coverage_file [str] Path to file containing the coverage info at the per-base level (as generated by samtools depth)

window [int] Size of sliding window

flowcraft.templates.fastqc module

Purpose

This module is intended to run FastQC on paired-end FastQ files.

Expected input

The following variables are expected whether using NextFlow or the `main()` executor.

- **`fastq_pair`** [*Pair of FastQ file paths*]
  - e.g.: `'SampleA_1.fastq.gz SampleA_2.fastq.gz'`

Generated output

The generated output are output files that contain an object, usually a string.

- **`pair_{1,2}_data`** [File containing FastQC report at the nucleotide level for each pair]
  - e.g.: `'pair_1_data'` and `'pair_2_data'`
- **`pair_{1,2}_summary`: File containing FastQC report for each category and for each pair**
  - e.g.: `'pair_1_summary'` and `'pair_2_summary'`

Code documentation

`flowcraft.templates.fastqc.`**`convert_adatpers`**`(`*adapter_fasta*`)`

Generates an adapter file for FastQC from a fasta file.

The provided adapters file is assumed to be a simple fasta file with the adapter's name as header and the corresponding sequence:

```
>TruSeq_Universal_Adapter
AATGATACGGCGACCACCGAGATCTACACTCTTTCCCTACACGACGCTCTTCCGATCT
>TruSeq_Adapter_Index 1
GATCGGAAGAGCACACGTCTGAACTCCAGTCACATCACGATCTCGTATGCCGTCTTCTGCTTG
```

Parameters

adapter_fasta [str] Path to Fasta file with adapter sequences.

Returns

adapter_out [str or None] The path to the reformatted adapter file. Returns `None` if the adapters file does not exist or the path is incorrect.

flowcraft.templates.fastqc_report module

Purpose

This module is intended parse the results of FastQC for paired end FastQ samples. It parses two reports:

- Categorical report
- Nucleotide level report.

Expected input

The following variables are expected whether using NextFlow or the `main()` executor.

- **`sample_id`** [Sample identification string]
  - e.g.: `'SampleA'`
- **`result_p1`** [Path to both FastQC result files for pair 1]
  - e.g.: `'SampleA_1_data SampleA_1_summary'`
- **`result_p2`** [Path to both FastQC result files for pair 2]
  - e.g.: `'SampleA_2_data SampleA_2_summary'`
- **`opts`** [*Specify additional arguments for executing fastqc_report. The arguments should be a string of command line arguments, The accepted arguments are:*]
  - `'--ignore-tests'` : Ignores test results from FastQC categorical summary. This is used in the first run of FastQC.

Generated output

The generated output are output files that contain an object, usually a string.

- **`fastqc_health`** [Stores the health check for the current sample. If it] passes all checks, it contains only the string 'pass'. Otherwise, contains the summary categories and their respective results - e.g.: `'pass'`
- **`optimal_trim`** [Stores a tuple with the optimal trimming positions for 5'] and 3' ends of the reads. - e.g.: `'15 151'`

Code documentation

`flowcraft.templates.fastqc_report.`**`write_json_report`**`(`*sample_id*, *data1*, *data2*`)`

Writes the report

Parameters

data1

data2

`flowcraft.templates.fastqc_report.`**`get_trim_index`**`(`*biased_list*`)`

Returns the trim index from a `bool` list

Provided with a list of `bool` elements (`[False, False, True, True]`), this function will assess the index of the list that minimizes the number of True elements (biased positions) at the extremities. To do so, it will iterate over the boolean list and find an index position where there are two consecutive `False` elements after a `True` element. This will be considered as an optimal trim position. For example, in the following list:

```
[True, True, False, True, True, False, False, False, False, ...]
```

The optimal trim index will be the 4th position, since it is the first occurrence of a `True` element with two False elements after it.

If the provided `bool` list has no `True` elements, then the 0 index is returned.

Parameters

biased_list: list List of `bool` elements, where `True` means a biased site.

Returns

x [index position of the biased list for the optimal trim.]

`flowcraft.templates.fastqc_report.`**`trim_range`**`(`*data_file*`)`

Assess the optimal trim range for a given FastQC data file.

This function will parse a single FastQC data file, namely the *'Per base sequence content'* category. It will retrieve the A/T and G/C content for each nucleotide position in the reads, and check whether the G/C and A/T proportions are between 80% and 120%. If they are, that nucleotide position is marked as biased for future removal.

Parameters

data_file: str Path to FastQC data file.

Returns

trim_nt: list List containing the range with the best trimming positions for the corresponding FastQ file. The first element is the 5' end trim index and the second element is the 3' end trim index.

`flowcraft.templates.fastqc_report.`**`get_sample_trim`**`(`*p1_data*, *p2_data*`)`

Get the optimal read trim range from data files of paired FastQ reads.

Given the FastQC data report files for paired-end FastQ reads, this function will assess the optimal trim range for the 3' and 5' ends of the paired-end reads. This assessment will be based on the *'Per sequence GC content'*.

Parameters

p1_data: str Path to FastQC data report file from pair 1

p2_data: str Path to FastQC data report file from pair 2

Returns

optimal_5trim: int Optimal trim index for the 5' end of the reads

optima_3trim: int Optimal trim index for the 3' end of the reads

See also:

`trim_range`

`flowcraft.templates.fastqc_report.`**`get_summary`**`(`*summary_file*`)`

Parses a FastQC summary report file and returns it as a dictionary.

This function parses a typical FastQC summary report file, retrieving only the information on the first two columns. For instance, a line could be:

```
'PASS   Basic Statistics        SH10762A_1.fastq.gz'
```

This parser will build a dictionary with the string in the second column as a key and the QC result as the value. In this case, the returned `dict` would be something like:

```
{"Basic Statistics": "PASS"}
```

Parameters

summary_file: str Path to FastQC summary report.

Returns

summary_info: :py:data:'OrderedDict' Returns the information of the FastQC summary report as an ordered dictionary, with the categories as strings and the QC result as values.

`flowcraft.templates.fastqc_report.`**`check_summary_health`**`(`*summary_file*, ***kwargs*`)`

Checks the health of a sample from the FastQC summary file.

Parses the FastQC summary file and tests whether the sample is good or not. There are four categories that cannot fail, and two that must pass in order for the sample pass this check. If the sample fails the quality checks, a list with the failing categories is also returned.

Categories that cannot fail:

```
fail_sensitive = [
    "Per base sequence quality",
    "Overrepresented sequences",
    "Sequence Length Distribution",
    "Per sequence GC content"
]
```

Categories that must pass:

```
must_pass = [
    "Per base N content",
    "Adapter Content"
]
```

Parameters

summary_file: str Path to FastQC summary file.

Returns

x [bool] Returns `True` if the sample passes all tests. `False` if not.

summary_info [list] A list with the FastQC categories that failed the tests. Is empty if the sample passes all tests.

flowcraft.templates.integrity_coverage module

flowcraft.templates.mapping2json module

flowcraft.templates.mashdist2json module

Purpose

This module is intended to generate a json output for mash dist results that can be imported in pATLAS.

Expected input

The following variables are expected whether using NextFlow or the `main()` executor.

- **`mash_output`** [String with the name of the mash screen output file.]
    - e.g.: `'fastaFileA_mashdist.txt'`

Code documentation

`flowcraft.templates.mashdist2json.`**`send_to_output`**(*master_dict*, *mash_output*, *sample_id*, *assembly_file*)

Send dictionary to output json file This function sends master_dict dictionary to a json file if master_dict is populated with entries, otherwise it won't create the file

Parameters

master_dict: dict dictionary that stores all entries for a specific query sequence in multi-fasta given to mash dist as input against patlas database

last_seq: str string that stores the last sequence that was parsed before writing to file and therefore after the change of query sequence between different rows on the input file

mash_output: str the name/path of input file to main function, i.e., the name/path of the mash dist output txt file.

sample_id: str The name of the sample being parse to .report.json file

flowcraft.templates.mashscreen2json module

flowcraft.templates.megahit module

Purpose

This module is intended execute megahit on paired-end FastQ files.

Expected input

The following variables are expected whether using NextFlow or the `main()` executor.

- **`sample_id`** [Sample Identification string.]
    - e.g.: `'SampleA'`
- **`fastq_pair`** [Pair of FastQ file paths.]
    - e.g.: `'SampleA_1.fastq.gz SampleA_2.fastq.gz'`
- **`kmers`** [Setting for megahit kmers. Can be either `'auto'`, `'default'` or a user provided list. All must be odd, in the range 15-255, increment <= 28]
    - e.g.: `'auto'` or `'default'` or `'55 77 99 113 127'`
- **`clear`** [If 'true', remove the input fastq files at the end of the] component run, IF THE FILES ARE IN THE WORK DIRECTORY

Generated output

- **`contigs.fa`** [Main output of megahit with the assembly]

  - e.g.: `contigs.fa`

- **`megahit_status`** [Stores the status of the megahit run. If it was successfully executed, it stores `'pass'`. Otherwise, it stores the `STDERR` message.]
  - e.g.: `'pass'`

Code documentation

`flowcraft.templates.megahit.`**`is_odd`**`(`*k_mer*`)`

`flowcraft.templates.megahit.`**`set_kmers`**`(`*kmer_opt*, *max_read_len*`)`

Returns a kmer list based on the provided kmer option and max read len.

Parameters

kmer_opt [str] The k-mer option. Can be either `'auto'`, `'default'` or a sequence of space separated integers, `'23, 45, 67'`.

max_read_len [int] The maximum read length of the current sample.

Returns

kmers [list] List of k-mer values that will be provided to megahit.

`flowcraft.templates.megahit.`**`fix_contig_names`**`(`*asseembly_path*`)`

Removes whitespace from the assembly contig names

Parameters

asseembly_path [path to assembly file]

Returns

str: Path to new assembly file with fixed contig names

`flowcraft.templates.megahit.`**`clean_up`**`(`*fastq*`)`

Cleans the temporary fastq files. If they are symlinks, the link source is removed

Parameters

fastq [list] List of fastq files.

flowcraft.templates.metaspades module

Purpose

This module is intended execute metaSpades on paired-end FastQ files.

Expected input

The following variables are expected whether using NextFlow or the `main()` executor.

- **`sample_id`** [Sample Identification string.]
  - e.g.: `'SampleA'`
- **`fastq_pair`** [Pair of FastQ file paths.]
  - e.g.: `'SampleA_1.fastq.gz SampleA_2.fastq.gz'`

- **kmers** [Setting for Spades kmers. Can be either `'auto'`, `'default'` or a user provided list.]
  - e.g.: `'auto'` or `'default'` or `'55 77 99 113 127'`

Generated output

- **contigs.fasta** [Main output of spades with the assembly]
  - e.g.: `contigs.fasta`
- **spades_status** [Stores the status of the spades run. If it was successfully executed, it stores `'pass'`. Otherwise, it stores the `STDERR` message.]
  - e.g.: `'pass'`

Code documentation

`flowcraft.templates.metaspades.`**`clean_up`**(*fastq*)

Cleans the temporary fastq files. If they are symlinks, the link source is removed

Parameters

fastq [list] List of fastq files.

`flowcraft.templates.metaspades.`**`set_kmers`**(*kmer_opt*, *max_read_len*)

Returns a kmer list based on the provided kmer option and max read len.

Parameters

kmer_opt [str] The k-mer option. Can be either `'auto'`, `'default'` or a sequence of space separated integers, `'23, 45, 67'`.

max_read_len [int] The maximum read length of the current sample.

Returns

kmers [list] List of k-mer values that will be provided to Spades.

flowcraft.templates.pATLAS_consensus_json module

Purpose

This module is intended to generate a json output from the consensus results from all the approaches available through options (mapping, assembly, mash screen)

Expected input

The following variables are expected whether using NextFlow or the `main()` executor.

- **mapping_json** [String with the name of the json file with mapping results.]
  - e.g.: `'mapping_SampleA.json'`
- **dist_json** [String with the name of the json file with mash dist results.]
  - e.g.: `'mash_dist_SampleA.json'`
- **screen_json** [String with the name of the json file with mash screen results.]

  - e.g.: `'mash_screen_sampleA.json'`

Code documentation

flowcraft.templates.pipeline_status module

Purpose

This module is intended to collect pipeline run statistics (such as time, cpu, RAM for each tasks) into a report JSON

Expected input

- `trace_file` : *Trace file generated by nextflow*

Code documentation

`flowcraft.templates.pipeline_status.`**`get_json_info`**`(`*fields*, *header*`)`

Parameters

fields

`flowcraft.templates.pipeline_status.`**`get_previous_stats`**`(`*stats_path*`)`

Parameters

workdir

flowcraft.templates.process_abricate module

flowcraft.templates.process_assembly module

flowcraft.templates.process_assembly_mapping module

Purpose

This module is intended to process the coverage report from the `assembly_mapping` process.

TODO: Better purpose

Expected input

The following variables are expected whether using NextFlow or the `main()` executor.

- **`sample_id`** [Sample Identification string.]
  - e.g.: `'SampleA'`
- **`assembly`** [Fasta assembly file.]
  - e.g.: `'SH10761A.assembly.fasta'`
- **`coverage`** [TSV file with the average coverage for each assembled contig.]

  - e.g.: `'coverage.tsv'`

- **`coverage_bp`** [TSV file with the coverage for each assembled bp.]
  - e.g.: `'coverage.tsv'`
- **`bam_file`** [BAM file with the alignment of reads to the genome.]
  - e.g.: `'sorted.bam'`
- **`opts`** [List of options for processing assembly mapping output.]
  1. **Minimum coverage for assembled contigs. Can be‘‘auto‘‘.**
     - e.g.: `'auto'` or `'10'`
  2. **Maximum number of contigs.**
     - e.g.: ‘100’
- **`gsize`: Expected genome size.**
  - e.g.: `'2.5'`

Generated output

- **`${sample_id}_filtered.assembly.fasta`** [Filtered assembly file in Fasta format.]
  - e.g.: `'SampleA_filtered.assembly.fasta'`
- **`filtered.bam`** [BAM file with the same filtering as the assembly file.]
  - e.g.: `filtered.bam`

Code documentation

`flowcraft.templates.process_assembly_mapping.`**`parse_coverage_table`**(*coverage_file*)

Parses a file with coverage information into objects.

This function parses a TSV file containing coverage results for all contigs in a given assembly and will build an `OrderedDict` with the information about their coverage and length. The length information is actually gathered from the contig header using a regular expression that assumes the usual header produced by Spades:

```
contig_len = int(re.search("length_(.+?)_", line).group(1))
```

Parameters

coverage_file [str] Path to TSV file containing the coverage results.

Returns

coverage_dict [OrderedDict] Contains the coverage and length information for each contig.

total_size [int] Total size of the assembly in base pairs.

total_cov [int] Sum of coverage values across all contigs.

`flowcraft.templates.process_assembly_mapping.`**`filter_assembly`**(*assembly_file*, *minimum_coverage*, *coverage_info*, *output_file*)

Generates a filtered assembly file.

This function generates a filtered assembly file based on an original assembly and a minimum coverage threshold.

Parameters

assembly_file [str] Path to original assembly file.

minimum_coverage [int or float] Minimum coverage required for a contig to pass the filter.

coverage_info [OrderedDict or dict] Dictionary containing the coverage information for each contig.

output_file [str] Path where the filtered assembly file will be generated.

`flowcraft.templates.process_assembly_mapping.`**`filter_bam`**(*coverage_info*, *bam_file*, *min_coverage*, *output_bam*)

Uses Samtools to filter a BAM file according to minimum coverage

Provided with a minimum coverage value, this function will use Samtools to filter a BAM file. This is performed to apply the same filter to the BAM file as the one applied to the assembly file in `filter_assembly()`.

Parameters

coverage_info [OrderedDict or dict] Dictionary containing the coverage information for each contig.

bam_file [str] Path to the BAM file.

min_coverage [int] Minimum coverage required for a contig to pass the filter.

output_bam [str] Path to the generated filtered BAM file.

`flowcraft.templates.process_assembly_mapping.`**`check_filtered_assembly`**(*coverage_info*, *coverage_bp*, *minimum_coverage*, *genome_size*, *contig_size*, *max_contigs*, *sample_id*)

Checks whether a filtered assembly passes a size threshold

Given a minimum coverage threshold, this function evaluates whether an assembly will pass the minimum threshold of `genome_size * 1e6 * 0.8`, which means 80% of the expected genome size or the maximum threshold of `genome_size * 1e6 * 1.5`, which means 150% of the expected genome size. It will issue a warning if any of these thresholds is crossed. In the case of an expected genome size below 80% it will return False.

Parameters

coverage_info [OrderedDict or dict] Dictionary containing the coverage information for each contig.

coverage_bp [dict] Dictionary containing the per base coverage information for each contig. Used to determine the total number of base pairs in the final assembly.

minimum_coverage [int] Minimum coverage required for a contig to pass the filter.

genome_size [int] Expected genome size.

contig_size [dict] Dictionary with the len of each contig. Contig headers as keys and the corresponding lenght as values.

max_contigs [int] Maximum threshold for contig number. A warning is issued if this threshold is crossed.

sample_id [str] Id or name of the current sample

Returns

x [bool] True if the filtered assembly size is higher than 80% of the expected genome size.

`flowcraft.templates.process_assembly_mapping.`**`get_coverage_from_file`**`(`*coverage_file*`)`

Parameters

coverage_file

`flowcraft.templates.process_assembly_mapping.`**`evaluate_min_coverage`**`(`*coverage_opt*, *assembly_coverage*, *assembly_size*`)`

Evaluates the minimum coverage threshold from the value provided in the coverage_opt.

Parameters

coverage_opt [str or int or float] If set to "auto" it will try to automatically determine the coverage to 1/3 of the assembly size, to a minimum value of 10. If it set to a int or float, the specified value will be used.

assembly_coverage [int or float] The average assembly coverage for a genome assembly. This value is retrieved by the *:py:func:parse_coverage_table* function.

assembly_size [int] The size of the genome assembly. This value is retrieved by the *py:func:get_assembly_size* function.

Returns

x: int Minimum coverage threshold.

`flowcraft.templates.process_assembly_mapping.`**`get_assembly_size`**`(`*assembly_file*`)`

Returns the number of nucleotides and the size per contig for the provided assembly file path

Parameters

assembly_file [str] Path to assembly file.

Returns

assembly_size [int] Size of the assembly in nucleotides

contig_size [dict] Length of each contig (contig name as key and length as value)

flowcraft.templates.skesa module

Purpose

This module is intended execute Skesa on paired-end FastQ files.

Expected input

The following variables are expected whether using NextFlow or the `main()` executor.

- **`sample_id`** [Sample Identification string.]
  - e.g.: `'SampleA'`
- **`fastq_pair`** [Pair of FastQ file paths.]
  - e.g.: `'SampleA_1.fastq.gz SampleA_2.fastq.gz'`
- **`clear`** [If 'true', remove the input fastq files at the end of the] component run, IF THE FILES ARE IN THE WORK DIRECTORY

Generated output

- **`${sample_id}_*.assembly.fasta`** [Main output of skesawith the assembly]
  - e.g.: `sample_1_skesa.fasta`
- **`clear`** [If 'true', remove the input fastq files at the end of the] component run, IF THE FILES ARE IN THE WORK DIRECTORY

Code documentation

`flowcraft.templates.skesa.`**`clean_up`**(*fastq*)

Cleans the temporary fastq files. If they are symlinks, the link source is removed

Parameters

fastq [list] List of fastq files.

flowcraft.templates.spades module

Purpose

This module is intended execute Spades on paired-end FastQ files.

Expected input

The following variables are expected whether using NextFlow or the `main()` executor.

- **`sample_id`** [Sample Identification string.]
  - e.g.: `'SampleA'`
- **`fastq_pair`** [Pair of FastQ file paths.]
  - e.g.: `'SampleA_1.fastq.gz SampleA_2.fastq.gz'`
- **`kmers`** [Setting for Spades kmers. Can be either `'auto'`, `'default'` or a user provided list.]
  - e.g.: `'auto'` or `'default'` or `'55 77 99 113 127'`
- **`opts`** [List of options for spades execution.]

1. **The minimum number of reads to consider an edge in the de Bruijn graph during the assembly.**

    - e.g.: `'5'`

2. **Minimum contigs k-mer coverage.**

    - e.g.: `['2' '2']`

- **`clear`** [If 'true', remove the input fastq files at the end of the] component run, IF THE FILES ARE IN THE WORK DIRECTORY

Generated output

- **`contigs.fasta`** [Main output of spades with the assembly]
    - e.g.: `contigs.fasta`
- **`spades_status`** [Stores the status of the spades run. If it was successfully executed, it stores `'pass'`. Otherwise, it stores the `STDERR` message.]
    - e.g.: `'pass'`

Code documentation

`flowcraft.templates.spades.`**`set_kmers`**`(`*kmer_opt*, *max_read_len*`)`

Returns a kmer list based on the provided kmer option and max read len.

Parameters

kmer_opt [str] The k-mer option. Can be either `'auto'`, `'default'` or a sequence of space separated integers, `'23, 45, 67'`.

max_read_len [int] The maximum read length of the current sample.

Returns

kmers [list] List of k-mer values that will be provided to Spades.

`flowcraft.templates.spades.`**`clean_up`**`(`*fastq*`)`

Cleans the temporary fastq files. If they are symlinks, the link source is removed

Parameters

fastq [list] List of fastq files.

flowcraft.templates.trimmomatic module

Purpose

This module is intended execute trimmomatic on paired-end FastQ files.

Expected input

The following variables are expected whether using NextFlow or the `main()` executor.

- **`sample_id`** [Pair of FastQ file paths.]

  - e.g.: `'SampleA'`

- **`fastq_pair`** [Pair of FastQ file paths.]
  - e.g.: `'SampleA_1.fastq.gz SampleA_2.fastq.gz'`
- **`trim_range`** [Crop range detected using FastQC.]
  - e.g.: `'15 151'`
- **`opts`** [List of options for trimmomatic]
  - e.g.: `'["5:20", "3", "3", "55"]'`
  - e.g.: `'[trim_sliding_window, trim_leading, trim_trailing, trim_min_length]'`
- **`phred`** [List of guessed phred values for each sample]
  - e.g.: `'[SampleA: 33, SampleB: 33]'`
- **`clear`** [If 'true', remove the input fastq files at the end of the] component run, IF THE FILES ARE IN THE WORK DIRECTORY

Generated output

The generated output are output files that contain an object, usually a string. (Values within `${}` are substituted by the corresponding variable.)

- **`${sample_id}_*P*`: Pair of paired FastQ files generated by Trimmomatic**
  - e.g.: `'SampleA_1_P.fastq.gz SampleA_2_P.fastq.gz'`
- **`trimmomatic_status`: Stores the status of the trimmomatic run. If it was successfully executed, it stores 'pass'. Other**
  - e.g.: `'pass'`

Code documentation

`flowcraft.templates.trimmomatic.`**`parse_log`**`(`*log_file*`)`

Retrieves some statistics from a single Trimmomatic log file.

This function parses Trimmomatic's log file and stores some trimming statistics in an `OrderedDict` object. This object contains the following keys:

- `clean_len`: Total length after trimming.
- `total_trim`: Total trimmed base pairs.
- `total_trim_perc`: Total trimmed base pairs in percentage.
- `5trim`: Total base pairs trimmed at 5' end.
- `3trim`: Total base pairs trimmed at 3' end.

Parameters

log_file [str] Path to trimmomatic log file.

Returns

x [`OrderedDict`] Object storing the trimming statistics.

flowcraft.templates.trimmomatic.**write_report**(*storage_dic*, *output_file*, *sample_id*)

Writes a report from multiple samples.

Parameters

storage_dic [dict or OrderedDict] Storage containing the trimming statistics. See *parse_log()* for its generation.

output_file [str] Path where the output file will be generated.

flowcraft.templates.trimmomatic.**trimmomatic_log**(*log_file*, *sample_id*)

flowcraft.templates.trimmomatic.**clean_up**(*fastq_pairs*, *clear*)

Cleans the working directory of unwanted temporary files

flowcraft.templates.trimmomatic.**merge_default_adapters**()

Merges the default adapters file in the trimmomatic adapters directory

Returns

str Path with the merged adapters file.

flowcraft.templates.trimmomatic_report module

Purpose

This module is intended parse the results of the Trimmomatic log for a set of one or more samples.

Expected input

The following variables are expected whether using NextFlow or the main() executor.

- **log_files: Trimmomatic log files.**
  - e.g.: 'Sample1_trimlog.txt Sample2_trimlog.txt'

Generated output

- trimmomatic_report.csv : Summary report of the trimmomatic logs for all samples

Code documentation

flowcraft.templates.trimmomatic_report.**parse_log**(*log_file*)

Retrieves some statistics from a single Trimmomatic log file.

This function parses Trimmomatic's log file and stores some trimming statistics in an OrderedDict object. This object contains the following keys:

- clean_len: Total length after trimming.
- total_trim: Total trimmed base pairs.
- total_trim_perc: Total trimmed base pairs in percentage.
- 5trim: Total base pairs trimmed at 5' end.
- 3trim: Total base pairs trimmed at 3' end.

Parameters

log_file [str] Path to trimmomatic log file.

Returns

x [`OrderedDict`] Object storing the trimming statistics.

`flowcraft.templates.trimmomatic_report.`**`write_report`**(*storage_dic*, *output_file*, *sample_id*)

Writes a report from multiple samples.

Parameters

storage_dic [dict or `OrderedDict`] Storage containing the trimming statistics. See *`parse_log()`* for its generation.

output_file [str] Path where the output file will be generated.

sample_id [str] Id or name of the current sample.

Module contents

Placeholder for template generation docs

18.1.3 flowcraft.tests package

Submodules

flowcraft.tests.data_pipelines module

flowcraft.tests.test_assemblerflow module

flowcraft.tests.test_engine module

flowcraft.tests.test_pipeline_parser module

`flowcraft.tests.test_pipeline_parser.`**`test_get_lanes`**()

`flowcraft.tests.test_pipeline_parser.`**`test_linear_connection`**()

`flowcraft.tests.test_pipeline_parser.`**`test_two_fork_connection`**()

`flowcraft.tests.test_pipeline_parser.`**`test_two_fork_connection_mismatch_lane`**()

`flowcraft.tests.test_pipeline_parser.`**`test_multi_fork_connection`**()

`flowcraft.tests.test_pipeline_parser.`**`test_linear_lane_connection`**()

`flowcraft.tests.test_pipeline_parser.`**`test_linear_multi_lane_connection`**()

`flowcraft.tests.test_pipeline_parser.`**`test_get_source_lane`**()

`flowcraft.tests.test_pipeline_parser.`**`test_get_source_lane_2`**()

`flowcraft.tests.test_pipeline_parser.`**`test_parse_pipeline`**()

`flowcraft.tests.test_pipeline_parser.`**`test_parse_pipeline_file`**()

`flowcraft.tests.test_pipeline_parser.`**`test_unique_id_len`**()

`flowcraft.tests.test_pipeline_parser.`**`test_remove_id`**`()`

flowcraft.tests.test_process_details module

flowcraft.tests.test_processes module

flowcraft.tests.test_sanity module

Module contents

18.2 Submodules

18.2.1 flowcraft.flowcraft module

18.3 Module contents

Python Module Index

Index